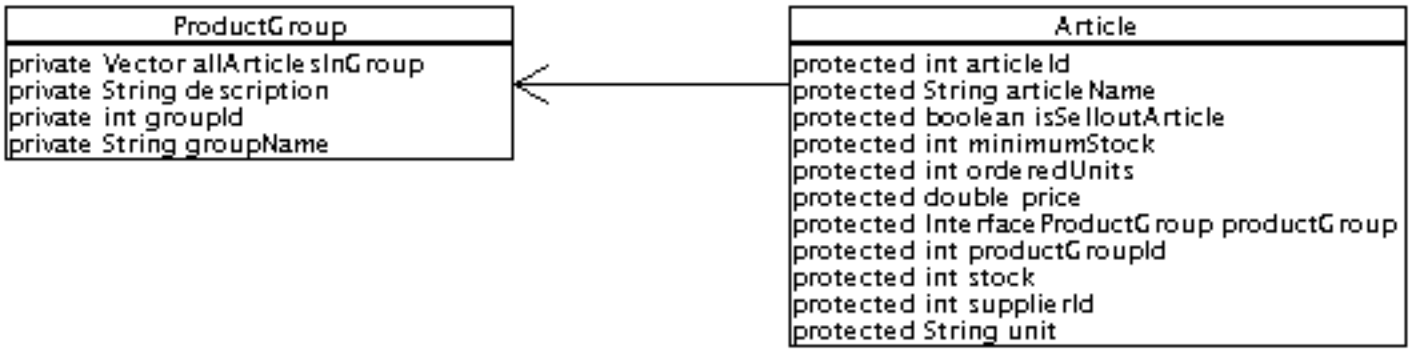
Basic Technique

by Thomas Mahler, Jakob Braeuchli, Armin Waibel

1. Mapping 1:1 associations

As a sample for a simple association we take the reference from an article to its productgroup.
This association is navigable only from the article to its productgroup. Both classes are modelled in the following class diagram. This diagram does not show methods, as only attributes are relevant for the O/R mapping process.



1:1 association

The association is implemented by the attribute `productGroup`. To automatically maintain this reference OJB relies on foreignkey attributes. The foreign key containing the `groupId` of the referenced `productgroup` is stored in the attribute `productGroupId`. To avoid FK attribute in persistent object class see section about anonymous keys.

This is the DDL of the underlying tables:

```
CREATE TABLE Artikel
(
    Artikel_Nr         INT NOT NULL PRIMARY KEY,
    Artikelname        VARCHAR(60),
    Lieferanten_Nr     INT,
    Kategorie_Nr       INT,
    Liefereinheit      VARCHAR(30),
    Einzelpreis        FLOAT,
    Lagerbestand       INT,
    BestellteEinheiten INT,
    MindestBestand     INT,
    Auslaufartikel     INT
)

CREATE TABLE Kategorien
(
    Kategorie_Nr       INT NOT NULL PRIMARY KEY,
    KategorieName      VARCHAR(20),
    Beschreibung       VARCHAR(60)
)
```

To declare the foreign key mechanics of this reference attribute we have to add a reference-descriptor to the class-descriptor of the Article class. This descriptor contains the following information:

- The attribute implementing the association (`name="productGroup"`) is productGroup.

Copyright © All rights reserved.

- The referenced object is of type (`class-ref="org.apache.ojb.broker.ProductGroup"`) `org.apache.ojb.broker.ProductGroup`.
- A reference-descriptor contains one or more foreignkey elements. These elements define foreign key attributes. The element

```
<foreignkey field-ref="productGroupId"/>
```

contains the name of the field-descriptor describing the foreignkey fields. The FieldDescriptor with the name "productGroupId" describes the foreignkey attribute productGroupId:

```
<field-descriptor
    name="productGroupId"
    column="Kategorie_Nr"
    jdbc-type="INTEGER"
/>
```

See the following extract from the repository.xml file containing the Article ClassDescriptor:

```
<!-- Definitions for org.apache.ojb.ojb.broker.Article -->
<class-descriptor
    class="org.apache.ojb.broker.Article"
    proxy="dynamic"
    table="Artikel"
>
    <extent-class class-ref="org.apache.ojb.broker.BookArticle" />
    <extent-class class-ref="org.apache.ojb.broker.CdArticle" />
    <field-descriptor
       name="articleId"
       column="Artikel_Nr"
       jdbc-type="INTEGER"
       primarykey="true"
       autoincrement="true"
    />
    <field-descriptor
       name="articleName"
       column="Artikelname"
       jdbc-type="VARCHAR"
    />
    <field-descriptor
       name="supplierId"
       column="Lieferanten_Nr"
       jdbc-type="INTEGER"
    />
    <field-descriptor
       name="productGroupId"
       column="Kategorie_Nr"
       jdbc-type="INTEGER"
    />
      ...
    <reference-descriptor
       name="productGroup"
       class-ref="org.apache.ojb.broker.ProductGroup"
    >
       <foreignkey field-ref="productGroupId"/>
    </reference-descriptor>
</class-descriptor>
```

This example provides unidirectional navigation only. Bidirectional navigation may be added by including a reference from a ProductGroup to a single Article (for example, a sample article for the productgroup). To accomplish this we need to perform the following steps:

1. Add a private Article attribute named `sampleArticle` to the class `ProductGroup`.
2. Add a private int attribute named `sampleArticleId` to the *ProductGroup* class representing the foreign key. To avoid FK attribute in persistent object class see section about anonymous keys.
3. Add a column `SAMPLE_ARTICLE_ID INT` to the table `Kategorien`.
4. Add a FieldDescriptor for the foreignkey attribute to the ClassDescriptor of the Class ProductGroup:

Copyright © All rights reserved.

```
<field-descriptor
  name="sampleArticleId"
  column="SAMPLE_ARTICLE_ID"
  jdbc-type="INTEGER"
/>
```

1. Add a ReferenceDescriptor to the ClassDescriptor of the Class ProductGroup:

```
<reference-descriptor
    name="sampleArticle"
    class-ref="org.apache.ojb.broker.Article"
>
    <foreignkey field-ref="sampleArticleId""/>
</reference-descriptor>
```

> **Note:**
> When using primitive primary key fields, please pay attention on how OJB manage `null` for primitive PK/FK

1.1. 1:1 auto-xxx setting

General info about the `auto-xxx` and `proxy` attributes can be found here

auto-retrieve
See here

auto-update

- **none** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced object will NOT be updated by default.The reference will not be *inserted* or *updated*, the link to the reference (foreign key value to the reference) on the main object will not be assigned automatically. The user has to link the main object and to store the reference *before* the main object to avoid violation of referential integrity.
- **link** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the FK assignment on the main object was done automatic. OJB reads the PK from the referenced object and sets these values as FK in main object. But the referenced object remains untouched. If no referenced object is found, the FK will be nullified. (On insert it is allowed to set the FK without populating the referenced object)
- **object** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced object will be stored first, then OJB does the same as in *link*.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

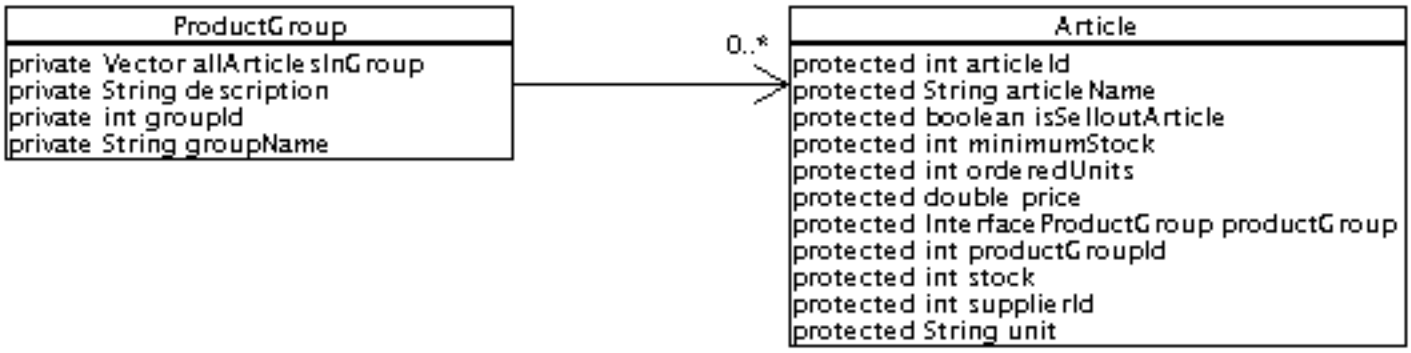
auto-delete

- **none** On deleting an object with `PersistenceBroker.delete(...)` the referenced object will NOT be touched.
- **link** Is equivalent to *none*.
- **object** On deleting an object with `PersistenceBroker.delete(...)` the referenced object will be deleted too.
- **false** Is equivalent to *none*.
- **true** Is equivalent to *object*.

2. Mapping 1:n associations

We will take a different perspective from the previous exmaple for a 1:n association. We will associate multiple Articles with a single ProductGroup. This association is navigable only from the ProductGroup to its Article instances. Both classes are modelled in the following class diagram. This diagram does not show methods, as only attributes are relevant for the O/R mapping process.

Copyright © All rights reserved.



1:n association

The association is implemented by the `Vector` attribute `allArticlesInGroup` on the ProductGroup class. As in the previous example, the Article class contains a foreignkey attribute named productGroupId that identifies an Article's ProductGroup. The Database table are the same as above.

To declare the foreign key mechanics of this collection attribute we must add a CollectionDescriptor to the ClassDescriptor of the ProductGoup class. This descriptor contains the following information:

1. The attribute implementing the association (`name="allArticlesInGroup"`)
2. The class of the elements in the collection (`element-class-ref="org.apache.ojb.broker.Article"`)
3. The name of field-descriptor of the element class used as foreign key attributes are defined in inverse-foreignkey elements:
   ```
   <inverse-foreignkey field-ref="productGroupId"/>
   ```
   This is again pointing to the field-descriptor for the attribute `productGoupId` in class Article.
4. optional attributes to define the sort order of the retrieved collection: `orderby="articleId" sort="DESC"`.

See the following extract from the repository.xml file containing the ProductGoup ClassDescriptor:

```
<!-- Definitions for org.apache.ojb.broker.ProductGroup -->
 <class-descriptor
    class="org.apache.ojb.broker.ProductGroup"
    table="Kategorien"
 >
    <field-descriptor
       name="groupId"
       column="Kategorie_Nr"
       jdbc-type="INTEGER"
       primarykey="true"
       autoincrement="true"
    />
    <field-descriptor
       name="groupName"
       column="KategorieName"
       jdbc-type="VARCHAR"
    />
    <field-descriptor
       name="description"
       column="Beschreibung"
       jdbc-type="VARCHAR"
    />
    <collection-descriptor
       name="allArticlesInGroup"
       element-class-ref="org.apache.ojb.broker.Article"
       orderby="articleId"
       sort="DESC"
    >
       <inverse-foreignkey field-ref="productGroupId"/>
    </collection-descriptor>
 </class-descriptor>
```

With the mapping shown above OJB has two possibilities to load the Articles belonging to a ProductGroup:

Copyright © All rights reserved.

