XJDataMapper
Java's ORM

Table of contents

XJDataMapper is an ***Object Relational Mapper*** written in Java. It is designed to map your Database tables into easy to work with objects, fully aware of the relationships between each other.

General Features

- Everything is an object!
- Easy to setup, easy to use.
- Lazy Loading (related objects are only loaded upon access).
- You concentrate on business logic and not on SQL.
- Relations and their integrity are automatically managed for you.
- ***One to One***, ***One to Many***, and ***Many to Many*** relations fully supported.
- Can work with different database with little change in code.
- Changes to the object model are made in one place.
- ORM often reduces the amount of code that needs to be written.
- ORM cuts down the development time.

If you are new to ORM, please start here to understand XJDataMapper ORM.

XJDataMapper
Java's ORM

Using This Manual

This manual has been designed with (mostly) standardized formatting to help you get started with XJDataMapper ORM as quickly as possible.

The code in this manual has been formatted to make it easier to see XJDataMapper-specific components, as well as making Java easier to read.

This format should be fairly consistent. Some areas bolded text has been used to highlight information as well.

Special Content

Besides the special content, there are also special sections that may be designed to stand out.

Important Section

Important sections highlight critical information than can affect your usage of XJDataMapper.

Note Section

Notes signify information that is important to getting the best out of XJDataMapper.

XJDataMapper
Java's ORM

Table of Contents

Top of document Top of Topic Table of contents **Next :** Getting Started

XJDataMapper
Java's ORM

Getting Started

The first step is to read all the topics in the General Topics section of the Table of Contents. You should read them in order as each topic builds on the previous one, and may include code examples that you are encouraged to try.

Once you understand the basics you'll be ready to feel the magic of "XJDataMapper" ORM. Below is a glimpse of what's to come!

Class Hierarchy:

XJDataMapper Package consists of following three classes:

- **XConfiguration** is used to setup connection with the database for once.
- **XJData** is a class which is handling all the basic queries for performing operations with database and extends **XConfiguration**.
- **XJDataMapper** is used by models for performing database operations. It extends **XJData**.

Models

XDataMapper models do the work of transforming your Database tables into easy to use objects. Your Model extends XJDataMapper.

Here's a simple example of a few XDataMapper models setup with relationships between each other.

```
public class student extends XJDataMapper{

   public student() throws ClassNotFoundException, SQLException {

      super();
      table = "xj_student";
      selfClass = "student";

      hasOne = "{'class'->class_session,class_session_id,id,xj_class_session,xj_student,student,students}"
            + "{'scholar'->scholar,scholar_id,id,xj_scholar,xj_student,student,students}"
            + "{'rollno'->rollno,id,student_id,xj_student_rollno,xj_student,student,student}"
            + "{'session'->session,session_id,id,xj_session,xj_student,student,students}"
+ "{'studentrecord'->feesapplicable,id,student_id,xj_feesapplicable,xj_student,student,studentinfo}";

      hasMany = "{'fees'->feesapplicable,id,student_id,xj_feesapplicable,xj_student,student,students}";

   }
}
```

```
public class session extends XJDataMapper{

   public session() throws ClassNotFoundException, SQLException {

       super();
       table = "xj_session";
       selfClass = "session";

       hasOne = "{'sessionrecord'->
class_session,id,session_id,xj_class_session,xj_session,class_detail,sessioninfo)}";

       hasMany = "{'classes'->class_session,id,session_id,xj_class_session,xj_session,session,sessions}"
              + "{'students'->student,id,session_id,xj_student,xj_session,session,session}"
              + "{'scholars'->student,id,session_id,xj_session,xj_student,session,sessions}";

   }
}
```

At this level just focus on “what is model used for ?” and not how it is working. Simple it is just setting the table name and relationships within the table.

Further down in the Controllers section, you'll see how easy it is to use models.

Controllers

This is the place where the business logic is implemented. Here models are used for performing operations on databases.

Here's a simple example of a few XJDataMapper controllers.

```
         XJData student = new student();

         student.get(10,5);

         for(XJData result : student.data){

             out.println(result.getSingle("student.id"));

        }
```

Understanding above example :-

In this example object of student model is prepared, 10 records are fetched from the table xj_student leaving starting 5 records, and then their 'id' are shown.

Again I'm saying at this level just focus on "what is controller used for ?" and not how it is working. Simple it is just doing what we are forcing it to do.

You will understand the working of controllers when you will go through the methods provided by XJData.

Cool huh?

I hope that's enough to wet your appetite! It's hard to show the full benefits of XDataMapper in one simple page but I'm sure you've glimpsed the power XDataMapper can give you and in such a simple and logical way!

Please continue on with the General Topics to learn more.

Database Tables

XJDataMapper ORM is implemented with Database normalization " Third normal form" in mind. In short, that means every table is aware only of itself, with fields relevant only to itself. If a table has a **many-many** relationship with another table, it is represented by a special joining table. In either case, the same two objects can only have one relationship between them.

Lets take a look at the below example.

xj_class_detail(id,name)

id	name
1	1a
2	1b
3	2a
4	2b
5	3a
6	3b
7	4a
8	4b
9	5a
10	5b

xj_session(id,name)

id	name
1	2005-2006
2	2006-2007
3	2007-2008
4	2008-2009
5	2009-2010
6	2010-2011
7	2011-2012

xj_class_session(id,class_id,session_id)

id	class_id	session_id
1	1	1
2	2	1
3	1	2
4	3	2
5	1	3
6	2	3

Here we have 3 tables. Tables **xj_class_detail** and **xj_session** are normal tables. Table **xj_class_session** is the joining table that stores the relations between the records of **xj_class_detail** and **xj_session.**

The joining table xj_class_session shows that class_id=2 (1b) is present in session_id=1 (2005-2006) and session_id=3 (2007-2008). The joining table also shows that session_id=1 (2005-2006) contains classes class_id=1(1a) and class_id=2(1b). Thus showing many-many relationship.

Table Naming Rules

Please read this section carefully, as these rules are the foundation of XJDatamapper ORM's methods.

- **Every** table must have a primary, numeric key named id that should be auto incremented. “Note that id should be the first field of every table used with XJDataMapper”.
- Normal tables can be given any names but be sure that the names should be meaningful. So for the table containing class details can be given the name as **class_detail**.
- A joining table must also be given meaningful name as the table showing class and session relationship is given the name as **class_session.**

In-Table Foreign Keys

XJDatamapper ORM is a little more flexible and allows in-table foreign keys as well.

For this example, let's look at the same data, but when there is only one class for each session.

Class_detail

id	**name**
1	1a
2	1b
3	2a
4	2b

Session

id	**Name**	**Class_id**
1	2005-2006	1
2	2006-2007	2
3	2007-2008	3
4	2008-2009	4

Notice we've removed the joining table, and added the column **class_id** directly to the table **session**. Now the relationships are preserved, but we have slightly faster queries as well.

That's pretty much it as far as your normal tables go. The setting to signify if tables are joined with a One to One, One to Many, or Many to Many relationship is setup in the XJDataMapper models.

