Hibernate and XDoclet

How to generate Hibernate mapping files with XDoclet

Pascal Betz (pascal@downside.ch)

Version: 1.4
Date: 26. February 2005

History

Version	What
1.4	Added @hibernate.column tag description in Chapter 3
1.3	Fixed error on page 29 (inverse="true instead of "true="true"). Small "run examples with ant explanation" in section 10.3 and mentioned the log4j.properties in section 10.4
1.2	Section on composite ids added
1.1	Spell checking, added description of the generate-hbm and schemaexport ant tasks
1.0	Updated inheritance mapping strategy chapter.
0.9	Added section on bidirectional parent/child relation, added cheatsheet (mapping overview)
0.85	Added chapter on one-to-one, added history
0.8	Initial version

Table of contents

1. Introduction 3
1.1. Hibernate 3
1.2. XDoclet 3
1.3. What is in this tutorial 3
1.4. Feedback 3
1.5. Thanks 3
2. Setup 3
2.1. Tutorial 3
2.2. Database 3
2.3. Hibernate 4
2.4. XDoclet 4
2.5. Ant 4
3. Basic Persistence 5
3.1. Our own Pet 6
4. Mapping Subclasses 10
4.1. Table per hierarchy 10
4.2. Table per subclass 12
4.3. Table per concrete class 15
5. Simple Queries 16
6. Composition 17
7. Many-to-One 19
8. One-to-One 21
8.1. Primary key association 21
8.2. Foreign key association 22
9. Collection Mapping 22
9.1. One-to-Many relation 22
9.2. Lazy Initialization 26
9.3. Many-to-Many 26
9.4. Bidirectional Parent Child 29
10. Composite Keys 30
11. Appendix 32
11.1. Links 32
11.2. IDE Hints 32
11.3. Running the examples with Ant 33
11.4. Log levels 33
11.5. Mapping Cheatsheet 34

 pascal@downside.ch

1. Introduction

1.1. Hibernate

Hibernate is a powerful Object Relational Mapping (ORM) service which is available under the LPGL license. It is very well documented (which is often the major drawback of open-source/free of charge tools) and has an active community. Of course you will loose some time studying Hibernate and how it works. But once you understand it, you don't want to miss it.

1.2. XDoclet

XDoclet is an attribute driven code generation engine. From XDoclet tags written in Java, we can create mapping files for Hibernate. Like this these files can be created automatically and are in sync with your java code. We use version 1.2.1, XDoclet 2 is still in development.

1.3. What is in this tutorial

I'll try to show you how to use XDoclet to generate Hibernate mapping files. Generating these files from the Hibernate tags can save you a lot of time and hassle. There a lots of possible mappings, only a selection is covered in this tutorial.

1.4. Feedback

Found a spelling mistake, a technical error, is code not working as expected? Did you like it or do you think it did not even worth the download? Should I add a chapter on feature XYZ?

Please send your feedback to pascal@downside.ch

1.5. Thanks

Thanks to Fabien Girardin for proofreading the tutorial, testing the code and pointing out things that were not clear.
Thanks to Scott Plante for detailed feedback and spell checking.

2. Setup

A few things are to be configured before we can start. Make sure all software is installed correctly before you proceed.

2.1. Tutorial

Download the tar or zip file for this tutorial and extract it to a location of your choice. This will give you the directory `hibernatetutorial` which holds all sources, mappings, properties, the build file, and the required libraries.

2.2. Database

Hibernate offers an abstraction from the underlying database. Use any DB that is supported by Hibernate (if your favorite DB is not supported, then download a copy of MySQL which is free for non-commercial use), create a database and a user that has sufficient access rights. If you do not use MySQL, add your JDBC driver to the `hibernatetutorial/lib/jdbc` directory.

2.3. Hibernate

All required libraries are in the distribution of this tutorial (`hibernatetutorial/lib/hibernate`). The only thing left to do is to adapt `hibernate.properties` to reflect your environment. You can find the Hibernate API documentation in `hibernatetutorial/doc/hibernate/api/index.html`. For a list of available dialects, you can check the package `net.sf.hibernate.dialect`.

```
hibernatetutorial/resources/hibernate.properties
```

```
hibernate.dialect net.sf.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql://localhost:3306/hibernate
hibernate.connection.username hibernate
hibernate.connection.password hibernate
```

This is a minimal Hibernate configuration and there are a lot of other things that can be configured. We don't care about them for now.

2.4. XDoclet

There is nothing to setup for XDoclet, all required libraries are in `hibernatetutorial/lib/xdoclet`. You can find the Hibernate tag documentation of XDoclet under `hibernatetutorial/doc/xdoclet/hibernate-tags.html`.

2.5. Ant

To run XDoclet we need the Ant build tool. If not already done, download and install it according to the documentation. The two most important targets are generate-hbm and the schemaexport.

```
The generate-hbm target
```

```
<target name="generate-hbm"
      description="Generates Hibernate class descriptor files."
      depends="compile-src">
      <!-- Define the hibernatedoclet task -->
      <taskdef name="hibernatedoclet"
            classname="xdoclet.modules.hibernate.HibernateDocletTask"
            classpathref="project.class.path">
      </taskdef>

      <!-- Execute the hibernatedoclet task -->
      <hibernatedoclet
            destdir="${hbm.dir}"
            excludedtags="@version,@author,@todo"
            force="true"
            verbose="true">
            <fileset dir="${src.dir}">
                  <include name="**/*/*.java"/>
            </fileset>
            <!-- hibernate 1 and hibernate 2
                mapping files are not fully compatible-->
            <hibernate version="2.0"/>
      </hibernatedoclet>
</target>
```

 pascal@downside.ch

This target takes all java source files in the `${src.dir}` and generates hibernate mapping files in the `${hbm.dir}`

```
The schemaexport target
<target name="schemaexport"
      description="Exports all hbm.xml files in {res.dir}/hbm">
      <taskdef name="schemaexport"
            classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
            classpathref="project.class.path">
      </taskdef>
      <schemaexport
            properties="${res.dir}/hibernate.properties"
            quiet="no"
            text="no"
            drop="no"
            output="${gen.files.dir}/schema-export.sql">
            <fileset dir="${gen.files.dir}">
                  <include name="**/*.hbm.xml"/>
            </fileset>
      </schemaexport>
</target>
```

Here we create a file containing all the create and update statements to initialize the Database at `${gen.files.dir}/schema-export.sql` and then we run it on the Database defined in `${res.dir}/hibernate.properties`.

You can find more information on XDoclet for Hibernate configuration on the Hibernate website and of course on the XDoclet site.

