HOWTO - Use Anonymous Keys

by Brian McCallister

Table of contents

1. Why Do We Need Anonymous Keys?

The core difference between referential integrity in Java and in an RDBMS lies in where the specific referential information is maintained. Java, and most modern OO languages, maintain referential integrity information in the runtime environment. Actual object relationships are maintained by the virtual machine so that the symbolic variable used in the application is dereferenced it will in fact provide access to the object instance which it is expected to provide access to. There is no need for a manual lookup or search across the heap for the correct object instance. Entity reference integrity is maintained and handled for the programmer by the environment.

Relational databases, on the other, purposefully place the referential integrity and lookups into the problem domain - that is the problem they are designed to solve. An RDBMS presumes you can design something more efficient for your specific circumstances than the JVM does (you trust its ability to do object lookups in the heap is sufficiently efficient). As an RDBMS has a much larger heap equivalent it is designed to not operate under that assumption (mostly). So, in an RDBMS the concept of specific foreign keys exists to maintain the referential integrity.

In crossing the object to relational entity barrier there is a mismatch between the referential integrity implementations. Java programmers do not want to have to maintain both object referential integrity and key referential integrity analogous to

```
{
    Foo child = new SomeOtherFooType();
    Foo parent = new SomeFooType();
    child.setParent(parent);
    child.setParentId(parent.getId());
}
```

This is double the work required - you set up the object relationship, then set up the key relationship. OJB knows about the relationship of the objects, thus it is only needed to do

```
{
    Foo child = new Foo();
    Foo parent = new Foo();
    child.setParent(parent);
}
```

OJB can provide transparent key relationship maintenance behind the scenes for 1:1 relations via **anonymous access fields**. As object relationships change, the relationships will be propogated into the key values without the *Java object ever being aware of a relational key* being in use. This means that the java object does not need to specify a FK field for the reference.

Without use of *anonymous keys* class `Foo` have to look like:

```
class Foo
{
    Integer id;
    Integer fkParentFoo;
    Foo parent;

    // optional getter/setter
    ....
{
```

When using *anonymous keys* the FK field will become obsolete:

```
class Foo
{
    Integer id;
    Foo parent;
```

```
    // optional getter/setter
    ....
{
```

Note:

Under specific conditions it's also possible to use anonymous keys for other relations or primary keys. More info in advanced-technique section.

2. How it works

To play for safety it is mandatory to understand how this feature is working. More information how it works please see here.

3. Using Anonymous Keys

Now we can start using of the *anonymous key* feature. In this section the using is detailed described on the basis of an example.

3.1. The Code

Take the following classes designed to model a particular problem domain. They may do it reasonably well, or may not. Presume they model it perfectly well for the problem being solved.

```
public class Desk
{
    private Integer id;
    private Finish finish;
    /** Contains Drawer instances */
    private List drawers;
    private int numberOfLegs;

    public Desk()
    {
        this.drawers = new ArrayList();
    }
....
// getter/setter
}

public class Drawer
{
    private Integer id;
    /** Contains Thing instances */
    private List stuffInDrawer;

    public Drawer()
    {
        this.stuffInDrawer = new ArrayList();
    }

....
// getter/setter
}

public class Finish
{
    private Integer id;
    private String wood;
    private String color;

....
// getter/setter
}
```

```
public class Thing
{
    private Integer id;
    private String name;

....
// getter/setter
}
```

A Desk will typically reference multiple drawers and one finish.

3.2. The Database

When we need to store our instances in a database we use a fairly typical table per class persistance model.

```
CREATE TABLE finish
(
    id          INTEGER PRIMARY KEY,
    wood        VARCHAR(255),
    color       VARCHAR(255)
);

CREATE TABLE desk
(
    id          INTEGER PRIMARY KEY,
    num_legs    INTEGER,
    finish_id   INTEGER,
    FOREIGN KEY (finish_id) REFERENCES finish(id)
);

CREATE TABLE drawer
(
    id          INTEGER PRIMARY KEY,
    desk_id     INTEGER,
    FOREIGN KEY (desk_id) REFERENCES desk(id)
);

CREATE TABLE thing
(
    id          INTEGER PRIMARY KEY,
    name        VARCHAR(255),
    drawer_id   INTEGER,
    FOREIGN KEY (drawer_id) REFERENCES drawer(id)
);
```

At the database level the possible relationships need to be explicitely defined by the foreign key constraints. These model all the possible object relationships according to the domain model (until generics enter the Java language for the collections API, this is technically untrue for the classes used here).