XJDataMapper Models

In order for XJDataMapper to map your Database tables into objects, you first need to create XJDataMapper model for each table. These models will extend XJDataMapper in order to gain the wonderful functionality of tables as objects.

Basic Template

Below is a basic template you can use to create XJDataMapper models.

```
public class Name extends XJDataMapper {

    public Name() throws Exception
    {
        super(); //calling constructor of base class.
        table = Table_Name;
        selfClass =Self_Class;
        hasOne = Has_One_Relation;
        hasMany = Has_Many_Relation;
    }
}
```

Terms	**Meaning**
Name	Replace this value with the name of your model. For example: class_detail.
XJDataMapper	Extending XJDataMapper is what makes your model a XJDataMapper model.
Table_Name	Name of the table with which the model will work.
Self_Class	Name of the class itself.
Has_One_Relation	Describes the One to One relationship with other tables.
Has_Many_Relation	Describes the Many to Many relationship with other tables.

XJDataMapper
Java's ORM

Relationship Types

There are 3 different types of relationships objects can have with one another. These are:

- One to One
- One to Many
- Many to Many

To describe these relationships, we will use the following Database tables as an example to work with:

Following tables are containing several records but here maximum 10 records are shown.

xj_class_detail

id	name
1	1a
2	1b
3	2a
4	2b
5	3a
6	3b
7	4a
8	4b
9	5a
10	5b

xj_class_session

id	class_id	session_id
1	1	1
2	2	1
3	1	2
4	3	2
5	1	3
6	2	3
7	3	3
8	1	4
9	3	4
10	1	5

xj_session

id	Name
1	2005-2006
2	2006-2007
3	2007-2008
4	2008-2009
5	2009-2010
6	2010-2011
7	2011-2012

xj_feeshead

id	name
1	Education
2	Admission
3	Extra co-curricular
4	Travelling
5	Hostel
6	Food
7	Other

xj_fees

Id	feeshead_id	amount	Name
1	1	12000	Tuition
2	1	500	Library
3	1	500	Science Lab
4	1	4000	Computer Lab
5	1	600	Magazine
6	2	20000	New Admission
7	2	10000	Re-Admission
8	2	100000	Donation
9	3	600	Sports
10	3	2000	Events

xj_feesapplicable

id	fees_id	student_id	amount	paid	due
1	1	794	12000	2000	10000
2	2	794	500	500	0
3	3	794	500	500	0
17	1	795	12000	5000	7000
18	2	795	500	500	0
19	3	795	500	500	0

xj_scholar	xj_student_rollno

id	name
1	Kiran parmar
2	chirab Singh
3	Mahadeva Jeeri
4	Abhijeet Desai
5	Thagana Chatterjee
6	Lasya Mahajan
7	Gnana Mukherjee
8	Abhijit Jaishree
9	Virupaksh Chander

id	student_id	rollno
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9

xj_student

id	session_id	class_session_id	scholar_id
1	1	1	1
2	1	1	2
3	1	1	3
4	1	1	4
21	2	3	21
22	2	3	22
23	2	3	23
24	2	3	24
25	2	3	25

If you've read Database Tables, you will recognize that xj_class, xj_session, xj_feeshead, xj_fees, xj_feesapplicable, xj_scholar and xj_student_rollno are all normal tables.

The other tables, being xj_class_session and xj_student are the joining tables which hold the relationship records between the normal tables.

One to One

One to One relationships occur when there is exactly one record in the first table that corresponds to exactly one record in the related table.
For example, a student has one roll number and a roll number belongs to one student. From the tables above we can see that :

- student_id=1 has rollno=1.
  student_id=2 has rollno=2.

Only student_id=1 will be identified by his/her rollno i.e. 1. Similarly for the others, they can be identified only by their own roll numbers .

One to Many

One to Many relationships occur when each record in the first table has many linked records in the related table but each record in the related table has only one corresponding record in the first table.
For example, one session can have many students but one student can be present in only one session. From the tables above we can see that:

- session_id=1 is having four students with scholar_id=1,2,3,4.
- session_id=2 is having five students with scholar_id=21,22,23,24,25.

Many to Many

Many to Many relationships occur when each record in the first table has many linked records in the related table and vice-versa. These are always stored using dedicated join tables.
For example, many students can give one type of fees and one student can give many types of fees. From the tables above we can see that:

- student_id=794 is giving fees of type fees_id=1,2,3.
- fees_id=1 is given by students with student_id=794 and 795.

So we can see that one student can pay many fees and one fees can be payable by many students.

Setting up Relationships

In order for your XJDataMapper models to know the relationships it has between other XJDataMapper models, you need to set its *has_One* and *$has_Many* variables. You do this by adding a class variable of *has_one* and *has_many*, both of which are strings.

The values you add to this string should follow the below syntax.

has_One/has_Many = "{ 'Relation_Name' -> Related_Class, join_self_as, join_other_as, Related_Table, Self_Table, Self_Class, Other_Model_Relation }";

For Example :- to define that one student can have one roll no ; we will establish the relationship as,

hasOne= `"{'rollno' ->rollno,id,student_id,xj_student_rollno,xj_student,student,student}";`

Understanding above example :-

As it is clear by the above example that we are setting one-one relationship between xj_student table and xj_student_rollno table. Now we'll have a look on the terms used in setting a relationship.

- Relation_Name :- It is the name of the relation which will be used by controllers.
- Related_Class = It is the name of the class where Other_Model_Relation is defined.
- join_self_as = It is the name of joining field present in self table.
- join_other_as = It is the name of joining field present in related table.
- Related_Table = It is the name of the other table with which relationship is set.
- Self_Table = It is the name of table who is initiating relationship.
- Self_Class = It is the name of the class in which this relationship is defined.
- Other_Model_Relation = It is the name of relation which is defined in related class.

Note : The below rules are necessary to be followed in order to make XJDataMapper to handle relationship between tables.

Rules to be followed:-

1) Every relation must be written between pair of braces i.e. {}.
2) Relation name must be enclosed within single quotes i.e. ‘ ’.
3) There must be arrow sign. i.e. -> after relation name.
4) Other attributes of relations are separated using comma.

Similarly has_Many relationship can be set in model.
For example :- This relationship tells one student can pay many types of fees.

```
hasMany = "{'fees'
->feesapplicable,id,student_id,xj_feesapplicable,xj_student,student
,students}";
```

Multiple Relations

You can setup as many relationships as you need. You simply append relationships in has_One and has_Many variables as needed. **For example** :-

```
hasOne= "{'class'
      ->class_session,class_session_id,id,xj_class_session,xj_student
      ,student,students}"

      + "{'scholar'
      ->scholar,scholar_id,id,xj_scholar,xj_student,student,students}"

      + "{'rollno'
      ->rollno,id,student_id,xj_student_rollno,xj_student,student
      ,student}"

      + "{'session'
      ->session,session_id,id,xj_session,xj_student,student,students}"

      + "{'studentrecord'
      ->feesapplicable,id,student_id,xj_feesapplicable,xj_student
      ,student,studentinfo}";

hasMany= "{'fees'
          ->feesapplicable,id,student_id,xj_feesapplicable,xj_student
          ,student,students}";
```

XJDataMapper
Java's ORM

DataMapper in Controllers

Here we can finally get to the good stuff! By now you've got your XJDataMapper models all setup so we can begin using our tables as objects.

