**UNIVERSITÉ LIBRE DE BRUXELLES**
**Faculté des Sciences**
**Département d'Informatique**

# INFO-H-415 - Advanced Databases
# Project – Java Data Objects

Benacha Ismael - Lakbeich Yassine

**Academic year 2015 - 2016**

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 What is object persistence ?

Initially we will start by explaining briefly what is an object and what is an object oriented database.

### 1.1.1 Object

In all object oriented programming languages, an object is an instance of a class. An object has a state (the attribute values) and a behaviour (the methods).

### 1.1.2 Object Oriented Database

An object oriented database, or Object Oriented Database Management System (OODBMS), is a database that can store objects.

To illustrate the difference between an Object Oriented Database and a relational database, let's say that we have an object of type Cat that we want to store in a database. In an object oriented database we simply store the Cat object as is, and the data inside Cat would be kept in its original form.

Figure 1.1: Object Cat in an object oriented database

In a relational database, we have some more work to do. Indeed, The object Cat is not a primitive data type , so we have to break it down into primitive data type and store it fragment by fragment

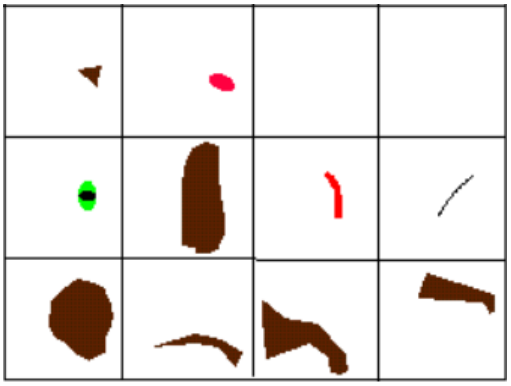

Figure 1.2: Object Cat in a relational database

We can also Serialize the entire object and save it to a text field ,but it is not practical. Indeed it would make it impossible to query the database. Therefore , the best method is to divide the object into primitives properties. In Figure 1.2 it can be seen that the cat object was divided into fragment representing cat properties. The developer using this database had to take extra time to find each attribute , determine the primitive data type, and store them in the database. This process is much more painful, and shows when object oriented database become necessary.

### 1.1.3 Object persistence

At the end of a session of a object-oriented application all data of existing objects in RAM is lost. Persist an object is saved its data to a non volatile media. so that an object, identical to this object, can be recreated in a session later.

## 1.2 What is Java Data Objects (JDO)?

Now that we have a good understanding of object persistence, we can explain what is JDO !

### 1.2.1 JDO

Java Data Objects is a standard for Java language. JDO allows storage , querying and retrieving objects in a database. JDO Is Extremely convincing due to the concept of transparent persistence ( Transparent persistence is the storage and retrieval of persistent data with little or no work from the developer).

## 1.3 What's next ?

in the following, we will talk about the state of the art. In this section we traced the state of knowledge in this field. We'll talk about the architecture, the operation and the specificity of JDO.We'll also talk about the benefit and inconvenient of this technology. Then we conclude with a description of our project.

# Chapter 2

# State of the art

## 2.1 Architecture

### 2.1.1 JDO implementations and vendors

The JDO package javax.jdo, provide the main interface definitions. It also contains a few concrete classes, notably JDOHelper and the JDO exception classes. It is through these interfaces that applications have access to the functionality of object persistence.
The most important one is PersistenceManager, through which transient instances can be made persistent, persistent instances deleted, and so on.

These standard interfaces are not in themselves sufficient to actually implement persistence. What is needed is a set of concrete classes implementing the respective interface definitions, which will perform persistence operations when invoked to do so. A set of such classes is known as a JDO implementation.

JDO implementations are data store-specific. Some work against any JDBC compliant database. Others may work with only a specific relational database in order to exploit potential optimizations. Still others work with certain object databases, file system formats, or provide integration to specific enterprise applications. A company that markets a JDO implementation is known as a JDO vendor. Most JDO implementations are shipped with an enhancement tool.

### 2.1.2 JDO instances

The term JDO instance is used to describe any instance of a Java language class which implements the PersistenceCapable interface which the implementation is capable of managing.

Some implementations, largely dictated by the underlying data store, require storage areas to be explicitly defined for each class before that class can be managed. This is more typical of object-relational mapping implementations (with an underlying relational database) than of object databases.

### 2.1.3 JDO Enhancer

- Enhancer makes transient class PersistenceCapable.
- Enhancer can be also part of compiler or loader.
- Enhancing can be modified via Persistence Descriptor.
- All Enhancers are compatible.

We can see that the enhancer is called after compiling our source code. It is used to make the persistent class. The enhancer uses a written description file, or generated with the implementation being used. This file will allow the enhancer to add to the compiled file the code necessary to persistence. It also ensures that the mapping is done. Moreover, it can generate a schema creation script of the database. The Enhancer is an amplifier that will allow to transform either annotations or XML file in code to be used by the database and the JVM.