3. Basic Persistence

Before Hibernate can persist your objects you need to tell it what attributes are to be persisted. This is done trough the Hibernate mapping files (hbm.xml files) which are loaded into the Hibernate `Configuration`. From this `Configuration` you can create a `SessionFactory` and this factory enables you to create `Session` objects. Please read the Hibernate documentation so you fully understand the concept of `Session`. Since this tutorial is mostly on the creation of mapping files with XDoclet, there will not be much code that explains the interaction with the `Session`. But the following template should give you an idea.

```
tutorial.DemoTemplate.java
package tutorial;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;

public class DemoTemplate {
      public static void main(String[] args) {
            try {
                  new DemoTemplate();
```

 pascal@downside.ch

```
            } catch(HibernateException he) {
                  he.printStackTrace();
            }
      }

      public DemoTemplate() throws HibernateException {
            Configuration config = new Configuration();
            // add the classes you want to persist
            // make sure the mapping files are on the classpath

            //config.addClass(SomePersistentClass.class);
            //config.addClass(AnotherPersistentClass.class);
            //config.addClass(YetPersistentClass.class);

            SessionFactory sf = config.buildSessionFactory();

            Session sess = sf.openSession();
            Transaction tx = null;
            try {
                  tx = sess.beginTransaction();

                  //sess.save(obj);
                  //sess.saveOrUpdate(obj);
                  //sess.find("HQL Query here");
                  //sess. ...

                  tx.commit();
            } catch (HibernateException e) {
                if (tx!=null) tx.rollback();
                throw e;
            }
            finally {
                sess.close();
            }
      }
}
```

3.1. Our own Pet

Let's start to write our first persistent class: a simple `Pet` class. It has an id and a name attribute. The id attribute identifies the `Pet` in the database. We use an artificial key, which can be set to null (no primitive data type like int or long). That way, Hibernate can determine if an object has been saved before just by checking if the id attribute is null.

tutorial.basic.Pet.java

```
package tutorial.basic;
/**
 *@hibernate.class
 *     table="PET"
 */
public class Pet {
      private Long id;
      private String name;
      /**
       * @hibernate.id
       *     column="ID"
```

 pascal@downside.ch

```
         *     generator-class="hilo"
         *     unsaved-value="null"
         */
        public Long getId() {return id;}
        public void setId(Long id) {this.id = id;}
        /**
         * @hibernate.property
         */
        public String getName() {return name;}
        public void setName(String name) {this.name = name;}
}
```

The *@hibernate.class* is required; otherwise XDoclet does not create a mapping file for this class. The *table* attribute is optional, if not specified Hibernate uses the class name. The *unsaved-value="null"* is also optional since null is the default value. For the name, we simply add a *@hibernate.property* tag and we are done.

Note: If your class has a name that is a reserved word in your database (e.g. Index, Order, Table), you need to specify another name trough the table attribute. Otherwise the tables wont be created correctly.

Running `ant generate-hbm` creates the mapping file for the `Pet` class.

hibernatetutorial/gen-files/hbm/basic/Pet.hbm.xml

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class
        name="tutorial.basic.Pet"
        table="PET"
        dynamic-update="false"
        dynamic-insert="false"
    >
        <id
            name="id"
            column="ID"
            type="java.lang.Long"
            unsaved-value="null"
        >
            <generator class="hilo">
            </generator>
        </id>

        <property
            name="name"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="name"
        />
```

 pascal@downside.ch

```
    </class>
</hibernate-mapping>
```

Note: If you get an Exception when exporting the schema (e.g. Schema text failed: java.lang.StringIndexOutOfBoundsException: String index out of range: 0) it could be you have a typo in your XDoclet tags.

I assume you prefer to type some XDoclet tags instead of creating such an XML file. `ant schemaexport` will create a file with the SQL statements to create and alter the tables and then executes the script. After that we can find following tables in the database:

```
mysql> show tables;
+----------------------+
| Tables_in_hibernate  |
+----------------------+
| hibernate_unique_key |
| pet                  |
+----------------------+
2 rows in set (0.00 sec)
```

The hibernate_unique_key table is required for the hilo strategy id generator (hilo is from high-low. Find more on the high-low strategy on http://www.agiledata.org/essays/dataModeling101.html). The pet table looks like this:

```
mysql> describe pet;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| ID    | bigint(20)   |      | PRI | 0       |       |
| name  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.09 sec)
```

As you can see, Hibernate picked varchar(255) for the name. If you think this is too big or too small, then you can specify the column length yourself. We add another property to show how to change the length of a varchar column:

A snipped of tutorial.basic.Pet.java

```
/**
 * @hibernate.property
 *   column="nick"
 *   length="30"
 */
 public String getNickname() {return nickname;}
 public void setNickname(String nickname) {this.nickname = nickname;}
```

Run `generate-hbm` and `schemaexport` again and check the tables:

```
mysql> describe pet;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
```

 pascal@downside.ch

```
| ID     | bigint(20)    |      | PRI | 0       |       |
| name   | varchar(255)  | YES  |     | NULL    |       |
| nick   | varchar(30)   | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

If you need to tweak the `sql-type or specify an index`, then a *`@hibernate.column`* tag is required.

```
A snipped of tutorial.basic.Pet.java
/**
       * @hibernate.property
       *    column="age"
       * @hibernate.column
       *  name="override_column"
       *  sql-type="numeric(8,2)"
       */
```

The name attribute of *`@hibernate.column`* will override the column attribute of the *`@hibernate.property`*. sql-type can be anything your DB supports.

```
mysql> describe pet;
+-----------------+--------------+------+-----+---------+-------+
| Field           | Type         | Null | Key | Default | Extra |
+-----------------+--------------+------+-----+---------+-------+
| ID              | bigint(20)   |      | PRI | 0       |       |
| name            | varchar(255) | YES  |     | NULL    |       |
| nick            | varchar(30)  | YES  |     | NULL    |       |
| override_column | decimal(8,2) | YES  |     | NULL    |       |
+-----------------+--------------+------+-----+---------+-------+
```

Note: If you put your XDoclet tags inside a normal comment spanning multiple lines , then your tags are not recognized. Use

```
/**
 * @hibernate.something
 */
```

instead of

```
/*
 * @hibernate.something
 */
```

Note: If you add *`your @hibernate.class`* tags above the `package` statement, then XDoclet will not create any mapping files.

That's all you need to know to persist your first Objects. You can run the `tutorial.basic.BasicDemo` to insert some values and then check the content of the Database:

 pascal@downside.ch

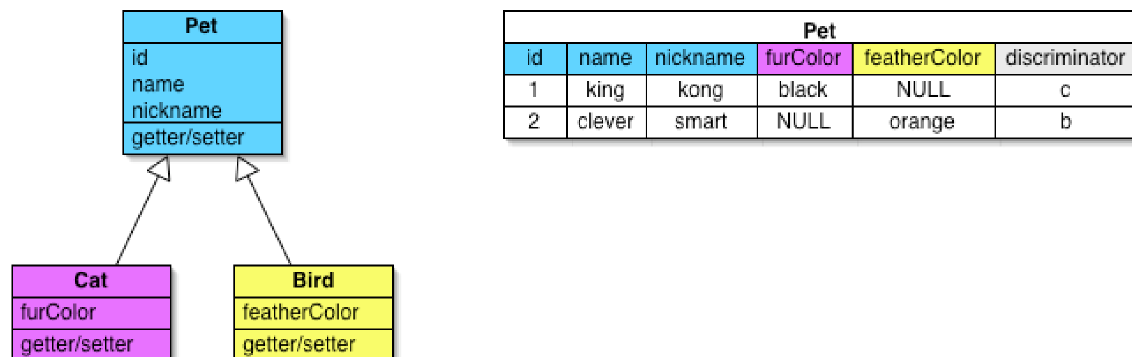
```
mysql> select * from pet;
+----+----------------+--------+
| ID | name           | nick   |
+----+----------------+--------+
|  1 | Some Name Here | Tweety |
|  2 | Silvester      | Sly    |
+----+----------------+--------+
2 rows in set (0.00 sec)
```

4. Mapping Subclasses

There are several ways to map a class hierarchy to a relational database: “Table per class”, “Table per concrete class” or “Table per hierarchy”. Hibernate supports those 3 but recommends the “Table per hierarchy” approach. There are limitations, pros and cons to each of these strategies. Please consult chapter 8 of the Hibernate documentation. It describes the three approaches and lists the benefits as well as the limitations. It is even possible to use different strategies for different branches of a class hierarchy.