I won't go into too much detail here as the **Functions** section of the Table of Contents contains detailed usage instructions on the Get, Save, Update and Delete functions, as well as others. The **Relationships** section also has detailed usage instructions for accessing and modifying your relationships between objects.

Let's get ready to enjoy the magic of XJDataMapper.

Get

You'll find Get is used where you want to retrieve the data from tables.

Now, let's look at all the available methods. We'll assume we have a XJDataMapper Model is setup with its Object. In this chapter, just understand the functioning of each method. When you'll go through with the methods then you can easily understand the Example Application.

Subsections

- Basic Get (This Section)
- Field Selection
- Aggregate Functions
- Limiting Results
- Query Grouping
- Other Features

object.get();

```
/* Runs the selection query and returns the result. Retrieve all
records from a table: */

XJData object = new Model();
object.get();

The object is populated with all objects from its corresponding table.
```

object.get(limit);

```
/* returns the record up to the limit specified */

XJData object = new Model();
object.get(10);
```

object.get(limit,offset);

```
/* returns the record up to the limit & offset specified */

XJData object = new Model();
object.get(10,20);

The object is populated with 10 objects from its corresponding table,
starting from record 20.
```

object.get(colArray);

```
/* returns the record as per the columns specified */

XJData object = new Model();
___________________________// some code to be written , see in example
object.get(colArray);

Where colArray is the ArrayList containing names of columns to be
displayed.
```

for example :-

```
      XJData student = new student();
      ArrayList colArray = new ArrayList();
      colArray.add("id");
      colArray.add("scholar_id");
      student.get(colArray);
```

Explanation :- only id, scholar_id of students will be fetched.

object.get_Where(field, op, value);

```
/* returns the record where field is having value [=,<,<=,>,>=,!=] the
value passed as parameter */

XJData object = new Model();
object.get_Where(field,op,value);

In this case records are fetched based on the condition set in the
where clause.
```

for example :-

```
      XJData student = new student();
      student.get_Where("id","=","1");
```

Explanation:- only record of that student will be fetched whose id=1.

Identical to the above function except that it permits you to add a "where" clause within the get function, instead of using the object.where(condition).get() function.

object.get_By(field,value);

```
/* returns the record where field is having value [=] the value passed
as parameter */

XJData object = new Model();
object.get_By(field,value);
```

for example :-

```
      XJData student = new student();
      student.get_By("id","1");
```

Explanation:- only record of that student will be fetched whose id=1.

Identical to get_Where(field,op,value) function but here only "=" operator will be used implicitly.

object.getSingle(fieldName);

/* This method is used for viewing the result */

Case 1: When only single row is obtained then same Model object will contain the result.

```
XJData object = new Model();
___________________________// some code to be written , see in example
out.println(object.getSingle(field));
```

For Example:-

```
XJData student = new student();
student.get_where("id","=","1");
out.println(student.getSingle("student.scholar_id"));
```

Understanding above example :-

Once get method is used, the result is stored in the student object. Now to retrieve particular column values we have to use getSingle method. This method takes one argument that is name of field whose value we want to show i.e. "student.scholar_id".

Case 2: When multiple rows obtained then temporary XJData variable is used to store the result.

```
XJData object = new Model();
___________________________// some code to be written , see in example
out.println(result.getSingle(Field));
```

For Example :-

```
XJData student = new student();
student.get(); // records of all the students is returned.
for( XJData result : student.data){

        out.println(result.getSingle("student.scholar_id"));
}
```

Understanding above example :-

Once get method is used, the result is stored in the data[] array of object. Now to retrieve particular column values we have to use getSingle method. The above example will print scholar id of every student.

Note :
1) getSingle method must follow get method. Otherwise you will not get the desired output.
2) It is necessary to give the name of model together with field name using dot operator. Otherwise you will get null value. Example: getSingle("Model.id")

Field Selection

Use the following methods to limit which fields are selected.

object.select(colArray);

```
/* this is used when we want to have selected columns . It permits you
to write the SELECT portion of your query:*/

XJData object = new Model();
object.select(colArray).get();

The object is populated with all objects from its corresponding table,
but with only the fields present in the colArray.

The above method will work same as object.get(colArray) method.
```

object.distinct();

```
/*Adds the "DISTINCT" keyword to a query*/

XJData object = new Model();
object.distinct.get();

When above method is called, a DISTINCT selection of records will be
made.
```

Aggregate Functions

object.select(type, field);

/* Using this method aggregate functions like (sum, max, min, count, average) can be used within query. **type** is the type of aggregate function and **field** is the column name */

For Example :-

```
XJData object = new Model();
object.select("max","id").get();
out.println(object.getAggregate());
```

Explanation :- will find the maximum id value from the table.

The object is populated with a single object from its corresponding table, but with only the id field populated, which contains the maximum id.

The result of the query is shown using getAggregate method.

Note : Only get method is used for finding the result of aggregate functions. Other variations of get method won't work. It is necessary to use the getAggregate method after using get method in order to show the results.

Similarly, aggregate functions like **min(),avg(),count(),sum()** can be used.

Limiting Results

Use the following methods to limit which rows are returned.

Where clause

object.where(field,op,value);

```
/* Generates where portion of query */

XJData object = new Model();
object.where(field,op,value);
```

for example :-

```
XJData student = new student();
student.where("id","=","1").get();
```

Explanation:-only record of that student will be fetched whose id = 1.

If you use multiple where method calls they will be chained together with ***AND*** between them:

for example :-

```
XJData student = new student();
student.where("id","=","1").where("scholar_id","=","1").get();
```

Explanation:-only record of that student will be fetched whose id = 1 and scholar_id=1.

object.where(array);

```
/* where condition specified in array. each array element specifies one
condition. 'field' , 'condition operator', 'value' are seperated  using
-> symbol */

XJData object = new Model();
___________________________// some code to be written , see in example
object.where(array).get();
```

for example :-

```
        XJData student = new student();
        ArrayList array = new ArrayList();
        array.add("id -> = -> 1");
        array.add("session_id -> = -> 1");
        student.where(array).get();
```

Explanation :- only record of that student will be fetched whose id=1 and session_id=1.

object.where(id, XJDataObject);

```
/* generates query where id = (id of record contained by
XJDataObject)*/

XJData object1 = new Model1();
___________________________// some code to be written , see in example
XJData object2 = new Model2();
Object2.where(id,object1).get();
```

for example :-

```
      XJData class_detail = new class_detail();
      class_detail.where("id","=","1").get();
      XJData class_session = new class_session();
      class_session.where("class_id",class_detail).get();
```

Explanation :- records from class_session will be retrieved where class_session table have class_id same as id of class_detail.

object.where_Or(field,op,value);

```
/* Generates where portion of query */

XJData object = new Model();
object.where_Or(field,op,value);
```

for example :-

```
      XJData student = new student();
      student.where(“id”,”=”,”1”).where_Or(“id”,”=”,”2”).get();
```

Explanation:- records of students will be fetched where id=1 or id=2.

