XDoclet OJB module documentation

by Thomas Dudziak

Table of contents

1. Acquiring and building

The XDoclet OJB module is part of OJB source. As such, the source of the module is part of the OJB source tree and can be found in directory src/xdoclet. Likewise, binary versions of the module and the required libraries (xjavadoc, xdoclet) are to be found in the lib folder.

In order to build the XDoclet OJB module from source, you'll need a source distribution of XDoclet version 1.2, either a source distribution from the sourceforge download site or a CVS checkout/drop. See the XDoclet website at http://xdoclet.sourceforge.net/install.html for details.

1.1. Building with a XDoclet source distribution

Unpack the source distribution of XDoclet which is contained in a file `xdoclet-src-<version>.<archive-format>` somewhere. If you unpacked it side-by-side of OJB, you'll get a directory layout similar to:

```
\xdoclet-1.2
    \config
    \core
    \lib
    ...
\db-ojb
    \bin
    \contrib
    ...
```

The XDoclet OJB module is then build using the `build-xdoclet-module.xml` ant script:

```
ant -Dxdoclet.src.dir=../xdoclet-1.2 -f build-xdoclet-module.xml
```

The build process will take some time, and after successful compilation the three jars `xjavadoc-<version>.jar`, `xdoclet-<version>.jar`, and `xdoclet-ojb-module-<version>.jar` are copied to the library directory of OJB.

1.2. Building with a XDoclet CVS checkout

When checking out from CVS (the `xdoclet-all` target), you'll get a directory like:

```
\xdoclet-all
    \xdoclet
        \config
        \core
        ...
    \xdocletgui
```

```
    \xjavadoc
\db-ojb
    \bin
    \contrib
    ...
```

Building is XDoclet OJB module is performed by calling:

```
ant -Dxdoclet.src.dir=../xdoclet-all/xdoclet -f build-xdoclet-module.xml
```

Since this is the default structure assumed by the build script, this can be shortend to:

```
ant -f build-xdoclet-module.xml
```

1.3. Other build options

The build script for the XDoclet OJB module uses the OJB build properties so the following line added to the `build.properties` file in the OJB root directory allows to omit the `-Dxdoclet.src.dir=<xdoclet src dir>` commandline option:

```
xdoclet.src.dir=<xdoclet src dir>
```

2. Usage

Using the XDoclet OJB module is rather easy. Put the module jar along with the xdoclet and xjavadc jars in a place where ant will find it, and then invoke it in your build file like:

```
<target name="repository-files">
    <taskdef name="ojbdoclet"
             classname="xdoclet.modules.ojb.OjbDocletTask"
             classpathref="build-classpath">
    <ojbdoclet destdir="./build">
        <fileset dir="./src"/>
        <ojbrepository destinationFile="repository_user.xml"/>
        <torqueschema databaseName="test"
destinationFile="project-schema.xml"/>
    </ojbdoclet>
</target>
```

The XDoclet OJB module has two sub tasks, `ojbrepository` and `torqueschema`, which generate the OJB repository part containing the user descriptors and the torque table schema, respectively. Please note that the XDoclet OJB module (like all xdoclet tasks) expects the directory structure of its input java source files to match their package structure. In this regard it is similar to the `javac` ant task.
Due to a bug in XDoclet, you should not call the `ojbdoclet` task more than once in the same `taskdef` scope. So, each `ojbdoclet` call should be in its own target with a leading `taskdef`.

The main `ojbdoclet` task has two attributes:

destdir
The destination directory where generated files will be placed.
checks : none | basic | strict (default)
The amount of the checks performed. Per default, `strict` checks are performed which means that for instance classes specified in an attribute (e.g. `collection-class`, `row-reader` etc.) are loaded from the classpath and checked. So in this mode it is necessary to have OJB as well as the processed classes on the classpath (using the `classpathref` attribute of the `taskdef` ant task above). If this is for some reason not possible, then use `basic` which performs most of the checks but does not load classes from the classpath. `none` does not perform any checks so use it with care and only if really necessary (in this case it would be helpful if you would post the problem to the ojb-user mailing list).

The `ojbrepository` subtask has the following attributes:

destinationFile
Specifies the output file. The default is `repository_user.xml`.
verbose : true | false (default)
Whether the task should output some information about its progress.

The `torqueschema` subtask has these attributes:

databaseName
This attribute gives the name of the database for torque (required).
destinationFile
The output file, default is `project-schema.xml`.
dtdUrl
Allows to specify the url of the torque dtd. This is necessary e.g. for XML parsers that have problems with the default dtd url (http://jakarta.apache.org/turbine/dtd/database.dtd), or when using a newer version of torque.
generateForeignkeys : true (default) | false
Whether foreignkey tags are generated in the torque database schema.
verbose : true | false (default)
Whether the task outputs some progress information.

The `classpathref` attribute in the taskdef can be used to define the classpath for xdoclet (containing the xdoclet and ojb module jars), e.g. via:

```
<path id="build-classpath">
    <fileset dir="lib">
```

```
        <include name="**/*.jar"/>
    </fileset>
</path>
```

Using the generated torque schema is a bit more tricky. The easiest way is to use the `build-torque.xml` script which is part of OJB. Include the lib subdirectory of the OJB distribution which also includes torque (e.g. in `build-classpath` as shown above). You will also want to use your OJB settings (if you're using the ojb-blank (../../docu/getting-started.html) project, then only `build.properties`), so include them at the beginning of the build script if they are not already there:

```
<property file="build.properties"/>
```