4.1. Table per hierarchy

In the table per hierarchy approach all properties are mapped to one database table and a discriminator column is added to determine the class that is represented by a specific row. Depending on your hierarchy you may have wide tables, where only a few columns are used. On the other hand, only one table is queried when an object is reconstructed.



Pet

id	name	nickname	furColor	featherColor	discriminator
1	king	kong	black	NULL	c
2	clever	smart	NULL	orange	b

Figure 1: Table per hierarchy

Subclasses are mapped with a *@hibernate.subclass* tag. In the superclass we specify the discriminator with *@hibernate.discriminator* and its value with *discriminator-value*. The subclasses only need the *discriminator-value*..

tutorial.inheritance.tableperhierarchy.Animal.java

```
package tutorial.inheritance.tableperhierarchy;
/**
 * @hibernate.class
 *     table="Animal"
 *     discriminator-value="A"
 * @hibernate.discriminator
 *     column="discriminator"
 *     type="char"
```

 pascal@downside.ch

```
 */
public abstract class Animal {
      private Long id;
      private String name;
      /**
       * @hibernate.id
       *     column="ID"
       *     generator-class="hilo"
       *     unsaved-value="null"
       */
      public Long getId() {return id;}
      public void setId(Long id) {this.id = id;}
      /**
       * @hibernate.property
       */
      public String getName() {return name;}
      public void setName(String name) {this.name = name;}

      public abstract void makeSound();
}
```

```
in tutorial.inheritance.tableperhierarchy.Cat.java
package tutorial.inheritance.tableperhierarchy;
/**
 * @hibernate.subclass
 *     discriminator-value="B"
 */
public class Cat extends Animal {
```

```
in tutorial.inheritance.tableperhierarchy.Cow.java
package tutorial.inheritance.tableperhierarchy;
/**
 * @hibernate.subclass
 *     discriminator-value="C"
 */
public class Cow extends Animal {
```

If we do not specify a value, then Hibernate will use the fully specified class name as discriminator value. The type and length of the discriminator column can be changed trough `type` and `length` attributes of the `@hibernate.discriminator` tag. XDoclet will only generate one mapping file (`Animal.hbm.xml`) for all three classes.
Here is what the table looks like after generating the mappings and exporting the schema:

```
mysql> describe animal;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| ID            | bigint(20)   |      | PRI | 0       |       |
| discriminator | char(1)      |      |     |         |       |
| name          | varchar(255) | YES  |     | NULL    |       |
| furColor      | varchar(255) | YES  |     | NULL    |       |
| nickName      | varchar(255) | YES  |     | NULL    |       |
```

 pascal@downside.ch

```
+---------------+--------------+------+-----+---------+-------+
5 rows in set (0.04 sec)
```

As you can see all properties of `Animal`, `Cat` and `Cow` have been mapped to the same table. Now we can ask Hibernate to store our `Animals`.

```
In tutorial.inheritance.tableperhierarchy.InheritanceDemo.java
tx = sess.beginTransaction();

Cat cat = new Cat();
cat.setName("Silvester");
cat.setNickName("Sly");
sess.save(cat);

Cow cow = new Cow();
cow.setName("Rose");
cow.setFurColor("Brown");
sess.save(cow);

tx.commit();
```

Two rows have been added to the Animal table:

```
mysql> select * from animal;
+----+---------------+-----------+----------+----------+
| ID | discriminator | name      | furColor | nickName |
+----+---------------+-----------+----------+----------+
|  1 | B             | Silvester | NULL     | Sly      |
|  2 | C             | Rose      | Brown    | NULL     |
+----+---------------+-----------+----------+----------+
2 rows in set (0.00 sec)
```

The ID column has been created by Hibernate, the discriminator will be used to determine the specific class that has to be instantiated.

Note: Even though you will never store an instance of the abstract `Animal` class you need to specify a *`discriminator-value`*. (only if custom discriminator values are being used).

Note: Defining your own discriminator value seems to be an easy way to save some space (after all a fully qualified class name can be quite long). But be careful to assign unique discriminator values. Otherwise your Cat might end up as a Cow.

4.2. Table per subclass

The table per subclass approach creates a table per class and does not need a discriminator column. Instead it adds an id column to the subclass tables that links it to the superclass table.

 pascal@downside.ch

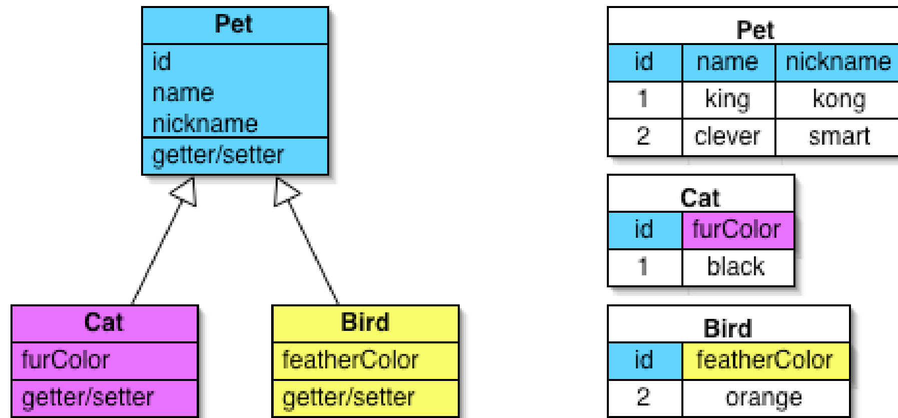


Figure 2: Table per class

We do not need discriminator and discriminator-value in the superclass a simple *@hibernate.class* is enough.

```
In tutorial.inheritance.tablepersubclass.Payment.java
package tutorial.inheritance.tablepersubclass;
/**
 * @hibernate.class
 */
public class Payment {
```

We map the subclasses with the *@hibernate.joined-subclass* tag and specify a *@hibernate.joined.subclass-key*.

```
In tutorial.inheritance.tablepersubclass.CreditCardPayment.java
package tutorial.inheritance.tablepersubclass;
/**
 * @hibernate.joined-subclass
 * @hibernate.joined-subclass-key
 *    column="id"
 */
public class CreditcardPayment extends Payment {
```

```
In tutorial.inheritance.Cat.java
package tutorial.inheritance.tablepersubclass;
/**
 * @hibernate.joined-subclass
 * @hibernate.joined-subclass-key
 *    column="id"
 */
public class CashPayment extends Payment {
```

