NeoDatis Object Database

2.1

Publication date 08/12/2009

Table of Contents

1. Overview 1
    1.1. Simple 1
    1.2. Small 1
    1.3. Fast 1
    1.4. Safe and robust 1
    1.5. One single database file 1
    1.6. Multiplatform 2
    1.7. Data are always available 2
    1.8. Productivity 2
    1.9. Easy to integrate 2
    1.10. Refactoring 2
    1.11. License 2
2. Download 3
3. Wiki and forum 4
4. How to execute NeoDatis 5
5. IDE Integration 6
    5.1. Eclipse 6
6. Migration from previous versions 7
7. How to open a NeoDatis database 8
8. Storing objects 9
9. Retrieving objects 13
    9.1. Retrieving all objects of a specific class 13
    9.2. Criteria queries 14
        9.2.1. Equality 15
        9.2.2. like 16
        9.2.3. greater than (gt) 16
        9.2.4. greater or equal (ge) 16
        9.2.5. less than (lt) 17
        9.2.6. less or equal (le) 17
        9.2.7. contain -To test if an array or a collection contains a specific value 18
        9.2.8. Null Objects 19
        9.2.9. Where on collection or array size 19
        9.2.10. Logical 19
        9.2.11. More criteria query examples 19
    9.3. Native Query 20
    9.4. Retrieving an object by its OID 22
    9.5. Query tuning 22
        9.5.1. Load Depth 22
        9.5.2. Indexes 23

9.5.2.1. Creating an index 23
9.5.2.2. Checking if an index exists 23
9.5.2.3. Deleting an index 23
9.5.2.4. Rebuilding an index 23
9.6. Query Factory 23
10. Object Values API 25
10.1. Aggregate functions 26
10.2. Dynamic Views 29
10.3. Custom Functions 30
11. Updating Objects 32
12. Deleting Objects 33
13. Using NeoDatis in client/server mode 35
14. Reconnecting Objects to Session 37
15. Database Encoding 38
16. Object Explorer 39
16.1. Browsing data 40
16.1.1. Table View 41
16.1.2. Object View 41
16.2. Queries 42
16.3. Updating objects 43
16.4. Creating new objects 45
17. XML 47
17.1. Using Object Exporer 47
17.1.1. Exporting 47
17.1.2. Importing 47
17.2. Using NeoDatis API 48
18. NeoDatis Extended API 50
19. User/Password protection 51
20. Best practices 52
21. Using NeoDatis in web applications 53
21.1. Web App Example 53
22. Advanced features 55
23. NeoDatis Context 57
24. Storage Engine Plugins 61
25. NeoDatis on Google Android 65
26. NeoDatis and Groovy 66
27. Annexes 67

Chapter 1. Overview

NeoDatis ODB is an open source Object Oriented Database. It is a real native and transparent persistence layer for Java, .Net, Groovy, Scala and Google Android.

With NeoDatis ODB you can store and retrieve your objects with a single line of code without the need of mapping your objects to any table.

So NeoDatis ODB will increase your productivity and let you concentrate on your business logic.

1.1 Simple

NeoDatis is very simple and intuitive: the learning time is very short. Have a look at the NeoDatis 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. NeoDatis simply stores the objects the way they are. NeoDatis requires zero administration and zero installation.

1.2 Small

The NeoDtais runtime is less than 800k and is distributed as a single jar/dll that can easily be packaged in any type of application.

1.3 Fast

NeoDatis can store more than 30000 objects per second.

1.4 Safe and robust

NeoDatis 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

NeoDatis uses a single file to store all data:

- The Meta-model

- The Objects
- The indexes

1.6 Multiplatform

NeoDatis works with Java, .Net, Groovy, Scala and Google Andoid

1.7 Data are always available

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

1.8 Productivity

NeoDatis 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.

1.9 Easy to integrate

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

1.10 Refactoring

NeoDatis currently supports 4 types of refactoring:

- Renaming a class (via API or Object Explorer)
- Renaming a Field (via API or Object Explorer)
- Adding a new Field (automatically detected)
- Removing a Field (automatically detected)

1.11 License

NeoDatis is distributed under the LGPL license.

Chapter 2. Download

The last NeoDatis Object Database download can be found at http://www.neodatis.org.

Chapter 3. Wiki and forum

Check the http://www.neodatis.org/documentation site to access the wiki that contains documentation about NeoDatis architecture!

For any question about NeoDatis, the best place is the source forge forum at http://sourceforge.net/forum/?group_id=179124.

Chapter 4. How to execute NeoDatis

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

```
java –cp neodatis-odb.jar [your-class-name]
```

Depending on which storage engine is configured, you may need extra jars on the classpath.

Chapter 5. IDE Integration

To use NeoDatis in your favorite IDE, the classpath of your project must be updated to contain the NeoDatis runtime jar and the plugin jars.

5.1 Eclipse

Using NeoDatis 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 NeoDatis runtime jar:

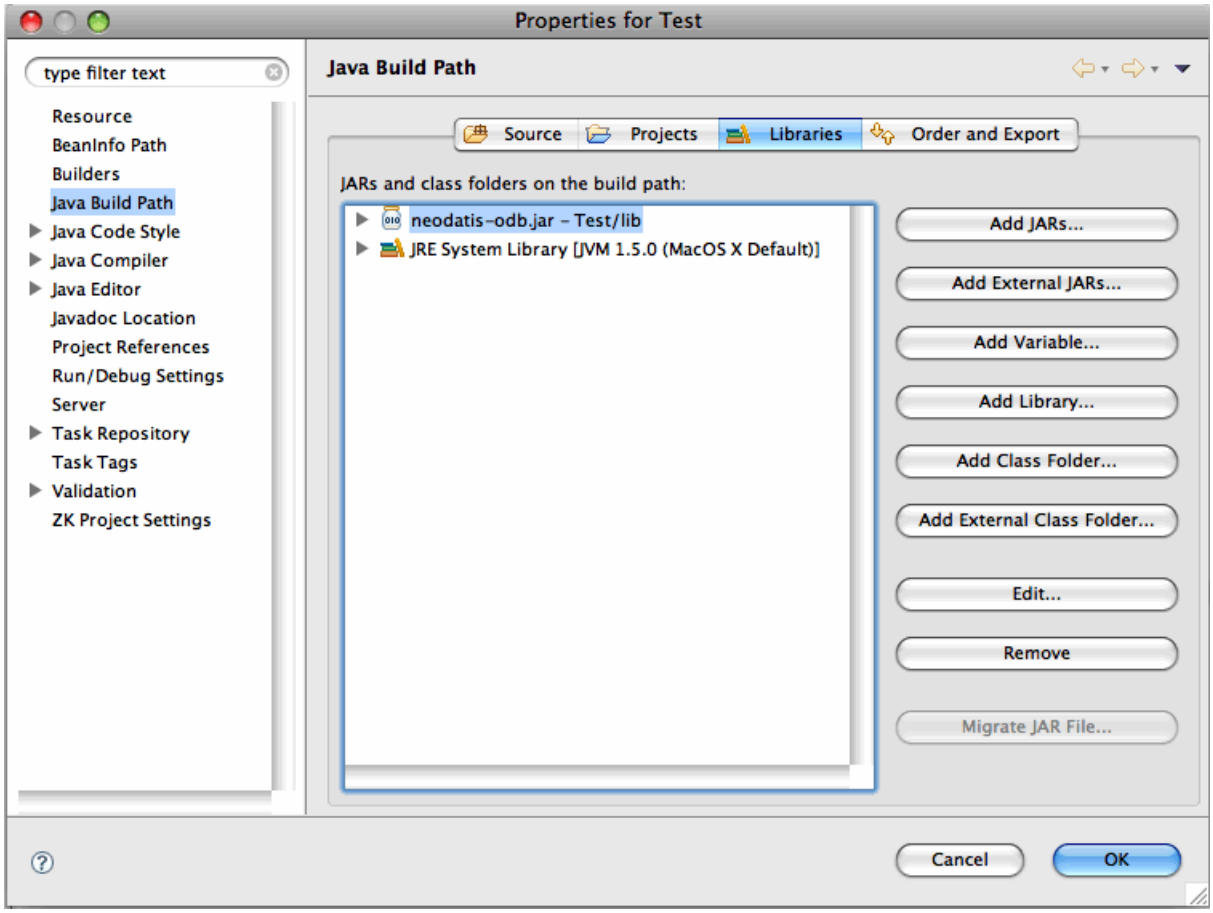

Figure 5.1. Eclipse integration

Chapter 6. Migration from previous versions

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 1.9 version. This can be done easily with the ObjectExplorer graphical application.

Chapter 7. How to open a NeoDatis database

In version 2, NeoDatis can store its information in different storage engine like BerkeleyDb, Voldemort, JDBM,..... So, when creating a new database, there is at least one information to give to NeoDatis, the storage engine. There are 3 ways to do that:

- (1) Using a configuration file to specify the config parameters
- (2) Using a NeoDatisConfig instance (NeoDatis.getConfig()) to specify the config parameters
- (3) Using the NeoDatisGlobalConfig global (NeoDatisGlobalConfig.get()) class to configure the parameters in a static way

Using a configuration file

Chapter 8. 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.

Note: getters, setters and toString are not necessary for NeoDatis.