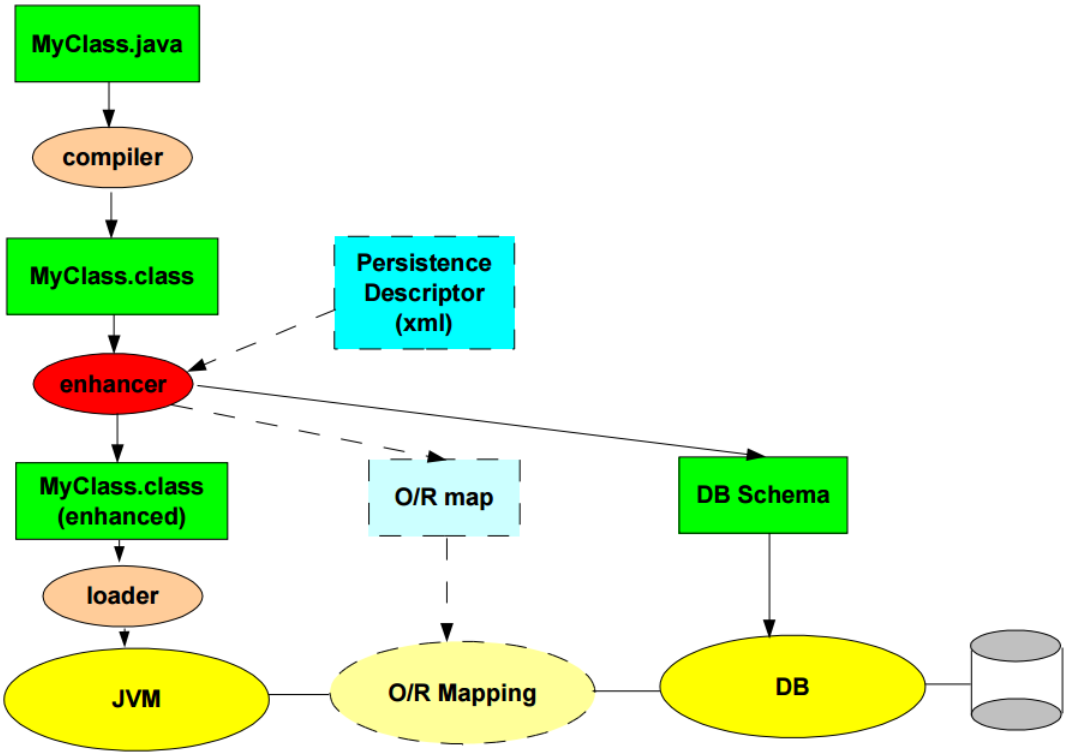


Figure 2.1: diagram of Enhancer impact

### 2.1.4 JDO environments

JDO is intended for use in two specific architectural spaces. The most simple environment is one in which an application directly invokes the services of an implementation.
This is the so-called non-managed environment.
The second environment is that in which the persistence functions of a persistence manager are invoked by application components running within a J2EE application server. This more complicated environment requires that vendors integrate their JDO implementations with the J2EE transaction and connector architectures. This is the so-called managed environment.

#### 2.1.4.1 Non-managed environment

In the non-managed environment, it is the application itself that handles all interactions with the implementation.

This includes the configuration of the PersistenceManagerFactory, obtaining the PersistenceManager, defining transactions (with appropriate begin(), commit() and rollback() invocations), and all persistence operations on instances.

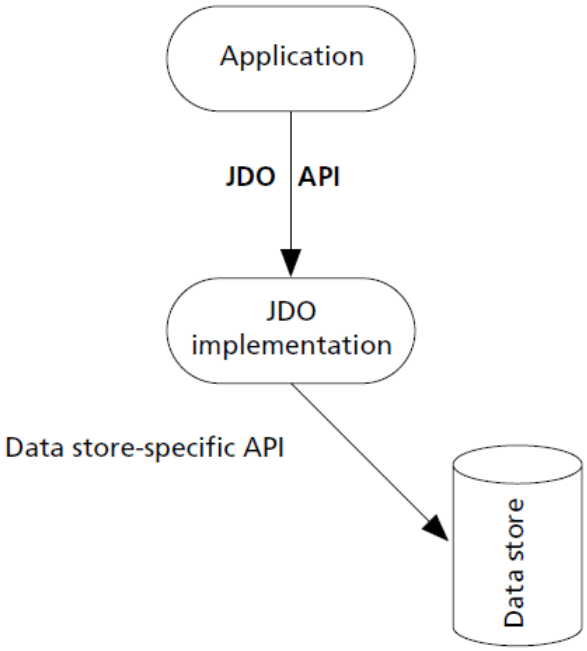


Figure 2.2: The non-managed environment

Applications using JDO in this way, generally configure a factory to obtain a reference PersistenceManager at startup and save this reference until the application is closed.
Heavily multithreaded applications, however, may rely on the pooling characteristics of the persistence manager factory, which maintains a pool of persistence managers.
In such cases the application will get a persistence manager from the factory, use it, and then immediately return it to the pool by invoking its close() method.