1. loading all Articles of the ProductGroup immediately after loading the ProductGroup. This is done with **two** SQL-calls: one for the ProductGroup and one for all Articles.
2. if Article is a proxy (using proxy classes), OJB will only load the keys of the Articles after the ProductGroup. When accessing an Article-proxy OJB will have to materialize it with another SQL-Call. Loading the ProductGroup and all it's Articles will thus produce **n+2** SQL-calls: one for the ProductGroup, one for keys of the Articles and one for each Article.

Both approaches have their benefits and drawbacks:

- **A.** is suitable for a small number of related objects that are easily instantiated. It's efficient regarding DB-calls. The major drawback is the amount of data loaded. For example to show a list of ProductGroups the Articles may not be needed.
- **B.** is best used for a large number of related heavy objects. This solution loads the objects when they are needed ("lazy loading"). The price to pay is a DB-call for each object.

Further down a third solution using a single proxy for a whole collection will be presented to circumvent the described drawbacks.

OJB supports different Collection types to implement 1:n and m:n associations. OJB detects the used type automatically, so there is no need to declare it in the repository file. But in some cases the *default* behaviour of OJB is undesired. Please read here for more information.

Note:
When using primitive primary key fields, please pay attention on how OJB manage `null` for primitive PK/FK

2.1. 1:n auto-xxx setting

General info about the *auto-xxx* and *proxy* attributes can be found here.

auto-retrieve
See here

auto-update

- **none** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, the referenced objects will not be linked (foreign key assignment on referenced objects) to the main object automatically. The user has to link and to store the referenced objects *after* storing the main object to avoid violation of referential integrity.
- **link** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, but the referenced objects will be linked automatically (FK assignment) the main object.
- **object** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects will be linked and stored automatically.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

auto-delete

- **none** On deleting an object with `PersistenceBroker.delete(...)` the referenced objects are NOT touched. This may lead to violation of referential integrity if the referenced objects are childs of the main object. In this case the referenced objects have to be deleted manually first.
- **link** Is equivalent to *none*.
- **object** On deleting an object with `PersistenceBroker.delete(...)` the referenced objects will be deleted too.
- **false** Is equivalent to *none*.
- **true** Is equivalent to *object*.

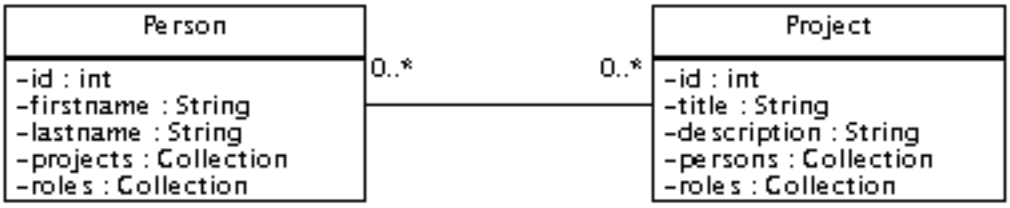
3. Mapping m:n associations

Copyright © All rights reserved.

OJB provides support for manually decomposed m:n associations as well as an automated support for non decomposed m:n associations.

3.1. Manual decomposition into two 1:n associations

Have a look at the following class diagram:



m:n association

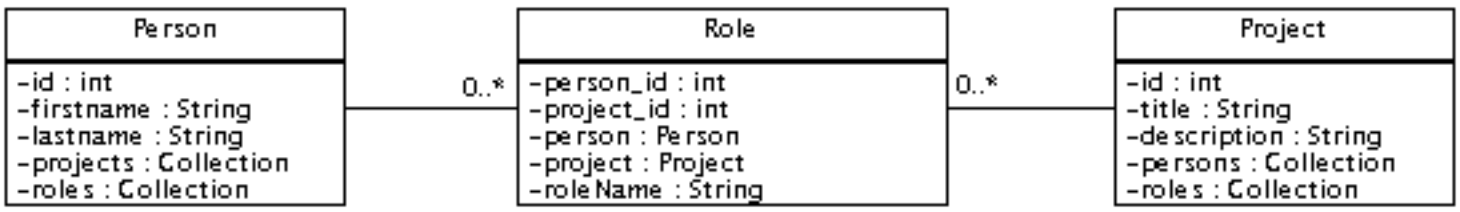
We see a two classes with a m:n association. A Person can work for an arbitrary number of Projects. A Project may have any number of Persons associated to it.
Relational databases don't support m:n associations. They require to perform a manual decomposition by means of an intermediary table. The DDL looks like follows:

```
CREATE TABLE PERSON (
    ID          INT NOT NULL PRIMARY KEY,
    FIRSTNAME   VARCHAR(50),
    LASTNAME    VARCHAR(50)
  );

CREATE TABLE PROJECT (
    ID          INT NOT NULL PRIMARY KEY,
    TITLE       VARCHAR(50),
    DESCRIPTION VARCHAR(250)
  );

CREATE TABLE PERSON_PROJECT (
    PERSON_ID   INT NOT NULL,
    PROJECT_ID  INT NOT NULL,
    PRIMARY KEY (PERSON_ID, PROJECT_ID)
  );
```

This intermediary table allows to decompose the m:n association into two 1:n associations. The intermediary table may also hold additional information. For example, the role a certain person plays for a project:

```
    CREATE TABLE PERSON_PROJECT (
        PERSON_ID   INT NOT NULL,
        PROJECT_ID  INT NOT NULL,
        ROLENAME    VARCHAR(20),
        PRIMARY KEY (PERSON_ID, PROJECT_ID)
      );
```

The decomposition is mandatory on the ER model level. On the object model level it is not mandatory, but may be a valid solution. It is mandatory on the object level if the association is qualified (as in our example with a rolename). This will result in the introduction of a association class. A class-diagram reflecting this decomposition looks like:



m:n association

Copyright © All rights reserved.