Again only one Mapping file was generated trough XDoclet. The subclasses appear as `<joined-subclass>` elements in `Payment.hbm.xml`.

```
A snippet of Payment.hbm.xml
```

 pascal@downside.ch

```
<joined-subclass
    name="tutorial.inheritance.tablepersubclass.CreditcardPayment"
    dynamic-update="false"
    dynamic-insert="false">
    <key
        column="id"/>
     <property
         name="cardNumber"
         type="java.lang.String"
         update="true"
         insert="true"
         access="property"
         column="cardNumber"/>
</joined-subclass>
```

There are 3 tables for the classes now, one for `Payment` and one each for the subclasses `CreditCardPayment` and `CashPayment`.

```
mysql> show tables;
+-----------------------+
| Tables_in_hibernate   |
+-----------------------+
| cashpayment           |
| creditcardpayment     |
| payment               |
+-----------------------+
3 rows in set (0.03 sec)
```

The creditcardpayment and the cashpayment tables have an additional column to reference the payment table.

```
mysql> describe creditcardpayment;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | bigint(20)   |      | PRI | 0       |       |
| cardNumber | varchar(255) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

Querying several tables (the number of table depends on the depth of your class hierarchy) is required to reconstruct the Object.

```
mysql> select payment.id, creditcardpayment.cardNumber from payment,
creditcardpayment where payment.id = creditcardpayment.id;
+-------+---------------------+
| id    | cardNumber          |
+-------+---------------------+
| 32770 | 1234-5678-1234-5678 |
+-------+---------------------+
1 row in set (0.02 sec)
```

 pascal@downside.ch

4.3. Table per concrete class

The table per concrete class strategy does not need tables for the abstract classes. The properties of abstract classes are mapped to the concrete classes tables.

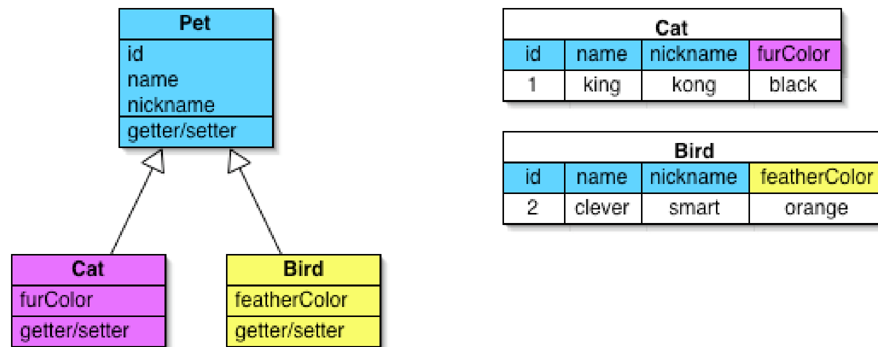


Figure 3: Table per concrete class

The abstract classes do not need an *@hibernate.class* tag at all. All the other tags can be used as usual:

```
tutorial.inheritance.tableperconcreteclass.AbstractClass.java
package tutorial.inheritance.tableperconcreteclass;
public abstract class AbstractClass {
	private Long id;
	private String name;
	/**
	 * @hibernate.id
	 *     column="ID"
	 *     generator-class="hilo"
	 *     unsaved-value="null"
	 */
	public Long getId() {return id;}
	public void setId(Long id) {this.id = id;}
	/**
	 * @hibernate.property
	 */
	public String getName() {return name;}
	public void setName(String name) {this.name = name;}
}
```

All the subclasses are mapped with *@hibernate.class* tags

```
tutorial.inheritance.tableperconcreteclass.SuperClass.java
package tutorial.inheritance.tableperconcreteclass;
/**
 * @hibernate.class
 *     table="superclass"
 */
public class Superclass extends AbstractClass {
	private String superName;
	/**
	 * @hibernate.property
	 */
```

 pascal@downside.ch

```
      public String getSuperName() {return superName;}
      public void setSuperName(String superName) {
          this.superName = superName;
      }
}
```

```
tutorial.inheritance.tableperconcreteclass.SubClass.java
package tutorial.inheritance.tableperconcreteclass;
/**
 * @hibernate.class
 *     table="subclass"
 */
public class Subclass extends Superclass {
      private String subName;
      /**
       * @hibernate.property
       */
      public String getSubName() {return subName;}
      public void setSubName(String subName) {
          this.subName = subName;
      }
}
```

Two tables are created, one for each concrete class:

```
+----------------------+
| Tables_in_hibernate  |
+----------------------+
| subclass             |
| superclass           |
+----------------------+
2 rows in set (0.03 sec)
```

Note: If you annotate abstract classes with the *@hibernate.class* tag, then a table is created but not used by Hibernate.

5. Simple Queries

Hibernate offers a powerful, SQL like, query language called HQL (Hibernate Query Language). To specify bounds (maximum results, position in result rows) you can obtain an instance of the `net.sf.hibernate.Query`. And finally there is the new Criteria API which can also be used to query the database.
I will not go into the details of either of query possibility. There are several chapters in the Hibernate documentation. I just show some simple queries we can use to check if our Classes really end up in the database.

```
In tutorial.ineritance.tableperhierarhcy.InheritanceDemo.java
tx = sess.beginTransaction();

List animals = sess.find("from " + Animal.class.getName());
for (Iterator it = animals.iterator(); it.hasNext();) {
    Animal animal = (Animal) it.next();
    System.out.println( "Animal '" + animal.getName() +
        "' its class is: " + animal.getClass().getName());
```

 pascal@downside.ch

```
    System.out.print("Makes sound: ");
    animal.makeSound();
}

tx.commit();
```

The interesting thing is that we do not mention a table name or a column name in the query. We just ask for all instances of `Animal` class.

You can get all `Cow` instances by changing the query to "`from tutorial.inheritance.tableperhierarchy.Cow`" and so on.

6. Composition

A `Person` is composed of a name, address and other properties. From the OO Design perspective it is a good decision to have a separate `Address` class and have the `Person` hold a reference to an `Address` instance. You can use *`@hibernate.component`* to map all the `Address` properties to the `Person` table.

tutorial.component.Address

```
package tutorial.component;

public class Address {
      private String country;
      private String city;
      private String street;
      private int number;

      public Address() {}
      public Address(String country, String city,
                     String street, int number) {
            this.country = country;
            this.city = city;
            this.street = street;
            this.number = number;
      }

      /**
       * @hibernate.property
       */
      public String getCity() {return city;}
      public void setCity(String city) {this.city = city;}
      /**
       * @hibernate.property
       */
      public String getCountry() {return country;}
      public void setCountry(String country) {this.country = country;}
      /**
       * @hibernate.property
       */
      public int getNumber() {return number;}
      public void setNumber(int number) {this.number = number;}
      /**
       * @hibernate.property
       */
```

 pascal@downside.ch

```
	public String getStreet() {return street;}
	public void setStreet(String street) {this.street = street;}

}
```

In the `Address` we just mapped the properties with *@hibernate.property*, no *@hibernate.class* is required for a component. Components do not have an id property themselves.

Note: Since I added a convenience constructor which takes some arguments, I had to add a no-arguments-constructor too. Otherwise Hibernate could not instantiate Objects from this class.

`tutorial.component.Person.java`

```
package tutorial.component;
/**
 * @hibernate.class
 */
public class Person {
	private Long id;
	private String username;
	private Address address;

	/**
	 * @hibernate.id
	 *    generator-class="hilo"
	 *    unsaved-value="null"
	 */
	public Long getId() {return id;}
	public void setId(Long id) {this.id = id;}
	/**
	 * @hibernate.property
	 *    length="15"
	 *    unique="true"
	 *    not-null="true"
	 */
	public String getUsername() {return username;}
	public void setUsername(String username) {
	    this.username = username;
	}
	/**
	 *
	 * @hibernate.component
	 */
	public Address getAddress() {return address;}
	public void setAddress(Address address) {this.address = address;}
}
```

The `Address` has been mapped with *@hibernate.component*, the username has been constrained with *unique="true" and not-null="true"* to null or duplicate usernames.