#### 2.1.4.2 Managed environment

In the managed environment JDO is integrated within a J2EE application server. Application components executing within the application server still invoke the PersistenceManager in the usual way.
However, they would expect to obtain the PersistenceManagerFactory reference from the Java Naming and Directory Interface (JNDI) context of the application server.
All transaction management is coordinated between JDO and the application server.

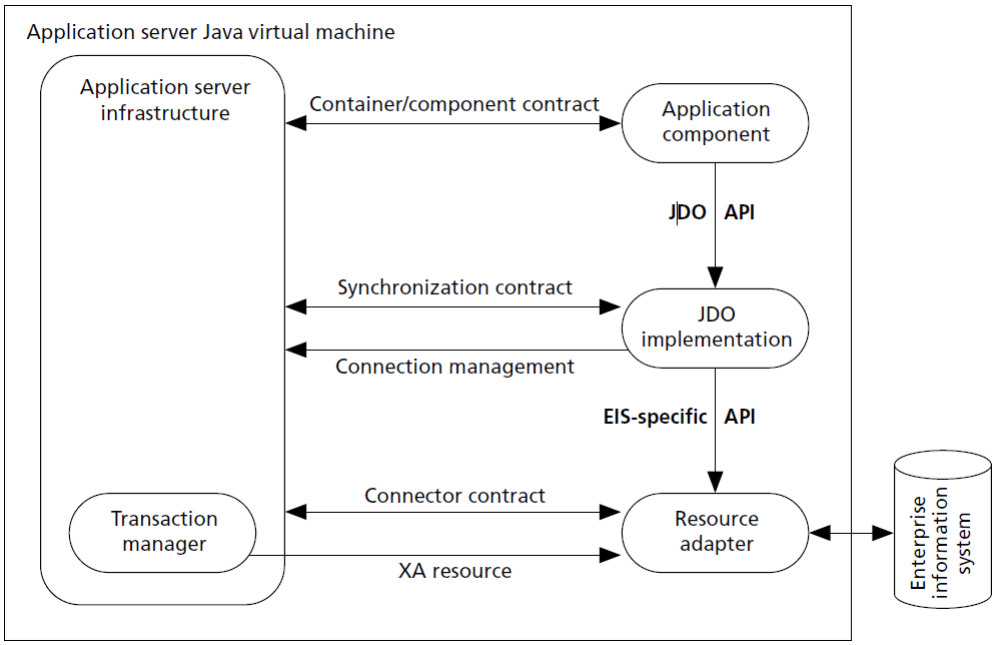


Figure 2.3: Managed environment

In the managed environment, applications tend to access data stores through JCA, providing maximum flexibility (see figure 2.3).
Finally PersistenceManagers are pooled to reduce resource usage and facilitate scalability. Thus an application component using the services of a PersistenceManager should obtain it from the factory, use it, and close it immediately

## 2.2 Working

### 2.2.1 Obtaining the JDO PersistenceManager Object

In order to access the functionalities of the JDO API you have to deal with a special facade object that serves as the main entry point to all JDO operations. This facade is specified by the Interface javax.jdo.PersistenceManager. A Vendor of a JDO compliant product must provide a specific implementation of the javax.jdo.PersistenceManager interface. JDO also specifies that a JDO implementation must provide a javax.jdo.PersistenceManagerFactory implementation that is responsible for generating javax.jdo.PersistenceManager instances

For our project we use the datanucleus implementations.

### 2.2.2 Detachable and CRUD

#### 2.2.2.1 Detachable object

The objects are called "attached" because they are always known to the persistantManager until the transaction is opened. Once closed, objects are detached. And we can not use it directly CRUD. A detachable object is a usable object even if the transaction is closed. The persistanceManager always recognizes and allows us to make changes to the object. Therefore the persistanceManager modify the database also.

#### 2.2.2.2 Update example

One way to update an object with JDO is to fetch the object, then modify it while the PersistenceManager that returned the object is still open. Changes are persisted when the PersistenceManager is closed. For example:

```
public void updateProductTitle(User user, String newTitle) {
    PersistenceManager pm = PMF.get().getPersistenceManager();
    try {
        Product p = pm.getObjectById(Product.class, user.getEmail());
        if (titleChangeIsAuthorized(p, newTitle) {
            p.setTitle(newTitle);
        } else {
            throw new UnauthorizedTitleChangeException(p, newTitle);
        }
    } finally {
        pm.close();
    }
}
```

Since the Product instance was returned by the PersistenceManager, the PersistenceManager knows about any modifications that are made to Persistent fields on the Product and automatically updates the datastore with these modifications when the PersistenceManager is closed. It knows this because the Product instance is "attached" to the PersistenceManager.