Now you can create the database with ant calls similar to these:

```
<target name="init-db" depends="repository-files">
    <!-- Torque's build file -->
    <property name="torque.buildFile"
              value="build-torque.xml"/>

    <!-- The name of the database which we're taking from the profile -->
    <property name="torque.project"
              value="${databaseName}"/>

    <!-- Where the schemas (your project and, if required, ojb's internal
tables) are -->
    <property name="torque.schema.dir"
              value="src/schema"/>

    <!-- Build directory of Torque -->
    <property name="torque.output.dir"
              value="build"/>

    <!-- Torque will put the generated sql here -->
    <property name="torque.sql.dir"
              value="${torque.output.dir}"/>

    <!-- Torque shall use the classpath (to find the jdbc driver etc.) -->
    <property name="torque.useClasspath"
              value="true"/>

    <!-- Which jdbc driver to use (again from the profile) -->
    <property name="torque.database.driver"
              value="${jdbcRuntimeDriver}"/>

    <!-- The url used to build the database; note that this may be
different
         from the url to access the database (e.g. for MySQL) -->
    <property name="torque.database.buildUrl"
              value="${urlProtocol}:${urlSubprotocol}:${urlDbalias}"/>
```

```
    <!-- Now we're generating the database sql -->
    <ant dir="."
         antfile="${torque.buildFile}"
         target="sql">
    </ant>
    <!-- Next we create the database -->
    <ant dir="."
         antfile="${torque.buildFile}"
         target="create-db">
    </ant>
    <!-- And the tables -->
    <ant dir="."
         antfile="${torque.buildFile}"
         target="insert-sql">
    </ant>
</target>
```

As you can see, the major problem of using Torque is to correctly setup Torque's build properties.

One important thing to note here is that the latter two calls modify the database and in the process remove any existing data, so use them with care. Similar to the above targets, you can use the additional targets `datadump` for storing the data currently in the database in an XML file, and `datasql` for inserting the data from an XML file into the database.
Also, these steps are only valid for the torque that is delivered with OJB, but probably not for newer or older versions.

3. Tag reference

Interfaces and Classes
ojb.class
ojb.extent-class
ojb.modify-inherited
ojb.object-cache
ojb.index
ojb.delete-procedure
ojb.insert-procedure
ojb.update-procedure
ojb.constant-argument
ojb.runtime-argument
Fields and Bean properties
ojb.field
References
ojb.reference
Collections

ojb.collection
Nested objects
ojb.nested
ojb.modify-nested

4. Interfaces and Classes

4.1. ojb.class

The **ojb.class** tag marks interfaces and classes that shall be present in the repository descriptor. This includes types that are used as reference targets or as collection elements, but for instance not abstract base classes not used elsewhere.

Attributes:

attributes
Optionally contains attributes of the class as a comma-separated list of name-value pairs.
determine-extents : true (default) | false
When set to `true`, then the XDoclet OJB module will automatically determine all extents (ojb-relevant sub types) of this type. If set to `false`, then extents need to be specified via the ojb.extent-class class tag (see below).
documentation
Optionally contains documentation on the class. If no `table-documentation` attribute is specified, then the value is also used for the table documentation in the database schema.
generate-repository-info : true (default) | false
Setting this to `false` prevents the generation of field/reference/collection descriptors in the repository XML file, and also automatically enforces `generate-table-info = false`.
Note that there is one case where the XDoclet module will still generate field descriptors. If the type is referenced by a reference or collection, then the corresponding foreign key fields (if 1:n collection) or primary keys (if reference or m:n collection) will be automatically included in the class descriptor, even if they are only defined in subtypes.
generate-table-info : true (default) | false
This attribute controls whether the type has an associated table. If set to `true`, a torque table descriptor will be created in the database schema. Otherwise, no table will be in the database schema for this type.
include-inherited : true (default) | false

Determines whether base type fields/references/collections with the appropriate tags (ojb.field, ojb.reference, ojb.collection) will be included in the descriptor and table definition of this class. Note that all base type fields/references/collections with an appropriate tag are included regardless of whether the base types have the **ojb.class** tag or not.

table

The name of the table used for this type. Is only used when table info is generated. If not specified, then the short name of the type is used.

table-documentation

Optionally contains documentation for the table in the database schema.

The following `class-descriptor` attributes are also supported in the **ojb.class** tag and will be written directly to the generated class descriptor (see the repository.dtd (repository.html#class-descriptor) for their meaning):

- **accept-locks**
- **factory-class**
- **factory-method**
- **initialization-method**
- **isolation-level**
- **proxy**
- **proxy-prefetching-limit**
- **refresh**
- **row-reader**

Example: (from the unit tests)

```
/**
 * @ojb.class generate-table-info="false"
 */
public abstract class AbstractArticle implements InterfaceArticle,
java.io.Serializable
...

/**
 * @ojb.class table="ARTICLE"
 *            proxy="dynamic"
 *            include-inherited="true"
 *            documentation="This is important documentation on the Article
class."
 *            table-documentation="And this is important documentation on
the ARTICLE table."
 *            attributes="color=blue,size=big"
 */
public class Article extends AbstractArticle implements InterfaceArticle,
java.io.Serializable
...
```

The `AbstractArticle` class will have an class descriptor in the repository file, but no field, reference or collection descriptors. The `Article` class however will not only have descriptors for its own fields/references/collections but also for those inherited from `AbstractArticle`. Also, its table definition in the torque file will be called "Artikel", not "Article". The resulting class descriptors look like:

```
<class-descriptor
    class="org.apache.ojb.broker.AbstractArticle"
>
    <extent-class class-ref="org.apache.ojb.broker.Article"/>
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.Article"
    proxy="dynamic"
    table="ARTICLE"
>
    <documentation>This is important documentation on the Article
class.</documentation>
    ...
    <attribute attribute-name="color" attribute-value="blue"/>
    <attribute attribute-name="size" attribute-value="big"/>
</class-descriptor>
...
```

4.2. ojb.extent-class

Use the **ojb.extent-class** to explicitly specify extents (direct persistent sub types) of the current type. The **class-ref** attribute contains the fully qualified name of the class. However, these tags are only evaluated if the **determine-extents** attribute of the ojb.class tag is set to `false`.

Attributes:

class-ref
The fully qualified name of the sub-class (required).