```
mysql> describe person;
+----------+-------------+------+-----+---------+-------+
```

 pascal@downside.ch

```
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| id       | bigint(20)   |      | PRI | 0       |       |
| username | varchar(15)  |      | UNI |         |       |
| city     | varchar(255) | YES  |     | NULL    |       |
| country  | varchar(255) | YES  |     | NULL    |       |
| number   | int(11)      | YES  |     | NULL    |       |
| street   | varchar(255) | YES  |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

The composition has been "flattened" and all properties are stored in the same table. Composition is not restricted to one level; the `Address` could again be composed of several components.

7. Many-to-One

A many-to-one relation is just a simple object reference. In the example code we have an employees having a boss. The relation between the employees and the boss is a many-to-one relation (many employees can have the same boss)

tutorial.manytoone.Employee.java

```
package tutorial.manytoone;
/**
 * @hibernate.class
 *     table="emp"
 */
public class Employee {
      private String id;
      private Employee boss;
      private String name;

      public Employee() {}
      public Employee(String name) {
            this.name = name;
      }
      /**
       * @hibernate.id
       *     column="ID"
       *     generator-class="uuid.hex"
       *     unsaved-value="null"
       */
      public String getId() {return id;}
      public void setId(String id) {this.id = id;}
      /**
       * @hibernate.many-to-one
       */
      public Employee getBoss() {return boss;}
      public void setBoss(Employee boss) {this.boss = boss;}
      /**
       * @hibernate.property
       */
      public String getName() {return name;}
      public void setName(String name) {this.name = name;}
}
```

 pascal@downside.ch

To show another id generator strategy we use the *uuid.hex* generator class. The boss, also of class `Employee`, is mapped with the *@hibernate.many-to-one* tag. If you store an `Employee` with a boss who is not yet stored, Hibernate will throw an Exception.

```
net.sf.hibernate.TransientObjectException: object references an unsaved
transient instance - save the transient instance before flushing:
tutorial.manytoone.Employee
      at
net.sf.hibernate.impl.SessionImpl.throwTransientObjectException(Session
Impl.java:2764)
```

So either you store the boss before you store his `Employee` or you specify a cascade mode of *"save-update".*

```
In hibernate.manytoone.Employee.java
/**
 * @hibernate.many-to-one
 *     cascade="save-update"
 */
public Employee getBoss() {return boss;}
public void setBoss(Employee boss) {this.boss = boss;}
```

Like this Hibernate stores the boss if it is not yet stored. In the following code snippet you see how Employees and their boss are stored. Since we specify *cascade="save-update"* for the employee-boss relation, we do not need to store the boss explicitly.

```
tx = sess.beginTransaction();

Employee marcellus = new Employee("Marcellus Wallace");

Employee vince = new Employee("Vincent Vega");
vince.setBoss(marcellus);

Employee jules = new Employee("Jules Winnfield");
jules.setBoss(marcellus);

sess.save(vince);
sess.save(jules);

tx.commit();
```

If we check the database, we see that Marcellus was inserted only once. The *uuid.hex* id is quite long, that is why I selected only the boss and name column

```
mysql> select boss, name from emp;
+----------------------------------+-------------------+
| boss                             | name              |
+----------------------------------+-------------------+
| NULL                             | Marcellus Wallace |
| 402881e5fd42fe9400fd42fe9d5d0002 | Vincent Vega      |
| 402881e5fd42fe9400fd42fe9d5d0002 | Jules Winnfield   |
```

 pascal@downside.ch

```
+----------------------------------+-------------------+
3 rows in set (0.13 sec)
```

Note: Be careful when specifying cascade. If you specify `delete` or `all` as cascade value and delete an Employee, you will not only delete the Employee but also his boss.

8. One-to-One

Hibernate can map one-to-one relationships in two ways, with a primary key or with a foreign key association.

8.1. Primary key association

With the primary key association the two associated objects have the same primary key.

tutorial.onetoone.Teacher.java

```
package tutorial.onetoone;
/**
 * @hibernate.class
 */
public class Teacher {
	private Long id;
	private String name;
	private HomeAddress address;

	public Teacher() {}
	public Teacher(String name, HomeAddress address) {
		this.name = name;
		this.address = address;
	}
	/**
	 * @hibernate.id
	 *	generator-class="foreign"
	 * @hibernate.generator-param
	 *	name="property"
	 *	value="homeAddress"
	 */
	public Long getId() {return id;}
	public void setId(Long id) {this.id = id;}
	/**
	 * @hibernate.property
	 */
	public String getName() {return name;}
	public void setName(String name) {this.name = name;}
	/**
	 * @hibernate.one-to-one
	 *	cascade="all"
	 */
	public HomeAddress getHomeAddress() {return address;}
	public void setHomeAddress(HomeAddress address) {
	  this.address = address;
	}

	public String toString() {
		return "[Name: " + name + " / Address: " + address+"]";
	}
```

 pascal@downside.ch

```
}
```

The `HomeAddress` is mapped with the *@hibernate.one-to-one* tag, I specifed *cascade="all"* so the store, update and delete operations are cascaded from the Teacher object to the `HomeAddress`. The most important thing is the way we create the id for `Teacher`. The *generator-class="foreign"* with the *@hibernate.generator-param* specifies where to take the id from.

When we check the tables created after we generated the mapping files and exported the schema, we can see that no additional column was inserted for the association but `Teacher` and `HomeAddress` have the same id.

```
mysql> select teacher.id, teacher.name, homeaddress.id from teacher,
homeaddress;
+----+---------+----+
| id | name    | id |
+----+---------+----+
|  1 | Someone |  1 |
+----+---------+----+
```

8.2. Foreign key association

The other strategy, mapping with a foreign key, is the same as a many-to-one relationship trough *@hibernate.many-to-one* with *unique="true"*.

```
In tutorial.onetoone.HomeAddress
/**
 * @hibernate.many-to-one
 *     cascade="all"
 *     unique="true"
 */
public Street getStreet() {return street;}
public void setStreet(Street street) {this.street = street;}
```

We can see that the `Homeaddress` has a Column for the id of Street

```
mysql> select homeaddress.id, homeaddress.street, street.id,
street.name from homeaddress, street;
+----+--------+-------+------------+
| id | street | id    | name       |
+----+--------+-------+------------+
|  1 |  32769 | 32769 | somestreet |
+----+--------+-------+------------+
1 row in set (0.00 sec)
```

9. Collection Mapping

9.1. One-to-Many relation

You can persist `Set`, `List` or arrays of persistent entities. A column (pointing to the owner of the collection) is added to the table of the many side by Hibernate. If you persist a `List` or an array of persistent entities, you'll need an additional index column to keep the ordering.

 pascal@downside.ch

One Side	
id	other cols
1	...
2	...