You can modify an object after the PersistenceManager has been closed by declaring the class as "detachable." To do this, add the detachable attribute to the @PersistenceCapable annotation:

```
import javax.jdo.annotations.PersistenceCapable;

@PersistenceCapable(detachable="true")
public class Product {
    // ...
}
```

Now you can read and write the fields of an Product object after the PersistenceManager that loaded it has been closed.

#### 2.2.2.3 Delete

To delete a tuple, given its id to retrieve it via getObjectById, and then gives it to persistentManager, which will remove it. This is possible because the object is still attached.

```
Product product = pm.getObjectById(Product.class, tupleId);
pm.makePersistent(product);
```

### 2.2.3 JDO constraints

JDO also implements the primary key constraints, unique, foreign key and others. There are 2 ways to do. Either one uses an XML file to describe each class and each attribute with its constraints and options, or with the annotations of direct use in the class concerned. A specific locations for each type of constraints.

#### 2.2.3.1 Primary Key

In RDBMS datastores, it is accepted as good practice to have a primary key on all tables. you define the name of the column DataNucleus should use for the primary key.

```
<class name="Booking">
    <primary-key name="BOOKING_PK"/>
    ...
</class>
```

We can use also annotations. This is a shortcut for **@Persistent(primaryKey="true")** meaning that the field/property is part of the primary key for the class. No attributes are needed when specified like this. Specified on the field/method.

```
@PersistenceCapable
public class MyClass
{
    @PrimaryKey
    String myOtherField;
    ...
}
```

JDO offers several strategies for the management of the primary key: Here are a few :

- Native - this is the default and allows DataNucleus to choose the most suitable for the datastore.
- Sequence - this uses a datastore sequence (if supported by the datastore)
- Identity - these use autoincrement/identity/serial features in the datastore (if supported by the datastore)
- Increment - this is datastore neutral and increments a sequence value using a table.
- ...

#### 2.2.3.2 Unique constraints

JDO provides a mechanism for defining such unique constraints.
How to specify this ? Like the previous constraints, we can use Metadata XML or annotations.
For Metadata XML file, we specify this like that :

```
<class name="Booking">
    <field name="bookingType">
        <unique name="BOOKING_TYPE_CONSTRAINT"/>
    </field>
</class>
```

The balise unique name=... allows to indicate the name of the constraints.

And annotations :

```
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Unique(name="MYFIELD1_IDX")
    String field1;
    ...
}
```

#### 2.2.3.3 Foreign Key

These are used to interrelate objects, and allow the datastore to keep the integrity of the data in the datastore.

When objects have relationships with one object containing, for example, a Collection of another object, it is common to store a foreign key in the datastore representation to link the two associated tables. Moreover, it is common to define behaviour about what happens to the dependent object when the owning object is deleted. Should the deletion of the owner cause the deletion of the dependent object maybe ? Lets take an example

```
public class Hotel
{
    private Set rooms;
    ...
}

public class Room
{
    private int numberOfBeds;
    ...
}
```

We now want to control the relationship so that it is linked by a named foreign key, and that we cascade delete the Room object when we delete the Hotel. We define the Meta-Data like this

```
<class name="Hotel">
    <field name="rooms">
        <collection element-type="com.mydomain.samples.hotel.Room"/>
        <foreign-key name="HOTEL_ROOMS_FK" delete-action="cascade"/>
    </field>
</class>
```

And with annotations

```
@PersistenceCapable
public class Hotel
{
    @Persistent
    @ForeignKey(name="HOTEL_ROOMS_FK", deleteAction=ForeignKeyAction.CASCADE)
    Set rooms;
    ...
}
```

#### 2.2.3.4 Inheritance

In Java it is a normal situation to have inheritance between classes. With JDO you have choices to make as to how you want to persist your classes for the inheritance tree. For each class you select how you want to persist that classes information. You have the following choices.

The first and simplest to understand option is where each class has its own table in the datastore. In JDO this is referred to as new-table. The second way is to select a class to have its fields persisted in the table of its subclass. In JDO this is referred to as subclass-table The third way is to select a class to have its fields persisted in the table of its superclass. In JDO this is known as superclass-table. JDO3.1 introduces support for having all classes in an inheritance tree with their own table containing all fields. This is known as complete-table and is enabled by setting the inheritance strategy of the root class to use this.

JDO imposes a "default" inheritance strategy if none is specified for a class. If the class is a base class and no inheritance strategy is specified then it will be set to new-table for that class. If the class has a superclass and no inheritance strategy is specified then it will be set to superclass-table. This means that, when no strategy is set for the classes in an inheritance tree, they will default to using a single table managed by the base class.

There are several ways to implement inheritance, below we show only the strategy used in the project.