A `Person` object has a Collection attribute `roles` containing `Role` entries. A `Project` has a Collection attribute `roles` containing `Role` entries. A `Role` has reference attributes to its `Person` and to its `Project`.
Handling of 1:n mapping has been explained above. Thus we will finish this section with a short look at the repository entries for the classes `org.apache.ojb.broker.Person`, `org.apache.ojb.broker.Project` and `org.apache.ojb.broker.Role`:

```
<!-- Definitions for org.apache.ojb.broker.Person -->
   <class-descriptor
      class="org.apache.ojb.broker.Person"
      table="PERSON"
   >
      <field-descriptor
         name="id"
         column="ID"
         jdbc-type="INTEGER"
         primarykey="true"
         autoincrement="true"
      />
      <field-descriptor
         name="firstname"
         column="FIRSTNAME"
         jdbc-type="VARCHAR"
      />
      <field-descriptor
         name="lastname"
         column="LASTNAME"
         jdbc-type="VARCHAR"
      />
      <collection-descriptor
         name="roles"
         element-class-ref="org.apache.ojb.broker.Role"
      >
         <inverse-foreignkey field-ref="person_id"/>
      </collection-descriptor>
      ...
   </class-descriptor>

<!-- Definitions for org.apache.ojb.broker.Project -->
   <class-descriptor
      class="org.apache.ojb.broker.Project"
      table="PROJECT"
   >
      <field-descriptor
         name="id"
         column="ID"
         jdbc-type="INTEGER"
         primarykey="true"
         autoincrement="true"
      />
      <field-descriptor
         name="title"
         column="TITLE"
         jdbc-type="VARCHAR"
      />
      <field-descriptor
         name="description"
         column="DESCRIPTION"
         jdbc-type="VARCHAR"
      />
      <collection-descriptor
         name="roles"
         element-class-ref="org.apache.ojb.broker.Role"
      >
         <inverse-foreignkey field-ref="project_id"/>
      </collection-descriptor>
      ...
```


Copyright © All rights reserved.

```
   </class-descriptor>

<!-- Definitions for org.apache.ojb.broker.Role -->
   <class-descriptor
      class="org.apache.ojb.broker.Role"
      table="PERSON_PROJECT"
   >
      <field-descriptor
         name="person_id"
         column="PERSON_ID"
         jdbc-type="INTEGER"
         primarykey="true"
      />
      <field-descriptor
         name="project_id"
         column="PROJECT_ID"
         jdbc-type="INTEGER"
         primarykey="true"
      />
      <field-descriptor
         name="roleName"
         column="ROLENAME"
         jdbc-type="VARCHAR"
      />
      <reference-descriptor
         name="person"
         class-ref="org.apache.ojb.broker.Person"
      >
         <foreignkey field-ref="person_id"/>
      </reference-descriptor>
      <reference-descriptor
         name="project"
         class-ref="org.apache.ojb.broker.Project"
      >
         <foreignkey field-ref="project_id"/>
      </reference-descriptor>
   </class-descriptor>
```

3.2. Support for Non-Decomposed m:n Mappings

If there is no need for an association class at the object level (we are not interested in role information), OJB can be configured to do the m:n mapping transparently. For example, a Person does not have a collection of `Role` objects but only a Collection of `Project` objects (held in the attribute `projects`). Projects also are expected to contain a collection of `Person` objects (hold in attribute `persons`).

To tell OJB how to handle this m:n association the CollectionDescriptors for the Collection attributes `projects` and `roles` need additional information on the intermediary table and the foreign key columns pointing to the `PERSON` table and the foreign key columns pointing to the `PROJECT` table:

Note:

OJB supports a multiplicity of collection implementations, inter alia `org.apache.ojb.broker.util.collections.RemovalAwareCollection` and `org.apache.ojb.broker.util.collections.RemovalAwareList`. By default the removal aware collections were used. This cause problems in m:n relations when `auto-update="true" or "object"` and `auto-delete="false" or "none"` is set, because objects deleted in the collection will be deleted on update of main object. Thus it is recommended to use a NOT removal aware collection class in m:n relations using the collection-class attribute.

Example for setting a collection class in the collection-descriptor:

```
collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"
```

An full example for a non-decomposed m:n relation looks like:

```
<class-descriptor
  class="org.apache.ojb.broker.Person"
  table="PERSON"
```


Copyright © All rights reserved.

```
>
  <field-descriptor
     name="id"
     column="ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />
  <field-descriptor
     name="firstname"
     column="FIRSTNAME"
     jdbc-type="VARCHAR"
  />
  <field-descriptor
     name="lastname"
     column="LASTNAME"
     jdbc-type="VARCHAR"
  />
  ...
  <collection-descriptor
     name="projects"
     collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"
     element-class-ref="org.apache.ojb.broker.Project"
     auto-retrieve="true"
     auto-update="true"
     indirection-table="PERSON_PROJECT"
  >
     <fk-pointing-to-this-class column="PERSON_ID"/>
     <fk-pointing-to-element-class column="PROJECT_ID"/>
  </collection-descriptor>
</class-descriptor>

<!-- Definitions for org.apache.ojb.broker.Project -->
<class-descriptor
  class="org.apache.ojb.broker.Project"
  table="PROJECT"
>
  <field-descriptor
     name="id"
     column="ID"
     jdbc-type="INTEGER"
     primarykey="true"
     autoincrement="true"
  />
  <field-descriptor
     name="title"
     column="TITLE"
     jdbc-type="VARCHAR"
  />
  <field-descriptor
     name="description"
     column="DESCRIPTION"
     jdbc-type="VARCHAR"
  />
  ...
  <collection-descriptor
     name="persons"
     collection-class="org.apache.ojb.broker.util.collections.ManageableArrayList"
     element-class-ref="org.apache.ojb.broker.Person"
     auto-retrieve="true"
     auto-update="false"
     indirection-table="PERSON_PROJECT"
  >
     <fk-pointing-to-this-class column="PROJECT_ID"/>
     <fk-pointing-to-element-class column="PERSON_ID"/>
  </collection-descriptor>
</class-descriptor>
```

That is all that needs to be configured! See the code in class `org.apache.ojb.broker.MtoNMapping` for JUnit

Copyright © All rights reserved.

testmethods using the classes `Person`, `Project` and `Role`.

