Guide to Java Persistence and Hibernate

Sebastian Hennebrueder

Guide to Java Persistence and Hibernate

Sebastian Hennebrueder

Table of Contents

List of Figures

List of Tables

About the author, book and versions

Revision Mai 2013

1. The author

Well, my name is Sebastian Hennebrueder. I am a freelancer and work as trainer, software developer and architect. I am located at Bad Vilbel, a small town close to Frankfurt in Germany. I am passionated about technologies, publish articles and participate in conferences like JavaOne in San Francisco or Devoxx in Antwerpen.

You can contract me for your projects.

Hibernate, Java Persistence and JEE training

I offer basic and advanced training in Europe, the training can happen onsite in your company but you will find as well a regular schedule for Frankfurt, Germany on my website

http://www.laliluna.de

With more than 25 trainings in Europe in the last 3 years, I have a solid experience. I can teach in German, English and French.

Software development

I am a freelancer and you can contract me as software developer and/or architect. I have a vast experience as software developer and architect in Hibernate / Java Persistence, Jboss Seam, Spring, GWT and a number of other web framework.

2. The book

I want to provide you with an easy-to-follow introduction to Java Persistence and Hibernate including not only code snippets but complete working examples. For this reason the guide is based on the following principles:

Explanations with complete working examples Whenever something is explained you will find a complete sample application. If a specific mapping is demonstrated you can find a sample application, too. It shows how to insert, update, delete or query the classes included in this mapping. Larger sample applications show the implementation of real applications using Hibernate in detail. Each application has an increasing complexity. Cache configurations, deployment configurations for particular application server rounds off the examples. Altogether the book provides about 30 mapping examples + variations, 4 real projects and some smaller projects showing a specific configuration.

Potential problems are investigated in detail Hibernate has some problem zones where you can easily make mistakes. These are covered in detail to help you to avoid these problems. Some topics are LazyLoading, Session and transaction handling, optimistic concurrency and version handling.

Best Practices One chapter is completely focussing on how to implement well designed Hibernate applications. DAO patterns and business layers are discussed. Pros and cons and the logic behind this structure are explained.

Focus on important features Deliberately, I did not explain some rare concepts which I consider to be either not stable or very rarely used. Instead I referred to the corresponding chapters of the Hibernate reference - the documentation available with Hibernate. In my opinion the Hibernate reference is far too complex for people learning Hibernate. However, it is a valuable resource for people who already know Hibernate.

What this book is not… The texts are short and frequently you are referred to parts of sample applications. If you prefer larger narrative parts you should not read this book. Feel free to contact me and to comment on the book in the forum at http://www.laliluna.de

Best Regards / Viele Grüße

Sebastian Hennebrueder

3. Library Versions

This book covers Java Persistence 2 and Hibernate 4.