If you use multiple where_Or method calls they will be chained together with ***OR*** between them.

Note :- where_Or(array) works same as where(array). The only difference is where clauses are joined using OR instead of AND.

object.where_*In*(field,values);

/*Generates a WHERE field IN ('value1', 'value2') SQL query joined with AND if appropriate. **Field** is the name of column for which where condition is built and **values** is the ArrayList containing the values that are compared with field’s value. Each array element specifies one value. Field value is compared with the values in array list */

```
XJData object = new Model();
___________________________// some code to be written , see in example
Object.where_In(field,values);
```

for example :-

```
       XJData student = new student();
       ArrayList values = new ArrayList();
       values.add("1");
       values.add("2");
       values.add("3");
       student.where_In("id", values).get();
       for(XJData result : student.data)
       {
           out.println(result.getSingle("student.id"));
       }
```

Explanation:- records where the id is 1,2 or 3 will be returned

object.where_Or_In(field,values);

/*Generates a WHERE field IN ('value1', 'value2') SQL query joined with OR if appropriate. It works same as where_In method, the only difference is when where_Or_In methods are used in chaining then they are chained with **OR** between them */

```
XJData object = new Model();
___________________________// some code to be written , see in example
Object.where_In(field1,values1).where_Or_In(field2,values2);
```

for example :-

```
       XJData student = new student();
       ArrayList values1 = new ArrayList();
       values1.add("1");
       values1.add("2");
       ArrayList values2 = new ArrayList();
       values2.add("3");
       values2.add("4");
       student.where_In("id", values1)
       .where_Or_In("scholar_id", values2).get();
       for(XJData result : student.data)
       {
           out.println(result.getSingle("student.id"));
       }
```

Explanation:- records where id is 1 or 2 or scholar_id is 3 or 4 will be returned

object.where_Not_In(field,values);

/*Generates a WHERE field NOT IN ('value1', 'value2') SQL query joined with AND if appropriate. It is opposite to where_In method because by using where_Not_In method those records are fetched where **field** value does not contained in **values** ArrayList*/

```
XJData object = new Model();
___________________________// some code to be written , see in example
Object.where_Not_In(field,values);
```

for example :-

```
XJData student = new student();
ArrayList values = new ArrayList();
values.add("1");
values.add("2");
values.add("3");
student.where_Not_In("id", values).get();
for(XJData result : student.data)
{
    out.println(result.getSingle("student.id"));
}
```

Explanation:- records where the id is other than 1,2 or 3 will be returned

object.where_Or_Not_In(field,values);

/*Generates a WHERE field NOT IN ('value1', 'value2') SQL query joined with OR if appropriate. It works same as where_Not_In method, the only difference is when where_Or_Not_In methods are used in chaining then they are chained with **OR** between them */

```
XJData object = new Model();
___________________________// some code to be written , see in example
Object.where_In(field1,values1).where_Or_In(field2,values2);
```

for example :-

```
XJData student = new student();
ArrayList values1 = new ArrayList();
values1.add("1");
values1.add("2");
ArrayList values2 = new ArrayList();
values2.add("3");
values2.add("4");
student.where_Not_In("id", values1)
.where_Or_Not_In("scholar_id", values2).get();
for(XJData result : student.data)
{
    out.println(result.getSingle("student.id"));
}
```

Explanation:- records where id is not 1 or 2 or where scholar_id is not 3 or 4 will be returned

Prev : Aggregate Functions Top of document Top of Topic Table of contents **Next** : Query Grouping

XJDataMapper
Java's ORM

object.where_Between(field,value1,value2);

```
/* Generates Sql where field is between value1 and value2 joined with
AND if appropriate */

XJData object = new Model();
object.where_Between(field,value1,value2);
```

for example :-

```
      XJData student = new student();
      student.where_Between("id","1","3").get();
```

Explanation:-record of that student will be fetched whose id is between 1 and 3.

object.where_Or_Between(field,value1,value2);

```
/* Generates Sql where field is between value1 and value2 joined with
OR if appropriate */

XJData object = new Model();
object.where(field,op,value).where_Or_Between(field,value1,value2);
```

for example :-

```
      XJData student = new student();
      student.where("session_id","=","1")
      .where_Or_Between("id","3","7").get();
```

Explanation:-record of that student will be fetched whose session_id is 1 or id is between 1 and 3.

object.where_Not_Between(field,value1,value2);

```
/* Generates Sql where field is NOT between value1 and value2 joined
with AND if appropriate */

XJData object = new Model();
object.where_Not_Between(field,value1,value2);
```

for example :-

```
XJData student = new student();
student.where_Not_Between(“id”,”1”,”830”).get();
```

Explanation:-record of that student will be fetched whose id is not between 1 and 830.

object.where_Or_Not_Between(field,value1,value2);

/* Generates Sql where field is **NOT** between value1 and value2 joined with **OR** if appropriate */

XJData object = new Model();
object.where(field,op,value).where_Or_Not_Between(field,value1,value2);

for example :-

```
XJData student = new student();
student.where_Between(“id”,”1”,”10”)
.where_Or_Not_Between(“id”,”11”,”830”).get();
```

Explanation:-record of that student will be fetched where id is between 1 and 10 or where id is not between 11 and 830.

Like Clause

This method enables you to generate LIKE clauses, useful for doing searches.

object.like(field,value);

```
/* Generates query using like clause. The query give the output on
perfect match only*/

XJData object = new Model();
object.like(field,value);
```

for example :-

```
      XJData feeshead = new feeshead();
      feeshead.like("name","education").get();
```

Explanation:-only that records will be fetched where name like education.

If you use multiple like method calls they will be chained together with ***AND*** between them:

for example :-

```
      XJData scholar = new scholar();
      scholar.like("name","Abhijit Jaishree").like("id","8").get();
```

Explanation:-only record of that scholar will be fetched whose name like *Abhijit Jaishree* and id like 8.

object.like(array);

```
/* Generates query using like clause. condition is given in array.
field and value is seperated by -> symbol. works when there is exact
match */

XJData object = new Model();
___________________________// some code to be written , see in example
object.like(array);
```

for example :-

```
XJData scholar = new scholar();
ArrayList values = new ArrayList();
values.add("name->Abhijit Jaishree");
values.add("id->8");
scholar.like(values).get();
```

Explanation:-only record of that scholar will be fetched whose name like *Abhijit Jaishree* and id like 8.

object.like(field,value,place);

/* Generates query using like clause. works also when there is not exact match. If you want to control where the wildcard (%) is placed, you can use third argument "place". place(after, before, both) decides where to look the pattern */

```
XJData object = new Model();
object.like(field,value,place);
```

for example :-

```
XJData feeshead = new feeshead();
feeshead.like("name","edu","after").get();
```

Explanation:-only that records will be fetched where name like edu%.