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.neodatis";
```

And then

```
public void step1() throws Exception{

  ODB odb = null;
  try{

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

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

         // Store the object
         odb.store(sport);
     } finally{
         if(odb!=null){
          // 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 = NeoDatis.open(ODB_NAME);

        // Get all object of type Sport
        Objects<Sport> sports = query(Sport.class).objects();

        // display each object
        Sport sport = null;
        while(sports.hasNext()){
            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;
}
    public Game(Date when, Sport sport, Team team1, Team team2) {
        this.when = when;
     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. NeoDatis 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 = NeoDatis.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, NeoDatis should contain:

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

Let's check this, here is the output of querying objects of each type: 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 be enough.

```
Step 2 : 1 games(s):
1 : Thu May 22 06:29:13 BRT 2009 : 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
```

Chapter 9. Retrieving objects

In the previous example, we learned how to store objects. Now, obviously, we need to get these objects back. For instance, NeoDatis 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

9.1 Retrieving all objects of a specific class

The ODB interface has a method query that receives query parameter or a query. Then with the query instance, one can call the objects() method that returns an instance 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
     odb = NeoDatis.open(ODB_NAME);

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

     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, retrieves a list of all objects of type Player and displays each one.

9.2 Criteria queries

CriteriaQuery let's you specify 'Where conditions' on objects that the query result must contain. The NeoDatis 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 = NeoDatis.open(ODB_NAME);

    Query query = odb.query(Player.class,W.equal("name", "olivier"));

    Objects<Player> players = query.objects();

    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.

```
Query query = odb.query(Player.class, W.equal("name", "olivier"));
```

A powerful feature of 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:

```
Query query = odb.query(Player.class,W.equal("favoriteSport.name","volley-ball"));
```

Another way to get the same result is :

- to get the object volley-ball
- then use the object in the query

Here is the code:

Retrieve the 'volley-ball' object:

```
Query query = odb.query(Sport.class,W.equal("name", "volley-ball"));

    Sport volleyBall = (Sport) query.objects().getFirst();
```

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

```
// build a query to get all players that play volley ball, using the
    // volley ball object
    Query query = odb.query(Player.class,W.equal("favoriteSport",volleyBall));


    Objects players = query.objects();
```

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:

9.2.1 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 iequal(String attributeName,Object value)`	case insensitive equal

9.2.2 like

`public static ICriterion like(String attributeName,String value)`	for patterns like 'name="pet%"'
`public static ICriterion ilike(String attributeName,String value)`	case insensitive option

9.2.3 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 value
`public static ICriterion gt(String attributeName,short value)`	for primitive short value
`public static ICriterion gt(String attributeName,byte value)`	for primitive byte value
`public static ICriterion gt(String attributeName,float value)`	for primitive float value
`public static ICriterion gt(String attributeName,double value)`	for primitive double value
`public static ICriterion gt(String attributeName,long value)`	for primitive long value
`public static ICriterion gt(String attributeName,char value)`	for primitive char value

9.2.4 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 value
`public static ICriterion ge(String attributeName,short value)`	for primitive short value

`public static ICriterion ge(String attributeName,byte value)`	for primitive byte value
`public static ICriterion ge(String attributeName,float value)`	for primitive float value
`public static ICriterion ge(String attributeName,double value)`	for primitive double value
`public static ICriterion ge(String attributeName,long value)`	for primitive long value
`public static ICriterion ge(String attributeName,char value)`	for primitive char value

9.2.5 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 value
`public static ICriterion lt(String attributeName,short value)`	for primitive short value
`public static ICriterion lt(String attributeName,byte value)`	for primitive byte value
`public static ICriterion lt(String attributeName,float value)`	for primitive float value
`public static ICriterion lt(String attributeName,double value)`	for primitive double value
`public static ICriterion lt(String attributeName,long value)`	for primitive long value
`public static ICriterion lt(String attributeName,char value)`	for primitive char value

9.2.6 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 value
`public static ICriterion le(String attributeName,short value)`	for primitive short value
`public static ICriterion le(String attributeName,byte value)`	for primitive byte value
`public static ICriterion le(String attributeName,float value)`	for primitive float value
`public static ICriterion le(String attributeName,double value)`	for primitive double value
`public static ICriterion le(String attributeName,long value)`	for primitive long value
`public static ICriterion le(String attributeName,char value)`	for primitive char value

9.2.7 contain -To test if an array or a collection contains 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

9.2.8 Null Objects

`public static ICriterion isNull(String attributeName)`	only null objects
`public static ICriterion isNotNull(String attributeName)`	only not null objects

9.2.9 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

9.2.10 Logical

`public static ComposedExpression or()`	a OR expression
`public static ComposedExpression and()`	a AND expression
`public static IExpression not(ICriterion criterion)`	negate a restriction

9.2.11 More criteria query examples

```
```

```
// users that have a profile which name is 'profile2'
Query query = odb.query(User.class, W.equal("profile.name","profile2"));

// users that have a specific profile p0
query = odb.query(User.class, W.equal("profile", p0));

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

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

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

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

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

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

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

// TestClass objects where the attribute 'string1' is equal to  'test
// class 1' or 'test class 2'
query = odb.query(TestClass.class,W.equal("string1", "test class1").or(W.equal("string1", "test class

// TestClass objects where the attribute 'string1' is not equal to 'test class 2'
query = odb.query(TestClass.class,W.equal("string1", "test class2").not());

// TestClass objects where the condition 'string1' is equal to 'test
// class 0' or the attribute 'bigDecimal1' is equal to 5 is not matched
Query query = odb.query(TestClass.class,W.equal("string1", "test class0").or(W.equal("bigDecimal1", new

// TestClass object where the attribute 'string1' is equal to 'test class 2' or 'test class 3' or 'test
//'test class 5'
//  The query result will be ordered by the fields 'boolean1' and'int1'
Criterion c = W.equal("string1", "test class2").or(W.equal("string1", "testclass 3")).or(W.equal("strin
"test class4")).or(W.equal("string1", "test class 5"));

Query query = odb.query(TestClass.class, c);
query.orderByDesc("boolean1,int1");

// Function objects where the name is not equal to 'function 2'
query = odb.query(Function.class, W.equal("name","function 2").not()));
```

9.3 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 NeoDatis, you must implement the class NativeQuery.

A native query that returns 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 = NeoDatis.open(ODB_NAME);
    Query query = new NativeQuery<Player>() {

     public boolean match(Player player) {
      return player.getFavoriteSport().getName().toLowerCase().startsWith("volley");
     }
    };

    Objects<Player> players = query.objects();

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

Warning : In client/Server mode, native query are executed on the server. So the NeoDatis 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 NeoDatis use serialization to send the Native Query object to the server.

9.4 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 (Object ID) 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!

9.5 Query tuning

Queries have a QueryParameters instance tha can be used to set parameters to tune the query execution:

```
odb.query(Country.class)
    .getQueryParameters()
     .setStartAndEndIndex(10, 20)
     .setInMemory(true)
     .setLazyLists(true);
```

The boolean inMemory is used by NeoDatis to know if all objects must be 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, NeoDatis 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 Query q = odb.query(Function.class); q.getqueryParameters().setStartAndEndIndex(0,10000) to to get the first 10000 objects and q.getqueryParameters().setStartAndEndIndex(10000,20000) to get the next 10000.

Default values are -1 (which disables query result paging).

9.5.1 Load Depth

In some situation, you may need to load objects without loading all its relations, this can be done using setLoadDepth(x);

```
odb.query(Country.class).getQueryParameters().setLoadDepth(1);
```

9.5.2 Indexes

To speed up query executions, you can add indexes to your classes. Indexes can be declared on various fields of a class. They can be unique or non unique.

9.5.2.1 Creating an index

Here is an example of a unique index declaration on the class Sport for the field ‘name’:

```
ODB odb = NeoDatis(ODB_NAME);
      String [] fieldNames = {"name"};

      odb.getClassRepresentation(Sport.class).addUniqueIndexOn("sport-index", fieldnames,true);
```

Here is an example of a non unique index declaration on the class User for the fields ‘name’ and 'email' (imagine the class has 2 String attributes name and email):

```
ODB odb = NeoDatis.open(ODB_NAME);
      String [] fieldNames = {"name","email"};

      odb.getClassRepresentation(User.class).addIndexOn("user-index", fieldnames,true);
```

The last parameter true is to ask NeoDatis to log what it is doing while creating the index.

9.5.2.2 Checking if an index exists

Here is an example of how to check if the index 'sport-index' exists on class Sport:

```
boolean exist = odb.getClassRepresentation(Sport.class).existIndex("sport-index");
```

9.5.2.3 Deleting an index

Here is an example of how to delete the index 'sport-index' of class Sport:

```
odb.getClassRepresentation(Sport.class).deleteIndex("sport-index", true);
```

9.5.2.4 Rebuilding an index

Here is an example of how to rebuild the index 'sport-index' on class Sport:

```
odb.getClassRepresentation(Sport.class).rebuildIndex("sport-index", true);
```

9.6 Query Factory

Sometimes you may want to create queries disconnected from a session or just create a class with most used queries to be able to reuse them. You can use the QueryFactory to create such queries. Here is an example:

```
// create some disconnected queries
public class MyQueries{
 public static final Query allFunctionsQuery = QueryFactory.query(Function.class);
 public static final Query adminUsersQuery =
   QueryFactory.query(User.class,w.equal("profile.name" , "admin" );
}

// How to use these queries
ODB odb = NeoDatis.open(baseName);

// reuse a previously created query
Objects<User>> adminUsers = odb.query(MyQueries.adminUsersQuery).objects();

odb.close();
```

Chapter 10. 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. 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.