Here we want to have a separate table for each class. This has the advantage of being the most normalised data definition. It also has the disadvantage of being slower in performance since multiple tables will need to be accessed to retrieve an object of a sub type. With metadata :

```
<class name="Product">
    <inheritance strategy="new-table"/>
    <field name="price">
        <column name="PRICE"/>
    </field>
</class>
<class name="Book">
    <inheritance strategy="new-table"/>
    <field name="isbn">
        <column name="ISBN"/>
    </field>
    <field name="author">
        <column name="AUTHOR"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
```

Or with annotations :

```
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.NEW_TABLE)
public class Product {...}

@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.NEW_TABLE)
public class Book {...}
```

## 2.3 JDO specificity

### 2.3.1 Transparent persistence

As mentioned in the introduction, JDO Is Extremely convincing due to the concept of transparent persistence ( Transparent persistence is the storage and retrieval of persistent data with little or no work from the developer).

This can be summarized in three points

- **The object-relational impedance mismatch**

  The object-relational impedance mismatch is a set of difficulties that are encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language

- **JDO is transparent to the Java objects being persisted**

  No specific methods or attributes must be added to the Java classes. Moreover, JDO does not encounter problems with attribute with private visibility, and without get and set methods. In summary, JDO persist an object without source code modification.

- **Different data storage paradigms**

  JDO can be used against a number of different data storage paradigms, like relational databases, object databases, file systems, XML documents and many more ...

### 2.3.2 JDOQL

**Java Data Objects Query Language (JDOQL) is the query language specified by the standard JDO. The language uses the Java syntax. The JDO Query Language allows users to search for persistent objects corresponding to specific criteria. The result of a JDO query execution is an Collection, comprised of objects for which the filter criteria evaluated to true.**

**Such a request can be implemented in two ways .**

- Declarative JDOQL :

```
Query q = pm.newQuery(mydomain.Person.class, "lastName ==
                    "Jones" && age < age_limit");
q.declareParameters("double age_limit");
List results = (List)q.execute(20.0);
```

- Single-String JDOQL :

```
Query q = pm.newQuery("SELECT FROM mydomain.Person WHERE lastName ==
          "Jones" +  " && age < :age_limit PARAMETERS double age_limit");
List results = (List)q.execute(20.0);
```

**In the example above, we select all objects " person " with the name " Jones" and where the age of the person is less than 20.**

#### 2.3.2.1 Declarative JDOSQL

It can be seen that the search request (in declarative JDOQL) is composed of distinct part

- the class being selected (the SELECT clause in SQL)
- the filter (the WHERE clause in SQL)
- the sorting (the ORDER BY clause in SQL)
- ...

#### 2.3.2.2 Single-String JDOSQL

the search request can also be specified with a single string. The distinct part is the follows

```
SELECT [UNIQUE] [<result>] [INTO <result-class>]
       [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
       [WHERE <filter>]
       [VARIABLES <variable declarations>]
       [PARAMETERS <parameter declarations>]
       [<import declarations>]
       [GROUP BY <grouping>]
       [ORDER BY <ordering>]
       [RANGE <start>, <end>]
```

## 2.4 Comparison

### 2.4.1 persistence technologies

Persistence requires the storage of object state for future retrieval. various technologies are in use. Here is a brief description of the different persistence technologies.

- **Serialization**
  Serialize an object is to convert this object to a byte array , which can then be written to a file. for safety reasons , all objects are not serializable. To be serializable , object just needs "to say it" ( implements Serializable )
- **JDBC**
  JDBC (Java DataBase Connectivity) allows Java applications to access a relational database
- **ODBMS**
  ODBMS (object-oriented Database Management System) is a database management system in which information is represented as used in object-oriented programming, i.e. as an object.
- **EJB**
  EJB (Enterprise Java Beans) is a standard for server-based Java component applications. EJB performs many of the tasks of applications, data life cycle management and transaction processing, like data persistence.
- **JDO**
  Java Data Objects is a standard for Java language. JDO allows storage , querying and retrieving objects in a database.
- ...

Of course, there are several other ways to persist an object.

### 2.4.2 comparison table

Below you will find a comparison table of the different persistence technology presented above

| | Serialization | JDBC | ODBMS | EJB | JDO |
|---|---|---|---|---|---|
| Transactional | X | ✓ | ✓ | ✓ | ✓ |
| Query facility | ✓ | ✓ | ✓ | ✓ | ✓ |
| Standard API | ✓ | ✓ | X | ✓ | ✓ |
| Standard query language | X | X | X | ✓ | ✓ |
| Supported data store paradigm | File-system | RDBMS | ODBMS | RDBMS, EAI | RDBMS, ODBMS, EAI, File-System, others |
| Transparent to closure of persistent instances | X | X | ✓ | X | ✓ |
| Transparent to domain model | X | X | ✓ | X | ✓ |
| True object database | X | X | ✓ | X | X |
| Supports existing table structure | X | ✓ | X | X | ✓ |

looking at the comparison table we can see the power of JDO. We can notice that JDO is the most comprehensive technology among those that have been presented. Developers can persist data

- without source code modification
- without any knowledge of SQL
- without being restricted by a single type of data store
- ...