Example:

```
/**
 * @ojb.class determine-extents="false"
 *            generate-table-info="false"
 * @ojb.extent-class class-ref="org.apache.ojb.broker.CdArticle"
 */
public abstract class AbstractCdArticle extends Article implements
java.io.Serializable
...
```

which results in:

```
<class-descriptor
    class="org.apache.ojb.broker.AbstractCdArticle"
>
    <extent-class class-ref="org.apache.ojb.broker.CdArticle"/>
</class-descriptor>
```

4.3. ojb.modify-inherited

Allows to modify attributes of inherited fields/references/collections (normally, all attributes are used without modifications) for this and all sub types. One special case is the specification of an empty value which leads to a reset of the attribute value. As a result the default value is used for this attribute.

Attributes: All of ojb.field, ojb.reference, and ojb.collection (with the exception of the attributes related to indirection tables (**indirection-table**, **remote-foreignkey**, **indirection-table-primarykeys**, **indirection-table-documentation**, **foreignkey-documentation**, **remote-foreignkey-documentation**), and also:

ignore : true | false (default)
Specifies that this feature will be ignored in this type (but only in the current type, not in subtypes).

name
The name of the field/reference/collection to modify (required).

Example:

```
/**
 * @ojb.class table="Artikel"
 * @ojb.modify-inherited name="productGroup"
 *                       proxy="true"
 *                       auto-update="object"
 */
public class ArticleWithReferenceProxy extends Article
```

produces the class descriptor

```
<class-descriptor
    class="org.apache.ojb.broker.ArticleWithReferenceProxy"
    table="Artikel"
>
    ...
    <reference-descriptor
        name="productGroup"
        class-ref="org.apache.ojb.broker.ProductGroup"
        proxy="true"
        auto-update="object"
    >
        <documentation>this is the reference to an articles
productgroup</documentation>
```

```
        <attribute attribute-name="color" attribute-value="red"/>
        <attribute attribute-name="size" attribute-value="tiny"/>
        <foreignkey field-ref="productGroupId"/>
    </reference-descriptor>
</class-descriptor>
```

4.4. ojb.object-cache

The **ojb.object-cache** tag allows to specify the ObjectCache implementation that OJB uses for objects of this class (instead of the one defined in the jdbc connection descriptor or in the `ojb.properties` file). Classes specified with this tag have to implement the `org.apache.ojb.broker.cache.ObjectCache` interface. Note that object cache specifications are not inherited.

Attributes:

attributes
Optionally contains attributes of the object cache as a comma-separated list of name-value pairs.
class
The fully qualified name of the object cache class (required).
documentation
Optionally contains documentation on the object cache specification.

Example:

```
/**
 * @ojb.class
 * @ojb.object-cache
class="org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl"
 *                   documentation="Some important documentation"
 */
public class SomeClass implements Serializable
{
    ...
}
```

and the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    <object-cache
class="org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl">
        <documentation>Some important documentation</documentation>
    </object-cache>
    ...
</class-descriptor>
```

4.5. ojb.index

The **ojb.index** tag is used to define possibly unique indices for the class. An index consists of at least one field of the class (either locally defined or inherited, anonymous or explicit). There is an default index (without a name) that is made up by all fields that have the **indexed** attribute set to `true`. All other indices have to be defined via the **ojb.index** tag. In contrast to the **indexed** attribute, indices defined via the **ojb.index** tag are not inherited.

Attributes:

documentation
Optionally contains documentation on the index.
fields
The fields that make up the index separated by commas (required).
name
The name of the index (required). If there are multiple indices with the same name, then only the first one is used (all others are ignored).
unique : true | false (default)
Whether the index is unique or not.

Example:

```
/**
 * @ojb.class table="SITE"
 * @ojb.index name="NAME_UNIQUE"
 *            unique="true"
 *            fields="name"
 */
public class Site implements Serializable
{
    /**
     * @ojb.field indexed="true"
     */
    private Integer nr;
    /**
     * @ojb.field column="NAME"
     *            length="100"
     */
    private String name;
    ...
}
```

the class descriptor

```
<class-descriptor
    class="org.apache.ojb.odmg.shared.Site"
    table="SITE"
>
```

```
    <field-descriptor
        name="nr"
        column="nr"
        jdbc-type="INTEGER"
        indexed="true"
    >
    </field-descriptor>
    <field-descriptor
        name="name"
        column="NAME"
        jdbc-type="VARCHAR"
        length="100"
    >
    </field-descriptor>
    ...
    <index-descriptor
        name="NAME_UNIQUE"
        unique="true"
    >
        <index-column name="NAME"/>
    </index-descriptor>
</class-descriptor>
```

and the torque table schema

```
<table name="SITE">
    <column name="nr"
            javaName="nr"
            type="INTEGER"
    />
    <column name="NAME"
            javaName="name"
            type="VARCHAR"
            size="100"
    />
    ...
    <index>
        <index-column name="nr"/>
    </index>
    <unique name="NAME_UNIQUE">
        <unique-column name="NAME"/>
    </unique>
</table>
```

4.6. ojb.delete-procedure

Declares a database procedure that is used for deleting persistent objects.

Attributes:

arguments
A comma-separated list of the names of constant or runtime arguments specified in the same class.

attributes
Optionally contains attributes of the procedure as a comma-separated list of name-value pairs.
documentation
Optionally contains documentation on the procedure.
include-pk-only : true | false (default)
Whether all fields of the class that make up the primary key, shall be passed to the procedure. If set to `true` then the **arguments** value is ignored.
name
The name of the procedure (required).
return-field-ref
Identifies a field of the class that will receive the return value of the procedure. Use only if the procedure has a return value.

Example:

```
/**
 * @ojb.class
 * @ojb.delete-procedure name="DELETE_PROC"
 *                       arguments="arg1,arg2"
 *                       return-field-ref="attr2"
 *                       documentation="Some important documentation"
 * @ojb.constant-argument name="arg1"
 *                        value="0"
 * @ojb.runtime-argument name="arg2"
 *                       field-ref="attr1"
 */
public class SomeClass
{
    /** @ojb.field */
    private Integer attr1;
    /** @ojb.field */
    private String attr2;
    ...
}
```

leads to the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    <field-descriptor
        name="attr1"
        column="attr1"
        jdbc-type="INTEGER"
    >
    </field-descriptor>
    <field-descriptor
```

```
        name="attr2"
        column="attr2"
        jdbc-type="VARCHAR"
        length="254"
    >
    </field-descriptor>
    ...
    <delete-procedure
        name="DELETE_PROC"
        return-field-ref="attr2"
    >
        <documentation>Some important documentation</documentation>
        <constant-argument
            value="0"
        >
        </constant-argument>
        <runtime-argument
            field-ref="attr2"
        >
        </runtime-argument>
    </delete-procedure>
</class-descriptor>
```

4.7. ojb.insert-procedure

Identifies the database procedure that shall be used for inserting objects into the database.

Attributes:

arguments
Comma-separated list of names of constant or runtime arguments that are specified in the same class.
attributes
Contains optional attributes of the procedure in a comma-separated list of name-value pairs.
documentation
Contains optional documentation on the procedure.
include-all-fields : true | false (default)
Specifies whether all persistent fields of the class shall be passed to the procedure. If so, then the **arguments** value is ignored.
name
The name of the procedure (required).
return-field-ref
The persistent field that receives the return value of the procedure (should only be used if the procedure returns a value).

For an example see constant argument.

4.8. ojb.update-procedure

The database procedure that will be used for updating persistent objects in the database.

Attributes:

arguments
A comma-separated list of names of constant or runtime arguments in the same class.
attributes
The optional attributes of the procedure in a comma-separated list of name-value pairs.
documentation
Optional documentation on the procedure.
include-all-fields : true | false (default)
Whether all persistent fields of the class shall be passed to the procedure in which case the **arguments** value is ignored.
name
Name of the database procedure (required).
return-field-ref
A persistent field that will receive the return value of the procedure (only to be used if the procedure returns a value).

For an example see runtime argument.

4.9. ojb.constant-argument

A constant argument for a database procedure. These arguments are referenced by the procedure tags in the **arguments** attribute via their names.

Attributes:

attributes
Optionally contains attributes of the argument.
documentation
Optionally contains documentation on the argument.
value
The constant value.
name
The identifier of the argument to be used the **arguments** attribute of a procedure tag (required).

Example:

```
/**
 * @ojb.class
 * @ojb.insert-procedure name="INSERT_PROC"
 *                       arguments="arg"
 * @ojb.constant-argument name="arg"
 *                        value="Some value"
 *                        attributes="name=value"
 */
public class SomeClass
{
    ...
}
```

will result in the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    ...
    <insert-procedure
        name="INSERT_PROC"
    >
        <constant-argument
            value="Some value"
        >
            <attribute attribute-name="name" attribute-value="value"/>
        </constant-argument>
    </insert-procedure>
</class-descriptor>
```

4.10. ojb.runtime-argument

An argument for a database procedure that is backed by a persistent field. Similar to constant arguments the name is important for referencing by the procedure tags in the **arguments** attribute.

Attributes:

attributes
Contains optionally attributes of the argument.
documentation
Optionally contains documentation on the argument.
field-ref
The persistent field that delivers the value. If unspecified, then in the procedure call `null` will be used.
name
Identifier of the argument for using it in the **arguments** attribute of a procedure

tag (required).
return
If the field receives a value (?).

Example:

```
/**
 * @ojb.class
 * @ojb.update-procedure name="UPDATE_PROC"
 *                       arguments="arg"
 * @ojb.runtime-argument name="arg"
 *                       field-ref="attr"
 *                       documentation="Some documentation"
 */
public class SomeClass
{
    /** @ojb.field */
    private Integer attr;
    ...
}
```

will result in the class descriptor