Similarly, "before","both" can be used as third argument in like method.

object.like_Or(field,value);

/* Generates query using like clause. The query give the output on perfect match only . This is identical to like method, except that multiple instances are joined by **OR** */

```
XJData object = new Model();
object.like_Or(field,value);
```

for example :-

```
      XJData feeshead = new feeshead();
      feeshead.like("name","education")
      .like_Or("name","admission").get();
```

Explanation:-only that records will be fetched where name like education or admission.

object.like_Or(array);

```
/* Generates query using like clause. condition is given in array.
field and value is seperated by -> symbol. works when there is exact
match. This is identical to like(array) method, except that multiple
instances are joined by OR */

XJData object = new Model();
___________________________// some code to be written , see in example
object.like_Or(array);
```

for example :-

```
      XJData scholar = new scholar();
      ArrayList values1 = new ArrayList();
      Values1.add("name->Abhijit Jaishree");
      ArrayList values2 = new ArrayList();
      Values2.add("id->8");
      scholar.like(values1).like_Or(values2).get();
```

Explanation:-only record of that scholar will be fetched whose name like *Abhijit Jaishree* or id = 8.

object.like_Or(field,value,place);

```
/* Generates query using like clause. works also when there is not
exact match. If you want to control where the wildcard (%) is placed,
you can use third argument "place". place(after, before, both) decides
where to look the pattern. This is identical to like(field,value,place)
method, except that multiple instances are joined by OR */
```

```
XJData object = new Model();
object.like_Or(field,value,place);
```

Similarly, “before”,”both” can be used as third argument in like_Or method.

object.like_Not(field,value);

/*This function is identical to **like(field,value)**, except that it generates NOT LIKE statements. works when there is exact match.*/

```
XJData object = new Model();
object.like_Not(field,value);
```

for example :-

```
    XJData scholar = new scholar();
    scholar.like_Not(“name”,”Abhijit Jaishree”).get();
```

Explanation:- records of that scholar will be fetched whose name is not like *Abhijit Jaishree*.

Note :- like_Not also have two more versions like_Not(array) and like_Not(field,value,place). Both of them works same as like(array) and like(field,value,place) respectively but they will work for NOT LIKE queries.

object.like_Or_Not(field,value);

/*This function is identical to **like_Not(field,value)**. This is identical to like method, except that multiple instances are joined by **OR** */

```
XJData object = new Model();
object.like_Or_Not(field,value);
```

for example :-

```
      XJData scholar = new scholar();
      scholar.like("id","8").like_Or_Not("id","1").get();
```

Explanation:- records of that scholar will be fetched whose id is like 8 or id is not like 1.

Note :- like_Or_Not also have two more versions like_Or_Not(array) and like_Or_Not(field,value,place). Both of them works same as like_Not(array) and like_Not(field,value,place) respectively but the multiple instances of methods are joined by OR rather than AND.

XJDataMapper
Java's ORM

Query Grouping

You can create more advanced queries by grouping your clauses. This allows you to specify construct such as (a OR b) AND (c OR NOT d).

Note: Every *group_start* must be balanced by exactly one *group_end*.

object.group_Start();

Starts a group. Every statement generated until *group_end* will be joined by an **AND** to the rest of the query. Groups can be nested.

object.group_Or_Start();

Every statement generated until *group_end* will be joined by an **OR** to the rest of the query.

object.group_Not_Start();

Every statement generated until *group_end* will be joined by an **AND** NOT to the rest of the query.

object.group_Or_Not_Start();

Every statement generated until *group_end* will be joined by an **OR** NOT to the rest of the query.

object.group_end();

Ends the most recently started group.

Grouping Example

```
XJData student = new student();
student.where("session_id","=","1").group_Start().where("id","=","1")
.where_Or("id","=","2").group_End().get();
```

Explanation :-

The above example will generate this query "select xj_student .* from xj_student where (xj_student.session_id='1')and ((xj_student.id = '1') or (xj_student.id = '2'))"

Prev : Limiting Results Top of document Top of Topic Table of contents **Next** : Other Features

Other Features

Other features related to generating your SQL query.

Groupby Clause

object.groupby(field);

```
/*Permits you to write the GROUP BY portion of your query*/

XJData object = new Model();
object.groupby(field);
```

for example :-

```
      XJData feesapplicable = new feesapplicable();
      feesapplicable.groupby("fees_id").get();
```

Explanation:- records will be fetched by grouping "fees_id" field.

object.groupby(fieldArray);

You can also pass an array of multiple values as well.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.groupby(fieldArray);
```

for example :-

```
      XJData feesapplicable = new feesapplicable();
      ArrayList fieldArray = new ArrayList();
      fieldArray.add("student_id");
      fieldArray.add("fees_id");
      feesapplicable.groupby(fieldArray).get();
```

Explanation:- records will be fetched by first grouping "fees_id" and then "student_id" field.

Having Clause

/*Permits you to write the HAVING portion of your query.*/

object.having(field,op,value);

```
XJData object = new Model();
object.having(field,op,value);
```

for example :-

```
XJData feesapplicable = new feesapplicable();
feesapplicable.having("student_id","=","795").get();
```

Explanation:- records will be fetched where student_id=795.

Note: *If having clause is used without groupby clause, then it works same as where clause. As it is shown in above example.*

Using having clause with Groupby clause.

for example :-

```
XJData feesapplicable = new feesapplicable();
feesapplicable.groupby("student_id")
.having("student_id","=","795").get();
```

Explanation:- records will be first grouped by "student_id" and then only record will be fetched where student_id=795.

object.having(array);

```
/*You can also pass an array of multiple values as well*/

XJData object = new Model();
___________________________// some code to be written , see in example
object.having(array);
```

for example :-

```
      XJData feesapplicable = new feesapplicable();
      ArrayList array = new ArrayList();
      array.add("student_id->=->795");
      array.add("fees_id->=->2");
      feesapplicable.having(array).get();
```

Explanation:- records will be fetched where student_id=795 and fees_id=2.

object.having_Or(field,op,value);

Identical to having(field,op,value), only separates multiple clauses with "OR".

object.having_Or(array);

Identical to having(array), only separates multiple clauses with "OR".

object.having_Not(field,op,value);

Opposite to having(field,op,value), generates having NOT clauses in sql.