> **Note:**
> When using primitive primary key fields, please pay attention on how OJB manage `null` for primitive PK/FK

3.3. m:n auto-xxx setting

General info about the `auto-xxx` and `proxy` attributes can be found here

auto-retrieve
See here

auto-update

- **none** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, the referenced objects will not be linked (creation of FK entries in the indirection table) automatically. The user has to store the main object, the referenced objects and to link the m:n relation after storing of all objects. establishing the m:n relationship *before* storing main and referenced objects may violate referential integrity.
- **link** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects are NOT updated by default. The referenced objects will not be *inserted* or *updated*, but the m:n relation will be linked automatically (creation of FK entries in the indirection table).

> **Note:**
> Make sure that the referenced objects exist in database before storing the main object with auto-update set *link* to avoid violation of referential integrity.

- **object** On updating or inserting of the main object with `PersistenceBroker.store(...)`, the referenced objects will be linked and stored automatically.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

auto-delete

- **none** On deleting an object with `PersistenceBroker.delete(...)` the referenced objects are NOT touched. The corresponding entries of the main object in the indirection table will not be removed. This may lead to violation of referential integrity depending on the definition of the indirection table.
- **link** On deleting an object with `PersistenceBroker.delete(...)` the m:n relation will be *unlinked* (all entries of the main object in the indirection table will be removed).
- **object** On deleting an object with `PersistenceBroker.delete(...)` all referenced objects will be deleted too.
- **false** Is equivalent to *link*.
- **true** Is equivalent to *object*.

4. Setting Load, Update, and Delete Cascading

As shown in the sections on 1:1, 1:n and m:n mappings, OJB manages associations (or object references in Java terminology) by declaring special Reference and Collection Descriptors. These Descriptor may contain some additional information that modifies OJB's behaviour on object materialization, updating and deletion.
The behaviour depends on specific attributes

- *auto-retrieve* - possible settings are *false*, *true*. If not specified in the descriptor the default value is *true*
- *auto-update* - possible settings are *none*, *link*, *object* and deprecated [*false*, *true*]. If not specified in the descriptor the default value is *false*
- *auto-delete* - possible settings are *none*, *link*, *object* and deprecated [*false*, *true*]. If not specified in the descriptor the default value is *false*

Copyright © All rights reserved.

Note:
When using a top-level api (ODMG, OTM, JDO) it is mandatory to use the *default* auto-XXX settings (or don't specify the attributes) for proper work.
This may change in future.

The attribute *auto-update* and *auto-delete* are described in detail in the corresponding sections for 1:1, 1:n and m:n references. The *auto-retrieve* setting is described below:

4.1. auto-retrieve setting

The `auto-retrieve` attribute used in `reference-descriptor` or `collection-descriptor` elements handles the loading behaviour of references (1:1, 1:n and m:n):

- **false** If set *false* the referenced objects will not be materialized on object materialization. The user has to materialize the n-side objects (or single object for 1:1) by hand using one of the following service methods of the `PersistenceBroker` class:

```
PersistenceBroker.retrieveReference(Object obj, String attributeName);
// or
PersistenceBroker.retrieveAllReferences(Object obj);
```

  The first method load only the specified reference, the second one loads all references declared for the given object.

  Note:
  Be careful when using "opposite" settings, e.g. if you declare a 1:1 reference with auto-retrieve="false" BUT auto-update="object" (or "true" or "link").
  Before you can perform an update on the main object, you have to "retrieve" the 1:1 reference. Otherwise you will end up with an nullified reference enty in main object, because OJB doesn't find the referenced object on update and assume the reference was removed.

- **true** If set *true* the referenced objects (single reference or all n-side objects) will be automatic loaded by OJB when the main object was materialized.

If OJB is configured to use proxies, the referenced objects are not materialized immmediately, but lazy loading proxy objects are used instead.

In the following code sample, a reference-descriptor and a collection-descriptor are configured to use cascading retrieval (`auto-retrieve="true"`), cascading insert/update (`auto-update="object"` or `auto-update="true"`) and cascading delete (`auto-delete="object"` or `auto-delete="true"`) operations:

```
<reference-descriptor
 name="productGroup"
 class-ref="org.apache.ojb.broker.ProductGroup"
 auto-retrieve="true"
 auto-update="object"
 auto-delete="object"
>
 <foreignkey field-ref="productGroupId"/>
</reference-descriptor>

<collection-descriptor
 name="allArticlesInGroup"
 element-class-ref="org.apache.ojb.broker.Article"
 auto-retrieve="true"
 auto-update="object"
 auto-delete="object"
 orderby="articleId"
 sort="DESC"
>
 <inverse-foreignkey field-ref="productGroupId"/>
</collection-descriptor>
```

4.2. Link references

Copyright © All rights reserved.

If in `reference-descriptor` or `collection-descriptor` the *auto-update* or *auto-delete* attributes are set to *none*, OJB does not touch the referenced objects on insert, update or delete operations of the main object. The user has to take care of the correct handling of referenced objects. When using referential integrity (who does not ?) it's essential that insert and delete operations are done in the correct sequence.

