# Mapping Tutorial

**by Brian McCallister**

## Table of contents

# 1. What is the Object-Relational Mapping Metadata?

The O/R mapping metadata is the specific configuration information that specifies how to map classes to relational tables. In OJB this is primarily accomplished through an xml document, the `repository.xml` file, which contains all of the initial mapping information.

## 1.1. The Product Class

This tutorial looks at mapping a simple class with no relations:

```
package org.apache.ojb.tutorials;

public class Product
{
    /** product name */
    private String name;

    /** price per item */
    private Double price;

    /** stock of currently available items */
    private int stock;

    ...
}
```

This class has three fields, `price, stock,` and `name`, that need to be mapped to the database. Additionally, we will introduce one artificial field used by the database that has no real meaning to the class, an artificial key primary id:

```
/** Artificial primary-key */
private Integer id;
```

Including the primary-key attribute in the class definition is mandatory, but under certain conditions anonymous keys can also be used to keep this database artifact hidden in the database. However, as access to an artifical unique identifier for a particular object instance can be useful, particularly in web-based applications, this tutorial will expose it

## 1.2. The Database

OJB is very flexible in terms of how it can map classes to database tables, however the simplest technique for mapping a single class to a relational database is to map the class to a single table, and each attribute on the class to a single column. Each row will then represent a unique instance of that class.

The DDL for such a table, for the `Product` class might look like:

```
CREATE TABLE Product
(
    id INTEGER PRIMARY KEY,
    name VARCHAR(100),
    price DOUBLE,
    stock INTEGER
)
```

The individual field names in the database and class definition match here, but this is no requirement. They may vary independently of each other as the metadata will specify what maps to what.

## 1.3. The Metadata

The `repository.xml` document is split into several physical documents. The `repository_user.xml` xml file is used to contain user-defined mappings. OJB uses the other ones for managing other metadata, such as database information.

In general each class will be defined within a `class-descriptor` element with `field-descriptoy` child elements for each field. In addition the mapping of references and collections is described in the basic technique section. This tutorial sticks to mapping a single, simplistic, class.

The complete mapping for the `Product` class is as follows:

```
<class-descriptor
       class="org.apache.ojb.tutorials.Product"
       table="Product"
>
    <field-descriptor
       name="id"
       column="id"
       primarykey="true"
       autoincrement="true"
    />
    <field-descriptor
       name="name"
       column="name"
    />
    <field-descriptor
       name="price"
       column="price"
    />
    <field-descriptor
       name="stock"
       column="stock"
    />
</class-descriptor>
```

Examine the `class-descriptor` element. It has two attributes:

- class - This attribute is used to specify the fully-qualified Java class name for this mapping.
- table - This attribute specifies which table is used to store instances of this class.

Other information can be specified here, such as proxies and custom row-readers as specified in the repository.xml documentation.

Examine now the first `field-descriptor` element. This is used to describe the `id` field of the `Product` class. Two required attributes are specified:

- name - This specifies the name of the instance variable in the Java class.
- column - This specifies the column in the table specified for this class used to store the value.

In addition to those required attributes, notice that the first element specifies two optional attributes:

- primary-key - This attribute specifies that this field is the primary key for this class.
- autoincrement - The `autoincrement` attribute specifies that the value will be automatically assigned by OJB sequence manager. This might use a database supplied sequence, or, by default, an OJB generated value.

## 1.4. Using the XDoclet module

OJB provides an XDoclet module to make generating the repository descriptor and the corresponding table schema easier. An XDoclet module basically processes custom JavaDoc tags in the source code, and generates files from them. In the case of OJB, two types of files can be generated: the repository descriptor (`repository_user.xml`) and a Torque schema which can

be used to create the tables in the database. This provides one important benefit: the descriptor and the database schema are much more likely in sync with the code thus avoiding errors that are usually hard to find. Furthermore, the XDoclet module contains some checks that find common mapping errors.

In the above example, the source code for Product class with JavaDoc tags would look like:

```
package org.apache.ojb.tutorials;

/**
 * @ojb.class
 */
public class Product
{
    /**
     * Artificial primary-key
     *
     * @ojb.field primarykey="true"
     *            autoincrement="ojb"
     */
    private Integer id;

    /**
     * product name
     *
     * @ojb.field length="100"
     */
    private String name;

    /**
     * price per item
     *
     * @ojb.field
     */
    private Double price;

    /**
     * stock of currently available items
     *
     * @ojb.field
     */
    private int stock;
}
```

As you can see, much of the stuff that is present in the descriptor (and the DDL) is generated automatically by the XDoclet module, e.g. the table/column names and the jdbc-types. Of course, you can also specify them in the JavaDoc tags, e.g. if they differ from the java names.

For details on OJB's JavaDoc tags and how to generate and use the mapping files please see the OJB XDoclet Module documentation.

## 2. Advanced Topics

### 2.1. Relations

As most object models have relationships between objects, mapping specific types of relationships (1:1, 1:Many, Many:Many) is important in mapping objects into a relational database. The basic technique tutorial discusses this in great detail.

It is important to note that this metadata mapping can be modified at runtime through the `org.apache.ojb.metadata.MetadataManager` class.

### 2.2. Inheritence

OJB can map inheritence hierarchies using a variety of techniques discussed in the Extents and Polymorphism section of the Advanced O/R Documentation

## 2.3. Anonymous Keys

This tutorial uses explicit keys mapped into the Java class. It is also possible to keep artificial keys completely hidden within the database. The Anonymous Keys HOWTO explains how this is accomplished.

## 2.4. Large Projects

Projects with small numbers of persistent classes can be mapped by hand, however, many projects can have hundreds, or even thousands, of distinct classes which must be mapped. In these circumstances managing the class-database mapping by hand is not viable. The How To Build Mappings HOWTO explores different tools which can be used for managing large-scale mapping.

## 2.5. Custom JDBC Mapping

OJB maps Java types to JDBC types according to the JDBC Types table. You can, however, define custom JDBC -> Java type mappings via custom field conversions.