# Chapter 3

# Implementation

Our project focuses on the use of the Java Data Objects standard for object persistence in a Java application. JDO implementation that we use is DataNucleus. Java Data Object intervenes to store objects in a Relational database (MySQL)

## 3.1 How to use DataNucleus

Our project was developed using the Eclipse IDE. DataNucleus provides its own plugin for use within Eclipse, giving access to many features of DataNucleus. Here are the steps to follow to use DataNucleus with Eclipse.

- **Installation**

  To obtain and install the DataNucleus Eclipse plugin :

  1. select Help → Software Updates
  2. On the panel that pops up set
     the field "work with" with this URL : http://www.datanucleus.org/downloads/eclipse-update/
     and the field "Name" with the following name : DataNucleus.

  To finish select the site it has added "DataNucleus", and click "Finish"

- **General Preferences**

  Go to Window → Preferences → DataNucleus Eclipse Plugin and configure the libraries needed by DataNucleus :

  1. jdo-api.jar
  2. asm.jar
  3. datanucleus-core
  4. datanucleus-enhancer
  5. datanucleus-api-jdo
  6. datanucleus-rdbms : for running SchemaTool
  7. Datastore driver jar (e.g JDBC) : for running SchemaTool

  These libraries must also be added to the project

- **Preferences : Enhancer**

  in this section, leave the default value

- **Preferences : SchemaTool**

  in this section you must enter the information details about the datastore

  1. Enter the driver path (path to the jar file)
  2. Enter the driver name (example : "com.mysql.jdbc.Driver" for MySql)
  3. Enter the connection URL (example : "jdbc:mysql://localhost:3306/projetadvanceddb" )
  4. Enter the user name of the database
  5. Enter the password about the database

- **Enable DataNucleus Support**

  After having configured the plugin you can now add DataNucleus support on your projects. Simply right-click on your project in Package Explorer and select DataNucleus → "Add DataNucleus Support" from the context menu.

- **Generate JDO MetaData**

  JDO uses XML files to map the attributes of a Java class to the corresponding fields in a database. These XML files are called metadata file, and takes as extension ".jdo"

  The xml files is placed at the project root or the root of a package , then it contains all the information on the classes of the project or package. It can also take the name of a class (for example Book.jdo) . In this case, it only informs characteristic of this class and must be in the same location as the corresponding class.

  To create a metadata file, you must do a right-click on a package in your project and select "Create JDO Metadata File" from DataNucleus context menu. The dialog prompts for the file name to be used and creates a basic Metadata file for all classes in the metadata file package

- **Generate persistence.xml**

  You can also use the DataNucleus plugin to generate a "persistence.xml" by following the same steps as previously

- **Run the Enhancer**

  The DataNucleus Eclipse plugin allows you to easily byte-code enhance your classes using the DataNucleus enhancer. Right-click on your project and select "Enable Auto-Enhancement" from the DataNucleus context menu

- **Run SchemaTool**

  Once your classes have been enhanced you are in a position to create the database schema Click on the project under "Package Explorer" and under "DataNucleus" there is an option "Run SchemaTool". This brings up a panel to define your database location (Driver path, URL, user name and password). You enter these details and the schema will be generated.
  note that you can omit this step if you already have your schema

- **Run application**

  After that it is simply a question of starting your application and all should be taken care of. then it is very easy to persist , delete, modify or retrieve object in the database. You just need to add some code in the Main method

## 3.2 Our project

The subject on which we built our application is the management of books in a bookstore. This application is very useful when the bookstore decides to make an inventory of his books. The database will contain all the books available and let you know which employee has which book listed.

Our database contains four tables :

- Product
  - id : contains the product ID
  - Name : contains the product name
  - Description : contains a short description of the product
  - Price : contains de product price in Euro
  - Product_name_own : Product_name_own is a foreign key of ID attribute in INVENTORY table
- Inventory
  - id : contains the Inventory ID
  - Name : contains the Inventory name
- User
  - id : contains the user ID
  - Name : contains the user name
  - User_id : User_id is a foreign key of ID attribute in USER table
- Book (Book is a child class of Product)
  - id : contains the book ID
  - Author : contains the name of the book's author
  - Isbn : contains the book isbn