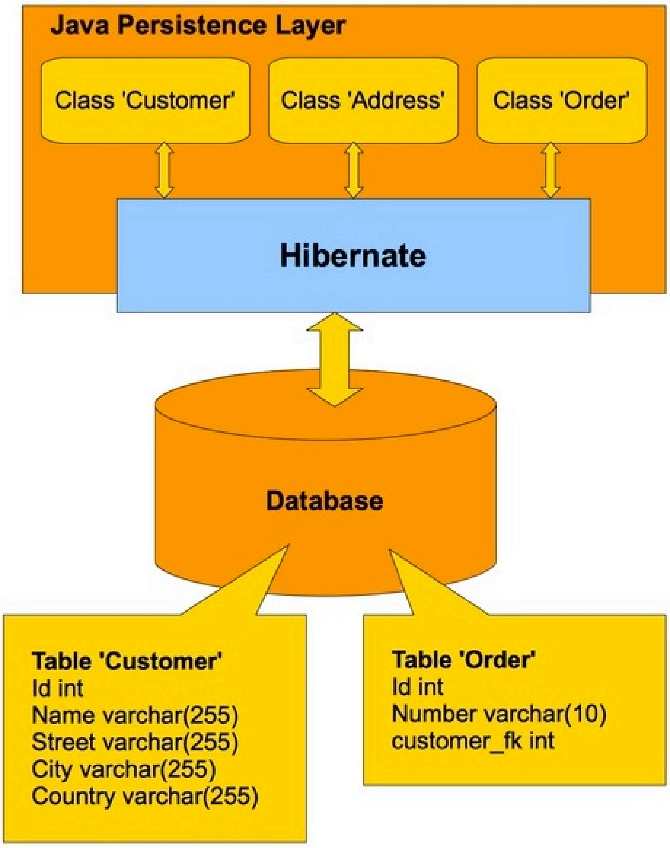
Part I. Introduction

Chapter 1. Introduction to Hibernate

Hibernate is a solution for object relational mapping and a persistence management solution or persistence layer. This is probably not understandable for anybody learning Hibernate.

What you can imagine is probably that you have your application with some functions (business logic) and you want to save data in a database. Using Java, the business logic normally works with objects of different class types. Your application is object-oriented. Your database tables are not object oriented but relational.

The basic idea of object relation mapping - ORM is to map database tables to a class. One row of the database data is copied to a class instance. On the other hand, if an object is saved, one row is inserted in the database table.



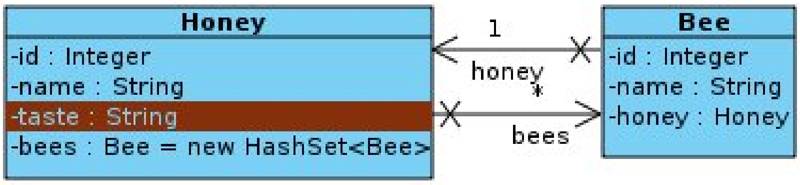
Saving data to a storage is called persistence. The process of copying of table rows to objects and vice versa is called object relational mapping.

1.1. A first Hibernate example

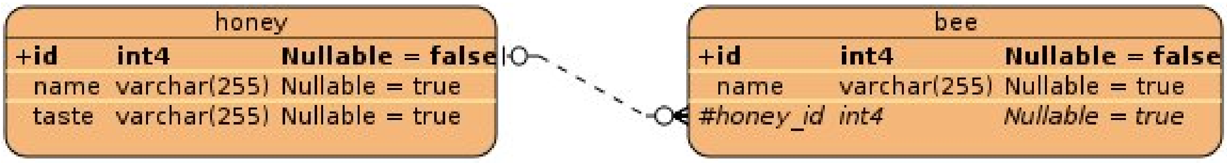
As it is easier to explain something after having shown a real example, we will develop our first Hibernate application now.

1.1.1. Introduction

Our use case is sweet. We will use an example from the honey production. You need a lot of bees to get honey. Our use case includes Honey and Bee as model and a 1:n relation from Bee to Honey. Using object oriented terms: There is a 1:n association between the class Bee and Honey.



We will create a class Honey which is mapped to the table honey in the database and a class Bee mapped to the table bee. The following picture shows our database schema.


honey
+id int4 Nullable = false
name varchar(255) Nullable = true
taste varchar(255) Nullable = true
bee
+id int4 Nullable = false
name varchar(255) Nullable = true
#honey_id int4 Nullable = true

In order to get complete our application, we must undertake the following steps:

- create classes
- create mapping from classes to tables
- configure Hibernate libraries
- configure the Hibernate datasource
- write code to use our mapped classes

Source code

For this example you will find the complete source code in the ebook. Normally, I show only the relevant part of the source code in the ebook. But you can find the complete source code in the provided sources.

1.1.2. Creating Java Project and classes

Create a new project using your development environment. I used Eclipse for this example. Using Eclipse press the keys Ctrl+n (Strg+n) to create a new project. Select Java project. We will call it FirstHibernateExample.