Many Side		
id	other cols	fk one
88	...	1
89	...	1

Figure 4: Mapping a Set

One Side	
id	other cols
1	...
2	...

Many Side (indexed)			
id	other cols	index	fk one
88	...	1	1
89	...	2	1

Figure 5: Mapping a List or array

Note: Hibernate replaces `List`, `Set` and `Map` with its own implementations. Therefore you must always work with the interfaces and not with classes (e.g. use `java.util.List` not `java.util.ArrayList`).

Here is the example class `One`, which represents the one side of the one-to-many relation. As usual it has an id attribute of type Long. In addition there is a `Set`, a `List` and an array of `ManyInArray` objects. I implemented three classes for the many side of the relation (`ManyInArray, ManyInSet, ManyInList`) so they will be stored in three different tables.

tutorial.ometomany.One

```
package tutorial.onetomany;

import java.util.List;
import java.util.Set;
/**
 * @hibernate.class
 */
public class One {
	private Long id;

	private List list;
	private Set set;
	private ManyInArray[] array;
	/**
	 * @hibernate.id
	 *     column="ID"
	 *     generator-class="hilo"
	 *     unsaved-value="null"
	 */
	public Long getId() {return id;}
	public void setId(Long id) {this.id = id;}

	/**
	 * @hibernate.array
	 *     cascade="save-update"
	 * @hibernate.collection-one-to-many
	 *     class="tutorial.onetomany.ManyInArray"
	 * @hibernate.collection-index
```

 pascal@downside.ch

```
         *    column="MANY_INDEX"
         * @hibernate.collection-key
         *    column="FK_ONE"
         */
        public ManyInArray[] getArray() {return array;}
        public void setArray(ManyInArray[] array) {this.array = array;}

        /**
         * @hibernate.list
         *    cascade="save-update"
         * @hibernate.collection-one-to-many
         *    class="tutorial.onetomany.ManyInList"
         * @hibernate.collection-index
         *    column="MANY_INDEX"
         * @hibernate.collection-key
         *    column="FK_ONE"
         */
        public List getList() {return list;}
        public void setList(List list) {this.list = list;}

        /**
         * @hibernate.set
         *    cascade="save-update"
         * @hibernate.collection-one-to-many
         *    class="tutorial.onetomany.ManyInSet"
         * @hibernate.collection-key
         *    column="FK_ONE"
         */
        public Set getSet() {return set;}
        public void setSet(Set set) {this.set = set;}
}
```

Mapping one-to-many relations is a little trickier than many-to-one and requires several tags:
One of *@hibernate.array*, *@hibernate.set* or *@hibernate.list* is required to indicate the type of the collection. In addition you need to specify the relation type, in this case *@hibernatet.collection-one-to-many* and the *@hibernate.collection-key* with *column* attribute. If you persist a `List` or an array, then you must specify the *@hibernate.collection-index* so the collection can be kept in the correct order. The many side of the relation (`ManyInSet, ManyInList, ManyInArray`) needs to be mapped too. Here is the important part of ManyInSet as a representatice example.

```
A snippet of tutorial.onetomany.ManyInSet
package tutorial.onetomany;
/**
 * @hibernate.class
 *    table="manyset"
 */
public class ManyInSet {
        private Long id;
        private String description;

        public ManyInSet() {}
```

 pascal@downside.ch

```
        public ManyInSet(String description) {
              this.description = description;
        }
        /**
         * @hibernate.id
         *      column="ID"
         *      generator-class="native"
         *      unsaved-value="null"
         */
        public Long getId() {return id;}
        public void setId(Long id) {this.id = id;}
```

Add a `@hibernate.class` and a `@hibernate.id` tag, no special tags are required for the relation.

The tables created by Hibernate look like this:

```
mysql> describe one;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| ID    | bigint(20) |      | PRI | 0       |       |
+-------+------------+------+-----+---------+-------+
1 row in set (0.00 sec)
```

No column is added for the one-to-many relations in the table for `One. The table for ManyInArray and ManyInList` look the same while `ManyInSet` does not have the column for the collection index.

```
mysql> describe manyarray;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| ID          | bigint(20)   |      | PRI | NULL    | auto_increment |
| description | varchar(255) | YES  |     | NULL    |                |
| FK_ONE      | bigint(20)   | YES  | MUL | NULL    |                |
| MANY_INDEX  | int(11)      | YES  |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

No index column is required if the collection is a `Set`.

```
mysql> describe manyset;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| ID          | bigint(20)   |      | PRI | NULL    | auto_increment |
| description | varchar(255) | YES  |     | NULL    |                |
| FK_ONE      | bigint(20)   | YES  | MUL | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

 pascal@downside.ch

9.2. Lazy Initialization

Even though lazy initialization is more related to Hibernate than to XDoclet, I will write a few words about it.

Hibernate supports lazy initialization of collection elements. Only when an element is accessed, then the data is loaded. Currently lazy loading is only supported for collections and not for simple properties like Strings or references to other persistent entities.

In tutorial.lazy.Orderlist.java

```
/**
 * @hibernate.list
 *     cascade="save-update"
 *   lazy="true"
 * @hibernate.collection-one-to-many
 *     class="tutorial.lazy.Product"
 * @hibernate.collection-index
 *     column="PRODUCT_INDEX"
 * @hibernate.collection-key
 *     column="FK_ORDER"
 */
public List getOrderedProducts() {return orderedProducts;}
public void setOrderedProducts(List orderedProducts) {
  this.orderedProducts = orderedProducts;
}
```

The *lazy="true"* tells Hibernate to load the elements of the collection as required. This works only as long as the Session is open, after the Session was closed access to the collection will result in a LazyInitializationException If you want to access elements of a lazy initialized collection after the Session was closed, then you can call the size() method of the collection before you close the Session.

In tutorial.lazy.LazyDemo.java

```
tx = sess.beginTransaction();

loadedAndInit = (Orderlist) sess.load(Orderlist.class, order.getId());
// the call to size() will initialize the collection
loadedAndInit.getOrderedProducts().size();

tx.commit();
```

Lazy loading is the reason Hibernate uses it's own implementations of the List, Set and Map interfaces. Take a look at tutorial.hibernate.lazy.LazyDemo.java to find a working example.

9.3. Many-to-Many

The many-to-many example consists of Course and Student objects. Every Student holds a List of Course objects (you can check the one-to-many section to see how a Set or a Course[] would be mapped). Many-to-many relations are stored with a

 pascal@downside.ch

separate mapping table, consisting of foreign keys to the two classes and an optional index table.