```
XJData object = new Model();
object.having_Not(field,op,value);
```

for example :-

```
XJData feesapplicable = new feesapplicable();
      feesapplicable.having_Not("student_id","=","795").get();
```

Explanation:- records will be fetched where student_id != 795.

object.having_Not(array);

```
Opposite to having(array).
```

object.having_Or_Not(field,op,value);

```
Identical to having_Not(field,op,value), only separates multiple
clauses with "OR".
```

object.having_Or_Not(array);

```
Identical to having_Not(array), only separates multiple clauses with
"OR".
```

XJDataMapper
Java's ORM

Orderby Clause

/*Permits you to write the ORDER BY portion of your query.*/

object.orderby(field,type);

/*The first parameter contains the name of the column you would like to order by. The second parameter lets you set the direction of the result. Options are asc or desc.*/

```
XJData object = new Model();
object.orderby(field,op,value);
```

for example :-

```
XJData feesapplicable = new feesapplicable();
feesapplicable.orderby("id","desc").get();
```

Explanation:- records will be fetched by ordering "id" in descending order.

object.orderby(list);

/*This method allows you to pass the sequence of fields on which ordering should be done*/

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.orderby(list);
```

for example :-

```
XJData feesapplicable = new feesapplicable();
HashMap list = new HashMap();
list.put("id","desc");
list.put("amount","asc");
feesapplicable.orderby(list).get();
```

Explanation:- records will be fetched by first ordering "id" in descending order and then "amount" in ascending order.

object.limit(limit);

```
/*Lets you limit the number of rows you would like returned by the
query*/

XJData object = new Model();
object.limit(limit);
```

for example :-

```
      XJData feesapplicable = new feesapplicable();
      feesapplicable.limit("10").get();
```

Explanation:- the number of records returned will be limited to 10

object.limit(limit,offset);

```
/*The second parameter lets you set a result offset.*/

XJData object = new Model();
object.limit(limit,offset);
```

for example :-

```
      XJData feesapplicable = new feesapplicable();
      feesapplicable.limit("10","1").get();
```

Explanation:- the number of records returned will be limited to 10, starting from record 1.

XJDataMapper
Java's ORM

Save

This methods inserts record into table. There are a number of ways to run Save. Let's have a look.

save();

```
/* This version of save method allows you to insert column values as
per your order. You can also give value for column id, although it is
auto-incremented */

XJData object = new Model();
___________________________// some code to be written , see in example
object.save();
```

for example :-

```
      XJData feeshead = new feeshead();
      feeshead.setValues("name","Hostel");
      feeshead.save();
```

Explanation:- new record will be inserted, with name = hostel.

save(array);

```
/* This version of save method allows you to insert column values as
per the order of fields in table.*/

XJData object = new Model();
___________________________// some code to be written , see in example
object.save();
```

for example :-

```
      XJData feeshead = new feeshead();
      ArrayList array = new ArrayList();
      array.add("Hostel");
      feeshead.save(array);
```

Explanation:- new record will be inserted, with name = hostel.

Note :-

- you don't have to specify the field name for which you are setting the value. You just have to set the column values as per the order of columns.
- You can't set value for column "id" here. Because it is auto-incremented. If you want to set the value for column id then you have to use save() method.

saveMultiple(list);

```
/* This version of save method allows you to insert multiple records
using single query. You can set multiple records by including a full
record in pair of () and then other records are separated by comma  */

XJData object = new Model();
object.saveMultiple(list);
```

for example :-

```
XJData feeshead = new feeshead();
feeshead.saveMultiple("('Hostel'),('Food')");
```

Explanation:- two records will be inserted, with name = hostel and food.

Note :-

- you don't have to specify the field name for which you are setting the value. You just have to set the column values as per the order of columns.
- You can't set value for column "id" here. Because it is auto-incremented. If you want to set the value for column id then you have to use save() method.

Update

If you want to update multiple objects or rows at the same time, you can do that easily using the **update** method. This method accepts one or more field-value pairs, and can use many of the existing DataMapper functions to determine which columns to update.

Basic Updates

object.update();

Set a Field in Every Row to the Same Value

The simplest form of **update** is to update every single row in a table.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.update();
```

for example :-

```
      XJData feeshead = new feeshead();
      feeshead.get();
      for(XJData datas : feeshead.data){
          datas.setValues("name","xyz");
          datas.update();
      }
```

Explanation:- the above code will update every row of feeshead table and change name of every row to "xyz".

Limiting Which Rows Are Updated

You can limit which rows are updated by setting condition.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.update();
```

for example :-

```
      XJData feeshead = new feeshead();
      feeshead.where("id","=","1").get();
      for(XJData datas : feeshead.data){
          datas.setValues("name","xyz");
          datas.update();
      }
```

Explanation:- the above code will update only that rows of feeshead table which get extracted using get method.

Updating Multiple Columns

Object.update(fieldArray,valueArray);

You can pass fields and their values as an array if you need to update more than one column at a time.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.update(fieldArray,valueArray);
```

for example :-

```
      XJData fees = new fees();
      fees.where("id","=","8").get();
      ArrayList fieldArray = new ArrayList();
      ArrayList valueArray = new ArrayList();
      fieldArray.add("feeshead_id");
      valueArray.add("1");
      fieldArray.add("name");
      valueArray.add("library");

      for(XJData datas : fees.data){
          datas.update(fieldArray,valueArray);
      }
```

Explanation:- the above code will update row with id=8 and sets feeshead_id=1 and name=library .

Note :- if you are using this method then it's your responsibility to assign values to appropriate fields. If it is not done then you'll not get desired output.

Object.update(array);

You can pass single array consisting of fields and their updated values to update one or more columns at a time.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.update(array);
```

for example :-

```
      XJData fees = new fees();
      fees.where("id","=","8").get();
      ArrayList array = new ArrayList();
      array.add("feeshead_id->2");// Field and its value is separated by -> symbol.
      array.add("name->donation");

      for(XJData datas : fees.data){
          datas.update(array);
      }
```

Explanation:- the above code will update row with id=8 and sets feeshead_id=2 and name=donation.

Note :- If compared with update(fieldArray,valueArray),this method is better option because you are assigning field and its value in the same array. So you don't have to worry about sequence of fields and values individually.

Object.update(list);

You can pass single list consisting of fields and their updated values to update one or more columns at a time.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.update(list);
```

for example :-

```
        XJData fees = new fees();
        fees.where("id","=","8").get();
        HashMap list = new HashMap();
        list.put("feeshead_id","1");
        list.put("name","library");

        for(XJData datas : fees.data){
            datas.update(list);
        }
```

Explanation:- the above code will update row with id=8 and sets feeshead_id=1 and name=library.

Note :- If compared with update(array),this method is better option because here you don't have to worry about syntax of fixing fields and values in array. You just have to put fields and their values in HashMap.

As you have seen there are many versions of the same method, now its users choice what he/she wants to use.

Delete

Running Delete on an existing object will delete its corresponding record from the database.

Basic deletes

object.remove();

Delete all rows of table

The simplest form of remove is to delete every single row in a table.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.remove();
```

for example :-

```
      XJData fees = new fees();
      fees.get();
      for(XJData result : fees.data )
      {
          result.remove();
      }
```

Explanation:- the above code will remove every row of fees table.

Limiting Which Rows to be deleted

You can limit which rows are deleted by setting condition.