Create a new class named *Honey* in the package *de.laliluna.example*. Our class has four fields:

- Integer id – an Integer value as primary key
- String name – name of the honey

- String taste – description of the taste

- java.util.Set<Bee> bees – bees, having produced the honey

Furthermore we need:

- Getter and setter methods for our fields. In eclipse you can generate them (Context menu → Source → Generate Getter and Setter).

- A default constructor

- Implementation of the Serializable interface

- Overwrite the toString method. We will need it for debugging.

Java version: I used Java generics, which was introduced with Version 5 (alias 1.5).

Annotations always starts with @.

Your source code should look like the code below:

Honey class.

```
package de.laliluna.example;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

public class Honey implements Serializable {

   private Integer id;
   private String name;
   private String taste;
   private Set<Bee> bees = new HashSet<Bee>();

   public Honey() {
   }
   public Honey(String name, String taste) {
      this.name = name;
      this.taste = taste;
   }
   public Integer getId() {
      return id;
   }
   public void setId(Integer id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getTaste() {
      return taste;
   }
   public void setTaste(String taste) {
      this.taste = taste;
   }
   public Set<Bee> getBees() {
      return bees;
   }
   public void setBees(Set<Bee> bees) {
      this.bees = bees;
   }
   public String toString() {
      return "Honey: " + getId() + " Name: " + getName() + " Taste: "
            + getTaste();
   }
}
```

Requirements of domain classes

There are a couple of requirements for a mapped class, you should always consider to meet:

- An id property
- A default constructor (= no parameter) with at least protected scope. Either omit the constructor or create an empty one.
- Implementation of the Serializable interface, if and only if you want to serialize your entity. This may happen indirectly if you put an entity for example into a HTTPSession. Hibernate does not require the *Serializable* interface in entities.
- A useful toString method for debugging. (recommendation)

Create a class Bee with the fields:

- Integer id
- String name
- Honey honey

The field id is once again the primary key. Please don’t forget the important requirements, I stated above.

Bee class.

```
package de.laliluna.example;

import java.io.Serializable;
import java.text.MessageFormat;

public class Bee implements Serializable {

   private Integer id;
   private String name;
   private Honey honey;

   public Bee() {
   }
   public Bee(String name) {
      this.name = name;
   }
   public Integer getId() {
      return id;
   }
   public void setId(Integer id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public Honey getHoney() {
      return honey;
   }
   public void setHoney(Honey honey) {
      this.honey = honey;
   }
   public String toString() {
      return MessageFormat.format("{0}: id={1}, name={2}", new Object[] {
            getClass().getSimpleName(), id, name });
   }
}
```

1.1.3. Hibernate configuration

The Hibernate configuration will define

- which database we connect to
- the type of database (MySQL, PostgreSQL, Oracle, …)
- Hibernate configuration settings
- classes or XML mapping files holding our mappings.

Create a new file named *hibernate.cfg.xml* in your src directory.

1.1.4. Annotation or XML

You have to choices to map documents: Annotations and XML. They’ll be discussed later in this book. I explain both approaches for this example.

If you can use Java 1.5 alias 5 or later, I recommend to use annotations.

Below you can find a configuration for PostgreSQL using annotation based mapping. Afterwards, I will explain required changes for other databases and XML based mapping. Do not forget to change the username and the password to suit your database configuration.