One important thing is assignment of the FK values. The assign of the FK values is transcribed with ***link references*** in OJB. In 1:1 references the main object has a FK to the referenced object, in 1:n references the referenced objects have FK pointing to the main object and in non-decomposed m:n relations a indirection table containing FK values from both sides of the relationship is used.

OJB provides some helper methods for linking references manually (assignment of the FK) in `org.apache.ojb.broker.util.BrokerHelper` class.

```
public void link(Object obj, boolean insert)
public void unlink(Object obj)
public boolean link(Object obj, String attributeName, boolean insert)
public boolean unlink(Object obj, String attributeName)
```

These methods are accessible via `org.apache.ojb.broker.PersistenceBroker`:

```
BrokerHelper bh = broker.serviceBrokerHelper();
```

Note:

The *link/unlink* methods are only useful if you set auto-update/-delete to *none*. In all other cases OJB handles the link/unlink of references internally. It is also possible to set all FK values by hand without using the link/unlink service methods.

Examples

Now we prepared for some example. Say class `Movie` has an m:n reference with class `Actor` and we want to store an Movie object with a list of Actor objects. The auto-update setting of collection-descriptor for Movie is *none*:

```
broker.beginTransaction();
// store main object first
broker.store(movie);
//now we store the right-side objects
Iterator it = movie.getActors().iterator();
while(it.hasNext())
{
    Object actor = it.next();
    broker.store(actor);
}
// now both side exist and we can link the references
broker.serviceBrokerHelper().link(movie, "actors", true);
/*
alternative call
broker.serviceBrokerHelper().link(movie, true);
*/
broker.commitTransaction();
```

First store the main object and the references, then use `broker.serviceBrokerHelper().link(movie, "actors", true)` to link the main object with the references. In case of a m:n relation linking create all FK entries in the indirection table.

In the next examples we want to manually delete a `Project` object with a 1:n relation to class `SubProject`. In the example, the Project object has load all SubProject objects and we want to delete the Project but **don't** want to delete the referenced SubProjects too (don't ask if this make sense ;-)). SubProject has an FK to Project, so we first have to *unlink* the reference from the main object to the references to avoid integrity constraint violation. Then we can delete the main object:

```
broker.beginTransaction();
// first unlink the n-side references
broker.serviceBrokerHelper().unlink(project, "subProjects");
```


Copyright © All rights reserved.

```
// update the n-side references, store SubProjects with nullified FK
Iterator it = project.getSubProjects().iterator();
while(it.hasNext())
{
    SubProject subProject = (SubProject) it.next();
    broker.store(subProject);
}

// now delete the main object
broker.delete(project);
broker.commitTransaction();
```

5. Using Proxy Classes

Proxy classes can be used for "lazy loading" aka "lazy materialization". Using Proxy classes can help you in reducing unneccessary database lookups.
There are two kind of proxy mechanisms available:

1. **Dynamic proxies** provided by OJB. They can simply be activated by setting certain switches in repository.xml. This is the solution recommemded for **most** cases.
2. **User defined proxies**. User defined proxies allow the user to write proxy implementations.

As it is important to understand the mechanics of the proxy mechanism I highly recommend to read this section before turning to the next sections "using dynamic proxies", "using a single proxy for a whole collection" and "using a proxy for a reference", covering dynamic proxies.

As a simple example we take a ProductGroup object `pg` which contains a collection of fifteen Article objects. Now we examine what happens when the ProductGroup is loaded from the database:

Without using proxies all fifteen associated Article objects are immediately loaded from the db, even if you are not interested in them and just want to lookup the description-attribute of the ProductGroup object.

If proxies are used, the collection is filled with fifteen proxy objects, that implement the same interface as the "real objects" but contain only an OID and a void reference. The fifteen article objects are not instantiated when the ProductGroup is initially materialized. Only when a method is invoked on such a proxy object will it load its "real subject" and delegate the method call to it. Using this dynamic delegation mechanism instantiation of persistent objects and database lookups can be minimized.

To use proxies, the persistent class in question (in our case the Article class) must implement an interface (for example InterfaceArticle). This interface is needed to allow replacement of the proper Article object with a proxy implementing the same interface. Have a look at the code:

```
    public class Article implements InterfaceArticle
    {
        /** maps to db-column "Artikel-Nr"; PrimaryKey*/
        protected int articleId;
        /** maps to db-column "Artikelname"*/
        protected String articleName;
        ...

        public int getArticleId()
        {
            return articleId;
        }

        public java.lang.String getArticleName()
        {
            return articleName;
        }
        ...
    }
```


Copyright © All rights reserved.

```
    public interface InterfaceArticle
    {
        public int getArticleId();
        public java.lang.String getArticleName();
        ...
    }
```

```
public class ArticleProxy extends VirtualProxy implements InterfaceArticle
{
    public ArticleProxy(ojb.broker.Identity uniqueId, PersistenceBroker broker)
    {
        super(uniqueId, broker);
    }

    public int getArticleId()
    {
        return realSubject().getArticleId();
    }

    public java.lang.String getArticleName()
    {
        return realSubject().getArticleName();
    }

    private InterfaceArticle realSubject()
    {
        try
        {
            return (InterfaceArticle) getRealSubject();
        }
        catch (Exception e)
        {
            return null;
        }
    }
}
```

The proxy is constructed from the identity of the real subject. All method calls are delegated to the object returned by `realSubject()`.
This method uses getRealSubject() from the base class VirtualProxy:

```
    public Object getRealSubject() throws PersistenceBrokerException
    {
        return indirectionHandler.getRealSubject();
    }
```

The proxy delegates the the materialization work to its `IndirectionHandler`. If the real subject has not yet been materialized, a PersistenceBroker is used to retrieve it by its OID:

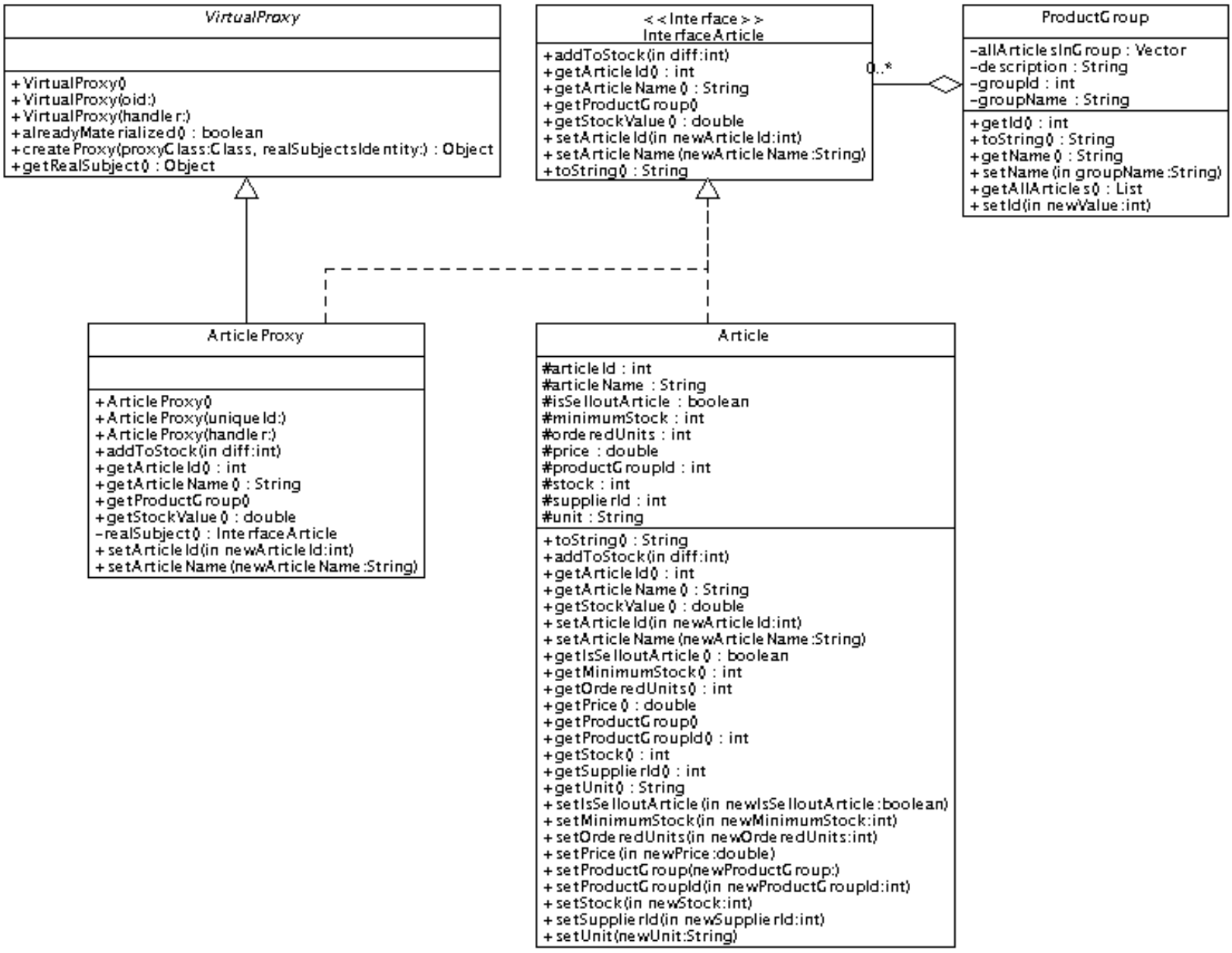
```
    public synchronized Object getRealSubject()
                        throws PersistenceBrokerException
    {
        if (realSubject == null)
        {
            materializeSubject();
        }
        return realSubject;
    }

    private void materializeSubject()
                        throws PersistenceBrokerException
    {
        realSubject = broker.getObjectByIdentity(id);
    }
```

To tell OJB to use proxy objects instead of materializing full Article objects we have to add the following section to the XML repository file:

Copyright © All rights reserved.

```
    <class-descriptor
        class="org.apache.ojb.broker.Article"
        proxy="org.apache.ojb.broker.ArticleProxy"
        table="Artikel"
    >
    ...
```

The following class diagram shows the relationships between all above mentioned classes:



proxy image

5.1. Using Dynamic Proxies

The implementation of a proxy class is a boring task that repeats the same delegation scheme for each new class. To liberate the developer from this unproductive job OJB provides a dynamic proxy solution based on the JDK 1.3 dynamic proxy concept. (For JDK1.2 we ship a replacement for the required `java.lang.reflect` classes. Credits for this solution to ObjectMentor.) The basic idea of the dynamic proxy concept is to catch all method invocations on the not-yet materialized

Copyright © All rights reserved.

(loaded from database) object. When a method is called on the object, Java directs this call to the invocation handler registered for it (in OJB's case a class implementing the `org.apache.ojb.broker.core.proxy.IndirectionHandler` interface). This handler then materializes the object from the database and replaces the proxy with the real object. By default OJB uses the class `org.apache.ojb.broker.core.proxy.IndirectionHandlerDefaultImpl`. If you are interested in the mechanics have a look at this class.

To use a dynamic proxy for lazy materialization of Article objects we have to declare it in the repository.xml file.

```
    <class-descriptor
      class="org.apache.ojb.broker.Article"
      proxy="dynamic"
      table="Artikel"
    >
    ...
```

Just as with normal proxies, the persistent class in question (in our case the Article class) must implement an interface (for example InterfaceArticle) to be able to benefit from dynamic proxies.