3.3. The Repository Configuration

When we go to map the classes to the database, it is almost a one-to-one property to field mapping. The exception here is the primary key on each entity. This is meaningless information in Java, so we would like to keep it out of the object model. Anonymous access keys allow us to do that.

The repository.xml must know about the database columns used for referential integrity, but OJB can maintain the foreign key relationships behind the scenes - freeing the developer to focus on more accurate modeling of her objects to the problem, instead of the the persistance mechanism. Doing this is also very simple - in the *repository.xml* file mark the field descriptors with a `access="anonymous"` attribute.

```
<class-descriptor
```

```
    class="Desk"
    table="desk">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="numberOfLegs"
        column="num_legs"
        jdbc-type="INTEGER"
        />
    <field-descriptor
        name="finishId"
        column="finish_id"
        jdbc-type="INTEGER"
        access="anonymous" />

    <reference-descriptor
        name="finish"
        class-ref="Finish">
            <foreignkey field-ref="finishId"/>
    </reference-descriptor>

    <collection-descriptor
        name="drawers"
        element-class-ref="Drawer"
        >
        <inverse-foreignkey field-ref="deskId"/>
    </collection-descriptor>
</class-descriptor>

<class-descriptor
    class="Finish"
    table="finish">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="wood"
        column="wood"
        jdbc-type="VARCHAR"
        size="255"
        />
   <field-descriptor
        name="color"
        column="color"
        jdbc-type="VARCHAR"
        size="255"
        />
</class-descriptor>

<class-descriptor
    class="Drawer"
    table="drawer">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
```

```
    <field-descriptor
        name="deskId"
        column="desk_id"
        jdbc-type="INTEGER"
        access="anonymous"
        />
    <collection-descriptor
        name="stuffInDrawer"
        element-class-ref="Thing"
        >
        <inverse-foreignkey field-ref="drawerId"/>
    </collection-descriptor>
</class-descriptor>

<class-descriptor
    class="Thing"
    table="thing">

    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="true"
        />
    <field-descriptor
        name="name"
        column="name"
        jdbc-type="VARCHAR"
        size="255"
        />
    <field-descriptor
        name="drawerId"
        column="drawer_id"
        jdbc-type="INTEGER"
        access="anonymous"
        />
</class-descriptor>
```

Look first at the class descriptor for the Thing class. Notice the field-descriptor with the name attribute "drawerId". This field is labeled as anonymous access. Because it is anonymous access OJB will not attempt to assign the value here to a "drawerId" field or property on the Thing class. Normally the name attribute is used as the Java name for the attribute, in this case it is not. The name is still required because it is used as an indicated for references to this anonymous field.

In the field descriptor for Drawer, look at the collection descriptor with the name *stuffInDrawer*. This collection descriptor references a foreign key with the `field-ref="drawerId"`. This reference is to the anonymous field descriptor in the Thing descriptor. The field-ref matches to the name in the descriptor whether or not the name also maps to the Java attribute name. This dual use of `name` can be confusing - be careful.

The same type mapping that is used for the collection descriptor in the Drawer descriptor is also used for the 1:1 reference descriptor in the Desk descriptor.

The primary keys are populated into the objects as it is generally a good practice to not implement primary keys as anonymous access fields. Primary keys may be anonymous-access but references will get lost if the cache is cleared or the persistent object is serialized.

4. Benefits and Drawbacks

There are both benefits and drawbacks to using anonymous field references for maintaining referential integrity between Java objects and database relations. The most immediate benefit is avoiding semantic code duplication. The second major benefit is avoiding cluttering class definitions with persistance mechanism artifacts. In a well layered application, the persistance

mechanism will not really need to be so obvious in the object model implementation. Anonymous fields helpt o achieve this - thereby making persistence mechanisms more flexible. Moving to a different one becomes easier.