hibernate.cfg.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <!--  postgre SQL configuration-->
      <property name="connection.url">
         jdbc:postgresql://localhost:5432/learninghibernate
      </property>
      <property name="connection.username">postgres</property>
      <property name="connection.password">p</property>
      <property name="connection.driver_class">
         org.postgresql.Driver
      </property>
      <property name="dialect">
         org.hibernate.dialect.PostgreSQLDialect
      </property>
      <property name="cache.provider_class">
         org.hibernate.cache.NoCacheProvider
      </property>
      <property name="current_session_context_class">thread</property>
      <property name="hibernate.show_sql">true</property>
      <property name="hibernate.hbm2ddl.auto">create</property>
      <mapping class="de.laliluna.example.Honey" />
      <mapping class="de.laliluna.example.Bee" />
   </session-factory>
</hibernate-configuration>
```

At the beginning of the configuration, you can find the database connection. Afterwards, the dialect is specified. Hibernate will translate all queries into this dialect. Then, I configured the cache implementation and the session and transaction behaviour. The setting *hbm2ddl.auto* instructs Hibernate to update or create the table schema when the configuration is initialized. You only have to create the database.

The tag <mapping class> at the end of the file defines which classes are mapped using annotations. If you want to use XML based mappings then you need to reference the XML mapping files instead:

```
<mapping resource="de/laliluna/example/Honey.hbm.xml" />
<mapping resource="de/laliluna/example/Bee.hbm.xml" />
```

Other databases

A configuration for MySQL requires the following changes:

```
<property name="connection.url">jdbc:mysql://localhost/learninghibernate</property>
<property name="connection.username">root</property>
<property name="connection.password">r</property>
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
```

If you want to use another database, you must find out the name of the driver class and the name of the dialect. You can find all available dialects in the java package org.hibernate.dialect of the hibernate.jar file.

A short extract of frequently used dialects:

- MySQL5Dialect
- OracleDialect
- SybaseDialect
- SQLServerDialect
- HSQLDialect
- DerbyDialect

The connection.url is a normal JDBC connection URL.

1.1.5. Mapping

With the mapping we define in which table column a field of a class is saved. As already stated, we have two options for the mapping. The first one is based on annotation, it is new and the future and I recommend to use it. But it requires Java 5. The second one is based on XML mapping files. I will explain both:

Annotation mapping

An annotation always starts with an @. You just have to add annotations to your java file. Annotations must be imported just like other classes.

Honey class with annotations.

```
.....
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.SequenceGenerator;
@Entity
@SequenceGenerator(name = "honey_seq", sequenceName = "honey_id_seq")
public class Honey implements Serializable {
   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="honey_seq")
   private Integer id;
   private String name;
   private String taste;
   @OneToMany(mappedBy="honey")
   private Set<Bee> bees = new HashSet<Bee>();
```

The annotation *@Entity* defines that our class is a mapped class. The primary key is defined by the *@Id* annotations. It is generated by Hibernate using a database sequence. It is defined with *@SequenceGenerator* before it is used by the id (*@GeneratedValue*).

The annotation *@OneToMany* describes that the field bees is a 1:n association to the class Bee. The foreign key is defined in the class Bee. Therefore, we have to add the parameter *mappedBy*.

The class Bee uses two other annotations. *@ManyToOne* describes the association/relation from class Bee to Honey. With the annotation *@JoinColumn* we configure that the table bee contains a foreign key column. We did not specify a column name, so Hibernate will choose *honey_id*.

Bee class with annotations.

```
.......
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;

@Entity
public class Bee implements Serializable {
   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "bee_gen")
   @SequenceGenerator(name = "bee_gen", sequenceName = "bee_id_seq")
   private Integer id;
   private String name;

   @ManyToOne
   @JoinColumn
   private Honey honey;
```

Other databases

If your database does not support sequences, you might try

```
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Integer id;
```

This selects a generator depending on the configured database dialect.

XML mapping

You only need the XML mapping files, if you don’t use annotations.

In the mapping file the mapping from our class Honey to the database table honey is configured. Create a file *Honey.hbm.xml* in the package *de.laliluna.example*.

Honey.hbm.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="de.laliluna.example">
  <class name="Honey" table="thoney" >
      <id name="id" column="id">
      <generator class="sequence">
            <param name="sequence">honey_id_seq</param>
         </generator>
      </id>
      <property name="name" column="name" type="string" />
      <property name="taste" column="taste" type="string" />
      <set name="bees" inverse="true">
        <key column="honey_id"></key>
       <one-to-many class="Honey" />
      </set>
   </class>
</hibernate-mapping>
```

The tag <id> specifies the primary key and how it is generated. Normal fields are mapped to a column using the <property> tag. If we didn’t specify a column name (column=”xyz”), than Hibernate would have chosen the name of the field as default column name.