5.2. Using a Single Proxy for a Whole Collection

A collection proxy represents a whole collection of objects, where as a proxy class represents a single object.
The advantage of this concept is a reduced number of db-calls compared to using proxy classes. A collection proxy only needs a **single** db-call to materialize all it's objects. This happens the first time its content is accessed (ie: by calling iterator();). An additional db-call is used to calculate the size of the collection if size() is called *before* loading the data. So collection proxy is mainly used as a deferred execution of a query.

OJB uses three specific proxy classes for collections:

1. **List proxies** are specific `java.util.List` implementations that are used by OJB to replace lists. The default set proxy class is `org.apache.ojb.broker.core.proxy.ListProxyDefaultImpl`
2. **Set proxies** are specific `java.util.Set` implementations that are used by OJB to replace sets. The default set proxy class is `org.apache.ojb.broker.core.proxy.SetProxyDefaultImpl`
3. **Collection proxies** are collection classes implementing the more generic `java.util.Collection` interface and are used if the collection is neither a list nor a set. The default collection proxy class is `org.apache.ojb.broker.core.proxy.CollectionProxyDefaultImpl`

Which of these proxy class is actually used, is determined by the `collection-class` setting of this collection. If none is specified in the repository descriptor, or if the specified class does not implement `java.util.List` nor `java.util.Set`, then the generic collection proxy is used.

The following mapping shows how to use a collection proxy for a relationship:

```
    <!-- Definitions for
    org.apache.ojb.broker.ProductGroupWithCollectionProxy -->
    <class-descriptor
      class="org.apache.ojb.broker.ProductGroupWithCollectionProxy"
      table="Kategorien"
    >
      <field-descriptor
         name="groupId"
         column="Kategorie_Nr"
         jdbc-type="INTEGER"
         primarykey="true"
      />
      ...
      <collection-descriptor
         name="allArticlesInGroup"
         element-class-ref="org.apache.ojb.broker.Article"
         proxy="true"
      >
         <inverse-foreignkey field-ref="productGroupId"/>
```


Copyright © All rights reserved.

```
      </collection-descriptor>
    </class-descriptor>
```

The classes participating in this relationship do not need to implement a special interface to be used in a collection proxy.

Although it is possible to mix the collection proxy concept with the proxy class concept, it is not recommended because it increases the number of database calls.

5.3. Using a Proxy for a Reference

A proxy reference is based on the original proxy class concept. The main difference is that the ReferenceDescriptor defines when to use a proxy class and not the ClassDescriptor.
In the following mapping the class ProductGroup is **not** defined to be a proxy class in its ClassDescriptor. Only for shown relationship a proxy of ProductGroup should be used:

```
    <!-- Definitions for org.apache.ojb.broker.ArticleWithReferenceProxy -->
       <class-descriptor
          class="org.apache.ojb.broker.ArticleWithReferenceProxy"
          table="Artikel"
       >
          <field-descriptor
             name="articleId"
             column="Artikel_Nr"
             jdbc-type="INTEGER"
             primarykey="true"
             autoincrement="true"
          />
          ...
          <reference-descriptor
             name="productGroup"
             class-ref="org.apache.ojb.broker.ProductGroup"
        proxy="true"
          >
             <foreignkey field-ref="productGroupId"/>
          </reference-descriptor>
       </class-descriptor>
```

Because a proxy reference is only about the location of the definition, the referenced class must implement a special interface (see using proxy classes).

5.4. Customizing the proxy mechanism

Both the dynamic and the collection proxy mechanism can be customized by supplying a user-defined implementation.

For dynamic proxies you can provide your own invocation handler which implements the `org.apache.ojb.broker.core.proxy.IndirectionHandler` interface. See OJB's default implementation `org.apache.ojb.broker.core.proxy.IndirectionHandlerDefaultImpl` for details on how to implement such an invocation handler.

Each of the three collection proxy classes can be replaced by a user-defined class. The only requirement is that such a class implements both the corresponding interface (`java.util.Collection`, `java.util.List`, or `java.util.Set`) as well as the `org.apache.ojb.broker.ManageableCollection` interface.

Proxy implementations are configured in the ojb properties file. These are the relevant settings:

```
    ...
        #-------------------------------------------------------------------------------
        # IndirectionHandler
        #-------------------------------------------------------------------------------
        # The IndirectionHandlerClass entry defines the class to be used by OJB's proxies to
        # handle method invocations
        #
```


Copyright © All rights reserved.

```
    IndirectionHandlerClass=org.apache.ojb.broker.core.proxy.IndirectionHandlerDefaultImpl
    #
    #-------------------------------------------------------------------------------
    # ListProxy
    #-------------------------------------------------------------------------------
    # The ListProxyClass entry defines the proxy class to be used for collections that
    # implement the java.util.List interface.
    #
    ListProxyClass=org.apache.ojb.broker.core.proxy.ListProxyDefaultImpl
    #
    #-------------------------------------------------------------------------------
    # SetProxy
    #-------------------------------------------------------------------------------
    # The SetProxyClass entry defines the proxy class to be used for collections that
    # implement the java.util.Set interface.
    #
    SetProxyClass=org.apache.ojb.broker.core.proxy.SetProxyDefaultImpl
    #
    #-------------------------------------------------------------------------------
    # CollectionProxy
    #-------------------------------------------------------------------------------
    # The CollectionProxyClass entry defines the proxy class to be used for collections that
    # do not implement java.util.List or java.util.Set.
    #
    CollectionProxyClass=org.apache.ojb.broker.core.proxy.CollectionProxyDefaultImpl
...
```

6. Type and Value Conversions

Say your database column contains INTEGER values but you have to use boolean attributes in your Domain objects. You need a type and value mapping described by a FieldConversion!

Copyright © All rights reserved.