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.remove();
```

for example :-

```
      XJData feeshead = new feeshead();
      feeshead.where("id","=","1").get();
      for(XJData datas : feeshead.data){
          datas.remove();
      }
```

Explanation:- the above code will remove only that row of fees table which get extracted using get method.

XJDataMapper
Java's ORM

Getting the Number of Affected Rows

XJDatamapper ORM also provides method to find the number of affected rows when table gets modified(insertion, updation, deletion).

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.getAffectedRows();
```

for example :-

```
      XJData fees = new fees();
      fees.saveMultiple("(1,'library',200),(2,'donation',500)");
      out.println(fees.getAffectedRows());
```

Explanation:- the above code will return the number of rows get affected.

Note :- using update,remove,save methods you always get no of affected rows = 1. Because calls to update(), remove(), save() modifies one row at a time.

Copying Records

You can copy records of table using copy method of XJDataMapper.

object.copy();

```
XJData object = new Model();
___________________________// some code to be written , see in example
object.copy();
```

for example :-

```
      XJData feesapplicable = new feesapplicable();
      feesapplicable.where("id","=","1").get();
      for(XJData result : feesapplicable.data)
      {
          XJData copied1 = result.copy();
          copied1.setValues("student_id","1");
          copied1.update();
          XJData copied2 = result.copy();
          copied2.setValues("student_id","2");
          copied2.update();
      }
```

Explanation:-The above code will copy two times the record where id=1. Benefit of above approach is you don't have to insert those records which are almost same. You just copy the desired record and update particular values you want to change in it.

XJDataMapper
Java's ORM

Accessing Relationships

You need to specify your XJDataMapper model's has_One and has_Many relationships before it is possible to access them.
Read setting up relationships to see how.

Now to access the relationship **setRelation** method of XJDataMapper is used.

Below is the example to use this method.

Query---------> Suppose we want to know the class name of class_id given in class_session table where id=1.

One Method :-

```
XJData class_session = new class_session();
class_session.where("id","=","1").get();//desired row is obtained

for(XJData result1 : class_session.data)
{
   result1.setRelation("class_session->classinfo").get();//relationship is set between tables
   for(XJData result2 : result1.data)
   {
  out.println(result2.getSingle("class_session.class_id")+"&nbsp;"+result2.getSingle("class_detail.name"));
   }
}
```

Second Method :-

First have a look at this code,

```
XJData class_session = new class_session();
class_session.where("id","=","2").get();
class_session.setRelation("class_session->classinfo").get();
out.println(class_session.getSingle("class_session.class_id")+"&nbsp;"+class_session.getSingle("class_det
ail.name"));
```

Confused??? How the above code is working.
Don't worry. I'm explaining it.

Since we know that class_session.where("id","=","1").get(); will bring one record because it is unique.
It is taught earlier that when get methods brings single record then it can be received in the same object.
The relationship which class_session sets is also one-one so the result set can also be occupied within the same object due to the same reason explained above.

Uptil now you have seen one method to access relationship. This can be done by other methods also in variety of ways.

Note :- In functions section of manual you have seen various methods provided by XJDataMapper. All those were methods that can be used with single table.

Now when we talk about relationships, then we have to deal two tables at once. One is self table and other is related table.

You have already seen functions for self table. XJDataMapper also provides same functions to perform operations with related table .

The only difference is that the functions for related table starts from letter 'R' for example :- Rwhere(field,op,value),Rremove() etc.

Note that save method is not used while accessing relationship.

object.where_Related(relation,field,op,value);

This method sets the relation and also specify the condition for the related table using second, third, fourth parameter.

where_Related(relation,field,op,value) = setRelation(relation) + Rwhere(field,op,value);

```
XJData object = new Model();
object.where_Related(relation,field,op,value).get();
```

object.get_By_Related(relation,field,op,value);

This method sets the relation and also specify the condition for the related table, gets the result set.

get_By_Related(relation,field,op,value) = setRelation(relation) + Rwhere(field,op,value) + get();

```
XJData object = new Model();
object.get_By_Related(relation,field,op,value).get();
```

object.where_Related(relation,array);

Works same as where_Related(relation,field,op,value) , the difference is here condition is specified in the form of array.

object.get_By_Related(relation,array);

Sets the relationship same as where_Related(relation,array) ,and then gives the result set too.

object.where_Related_In(relation,field,valueArray);

As in where_Related(relation,field,op,value) method condition is specified in Rwhere(field,op,value), here the condition is specified using Rwhere_In(field,valueArray).

object.get_By_Related_In(relation,field,valueArray);

As in get_By_Related(relation,field,op,value) method condition is specified in Rwhere(field,op,value), here the condition is specified using Rwhere_In(field,valueArray).

Sets the relationship , and then gives the result set too.

object.where_Related_Not_In(relation,field,valueArray);

Sets the relationship same as where_Related_In(relation,field,valueArray) but the condition is applied opposite to that method.

object.get_By_Related_Not_In(relation,field,valueArray);

Sets the relationship same as get_By_Related_Not_In(relation,field,valueArray) but the condition is applied opposite to that method.

Sets the relationship , and then gives the result set too.

XJDataMapper
Java's ORM

Example Application

Now it's time to see an example application that implements XJDataMapper . Let's create a **school management application.**

What to do for using XJDataMapper?

Let us understand it by taking an example. Say we have a database with name "dbtrial". So what we need to change in XConfiguration class is:

```
public class XConfiguration {

   String dbName="dbtrial";  //set your database name here
   String userName="root";  //set your database user name here
   String password="";  //set your database password here
   String portNo="3306";  //set port number on which the application is running
   static Connection con;
//that's all you need to do. Don't change the code written below.
   public XConfiguration() throws ClassNotFoundException, SQLException {

      Class.forName("com.mysql.jdbc.Driver");
      if(con == null)
         con=DriverManager.getConnection("jdbc:mysql://localhost:"+portNo+"/"+dbName
         ,userName,password);
   }
}
```

Let's have a quick review of database tables that will be used in the application.