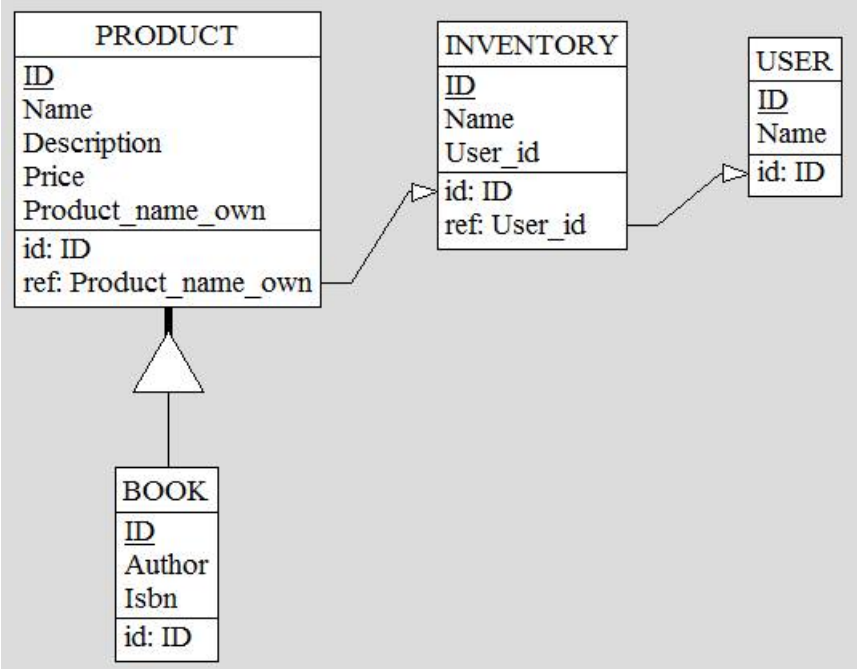


Figure 3.1: diagram of the database

About the GUI, it allows the user to read, add, delete and update information in the database (see figures 3.2 and 3.3). It should be noted that the GUI is built with Java Swing.

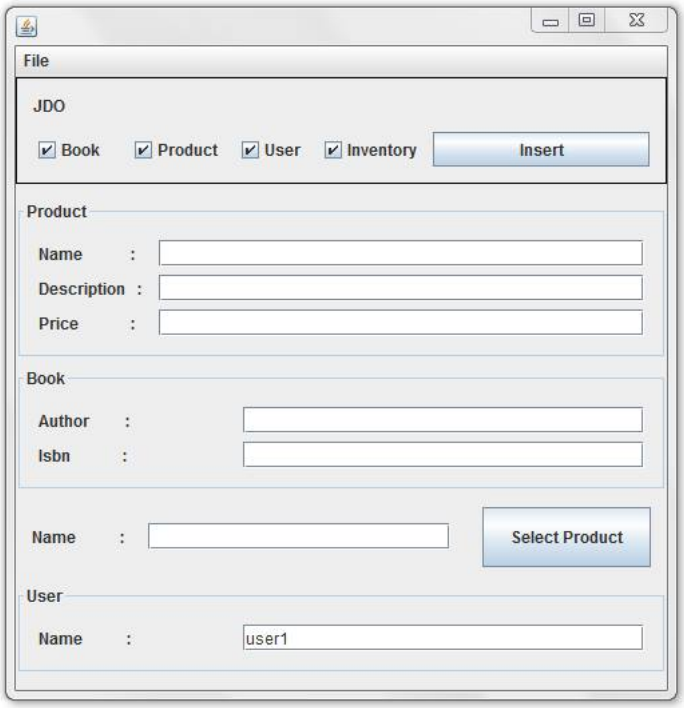


Figure 3.2: project interface

We have a main panel containing all the names of objects that can be persist ( Product, inventory, Book, User). By default , the Product panel is displayed. The checkbox allow to display the panel that allows to enter Information to persist this object. Objects can be separately persist or not. Furthermore, the inventory receives either a new User or an existing user, via either the name or the ID. And there is a button "select Products" for selecting products to add to inventory.

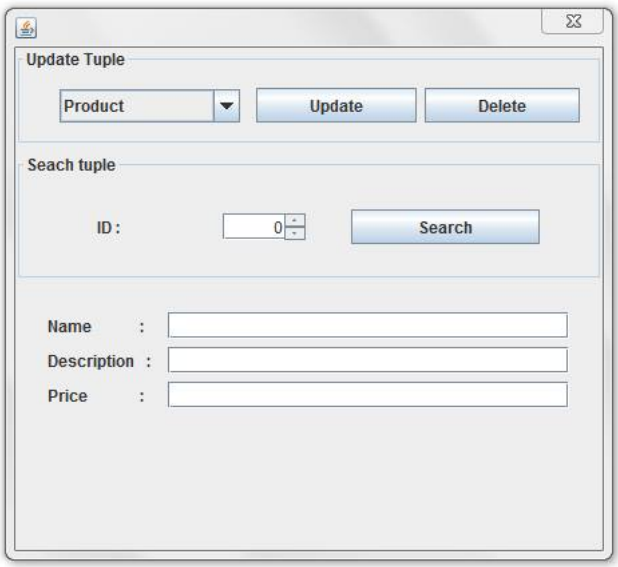


Figure 3.3: project interface

Via this window, the selected object by its id will be, as selected by the user, either delete or edit by selecting the tuple type in the comboBox