Student in Course		
fk_student	fk_course	index
1	55	1
1	56	2

Student	
id	name
1	fabien
2	pascal

Course	
id	description
55	CS
56	Spanish

Figure 6: Many-to-many mapping with index

A `Course` just contains an id and a description.

tutorial.manytomany.Course.java

```
package tutorial.manytomany;

/**
 * @hibernate.class
 */
public class Course {
	private Long id;
	private String description;

	public Course(){}
	public Course(String description) {
		this.description = description;
	}
	/**
	 * @hibernate.id
	 *     column="ID"
	 *     generator-class="hilo"
	 *     unsaved-value="null"
	 */
	public Long getId() {return id;}
	public void setId(Long id) {this.id = id;}
	/**
	 * @hibernate.property
	 *     type="text"
	 */
	public String getDescription() {return description;}
	public void setDescription(String description) {
	    this.description = description;
	}
}
```

There is not much to discover here. The *@hibernate.property* is complemented with the *type="text"*. For MySQL this results in following table:

```
mysql> describe course;
+-------------+------------+------+-----+---------+-------+
```


pascal@downside.ch

```
| Field       | Type       | Null | Key | Default | Extra |
+-------------+------------+------+-----+---------+-------+
| ID          | bigint(20) |      | PRI | 0       |       |
| description | text       | YES  |     | NULL    |       |
+-------------+------------+------+-----+---------+-------+
2 rows in set (0.13 sec)
```

The `Student` class has an id, the `List` of courses and a name:

tutorial.manytomany.Student

```
package tutorial.manytomany;
import java.util.List;

/**
 * @hibernate.class
 */
public class Student {
	private Long id;
	private List courses;
	private String name;

	public Student() {}

	public Student(String name) {
		this.name = name;
	}

	/**
	 * @hibernate.id
	 *    column="ID"
	 *    generator-class="hilo"
	 *    unsaved-value="null"
	 */
	public Long getId() {return id;}
	public void setId(Long id) {this.id = id;}
	/**
	 * @hibernate.list
	 *    table="students_in_course"
	 *    cascade="save-update"
	 * @hibernate.collection-many-to-many
	 *    column="fk_course"
	 *    class="tutorial.manytomany.Course"
	 * @hibernate.collection-key
	 *    column="fk_student"
	 * @hibernate.collection-index
	 *    column="course_index"
	 */
	public List getCourses() {return courses;}
	public void setCourses(List courses) {this.courses = courses;}

	/**
	 * @hibernate.property
	 */
	public String getName() {return name;}
	public void setName(String name) {this.name = name;}
}
```

 pascal@downside.ch

The *@hibernate.list* specifies the mapping table with *table="students_in_course"* and cascade="save-update" so we do not need to store `Course` objects explicelty. *@hibernate.collection-many-to-many* identifies the class that is contained in the collection as well as the name of the column with the foreign key. As in the one-to-many relationship the *@hibernate.collection-key* defines the column name for the `Student` foreign key. *@hibernate.collection-index* is only required for arrays of persistent entities or `List`.

9.4. Bidirectional Parent Child

Imagine you need to store information that is represented as a tree (e.g. a family tree, the class hierarchy of j2se and so on). Each parent has a list or set of child objects, each child object knows its parent. This is represented in a bidirectional (from parent to child and vice versa) parent child relationship.

```
A part of tutorial.parentchild.ParentObject.java
package tutorial.parentchild;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
/**
 * @hibernate.class
 */
public class ParentObject {

      /**
       * @hibernate.list
       *    cascade="save-update"
       *    inverse="true"
       * @hibernate.collection-one-to-many
       *    class="tutorial.parentchild.ChildObject"
       * @hibernate.collection-index
       *    column="child_index"
       * @hibernate.collection-key
       *    column="parent_id"
       */
      public List getChildren() {return children;}
      public void setChildren(List children) {
          this.children = children;
      }


}
```

```
A part of tutorial.parentchild.ChildObject.java
package tutorial.parentchild;

/**
 * @hibernate.class
 */
public class ChildObject {
```

 pascal@downside.ch

```
        /**
         * @hibernate.many-to-one
         *     cascade="save-update"
         *     column="parent_id"
         */
        public ParentObject getParent() {return parent;}
        /**
         * @param parent The parent to set.
         */
        public void setParent(ParentObject parent) {
            this.parent = parent;
        }
        /**
         * @hibernate.property
         *     column="child_index"
         */
        public int getIndex() {
               return parent.getChildren().indexOf(this);
        }
        public void setIndex(int index) {
               // not needed, calculated property
        }

}
```

I cut out some parts (getters/setters, custom toString() method, convenience methods) to save some space and prevent repeating code you have seen before. You can find the full source code in `hibernatetutorial/src/tutorial/parentchild/Parent.java` and `hibernatetutorial/src/tutorial/parentchild/Child.java`. The `ParentObject` class has a `List` of `ChildObject` objects which is mapped with a *`@hibernate.one-to-many`* tag. We use inverse="true" as its recommended for most bidirectional associations. The `ChildObject` has a reference to its parent which is mapped with a *`@hibernate.many-to-one`* tag. The calculated property `index` is required since Hibernate does not support bidirectional one-to-many associations with *`inverse="true"`* for indexed collections (see also the notes below). The property is mapped to *`column="child_index"`* which is the same column as we specified for the *`@hibernate.collection-index`* in the ParentObject. Of course this is not required if you are using an unindexed collection (e.g. `Set`)

Note: There are some good tutorials on the Hibernate website explaining the inverse setting, bidirectional one-to-many with indexed collections and parent child relations.
Inside explanation of inverse="true": http://www.hibernate.org/155.html
Some Parent-Child Principles: http://www.hibernate.org/209.html
Bidirectional one-to-many with an indexed collection: http://www.hibernate.org/193.html

10. Composite Keys

Composite keys are supported by hibernate but you are discouraged from using them (you can not use the hibernate key generators, you need an extra class for the key). The sample code for these examples is in the `tutorial.compositeid` package.

 pascal@downside.ch

For all your composite primary keys you need to write a class combining the different elements of the key.

```
tutorial.compositeid.CompositeId
```

```java
package tutorial.compositeid;
import java.io.Serializable;
public class CompositeId implements Serializable {
	private int partA;
	private int partB;

	public CompositeId() {}
	public CompositeId(int partA, int partB) {
		this.partA = partA;
		this.partB = partB;
	}

	/**
	 * @hibernate.property
	 */
	public int getPartA() {return partA;}
	public void setPartA(int partA) {this.partA = partA;}
	/**
	 * @hibernate.property
	 */
	public int getPartB() {return partB;}
	public void setPartB(int partB) {this.partB = partB;}

	public boolean equals(Object o) {
		// must override equals()
		if (o == null) {
			return false;
		} else if (o == this) {
			return true;
		} else if (o.getClass() == CompositeId.class) {
			CompositeId other = (CompositeId)o;
			return this.partA == other.partA &&
			       this.partB == other.partB;
		} else {
			return false;
		}
	}
	public int hashCode() {
		// must override hashCode()
		return partA * partB;
	}
}
```

The class `CompositeId` is used to represent a primary key composed of two `int` fields. The fields partA and partB are simply marked with a *@hibernate.property* tag, no *@hibernate.class* tag is required.