Here is a simple example of what Object Values API can do.

The following code create 10 players. Each player has its own favorite sport.

Then we retrieve the name of the player and the name of its favorite sport (this actually retrieve only the name of the sport and not the whole Sport object).

```
ODB odb = null;
  System.out.println("\nStep 18 : Object Values");
  try {
   // Open the database
   odb = NeoDatis.open(ODB_NAME);

   // Creates 100 players
   for(int i=0;i<100;i++){
    odb.store(new Player("player "+i,new Date(),new Sport("Sport "+i)));
   }
   // Close the database
   odb.close();

   // Opens the database
   odb = NeoDatis.open(ODB_NAME);
   // Executes the Object Values query
   Values values = odb.getValues(new ValuesCriteriaQuery(Player.class)
    .field("name")
    .field("favoriteSport.name", "sport"));

   // Iterate of the result
   while(values.hasNext()){
    // Each object is an ObjectValues that gives access to the fields
    ObjectValues objectValues= (ObjectValues) values.next();
    // Prints the name of the player and the name of the sport
    // Retrieve the player name by alias and the sport name by index
    System.out.println(
     objectValues.getByAlias("name") + " plays " + objectValues.getByIndex(1));
```

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

Here is the output of the program:

```
player 0 plays sport 0
player 1 plays sport 1
player 2 plays sport 2
player 3 plays sport 3
player 4 plays sport 4
```

Now, let's take a look at the API. In the following sections, we will be using three classes in the examples:

- Class Function : with a name attribute (String)
- Class Profile : with a name attribute (String) and a list of Functions
- Class User : with a name(String), an email(String), number of login(integer) and profile (Profile)

10.1 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:

`IValuesQuery.sum(StringfieldName)`	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 the results by the fields

`IValuesQuery.`
`sublist(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
`ICriterion iequal (String attributeName,Object value)`	case insensitive equal

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

1 Sum function

Retrieving the sum of logins

API	`odb.getValues(new ValuesCriteriaQuery(User.class).sum("nbLogins")`
Sql	`select sum(nbLogins) from User`

2 Average function

Retrieving the average number of logins

API	`odb.getValues(new ValuesCriteriaQuery(User.class).avg("nbLogins")`
Sql	`select avg(nbLogins) from User`

3 Minimum and Maximum

Retrieving the minimum and maximum number of logins

API	`odb.getValues(new ValuesCriteriaQuery(User.class)`
	`.min("nbLogins","min of nbLogins")`
	`.max("nbLogins","max of nbLogins");`

Sql	`select min(nbLogins) , max(nbLogins) from User`

4 Counting values

Counting the number of users

API	`odb.getValues(new ValuesCriteriaQuery(User.class).count("nb users");`
Sql	`select count(*) from User`

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

API	`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`

6 Sublist and List size

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.

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

Sql	`?`

The result 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
```

10.2 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:

API	`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:

API	`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.

10.3 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 five 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, notice that 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"));
```

Chapter 11. Updating Objects

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

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

 try {
  // Open the database
  odb = NeoDatis.open(ODB_NAME);
  IQuery query = new CriteriaQuery(Sport.class,
     Where.equal("name", "volley-ball"));

  Objects<Sport> 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 = NeoDatis.open(ODB_NAME);

  // Now query the database to 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:

```
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 NeoDatis, calling the store method will create a new one!

Chapter 12. Deleting Objects

There are two ways to delete an object:

- Getting the object and ask NeoDatis to delete it
- If you have the id of the object, ask NeoDatis to delete the object with this specific id

1 Deleting an object

```
public void step13() throws Exception {

 ODB odb = null;

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

  Objects<Player> players = odb.getObjects(query);

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

  odb.delete(agassi);

  odb.close();

  odb = NeoDatis.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();
  }
 }
}
```

2 Deleting an object using its internal ID

```
 public void step14() throws Exception {
```

```
 ODB odb = null;

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

  // Now query the databas eto check the change
  Objects<Player> 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();
  }
 }
}
```

Chapter 13. Using NeoDatis in client/server mode

NeoDatis 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 machines
- Optimized client/server mode where Clients and Server run in the same Virtual Machine (version 1.9+). Useful when using NeoDatis ODB in a Web application. This mode is much more faster.

The first thing to do this is to start the NeoDatis ODB server. A server needs some parameters to be created:

- The port on which it must be executed: port that will receive client connections. Make sure this port is free on the server.
- 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 = NeoDatis.openServer(8000);

// Tells the server to manage base 'base1' that points to the physical
// file /users/neodatis/db/base1.neodatis
server.addBase("base1", “/users/neodatis/db/base1.neodatis);

// 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 you are 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 = NeoDatis.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.

Here is the 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 = NeoDatis.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 = NeoDatis.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();
  }
 }
}
```

Warning: when using the Same VM client server mode, you can not open 2 connections in the same thread, each connection must be opened in its own thread.

Chapter 14. Reconnecting Objects to Session

When opening a NeoDatis base, all objects that are stored or selected from the NeoDatis instance are connected to the current session. Sometimes, you may need to reconnect objects (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 unable the reconnectObjectsToSession mode by :

- calling

  ```
  NeoDatisGlobalConfig.get().setReconnectObjectsToSession(true);
  ```

  (this enables reconnect mode for all NeoDatis instances)

- or create a config

  ```
  NeoDatisConfig config = NeoDatis.getConfig().setReconnectObjectsToSession(true);
  ```

  and use this config when opening the db :

  ```
   ODB odb = NeoDatis.open(ODB_NAME, config);
  ```

  or just call directly

  ```
  ODB odb = NeoDatis.open(ODB_NAME,NeoDatis.getConfig().setReconnectObjectsToSession(true));
  ```

This will automatically re-connect previously loaded objects the new session allowing a direct update (without having to reload it before). The default value of 'ReconnectObjectsToSession is false

The following code creates two objects of type Function in the database. One with name 'Function 1' and the other with name 'Function A'

```
ODB odb = NeoDatis.open(ODB_NAME);
Function f1 = new Function("Function 1");
odb.store(f1);
odb.close();

odb = NeoDatis.open(ODB_NAME);
f1.setName("Function A");
odb.close();
```

But, if ReconnectObjectsToSession is on, then the second odb.store call will be understood by NeoDatis as an update and not an insert, so the code will create a single object and update it with the name 'Function A'.

Chapter 15. Database Encoding

NeoDatis uses "ISO8859-1" as its default encoding.

The NeoDatisGlobalConfig.get().setDatabaseCharacterEncoding(Java Encoding) can be called to set the desired encoding.

The NeoDatisGlobalConfig.get().setLatinDatabaseCharacterEncoding() uses the ISO8859-1 encoding and is suitable for most of Latin languages applications.

Warning 1 :Encoding must always be configured before opening the NeoDatis database
Warning 2 :In client server mode, the correct encoding must be set on both client and server side.

Chapter 16. Object Explorer

NeoDatis Object Explorer is a graphical tool that comes with NeoDatis Database to manage the data. The tool has the following features:

- Browse objects
- Query objects
- Create objects
- Update objects
- Delete objects
- Export/import a NeoDatis ODB Database
- Refactor the database

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

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

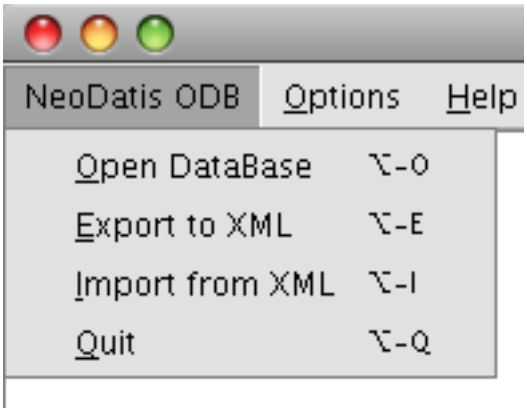

Figure 16.1. NeoDatis Object Explorer main menu

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

Clicking on a class, ObjectExplorer displays a contextual menu:

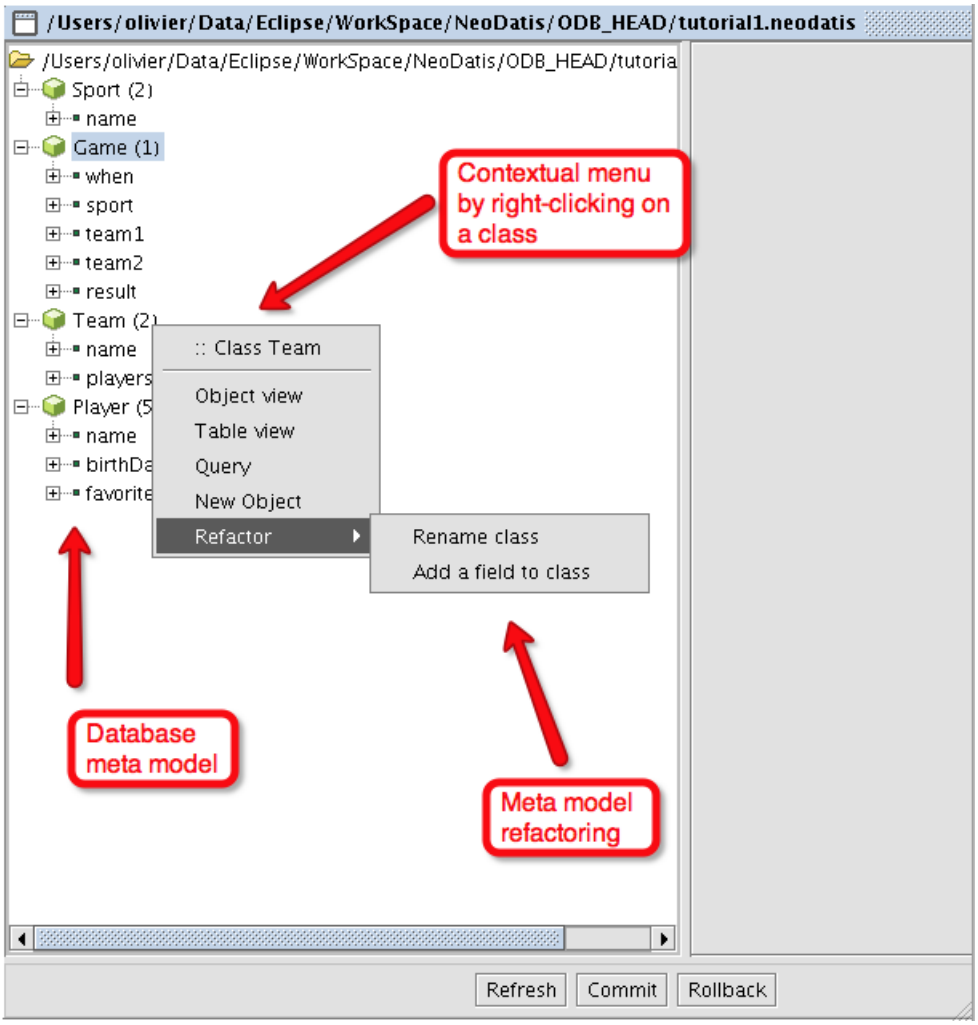

Figure 16.2. Meta model browser

- 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.

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.

16.1.1 Table View

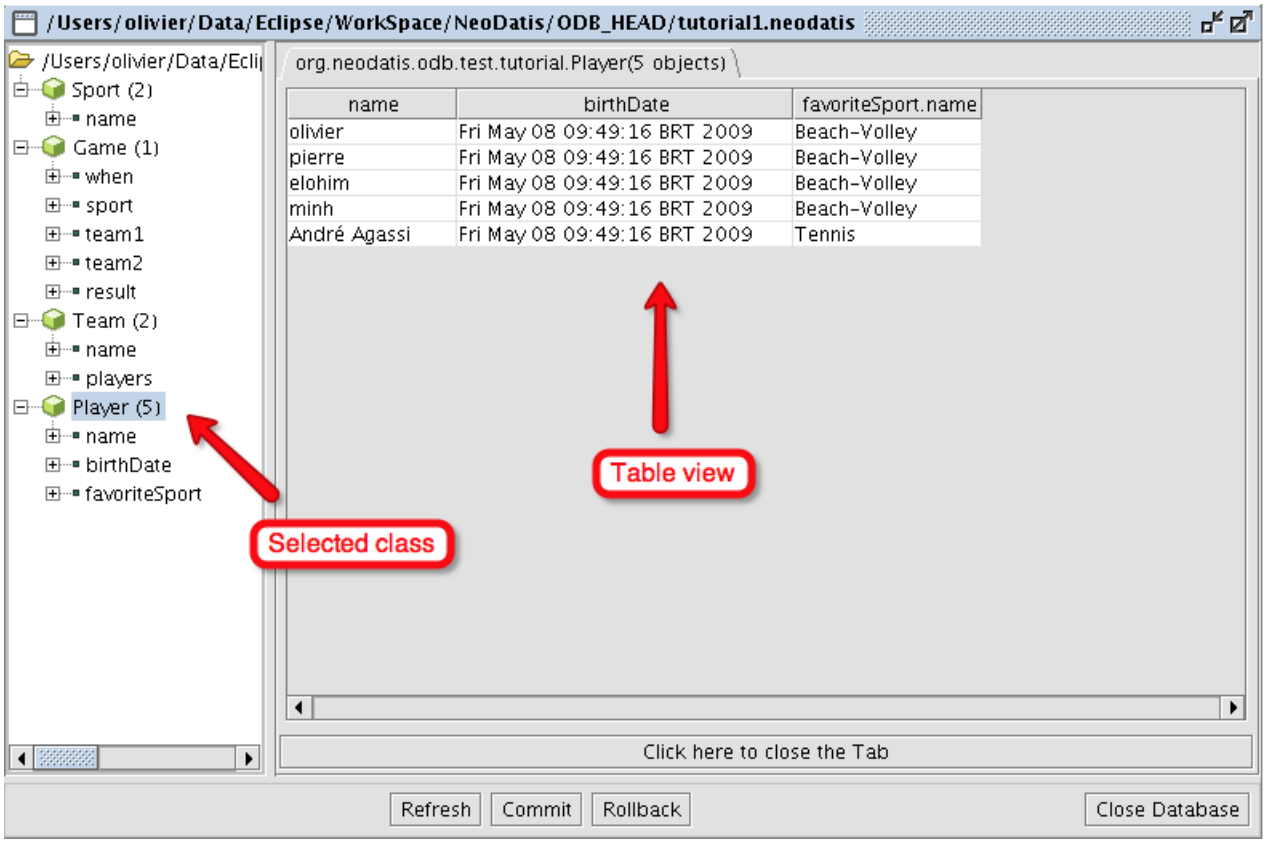

Figure 16.3. Browsing object using Table View

16.1.2 Object View

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

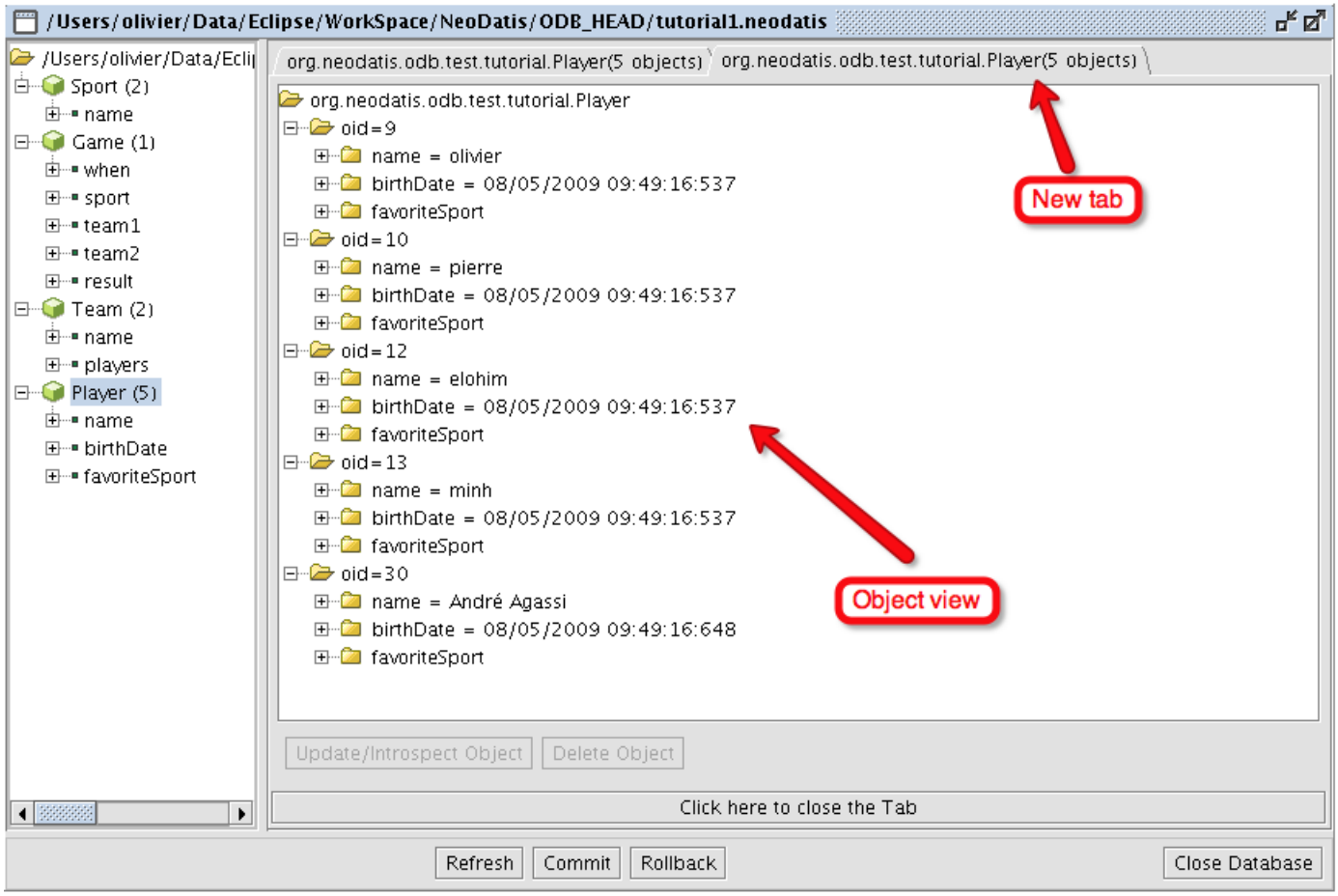

Figure 16.4. Browsing object using Object View

16.2 Queries

The Object Explorer offers a graphic interface to build a CriteriaQuery to query a subset of objects:

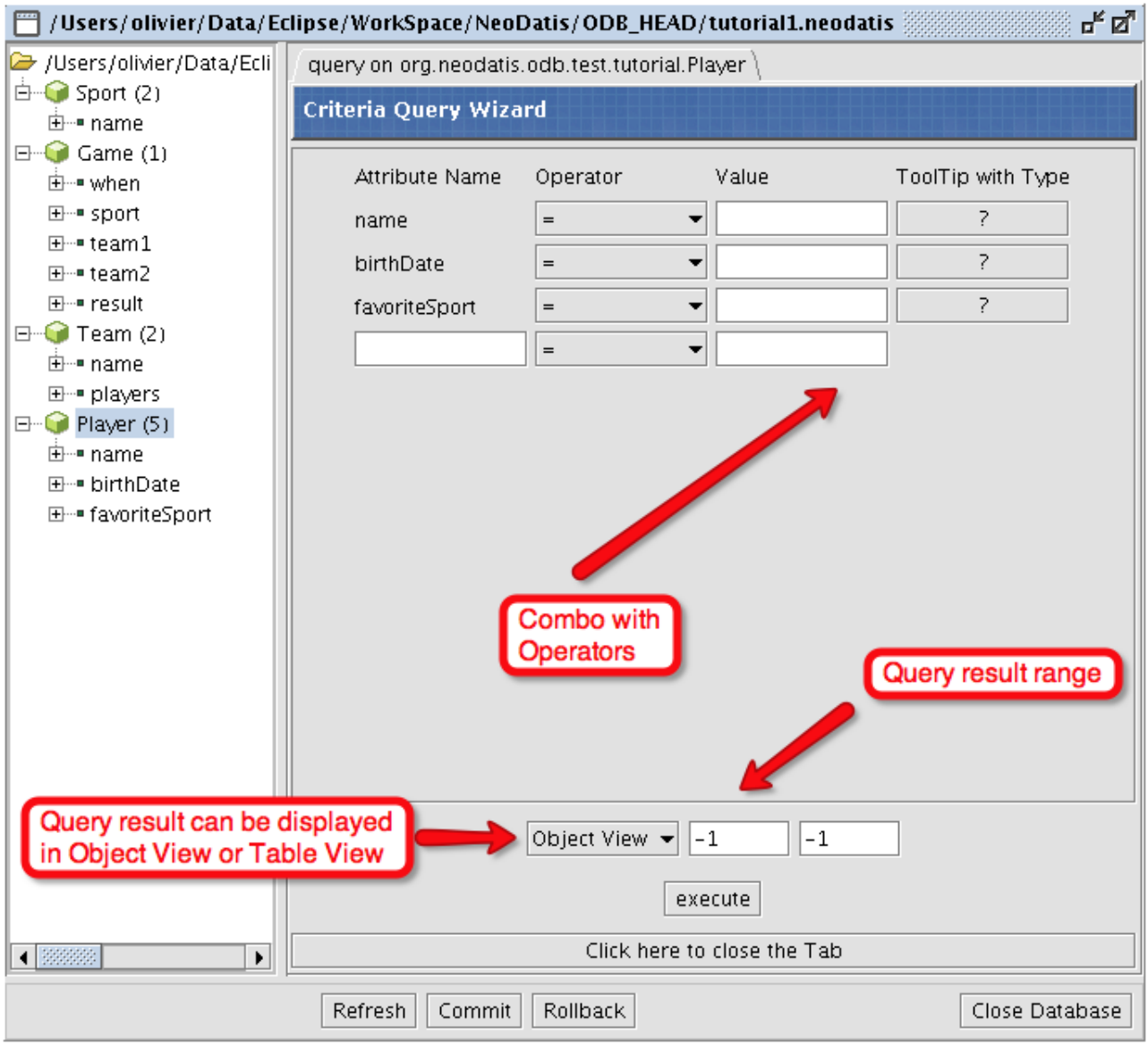

Figure 16.5. The query builder

16.3 Updating objects

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

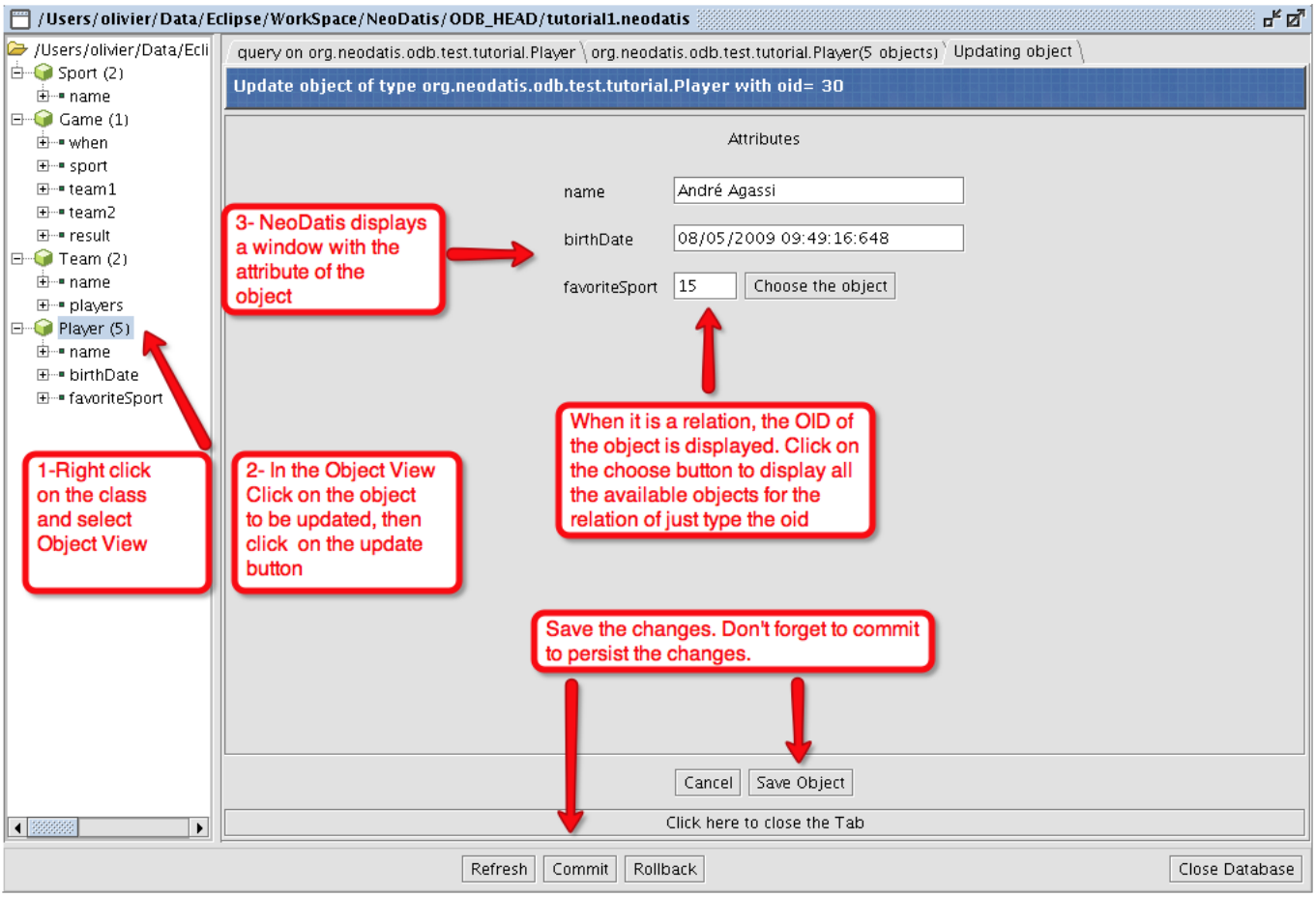

Figure 16.6. Updating an object

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

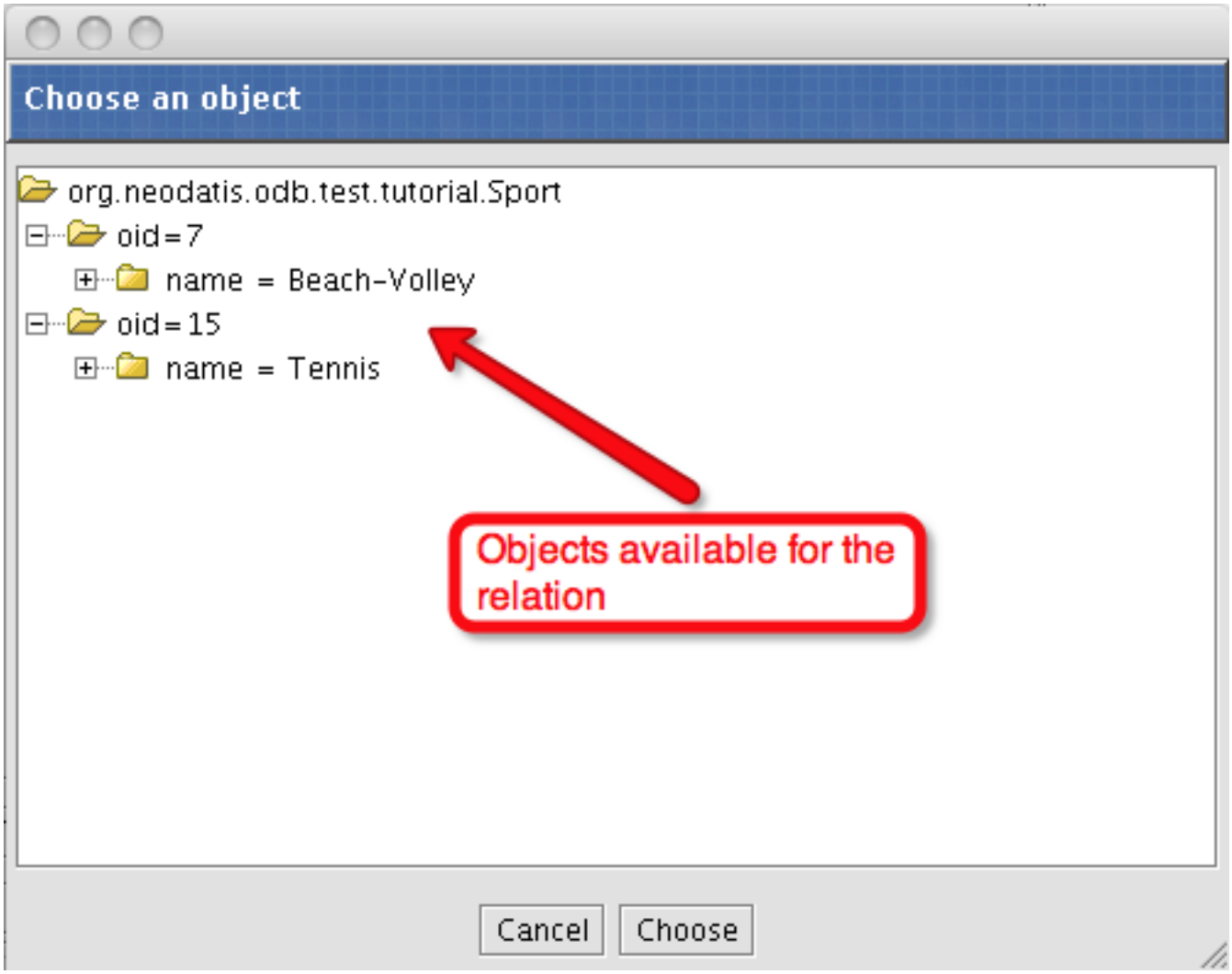

Figure 16.7. How to define a relation

16.4 Creating new objects

It is also possible to create objects using Object Explorer. Just select a class on the left panel and click the `New Object` button:

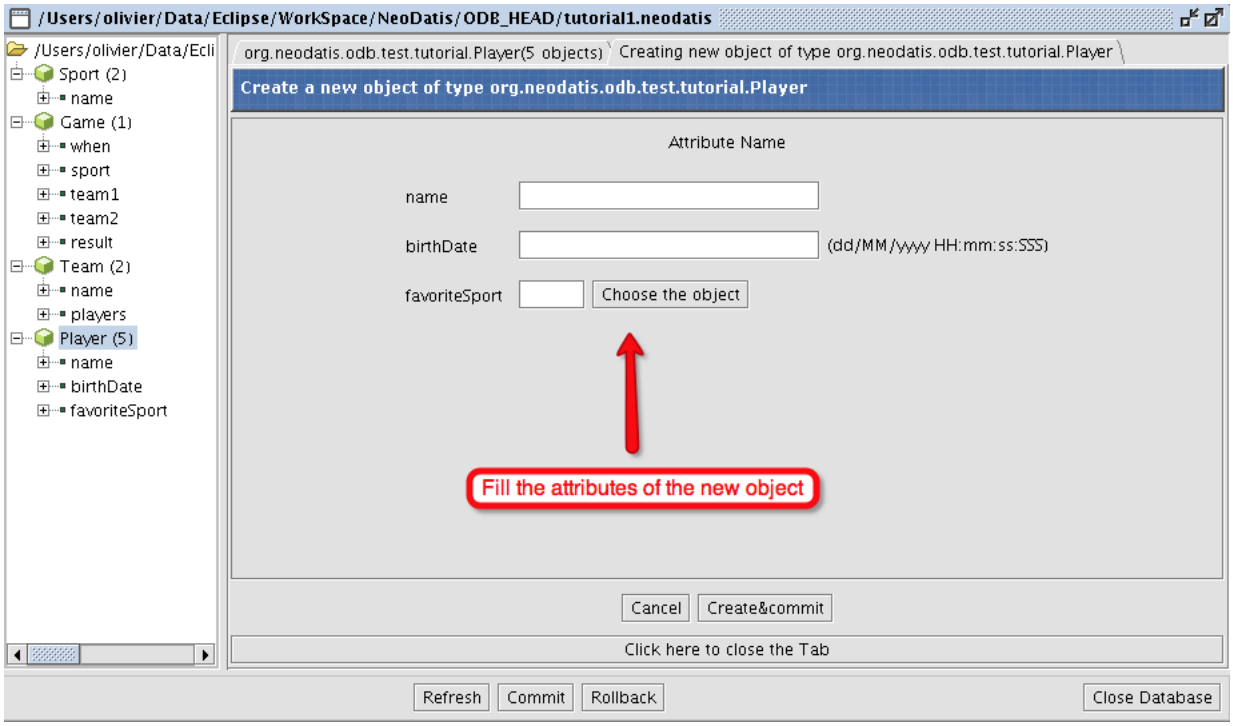

Figure 16.8. Creating a new object

Chapter 17. XML

Using Object Explorer or NeoDatis API, an entire NeoDatis database can be exported to XML and later imported back. Warning:XML import/export currently only works in local mode.

17.1 Using Object Exporer

17.1.1 Exporting

Click on the main menu and choose 'Export to XML'

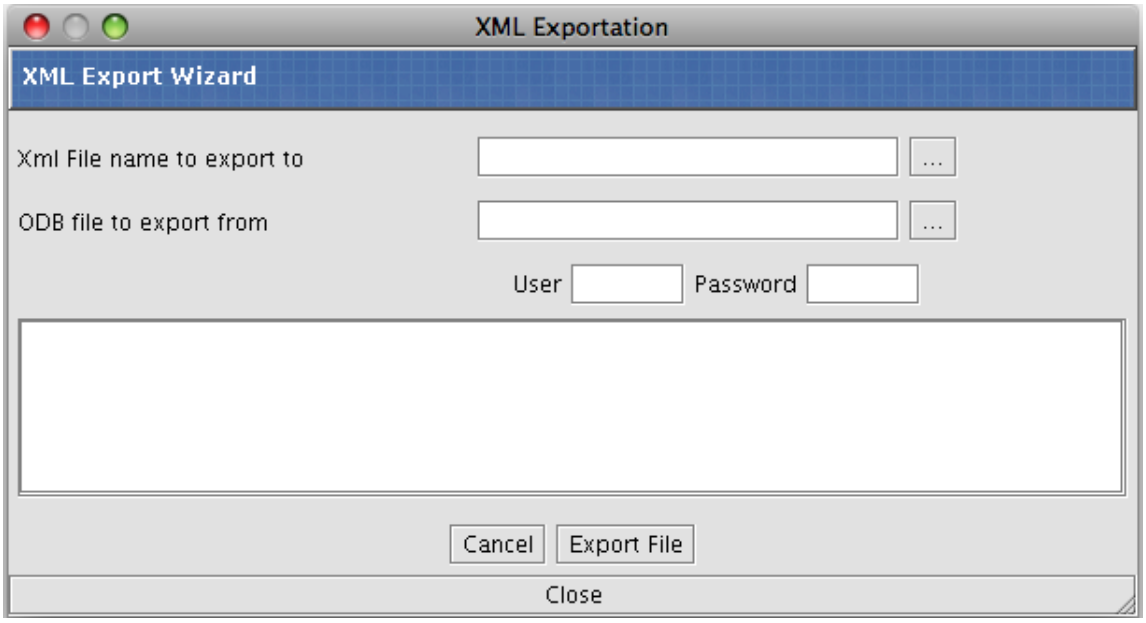

Figure 17.1. Xml Exporter panel

17.1.2 Importing

Click on the main menu and choose 'Import from XML'