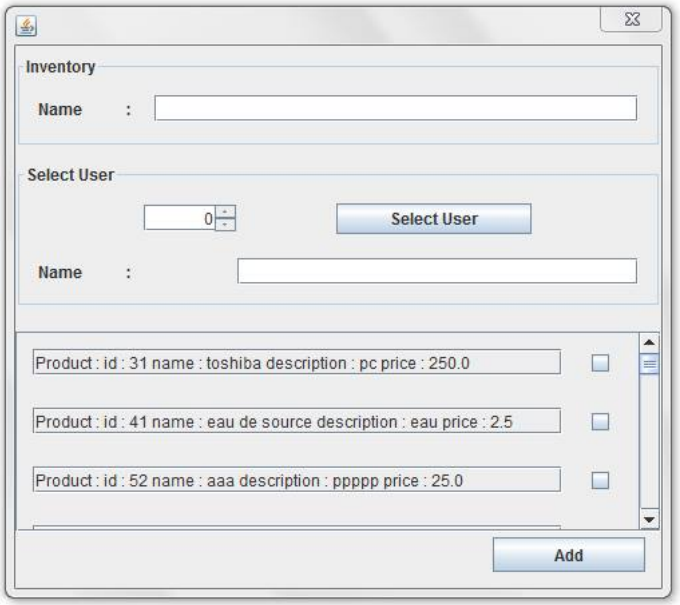


Figure 3.4: project interface

In this menu, the user who made the inventory is recovering through its id. It selects the id of product to add to the inventory. Either the user exists we only persist the inventory, or the user is also persist.

It is important to note that the main aim of our project is to show the advantage of JDO. In fact what differentiates several other JDO technology is its transparency. As you can see below all operations on the database requires very little code and therefore greatly facilitates the developer work.

- Read a product

```
tx.begin();
Product product = pm.getObjectById(Product.class, tupleId);
tx.commit();
```

- Add a product

```
tx.begin();
Product aProduct = new Product(productName, description, price);
pm.makePersistent(aProduct);
tx.commit();
```

- Delete a product

```
tx.begin();
Product product = pm.getObjectById(Product.class, tupleId);
pm.deletePersistent(product)
tx.commit();
```

- Update a product

```
tx.begin();
Product product = pm.getObjectById(Product.class, tupleId);
product.setName(jPanelProduct1.getProductName());
product.setDescription(jPanelProduct1.getDescription());
product.setPrice(jPanelProduct1.getPrice());
pm.makePersistent(product);
tx.commit();
```

# Chapter 4

# Conclusion

In conclusion, this project was a rich experience. Indeed, the persistence of objects is an important concept to consider when we want to develop an application that needs to store objects. Java Data Objects is then used to make transparent the tedious manipulations to persist an object in an database.

This project has allowed us to better understand the advantage of JDO.

- JDO is transparent for the java objects being persisted
- JDO can store data in different storage paradigms
- ...

What is more significant for developers. It can greatly reduce the working time and not be dependent to a database type.

This project has also enabled us to improve our project management competence and improve our ability to search informations and to be able to structure it, in order to prepare a comprehensive report on the subject study.

# Bibliography


[1] Java Data Objects, ROBIN M. ROOS
**Year** 2003
**ISBN** 0-321-12380-8
**URL** http://www.datanucleus.org/downloads/documents/jdo-robinroos-1.0.pdf

[2] Object Oriented Databases. A guide for implementing your own Object Oriented Database with the use of Java Data Objects, ASHLEY NELSON
**URL** http://web.cs.iastate.edu/ smkautz/cs430s14/tutorials/examples/Object%20Oriented%20Databases.pdf

[3] Wikipediaa
**RDBMS** https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es_relationnelle
**Serialization** https://fr.wikipedia.org/wiki/S%C3%A9rialisation
**JDBC** https://en.wikipedia.org/wiki/Java_Database_Connectivity
**ODBMS** https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es_orient%C3%A9e_objet
**EJB** https://fr.wikipedia.org/wiki/Enterprise_JavaBeans
**JDO** https://fr.wikipedia.org/wiki/Java_Data_Objects

[4] DataNucleus
**URL** http://www.datanucleus.org/products/datanucleus/jdo/guides/eclipse.html#install

[5] Persistence Java Types
**URL** http://www.datanucleus.org/products/datanucleus/jdo/types.html

[6] Inheritence
**URL** http://www.datanucleus.org/products/datanucleus/jdo/orm/inheritance.html

[7] Value generation
**URL** http://www.datanucleus.org/products/datanucleus/jdo/value_generation.html

[8] CRUD
**URL** https://cloud.google.com/appengine/docs/java/datastore/jdo/creatinggettinganddeletingdata