```
<class-descriptor
    class="SomeClass"
    table="SomeClass"
>
    <field-descriptor
        name="attr"
        column="attr"
        jdbc-type="INTEGER"
    >
    </field-descriptor>
    ...
    <update-procedure
        name="UPDATE_PROC"
    >
        <runtime-argument
            value="attr"
        >
            <documentation>Some documentation</documentation>
        </runtime-argument>
    </update-procedure>
</class-descriptor>
```

5. Fields and Bean properties

5.1. ojb.field

Fields or accessor methods (i.e. get/is and set methods) for properties are marked with the **ojb.field** tag to denote a persistent field. When a method is marked, then the corresponding bean property is used for naming purposes (e.g. "value" for a method `getValue()`). The XDoclet OJB module ensures that a field is not present more than once, therefore it is safe to mark both fields and their accessors. However, in that case these **ojb.field** tags are required to have the same attributes.

Due to a bug in XDoclet, it is currently not possible to process `final` or `transient` fields.

Marked fields are used for descriptor generation in the same type (if it has an ojb.class tag) and all sub types with the ojb.class tag having the **include-inherited** attribute set to `true`.

It is also possible to use the **ojb.field** tag at the class level (i.e. in the JavaDoc comment of the class). In this case, the tag is used to define an *anonymous* field, e.g. a "field" that has no counterpart in the class but exists in the database. For anonymous fields, both the **name** and the **jdbc-type** attributes are required, and the **access** attribute is ignored (it defaults to the value `anonymous`). Beside these differences, anonymous fields are handled like other fields, e.g. they result in field-descriptor entries in the repository descriptor, and in columns in the table schema, and they are inherited and can be modified via the ojb.modify-inherited tag.

The XDoclet OJB module orders the fields in the repository descriptor and table schema according to the following rules:

1. Fields (anonymous and non-anonymous) from base types/nested objects and from the current file that have an id, sorted by the id value. If fields have the same id, then they are sorted following the rules for fields without an id.
2. Fields (anonymous and non-anonymous) from base types/nested objects and from the current file that have no id, in the order they appear starting with the farthest base type. Per class, the anonymous fields come first, followed by the non-anonymous fields.

Attributes:

access : readonly | readwrite (default)
Specifies the accessibility of the field. `readonly` marks fields that are not to modified. `readwrite` marks fields that may be read and written to. Anonymous fields do not have to be marked (i.e. `anonymous` value) as the position of the **ojb.field** tag in the class JavaDoc comment suffices.
attributes
Optionally contains attributes of the field as a comma-separated list of name-value pairs.
autoincrement : none (default) | ojb | database

Defines whether this field gets its value automatically. If `ojb` is specified, then OJB fills the value using a sequence manager. If the value is `database`, then the column is also defined as `autoIncrement` in the torque schema (i.e. the database fills the field), and in the repository descriptor, the field is marked as `access='readonly'` (if it isn't an anonymous field). The `database` value is intended to be used with the `org.apache.ojb.broker.util.sequence.SequenceManagerNativeImpl` sequence manager. For details, see the Sequence Manager documentation (sequencemanager.html#nativeSequenceManager) .
The default value is `none` which means that the field is not automatically filled.

column
The name of the database column for this field. If not given, then the name of the attribute is used.

column-documentation
Optionally contains documentation on the column in the database schema.

conversion
The name of the class to be used for conversion between the java type of the field (e.g. `java.lang.Boolean` or `java.util.Date`) and the java type for the JDBC type (e.g. `java.lang.Integer` or `java.sql.Date`). Conversion classes must implement the `org.apache.ojb.broker.accesslayer.conversions.FieldConversion` interface. If no explicit JDBC type is defined and the java type has no defined conversion (see below), then per default the `org.apache.ojb.broker.accesslayer.conversions.Object2ByteArrFieldConve` conversion class is used.
Default conversion is also used for the following java types when no jdbc type is given (default type is used instead), and no conversion is specified:

Java type	Default conversion
`org.apache.ojb.broker.util.GUID`	`org.apache.ojb.broker.accesslayer.conversions.GUI`

documentation
Optionally contains documentation on the field. If no `column-documentation` attribute value is specified, then this value is also used for the documentation of the column in the database schema.

id
An integer specifying the position in the repository descriptor and table schema. For the placement rules see above.

jdbc-type : BIT | TINYINT | SMALLINT | INTEGER | BIGINT | DOUBLE | FLOAT | REAL | NUMERIC | DECIMAL | CHAR | VARCHAR | LONGVARCHAR | DATE | TIME | TIMESTAMP | BINARY | VARBINARY |

LONGVARBINARY | CLOB | BLOB | STRUCT | ARRAY | REF | BOOLEAN | DATALINK
The JDBC type for the column. The XDoclet OJB module will automatically determine a jdbc type for the field if none is specified. This means that for anonymous fields, the **jdbc-type** attribute is required. The automatic mapping performed by the XDoclet OJB module from java type to jdbc type is as follows:

Java type	JDBC type
`boolean`	`BIT`
`byte`	`TINYINT`
`short`	`SMALLINT`
`int`	`INTEGER`
`long`	`BIGINT`
`char`	`CHAR`
`float`	`REAL`
`double`	`FLOAT`
`java.lang.Boolean`	`BIT`
`java.lang.Byte`	`TINYINT`
`java.lang.Short`	`SMALLINT`
`java.lang.Integer`	`INTEGER`
`java.lang.Long`	`BIGINT`
`java.lang.Character`	`CHAR`
`java.lang.Float`	`REAL`
`java.lang.Double`	`FLOAT`
`java.lang.String`	`VARCHAR`
`java.util.Date`	`DATE`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
`java.sql.Blob`	`BLOB`

`java.sql.Clob`	`CLOB`
`java.sql.Ref`	`REF`
`java.sql.Struct`	`STRUCT`
`java.math.BigDecimal`	`DECIMAL`
`org.apache.ojb.broker.util.GUID`	`VARCHAR`

For any other type (including array types) the default mapping is to `LONGVARBINARY` using the `Object2ByteArrFieldConversion` conversion (see **conversion** attribute above).

length
The length of the column which might be required by the jdbc type in some databases. This is the reason that for some jdbc types, the XDoclet OJB module imposes default lengths if no length is specified:

Jdbc type	Default length
`CHAR`	`1`
`VARCHAR`	`254`

name
The name of the field. This attribute is required for anonymous fields, otherwise it is ignored.

precision
scale
The precision and scale of the column if required by the jdbc type. They are usually used in combination with the `DECIMAL` and `NUMERIC` types, and then specifiy the number of digits before (**precision**) and after (**scale**) the comma (excluding the plus/minus sign). Due to restrictions in some databases (e.g. MySQL), the XDoclet OJB module imposes default values for some types if none are specified:

Jdbc type	Default values for precision, scale
`DECIMAL`	`20,0` (this corresponds to the range of `long` where the longest number is -9223372036854775808).
`NUMERIC`	`20,0`

For other types, if only the precision is specified, the scale defaults to 0. If only scale is specified, precision defaults to 1.

Other attributes supported in the **ojb.field** tag that have the same meaning as in the repository

descriptor (repository.html#field-descriptor) (and partly in the torque table schema) are:

- **default-fetch**
- **indexed**
- **locking**
- **nullable**
- **primarykey**
- **sequence-name**
- **update-lock**

Examples:

```
/**
 * @ojb.field column="Auslaufartikel"
 *            jdbc-type="INTEGER"
 *
conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
 *            column-documentation="Some documentation on the column"
 *            id="10"
 *            attributes="color=green,size=small"
 */
protected boolean isSelloutArticle;
```

will result in the following field descriptor:

```
<field-descriptor
    name="isSelloutArticle"
    column="Auslaufartikel"
    jdbc-type="INTEGER"
conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
>
    <attribute attribute-name="color" attribute-value="green"/>
    <attribute attribute-name="size" attribute-value="small"/>
</field-descriptor>
```

The column descriptor looks like:

```
<table name="Artikel">
    ...
    <column name="Auslaufartikel"
            javaName="isSelloutArticle"
            type="INTEGER"
            description="Some documentation on the column"
    />
    ...
</table>
```

An anonymous field is declared like this:

```
/**
 * @ojb.class table="TABLE_F"
```

```
 *             include-inherited="false"
 * @ojb.field name="eID"
 *             column="E_ID"
 *             jdbc-type="INTEGER"
 * @ojb.reference class-ref="org.apache.ojb.broker.E"
 *                auto-retrieve="true"
 *                auto-update="object"
 *                auto-delete="object"
 *                foreignkey="eID"
 */
public class F extends E implements Serializable
...
```

In this case an anonymous field is declared and also used as the foreignkey of an anonymous reference. The corresponding class descriptor looks like:

```
<class-descriptor
    class="org.apache.ojb.broker.F"
    table="TABLE_F"
>
    <field-descriptor
        name="eID"
        column="E_ID"
        jdbc-type="INTEGER"
        access="anonymous"
    >
    </field-descriptor>
    ...
    <reference-descriptor
        name="super"
        class-ref="org.apache.ojb.broker.E"
        auto-retrieve="true"
        auto-update="object"
        auto-delete="object"
    >
        <foreignkey field-ref="eID"/>
    </reference-descriptor>
</class-descriptor>
```

Here the anonymous field and reference (which implicitly refers to `super`) are used to establish the super-subtype relationship between `E` and `F` on the database level. For details on this see the advanced technique section (../../docu/guides/advanced-technique.html) .

6. References

6.1. ojb.reference

Similar to fields, references (java fields or accessor methods) are marked with the **ojb.reference** tag. We have a reference when the type of the java field is itself a persistent

class (has an ojb.class tag) and therefore the java field represents an association. This means that the referenced type of an association (or the one specified by the **class-ref** attribute, see below) is required to be present in the repository descriptor (it has the ojb.class tag).
Foreign keys of references are also declared in the torque table schema (see example below).
OJB currently requires that the referenced type has at least one field used to implement the reference, usually some id of an integer type.
A reference can be stated in the JavaDoc comment of the class (anonymous reference), but in this case it silently refer to `super` (see the example of ojb.field) which can be used to establish an inheritance relationship. Note that anonymous references are not inherited (in contrast to anonymous fields and normal references).

Attributes:

attributes
Optionally contains attributes of the reference as a comma-separated list of name-value pairs.
class-ref
Allows to explicitly specify the referenced type. Normally the XDoclet OJB module searches the type of the field and its sub types for the nearest type with the ojb.class tag. If the type is specified explicitly, then this type is used instead. For anonymous references, the **class-ref** has to specified as there is no field to determine the type from.
Note that the type is required to have the ojb.class tag.
database-foreignkey : true (default) | false
Specifies whether a database foreignkey shall be generated for the reference. Note that this attribute is only evaluated if the XDoclet module has determined that a database foreignkey could be generated. You cannot force the generation with this attribute, and the value of the attribute is not considered when checking if database foreignkeys can be generated in case the referencing class has subtypes (in which case database foreignkeys can only be generated if all subtypes map to the same table or don't map to a table or the inheritance is mapped via a super-reference).
documentation
Optionally contains documentation on the reference.
foreignkey
Contains one or more foreign key fields separated by commas (required). The foreign key fields are fields with the ojb.field tag in the same class as the reference, which implement the association, i.e. contains the values of the primarykeys of the referenced object.

Other supported attributes (see repository.dtd (repository.html#reference-descriptor) for their meaning) written directly to the repository descriptor file:

- **auto-delete**
- **auto-retrieve**
- **auto-update**
- **otm-dependent**
- **proxy**
- **proxy-prefetching-limit**
- **refresh**

Example:

```
public abstract class AbstractArticle implements InterfaceArticle,
java.io.Serializable
{
    protected InterfaceProductGroup productGroup;

    /**
     * @ojb.reference class-ref="org.apache.ojb.broker.ProductGroup"
     *                foreignkey="productGroupId"
     *                documentation="this is the reference to an articles
productgroup"
     *                attributes="color=red,size=tiny"
     */
    protected InterfaceProductGroup productGroup;
    /**
     * @ojb.field
     */
    protected int productGroupId;
    ...
}
```

Here the java type is `InterfaceProductGroup` although the repository reference uses the sub type `ProductGroup`. The generated reference descriptor looks like:

```
<field-descriptor
    name="productGroupId"
    column="Kategorie_Nr"
    jdbc-type="INTEGER"
>
</field-descriptor>
<reference-descriptor
    name="productGroup"
    class-ref="org.apache.ojb.broker.ProductGroup"
>
    <documentation>this is the reference to an articles
productgroup</documentation>
    <attribute attribute-name="color" attribute-value="red"/>
    <attribute attribute-name="size" attribute-value="tiny"/>
    <foreignkey field-ref="productGroupId"/>
</reference-descriptor>
```

In the torque table schema for the `Article` class, the foreign key for the product group is

explicitly declared:

```
<table name="Artikel">
    ...
    <column name="Kategorie_Nr"
            javaName="productGroupId"
            type="INTEGER"
    />
    ...
    <foreign-key foreignTable="Kategorien">
        <reference local="Kategorie_Nr" foreign="Kategorie_Nr"/>
    </foreign-key>
</table>
```

For an example of an anonymous reference, see the examples of ojb.field.

7. Collections

7.1. ojb.collection

Persistent collections which implement 1:n or m:n associations are denoted by the **ojb.collection** tag. If the collection is an array, then the XDoclet OJB module can determine the element type automatically (analogous to references). Otherwise the type must be specified using the **element-class-ref** attribute. m:n associations are also supported (collections on both sides) via the **indirection-table**, **foreignkey** and **remote-foreignkey** attributes.

Attributes:

attributes
Optionally contains attributes of the collection as a comma-separated list of name-value pairs.

collection-class
Specifies the class that implements the collection. This attribute is usually only required if the actual type of the collection object shall be different from the variable type, e.g. if an interface like `java.util.Collection` is used as the declared type.
Collections that use `java.util.Collection`, `java.util.List` or `java.util.Set` can be handled by OJB as-is so specifying **collection-class** is not necessary. For the types that do not, the XDoclet OJB module checks whether the declared collection field type implements the `org.apache.ojb.broker.ManageableCollection` interface, and if so, generates the **collection-class** attribute automatically. Otherwise, you have to specify it.

database-foreignkey : true (default) | false
Specifies whether a database foreignkey shall be generated for the collection. Note that this attribute is only evaluated if the XDoclet module has determined that a database foreignkey could be generated. You cannot force the generation with this attribute, and the value of the attribute is not considered when checking if database foreignkeys can be generated in the case of subtypes of the element type or the type having the collection (if m:n collection). For 1:n collections, database foreignkeys can only be generated if all subtypes of the element type map to the same table or don't map to a table or the inheritance is mapped via a super-reference. For m:n collections, the same applies to the class owning the collection.
documentation
Optionally contains documentation on the collection.
element-class-ref
Allows to explicitly specify the type of the collection elements. Note that the type is required to have the ojb.class tag.
foreignkey
Contains one or more foreign key field or columns separated by commas (required).
If the collection implements an 1:n association, then this attribute specifies the fields in the element type that implement the association on the element side, i.e. they refer to the primary keys of the class containing this collection. Note that these fields are required to have the ojb.field tag.
When the collection is one part of an m:n association (e.g. with an indirection table), this attribute specifies the columns in the indirection table that point to the type owning this collection. This attribute is required of both collections. If the other type has no explicit collection, use the **remote-foreignkey** attribute to specify the foreign keys for the other type.
foreignkey-documentation
Optionally contains documentation for the columns in the indirection table in the database schema that point to this class.
indirection-table
Gives the name of the indirection table used for m:n associations. The XDoclet OJB module will create an appropriate torque table schema. The specified foreign keys are taken from the **foreignkey** attribute in this class and the corresponding collection in the element class, or if the element class has no collection, from the **remote-foreignkey** attribute of this collection. The XDoclet OJB module associates the foreignkeys (in the order they are stated in the **foreignkey**/ **remote-foreignkey** attributes) to the ordered primarykey fields (for

the ordering rules see the ojb.field tag), and use ther jdbc type (and length setting if necessary) of these primarey keys for the columns.

indirection-table-documentation
Optionally contains documentation for the indirection table in the database schema.

indirection-table-primarykeys : true | false (default)
Specifies that the columns in the indirection table that point to this type, are primary keys of the table. If the element type has no corresponding collection, then this setting is also applied to the columns pointing to the element type.

orderby
Contains the fields used for sorting the collection and, optionally, the sorting order (either `ASC` or `DESC` for ascending or descending, respectively) as a comma-separated list of name-value pairs. For instance, `field1=DESC,field2,field3=ASC` specifies three fields after which to sort, the first one in descending order and the other two in ascending order (which is the default and can be omitted).

query-customizer
Specifies a query customizer for the collection. The type is required to implement `org.apache.ojb.broker.accesslayer.QueryCustomizer`.

query-customizer-attributes
Specifies attributes for the query customizer. This attribute is ignored if no query customizer is specified for this collection.

remote-foreignkey
Contains one or more foreign key columns (separated by commas) in the indirection table pointing to the elements. Note that this field should only be used if the other type does not have a collection itself which the XDoclet OJB module can use to retrieve the foreign keys. This attribute is ignored if used with 1:n collections (no indirection table specified).

remote-foreignkey-documentation
Optionally contains documentation for the columns in the indirection table in the database schema that point to the element type. This value can be used when the element type has no corresponding collection (i.e. remote-foreignkey is specified) or if the corresponding collection does not specify the `foreignkey-documentation` attribute.

The same attributes as for references are written directly to the repository descriptor file (see repository.dtd (repository.html#collection-descriptor)) :

- **auto-delete**
- **auto-retrieve**
- **auto-update**

- **otm-dependent**
- **proxy**
- **proxy-prefetching-limit**
- **refresh**

Examples:

```
/**
 * @ojb.collection element-class-ref="org.apache.ojb.broker.Article"
 *                 foreignkey="productGroupId"
 *                 auto-retrieve="true"
 *                 auto-update="link"
 *                 auto-delete="object"
 *                 orderby="productGroupId=DESC"
 *
query-customizer="org.apache.ojb.broker.accesslayer.QueryCustomizerDefaultImpl"
 *                 query-customizer-attributes="attr1=value1"
 */
private ArticleCollection allArticlesInGroup;
```

The corresponding collection descriptor is:

```
<collection-descriptor
    name="allArticlesInGroup"
    element-class-ref="org.apache.ojb.broker.Article"
    collection-class="org.apache.ojb.broker.ArticleCollection"
    auto-retrieve="true"
    auto-update="link"
    auto-delete="object"
>
    <orderby name="productGroupId" sort="DESC"/>
    <inverse-foreignkey field-ref="productGroupId"/>
    <query-customizer
class="org.apache.ojb.broker.accesslayer.QueryCustomizerDefaultImpl">
        <attribute attribute-name="attr1" attribute-value="value1"/>
    </query-customizer>
</collection-descriptor>
```

An m:n collection is defined using the **indirection-table** attribute:

```
/**
 * @ojb.class generate-table-info="false"
 */
public abstract class BaseContentImpl implements Content
{
    /**
     * @ojb.collection element-class-ref="org.apache.ojb.broker.Qualifier"
     *                 auto-retrieve="true"
     *                 auto-update="link"
     *                 auto-delete="none"
     *                 indirection-table="CONTENT_QUALIFIER"
     *                 foreignkey="CONTENT_ID"
```