Note: there are a couple of rules for a composite key class:
- Must override `equals()`
- Must override `hashCode()`

 pascal@downside.ch

- Must implement `java.io.Serializable`

Now this key can be used in mapped entities:

```
Part of tutorial.compositeid.Comp
/**
 * @hibernate.id
 *     unsaved-value="any"
 */
public CompositeId getId() {return id;}
public void setId(CompositeId id) {this.id = id;}
```

The *@hibernate.id* tag is used as with `Long/String/…` ids but no *generator-class* is used and for the *unsaved-value* only "any" or "none" are allowed.

Mapping a collection also requires some special tags:

```
Part of tutorial.compositeid.Comp
/**
       * @hibernate.set
       *    cascade="all"
       * @hibernate.collection-one-to-many
       *    class="tutorial.compositeid.FooBar"
       * @hibernate.collection-key
       * @hibernate.collection-key-column
       *    name="partA"
       * @hibernate.collection-key-column
       *    name="partB"
       */
      public Set getFooBars() {return fooBars;}
      public void setFooBars(Set fooBars) {this.fooBars = fooBars;}
```

One *@hibernate.collection-key* und two *@hibernate.collection-key-column* (one for partA, one for partB) tags are required.

11. Appendix

11.1. Links

Hibernate: http://www.hibernate.org
XDoclet: http://xdoclet.sourceforge.net
PostgreSQL: http://www.postgresql.org/
Ant: http://ant.apache.org
MySQL: http://www.mysql.com/
Eclipse: http://www.eclipse.org

11.2. IDE Hints

Make sure the generated mapping files and the Hibernate property file are on the classpath. Otherwise Hibernate can not load them and you'll be faced with some pretty exceptions.

Some more files which are required on the classpath:
`hibernatetutorial/lib/hibernate/*.jar`,
`hibernatetutorial/lib/xdoclet/*.jar`,
`hibernatetutorial/lib/jdbc/*.jar` (only the MySQL JDBC driver is in this distribution)

11.3. Running the examples with Ant

Even though it is not verry thrilling to watch the example code do its work you can easely run the examples using Ant. Go to the console, change to the `hibernatetutorial` directory and run following command:

```
ant –Ddemo=<classname> rundemo
```

Replace `<class name>` with the fully qualified name of the class you want to run. Valid classes are:

- `tutorial.basic.BasicDemo`
- `tutorial.component.ComponentDemo`
- `tutorial.compositeid.CompositeIdDemo`
- `tutorial.inheritance.tableperconcreteclass.InheritanceDemo`
- `tutorial.inheritance.tableperhierarchy.InheritanceDemo`
- `tutorial.inheritance.tablepersubclass.InheritanceDemo`
- `tutorial.lazy.LazyDemo`
- `tutorial.manytomany.ManyToManyDemo`
- `tutorial.manytoone.ManyToOneDemo`
- `tutorial.onetoone.OneToOneDemo`
- `tutorial.onetomany.OneToManyDemo`
- `tutorial.parentchild.ParentChildDemo`
- `tutorial.projection.PropjectionDemo`
- `tutorial.timestamp.TimestampedDemo`
- `tutorial.version.VersionedDemo`

11.4. Log levels

There are two places where you can configure logging. One is the resources/hibernate.properties where you can turn on or off the SQL statement logging:

```
hibernate.show_sql=false
```

The other one is `resources/log4j.properties` where you can play with the logging levels of the different packages. E.g. if you want to see what parameter hibernate binds to the statements, then change the level for
`log4j.logger.net.sf.hibernate.type` from `info` to `debug`

 pascal@downside.ch

11.5. Mapping Cheatsheet

Table per hierarchy

Mapping in superclass

```
/**
 *@hibernate.class
 *@hibernate.discriminator
 *    column="discriminator"
 */
```

Mapping in subclass

```
/**
 * @hibernate.subclass
 */
```

Table per subclass

Mapping in superclass

```
/**
 *@hibernate.class
 */
```

Mapping in subclass

```
/**
 * @hibernate.joined-subclass
 * @hibernate.joined-subclass-key
 *    column="pet"
 */
```

Table per concrete class

Mapping in concrete classes

```
/**
 * @hibernate.class
 */
```

One-to-One with primary key

The primary key of the one-to-one property is used -> use foreign key id generator

```
/**
 * @hibernate.id
 *    generator-class="foreign"
 * @hibernate.generator-param
 *    name="property"
 *    value="name of one-to-one
 *           property"
 */

/**
 * @hibernate.one-to-one
 */
```

One-to-One with foreign key

Represented with many-to-one

```
/**
 * @hibernate.many-to-one
 *    unique="true"
 */
```

 pascal@downside.ch

One-to-Many

If collection is not indexed (e.g. Set) then drop the @hibernate.collection-index

```
/**
 * @hibernate.list
 *     cascade="save-update"
 * @hibernate.collection-one-to-many
 *     class="class in collection"
 * @hibernate.collection-index
 *     column="MANY_INDEX"
 * @hibernate.collection-key
 *     column="FK_ONE"
 */
```

Many-to-Many

If collection is not indexed (e.g. Set) then drop the @hibernate.collection-index

```
/**
 * @hibernate.list
 *     table="students_in_course"
 *     cascade="save-update"
 * @hibernate.collection-many-to-many
 *     column="fk_course"
 *     class="tutorial.manytomany.Course"
 * @hibernate.collection-key
 *     column="fk_student"
 * @hibernate.collection-index
 *     column="course_index"
 */
```

Component

Component owner.

```
/**
 * @hibernate.component
 */
```

Component: No id property required. Just map the persistent properties as in any other class.

```
/**
 * @hibernate.property
 */
```

Property

A property (e.g. String, char, Long, …)

```
/**
 * @hibernate.property
 *      type="a type"
 *      length="10"
 */
```

Specify column details

```
/**
 * @hibernate.column
 *      sql-type="a type"
 *      column="foo"
 */
```

ID

```
/**
 * @hibernate.id
 *     column="ID"
 *     generator-class="hilo"
 *     unsaved-value="null"
 */
```


pascal@downside.ch

<table>
<tr><th colspan="2">Composite ID</th></tr>
<tr><td colspan="2">The ID property</td></tr>
<tr><td colspan="2"><pre>/**
 * @hibernate.id
 *   column="ID"
 *   unsaved-value="any"
 */</pre></td></tr>
<tr><td colspan="2">When mapping collections</td></tr>
<tr><td colspan="2"><pre>* @hibernate.collection-key
* @hibernate.collection-key-column
*   name="column_1"
* @hibernate.collection-key-column
*   name="column_2"</pre></td></tr>
<tr><th colspan="2">Optimistic Locking</th></tr>
<tr><td>Version property (recommended).</td><td>Timestamp</td></tr>
<tr><td><pre>/**
 * @hibernate.version
 *   unsaved-value="null"
 */</pre></td><td><pre>/**
 * @hibernate.timestamp
 *   unsaved-value="null"
 */</pre></td></tr>
</table>

pascal@downside.ch