Below given are the models of our application.

```
public class class_detail extends XJDataMapper{

   public class_detail () throws ClassNotFoundException, SQLException, InstantiationException,
IllegalAccessException
   {
      super();
      table = "xj_class_detail";
      selfClass = "class_detail";
      hasOne = "{'classrecord'
->class_session,id,class_id,xj_class_session,xj_class_detail,class_detail,classinfo)}";

      hasMany = "{'sessions'
->class_session,id,class_id,xj_class_session,xj_class_detail,class_detail,classes}";
   }
}
```

```
public class class_session extends XJDataMapper{

   public class_session() throws ClassNotFoundException, SQLException, InstantiationException {

       super();
       table = "xj_class_session";
       selfClass = "class_session";
       hasOne="{'classinfo'
->class_detail,class_id,id,xj_class_detail,xj_class_session,class_session,classrecord)}"
+"{'sessioninfo’
->session,session_id,id,xj_session,xj_class_session,class_session,sessionrecord)}";

       hasMany ="{'students'->student,id,class_session_id,xj_student,xj_class_session,class_session,class}";
   }
}
```

```
public class fees extends XJDataMapper{

   public fees() throws ClassNotFoundException, SQLException {

       super();
       table = "xj_fees";
       selfClass = "fees";
       hasOne = "{'feesmaster'->fees,feeshead_id,id,xj_fees,xj_feeshead,fees,fees}"
             + "{'feesrecord'->feesapplicable,id,fees_id,xj_feesapplicable,xj_fees,fees,feeinfo}";

       hasMany = "{'students'->student,student_id,id,xj_student,xj_feesapplicable,feesapplicable,fees}";
   }
}
```

```
public class feesapplicable extends XJDataMapper{

   public feesapplicable() throws ClassNotFoundException, SQLException {

       super();
       table = "xj_feesapplicable";
       selfClass = "feesapplicable";
       hasOne = "{'feeinfo'->fees,fees_id,id,xj_fees,xj_feesapplicable,feesapplicable,feesrecord}"
             + "{'studentinfo'
->student,student_id,id,xj_student,xj_feesapplicable,feesapplicable,studentrecord}";
   }
}
```

```
public class feeshead extends XJDataMapper{

   public feeshead() throws ClassNotFoundException, SQLException {

       super();
       table = "xj_feeshead";
       selfClass = "feeshead";
       hasMany = "{'fees'->fees,id,feeshead_id,xj_fees,xj_feeshead,feeshead,feesmaster}";
   }
 }
```

```
public class rollno extends XJDataMapper{

   public rollno() throws ClassNotFoundException, SQLException {

       super();
       table = "xj_student_rollno";
       selfClass = "rollno";
       hasOne = "{'student'->student,student_id,id,xj_student,xj_student_rollno,rollno,rollno}";
   }
 }
```

XJDataMapper
Java's ORM

```
public class scholar extends XJDataMapper{

   public scholar() throws ClassNotFoundException, SQLException {
       super();
       table = "xj_scholar";
       selfClass = "scholar";
       hasMany = "{'students'->student,id,scholar_id,xj_student,xj_scholar,scholar,scholar}"
              + "{'sessions'->student,id,scholar_id,xj_student,xj_session,scholar,scholars}";
       }
}
```

```
public class session extends XJDataMapper{

   public session() throws ClassNotFoundException, SQLException {
       super();
       table = "xj_session";
       selfClass = "session";
       hasOne = "{'sessionrecord'
->class_session,id,session_id,xj_class_session,xj_session,class_detail,sessioninfo)}";

       hasMany = "{'classes'->class_session,id,session_id,xj_class_session,xj_session,session,sessions}"
              + "{'students'->student,id,session_id,xj_student,xj_session,session,session}"
              + "{'scholars'->student,id,session_id,xj_session,xj_student,session,sessions}";
      }
}
```

```
public class student  extends XJDataMapper{

   public student() throws ClassNotFoundException, SQLException {

       super();
       table = "xj_student";
       selfClass = "student";
       hasOne = "{'class'->class_session,class_session_id,id,xj_class_session,xj_student,student,students}"
             + "{'scholar'->scholar,scholar_id,id,xj_scholar,xj_student,student,students}"
             + "{'rollno'->rollno,id,student_id,xj_student_rollno,xj_student,student,student}"
             + "{'session'->session,session_id,id,xj_session,xj_student,student,students}"
             + "{'studentrecord'
->feesapplicable,id,student_id,xj_feesapplicable,xj_student,student,studentinfo}";

       hasMany = "{'fees'->feesapplicable,id,student_id,xj_feesapplicable,xj_student,student,students}";
    }
}
```

Now I think models will be clearly understood by; Now try to solve some of the queries given below;

Query-----> To retrieve the session names of a class.

```
 XJData class_session = new class_session();
 class_session.get_Where("id","=","1");
 class_session.setRelation("class_session->sessioninfo").get();
out.println("<tr><td>"+class_session.getSingle("class_session.class_id")+"</td><td>"+
class_session.getSingle("class_session.session_id")+"</td><td>"+class_session.getSingle("session.name")
+"</td></tr>");
```

Query-----> To retrieve the class names of a session.

```
XJData class_session = new class_session();
class_session.get_Where("id","=","1");
class_session.setRelation("class_session->classinfo").get();
out.println("<tr><td>"+class_session.getSingle("class_session.session_id")+"</td><td>"+
class_session.getSingle("class_session.class_id")+"</td><td>"+class_session.getSingle("class_detail.name
")+"</td></tr>");
```

Query-----> To retrieve the students of a class.

```
XJData class_session = new class_session();
class_session.get_Where("id","=","1");
class_session.setRelation("class_session->students").get();
for(XJData result : class_session.data){

   result.setRelation("student->scholar").get();
out.println("<tr><td>"+result.getSingle("class_session.session_id")+"</td><td>"+result.getSingle("class_s
ession.class_id")+"</td><td>"+result.getSingle("student.id")+"</td><td>"+result.getSingle("scholar.name"
)+"</td></tr>");

}
```

Query-----> To retrieve roll no of student.

```
XJData student = new student();
student.get_By("id","1");
student.setRelation("student->rollno").get();
student.setRelation("student->scholar").get();
out.println("<tr><td>"+student.getSingle("student.id")+"</td><td>"+student.getSingle("scholar.name")+"<
/td><td>"+student.getSingle("rollno.rollno")+"</td></tr>");
```

Query-----> To retrieve class of student.

```
XJData student = new student();
student.where("id","=","1").get();
for(XJData result : student.data){

result.setRelation("student->class").get();
result.setRelation("class_session->classinfo").get();
result.setRelation("class_session->sessioninfo").get();

out.println("<tr><td>"+result.getSingle("student.id")+"</td><td>"+result.getSingle("class_detail.name")+"
</td><td>"+result.getSingle("session.name")+"</td></tr>");

}
```

Query-----> To retrieve students of scholar.

```
XJData scholar = new scholar();
scholar.get();
for(XJData result1 : scholar.data){

result1.setRelation("scholar->students").get();
for(XJData result2 : result1.data){

out.println("<tr><td>"+result2.getSingle("scholar.id")+"</td><td>"+result2.getSingle("scholar.name")+"</
td><td>"+result2.getSingle("student.id")+"</td></tr>");
            }
        }
```

There can be endless examples once we start working with XJDataMapper.

Object-relational mapping

Object-relational mapping' (**ORM**, **O/RM**, and **O/R mapping**) in computer software is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools (as we have done).

Overview

Data management tasks in object-oriented (OO) programming are typically implemented by manipulating objects that are almost always non-scalar values. For example, consider an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person object" with attributes/fields to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "Phone Number objects" and so on. The address book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

Downloading XJDataMapper

Great !!!!!! You have gone through this tutorial.

If you have not read the tutorial then I recommend you to read the tutorial first and then proceed. This is because you should be able to think why are we actually using XJDataMapper and how it is helpful.

Well you are not going to get XJDataMapper so easily. You have to buy a license for getting it. License is available from our official site i.e. http://www.xavoc.com .

What happened? Confused? Are you thinking whether XJDataMapper is worthy or not?

Ok !!!!!!!!

Don't think too much of it now. Let's read the whole tutorial first. And I'm sure at the end of this tutorial you won't worry about license, when you will understand the need and benefits of XJDataMapper.

Let's begin the journey.