```
     *                  remote-foreignkey="QUALIFIER_ID"
     */
    private List qualifiers;
    ...
}

/**
 * @ojb.class table="NEWS"
 */
public class News extends BaseContentImpl
{
    ...
}

/**
 * @ojb.class generate-table-info="false"
 */
public interface Qualifier extends Serializable
{
    ...
}
```

The `BaseContentImpl` has a m:n association to the `Qualifier` interface. for the `BaseContentImpl` class, this association is implemented via the `CONTENT_ID` column (specified by the **foreignkey**) in the indirection table `CONTENT_QUALIFIER`. Usually, both ends of an m:n association have a collection implementing the association, and for both ends the **foreignkey** specifies the indirection table column pointing to the class at this end. The `Qualifier` interface however does not contain a collection which could be used to determine the indirection table column that implements the association from its side. So, this column is also specified in the `BaseContentImpl` class using the **remote-foreignkey** attribute. The class descriptors are:

```
<class-descriptor
    class="org.apache.ojb.broker.BaseContentImpl"
>
    <extent-class class-ref="org.apache.ojb.broker.News"/>
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.News"
    table="NEWS"
>
    ...
    <collection-descriptor
        name="qualifiers"
        element-class-ref="org.apache.ojb.broker.Qualifier"
        indirection-table="CONTENT_QUALIFIER"
        auto-retrieve="true"
        auto-update="link"
        auto-delete="none"
    >
```

```
        <fk-pointing-to-this-class column="CONTENT_ID"/>
        <fk-pointing-to-element-class column="QUALIFIER_ID"/>
    </collection-descriptor>
</class-descriptor>

<class-descriptor
    class="org.apache.ojb.broker.Qualifier"
>
    <extent-class class-ref="org.apache.ojb.broker.BaseQualifierImpl"/>
</class-descriptor>
```

As can be seen, the collection definition is inherited in the `News` class and the two indirection table columns pointing to the ends of the m:n associaton are correctly specified.

8. Nested objects

8.1. ojb.nested

The features of a class can be included in another class by declaring a field of that type and using this tag. The XDoclet OJB module will then add every tagged feature (i.e. fields/bean properties with ojb.field, ojb.reference or ojb.collection tag, or even with **ojb.nested**) from the type of the field to the current class descriptor. It is not required that the field's type has the ojb.class tag, though.
All attributes of the features are copied (even **primarykey**) and modified if necessary (e.g. the **foreignkey** of a reference is adjusted accordingly). For changing an attribute use the ojb.modify-nested tag.

For an example of nesting, see the example of ojb.modify-nested.

8.2. ojb.modify-nested

Similar to ojb.modify-inherited, this tag allows to modify attributes of a nested feature.

Attributes: All of ojb.field, ojb.reference, and ojb.collection with the exception of the attributes related to indirection tables (**indirection-table**, **remote-foreignkey**, **indirection-table-primarykeys**, **indirection-table-documentation**, **foreignkey-documentation**, **remote-foreignkey-documentation**), and also:

ignore : true | false (default)
Specifies that this feature will not be nested.
name
The name of the field/reference/collection to modify (required). Use here the name of the feature in the nested type.

Example:

The two classes:

```
public class NestedObject implements java.io.Serializable
{
    /** @ojb.field primarykey="true" */
    protected int id;

    /** @ojb.field */
    protected boolean hasValue;

    /** @ojb.field */
    protected int containerId;

    /**
     * @ojb.reference foreignkey="containerId"
     */
    protected ContainerObject container;

    ...
}

/** @ojb.class */
public class ContainerObject implements java.io.Serializable
{
    /**
     * @ojb.field primarykey="true"
     *            autoincrement="ojb"
     *            id="1"
     */
    protected int id;

    /** @ojb.field id="2" */
    protected String name;

    /**
     * @ojb.nested
     * @ojb.modify-nested name="hasValue"
     *                    jdbc-type="INTEGER"
     *
conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
     *                    id="3"
     * @ojb.modify-nested name="id"
     *                    primarykey=""
     */
    protected NestedObject nestedObj;

    ...
}
```

result in the one class descriptor

```
<class-descriptor
    class="ContainerObject"
    table="ContainerObject"
>
    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
    />
    <field-descriptor
        name="name"
        column="name"
        jdbc-type="VARCHAR"
        length="24"
    />
    <field-descriptor
        name="nestedObj::hasValue"
        column="nestedObj_hasValue"
        jdbc-type="INTEGER"
conversion="org.apache.ojb.broker.accesslayer.conversions.Boolean2IntFieldConversion"
    />
    <field-descriptor
        name="nestedObj::id"
        column="nestedObj_id"
        jdbc-type="INTEGER"
    />
    <field-descriptor
        name="nestedObj::containerId"
        column="nestedObj_containerId"
        jdbc-type="INTEGER"
    />
    <reference-descriptor
        name="nestedObj::container"
        class-ref="ContainerObject"
    >
        <foreignkey field-ref="nestedObj::containerId"/>
    </reference-descriptor>
    ...
</class-descriptor>
```

and the table descriptor

```
<table name="ContainerObject">
    <column name="id"
            javaName="id"
            type="INTEGER"
            primaryKey="true"
            required="true"
    />
    <column name="name"
            javaName="name"
            type="VARCHAR"
```

```
            size="24"
    />
    <column name="nestedObj_hasValue"
            type="INTEGER"
    />
    <column name="nestedObj_id"
            type="INTEGER"
    />
    <column name="nestedObj_containerId"
            type="INTEGER"
    />
    <foreign-key foreignTable=\"ContainerObject\">\n"+
        <reference local=\"nestedObj_containerId\" foreign=\"id\"/>\n"+
    </foreign-key>\n"+
    ...
</table>
```

Note how one **ojb.modify-nested** tag changes the type of the nested `hasValue` field, add a `conversion` and specifies the position for it. The other modification tag removes the `primarykey` status of the nested `id` field.