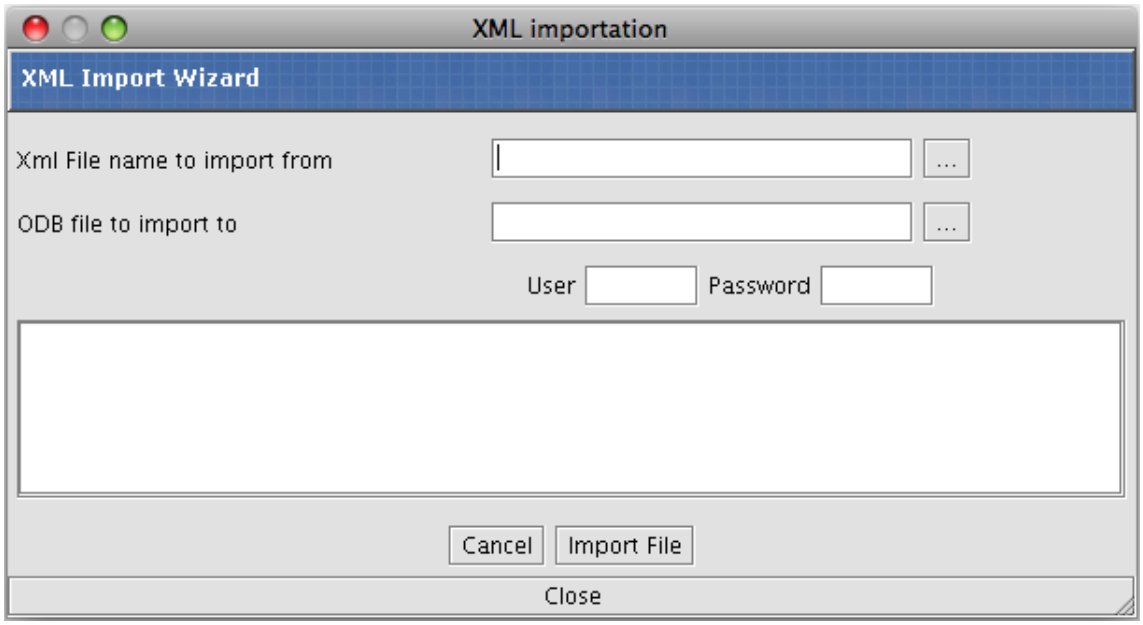

Figure 17.2. Xml Importer panel

17.2 Using NeoDatis API

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

1 Exporting using API

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

 try {
  // Open the database
  odb = NeoDatis.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();
  }
 }
}
```

2 Importing using API

```
public void step16() throws Exception {
 ODB odb = null;

 try {
  // Open a database to receive imported data
  odb = NeoDatis.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 = NeoDatis.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();
  }
 }
}
```

Chapter 18. NeoDatis Extended API

The extended NeoDatis API provides some advanced functions like:

`odb.ext().getLastTransactionId()`	To get the last transaction id of the database
`odb.ext().getObjectVersion(OID oid)`	To get the version of an object with the specific OID
`odb.ext().getObjectCreationDate(OID oid)`	To get the creation date of an object with the specified OID
`odb.ext().convertToExternalOID(oid);`	To convert an internal OID (Object ID) to an external OID : this guarantees uniqueness across databases

Chapter 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 = NeoDatis.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 = NeoDatis.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 = NeoDatis.open(ODB_NAME_2, "user", "password");
  System.out.println("\nStep 17 : user/password : database opened");

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

Chapter 20. Best practices

1 Open/Close Database

When working with NeoDatis 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 = NeoDatis.open(…);

  // work with odb
  // …

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

It is also a good practice to put the NeoDatis.open(ODB_NAME) code in a separated class to isolate the opening of the database.

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 NeoDatis that they do not need to be persisted.

Chapter 21. Using NeoDatis in web applications

To use NeoDatis in WEB applications, you just need to put the NeoDatis jar in the WEB-INF/lib of the war. The default place of the NeoDatis database file (if not specified when opening the NeoDatis file) will be the execution directory of the web container. For example, if you use Tomcat, the NeoDatis 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.

21.1 Web App Example

You can use both local mode or Client server mode for NeoDatis. If you know you will have concurrent connections it is better to the Client / Server one.

As clients will run in the same JVM (your web server), you can opt for the 'SameVm' client server mode (as it is faster).

How and when can I start the NeoDatis server?

You can use a ServletContextListener to start your server:

```
public class NeoDatisServerContextListener implements ServletContextListener {

 private static final int NEODATIS_SERVER_PORT = 10001;
 public static ODBServer server;
 public static boolean isOk;

 public void contextDestroyed(ServletContextEvent event) {
  System.out.println("NeoDatis server context destroyed");

  if (server != null) {
   try {
    server.close();
   } catch (IOException e) {
    e.printStackTrace();
   }
  }
 }

 public void contextInitialized(ServletContextEvent context) {
  try {
   System.out.println("Starting NeoDatis Server");
   // Creates the server
   server = NeoDatis.openServer(NEODATIS_SERVER_PORT);
   // Starts the server to run in an independent thread
   server.startServer(true);
   isOk = true;
  } catch (Exception e) {
   e.printStackTrace();
```

```
    throw new RuntimeException(e);
   } finally {
   }
  }
}
```

When you need to build an url to identify an object, you can use the NeoDatis OID to do that. Example : http://localhost/MyApp/UpdateObject?oid=1000. The OID would be the NeoDatis Object OID as you already have some API to retrieve objects by OID. See the OID chapter for more information [56]

The default location of database creation in the webserver execution directory. For example, if you are using tomcat, if you don't specify absolute path, the NeoDatis file will be created in the bin directory

Use commit instead of Close. If you use the odb.commit() instead of close(), all your objects will be in the NeoDatis cache and you won't need to reload objects when updating

Chapter 22. Advanced features

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

1 Multi-thread

For instance, NeoDatis ODB used in local mode 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 NeoDatis ODB that you are using multi-thread and specify the thread pool size. This can be done using:

```
OdbConfiguration.useMultiThread(true,[thread pool size])
```

Normally, if you try to open a database file that is already open, NeoDatis will throw an exception, when useMutiThread is called, instead of throwing an exception, NeoDatis will wait x milliseconds and retries to open the database. The time NeoDatis will wait and the number of retries depends on the number of threads that have been configured.

2 Defragmentation

to do.

3 Classes without default constructor

NeoDatis 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 NeoDatis with the right data to execute a constructor. The InstanceHelper interface may be used to help NeoDatis create the instance. The “No Calling Constructor” feature is enabled by default. To enable/disable, use

```
    OdbConfiguration.setEnableEmptyConstructorCreation(true/false);
```

4 NeoDatis logging

Default NeoDatis log behavior is logging to console(with System.out.println). But if you need a specific log you can then create your logger implementing org.neodatis.tool.ILogger to do what you need, logging to log4j, for example. Here is an example

```
public class MyLogger
     implements ILogger {

     public void debug(Object object) {
     ...
     }

     public void error(Object object) {
     ...
     }

     public void error(Object object, Throwable t) {
     ...
     }

     public void info(Object object) {
     }
   }
```

Then you must call org.neodatis.tool.DLogger.register(new MyLogger());

Then all log call will be forwarded to your logger too.

There is currently no way to disable log to console.

5 OIDs

In some cases, you may need the OID to retrieve an object. This can be the case in web application when you want to update an object for example. You can use the oid of the object as being the key of the object in the url: http://www.mydomain.com/update?oid=[NeoDatis OID] You can use the

```
String soid = oid.oidToString();
```

to retrive a string representation of the OID. Then you can use

```
OID oid = OIDFactory.oidFromString(soid);
```

to build a real OID from the string representation. Once you have the OID, you can retrieve the object using

```
odb.getObjectFromId(oid)
```

Chapter 23. NeoDatis Context

Default behavior of NeoDatis does not impose any restriction on the objects being stored. But some features (like object reconnecting and lazy loading) require that objects implement or extend some NeoDatis classes/interfaces.

There are 3 ways to do that:

- Manually implement NeoDatisObject interface
- Manually extend NeoDatisObjectAdapter class
- Use NeoDatis Ant task to byte code instrument your objects

Doing this, your objects will receive an attribute of type NeoDatisContext that will contain some NeoDatis information like the object OID (that can be seen as an auto generated primary key). This information will allow NeoDatis to know if an object has been loaded in a previous session and keep information about lazy loading of attributes.

Let’s start doing it using the manual method. We will create a class Computer that inherits from the org.neodatis.odb.core.context.NeoDatisObjectAdapter class.

```
public class Computer
    extends NeoDatisObjectAdapter {

    protected String name;
    protected int type;

    protected List <Monitor> monitors;
    protected List <Component> components;


    public String getName() {
     return name;
    }

    public void setName(String name) {
     neoDatisContext.markAsChanged();
     this.name = name;
    }

    public int getType() {
     return type;
    }

    public void setType(int type) {
     neoDatisContext.markAsChanged();
     this.type = type;
    }

    public Computer(String name, int type) {
     super();
```

```
     this.name = name;
     this.type = type;
     components = new ArrayList <Component>();
     monitors = new ArrayList <Monitor>();
    }

    public List <Monitor> getMonitors() {
     return monitors;
    }

    public void setMonitors(List <Monitor> monitors) {
     this.monitors = monitors;
    }

    public List <Component> getComponents() {
     return components;
    }

    public void setComponents(List <Component> components) {
     this.components = components;
    }

    public void addComponent(Component c){
     components.add(c);
    }

    public void addMonitor(Monitor m){
     monitors.add(m);
    }
   }
```

Here are the definitions of the 2 other classes

Class Monitor

```
public class Monitor extends NeoDatisObjectAdapter {

    protected String resolution;
    protected String model;

    public Monitor(String resolution, String model) {
     super();
     this.resolution = resolution;
     this.model = model;
    }

    public String getResolution() {
     return resolution;
    }

    public void setResolution(String resolution) {
     this.resolution = resolution;
    }

    public String getModel() {
     return model;
    }

    public void setModel(String model) {
     this.model = model;
    }
```

```
   }
```

Class Compomnent

```
public class Component extends NeoDatisObjectAdapter {
     protected String name;
     protected String description;

    public Component(String name, String description) {
     super();
     this.name = name;
     this.description = description;
    }

    public String getName() {
     return name;
    }

    public void setName(String name) {
     this.name = name;
    }

    public String getDescription() {
     return description;
    }

    public void setDescription(String description) {
     this.description = description;
    }
   }
```

Now, our classes are prepared to use advanced features like automatic object reconnection. Here is an example:

```
   String baseName = "reconnect.neodatis";

   ODB odb = NeoDatis.open(baseName);

   // Create a computer with some components
   Computer cpu1 = new Computer("MacBookPro", 1);
   cpu1.addComponent(new Component("HD1", "Harddrive 500GB 7200rpm"));
   cpu1.addComponent(new Component("HD2", "Harddrive 2TB 10000rpm"));
   cpu1.addComponent(new Component("GC1", "GraphicCard with internal Processor"));
   cpu1.addMonitor(new Monitor("1920*1200", "Samsung"));

   // Store the computer into NeoDatis
   odb.store(cpu1);
   odb.close();

   // re open database and check if NeoDatis will understand that the
   // object is the same
   odb = NeoDatis.open(baseName);

   // Use the computer without reloading it
   cpu1.setName("Mac_Book_Pro");
   odb.store(cpu1);
   odb.close();

   // Then load computers to check if the computer has been updated
```

```
   odb = NeoDatis.open(baseName);

   Objects <Computer> computers = odb.query(Computer.class).objects();
   odb.close();

   System.out.println("Number of computers : "+ computers.size());
   System.out.println(computers.getFirst().getName());
```

As you can see, the update (cpu1.setName("Mac_Book_Pro")) is done on an object that has been loaded/inserted in a previous session but, as the object holds some context information, NeoDatis detects the situation and updates correctly the object.

Once your object holds the NeoDatis context, you can retrieve the OID by calling :

```
object.getNeoDatisContext().getOid();
```

This will return the OID of the object without any cost as it is stored in the object. This OID can be used later to retrieve the object using:

```
odb.getObjectFromId(oid);
```

Chapter 24. Storage Engine Plugins

1 Architecture

NeoDatis V2 has a pluggable storage engine architecture. This means that it is possible to choose which storage engine best suits your needs while keeping the simplicity of NeoDatis.

The choice can bedone using the API or using configuration file and does not require any change to the NeoDatis API.

2 How to use a plugin

You just need to specify the storage engine class. You can do that in 3 ways:

- Creating a NeoDatisConfig instance :

```
NeoDatisConfig config =
        NeoDatis.getConfig().setStorageEngineClass(NeoDatisBerkeleyDbPlugin.class)
```

  and use the instance to open the database.

```
ODB odb =
        NeoDatis.open("base-name", config);
```

- Setting storage engine statically using

```
NeoDatisGlobalConfig.get().setStorageEngineClass(NeoDatisBerkeleyDbPlugin.class);
```

- Using a config file and setting the property storage.engine.class to the plugin class

Just remember to include in your classpath the plugin jar and the storage engine jars to execute your application.

3 How to build a plugin

The plugin architecture is simple and easy to extend : to create a new storage engine, it is necessary to extend the org.neodatis.odb.core.layers.layer4.StorageEngineAdapter abstract class. This class has 8 abstract methods that must be implemented:

`void open(`
`String baseName,`
`NeoDatisConfig config)`
	Opens the storage engine for database 'baseName' and the configuration. In Local mode (no network IO)

`void open( String host, int port, String baseName, NeoDatisConfig config)`	Opens the storage engine for database 'baseName' and the configuration. In Client Server mode
`OidAndBytes read( OID oid, boolean useCache)`	Reads data of the object with the given oid
`void write(OidAndBytes oidAndBytes)`	Writes data of an object
`void close()`	Closes the storage engine
`void commit()`	Commits changes to the storage engine
`void rollback()`	Rollbacks changes
`void deleteObjectWithOid(OID oid)`	Delete data of the object with the given oid
`boolean existOid(OID oid)`	Check if an object with the given oid exists in the base
`String getEngineDirectoryForBaseName (String theBaseName)`	Get the directory where the storage keeps its data for the given base name
`String getStorageEngineName()`	Get the name of the storage engine (just for information)

To simplify the explanation, we are going to show an example implementation of a In Memory plugin that uses an hashMap as a storage backend:

```
package org.neodatis.odb.core.layers.layer4.memory;

    import java.util.HashMap;
    import java.util.Map;

    import org.neodatis.odb.NeoDatisConfig;
    import org.neodatis.odb.OID;
    import org.neodatis.odb.core.layers.layer3.OidAndBytes;
    import org.neodatis.odb.core.layers.layer4.StorageEngineAdapter;
    import org.neodatis.tool.DLogger;

    /** A NeoDatis storage engine backed by a
    memory HashMap
    * @author olivier
    *
    */
    public class InMemoryStorageEngine extends StorageEngineAdapter {
```

```
    /** A boolean value to indicate whether to log debug message
    *
    */
    protected boolean debug = false;

    /** A hashmap to store the storage engine data
    *
    */
    protected Map<String,OidAndBytes> store;


    public InMemoryStorageEngine() {
     super(true);
    }

    public OidAndBytes read(OID oid, boolean useCache) {
     if (debug) {
      DLogger.info("Reading OID " + oid.oidToString());
     }
     return store.get(oid.oidToString());
    }

    public void write(OidAndBytes oidAndBytes) {
     if (debug) {
      DLogger.info("Writing OID " + oidAndBytes.oid.oidToString() +
       " | bytes = " + oidAndBytes.bytes);
     }
     store.put(oidAndBytes.oid.oidToString(), oidAndBytes);
    }

    public void close() {
     //nothing to do
    }

    public void commit() {
     // nothing to do
    }

    public void open(String baseName, NeoDatisConfig config) {
     this.store = new HashMap<String, OidAndBytes>();
    }

    public void open(String host, int port, String baseName, NeoDatisConfig config) {
     throw new RuntimeException("Client server mode is not supported by InMemory Mode");
    }

    public void rollback() {
     throw new RuntimeException("rollback not supported by InMemory Mode");
    }

    public void deleteObjectWithOid(OID oid) {
     store.remove(oid.oidToString());
    }

    public boolean existOid(OID oid) {
     return store.containsKey(oid.oidToString());
    }

    public String getEngineDirectoryForBaseName(String theBaseName) {
     return null;
    }
```

```
    public String getStorageEngineName() {
     return "in memory";
    }
   }
```

The code is the actual implementation of the InMemoryStorage engine of NeoDatis.

Chapter 25. NeoDatis on Google Android

As Android supports java 1.5, to run NeoDatis ODB on Android, just use the Java version and use it normally. Check http://www.neodatis.org/android

Chapter 26. NeoDatis and Groovy

Thanks to Guilherme Gomes, NeoDatis Odb can be used to persist Groovy objects

```
// Gets the Groovy ClassLoader
GroovyClassLoader gcl = new GroovyClassLoader();
// Gets a Groovy Engine
GroovyScriptEngine gse = new GroovyScriptEngine("", gcl);

// Sets the Groovy ClassLoader as the default NeoDatis Odb ClassLoader
OdbConfiguration.setClassLoader(gcl);

// Loads Groovy classes in the Groovy ClassLoader
gcl.parseClass(new File("scripts\\Costumer.groovy"));
```

Check http://www.neodatis.org/neodatis-and-groovy

Chapter 27. Annexes

1 Xml Exported file of the tutorial NeoDatis base

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<odb name="tutorial1.neodatis" export-date-time="1241786956672" max-oid="31" file-format-version="9">
  <meta-model >
    <class id="1" name="org.neodatis.odb.test.tutorial.Sport">
      <attribute id="1" name="name" type="java.lang.String"/>
    </class>
    <class id="3" name="org.neodatis.odb.test.tutorial.Game">
      <attribute id="1" name="when" type="java.util.Date"/>
      <attribute id="2" name="sport" type="org.neodatis.odb.test.tutorial.Sport"/>
      <attribute id="3" name="team1" type="org.neodatis.odb.test.tutorial.Team"/>
      <attribute id="4" name="team2" type="org.neodatis.odb.test.tutorial.Team"/>
      <attribute id="5" name="result" type="java.lang.String"/>
    </class>
    <class id="4" name="org.neodatis.odb.test.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="org.neodatis.odb.test.tutorial.Player">
      <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.test.tutorial.Sport"/>
    </class>
  </meta-model>
  <objects >
    <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="6" class-id="3">
      <attribute id="1" name="when" value="1241786956537"/>
      <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="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="1241786956537"/>
      <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="1241786956537"/>
      <attribute id="3" name="favoriteSport" ref-oid="7"/>
    </object>
    <object oid="12" class-id="5">
      <attribute id="1" name="name" value="elohim"/>
      <attribute id="2" name="birthDate" value="1241786956537"/>
      <attribute id="3" name="favoriteSport" ref-oid="7"/>
    </object>
    <object oid="13" class-id="5">
      <attribute id="1" name="name" value="minh"/>
      <attribute id="2" name="birthDate" value="1241786956537"/>
      <attribute id="3" name="favoriteSport" ref-oid="7"/>
    </object>
    <object oid="30" class-id="5">
      <attribute id="1" name="name" value="Andr%8E+Agassi"/>
      <attribute id="2" name="birthDate" value="1241786956648"/>
      <attribute id="3" name="favoriteSport" ref-oid="15"/>
    </object>
  </objects>
</odb>
```