<set> describes the relation to the class Bee.

Our example used a sequence to generate the id. If your database does not support sequences, you might try

```
<generator class="native"/>
```

This will choose a generator depending on your database dialect.

Why using type=”string” and not type=”java.lang.String”?

The Hibernate type can be more precise than the java types. A database field like date, timestamp, time is always handled by a java.util.Date. When you define a type as java.util.Date Hibernate does not know which kind of database type to choose when it generates tables. This is why I use Hibernate types in general.

In most cases you don’t need to set the type. Hibernate will guess the column type from the Java class.

Needed Libraries

To use Hibernate, we need a couple of libraries, alias JAR files.

Maven Setup

In the sample projects, I have used Maven dependencies. If you know Maven, just copy the POM from the source code and use it. I have declared all dependencies in a parent *pom.xml*, which is in the root of the workspace.

Just switch into the *FirstHibernateAnnotation* directory and input

```
mvn dependency:resolve
```

in a shell. Alternatively, you can choose a run task from within Eclipse as well by right-clicking on a pom.xml and choosing "Run…"

Non Maven Setup

If you want to add the libraries manually, here is a list of required libraries and versions.

In Eclipse Open your project properties, select “Java Build Path”, click on “Add External Jars” and add at least the libraries shown below to your project path.

Included in the Hibernate Core Download.

```
+- junit:junit:jar:4.5:test
+- org.slf4j:slf4j-log4j12:jar:1.6.0:compile
|  +- org.slf4j:slf4j-api:jar:1.6.0:compile
|  \- log4j:log4j:jar:1.2.14:compile
+- c3p0:c3p0:jar:0.9.1.2:compile
+- org.hibernate:hibernate-core:jar:3.6.4.Final:compile
|  +- antlr:antlr:jar:2.7.6:compile
|  +- commons-collections:commons-collections:jar:3.1:compile
|  +- dom4j:dom4j:jar:1.6.1:compile
|  +- org.hibernate:hibernate-commons-annotations:jar:3.2.0.Final:compile
|  +- org.hibernate.javax.persistence:hibernate-jpa-2.0-api:jar:1.0.0.Final:compile
|  \- javax.transaction:jta:jar:1.1:compile
+- javassist:javassist:jar:3.9.0.GA:compile
```

Some projects use caching and Hibernate validation.

Included in Hibernate Core, Hibernate Validator and Ehcache Download.

```
+- org.hibernate:hibernate-ehcache:jar:3.6.4.Final:compile
+- net.sf.ehcache:ehcache-core:jar:2.4.2:compile
+- org.hibernate:hibernate-validator:jar:4.1.0.Final:compile
|  \- javax.validation:validation-api:jar:1.0.0.GA:compile
```

Database Driver.

```
+- postgresql:postgresql:jar:8.3-603.jdbc3:compile
\- hsqldb:hsqldb:jar:1.8.0.7:compile
```

Database Driver

We need a database driver as well. Find the appropriate JDBC Driver for your database and add it to the project libraries.

I used PostgreSQL. You can get the driver from http://jdbc.postgresql.org/ . You can use the JDBC 3 driver if you are running a current j2sdk like 1.4 and 1.5/5.0.

For MySQL you can use the MySQL connector which can be found at http://www.mysql.com/products/connector/j/

An Oracle database driver is available at Oracle: http://www.oracle.com

1.1.6. Create a session factory

A session factory is important for Hibernate. As the name already indicates, it creates Hibernate sessions for you. A session is all you need to access a database using Hibernate. In addition, a session factory initialises your Hibernate configuration.

We need to distinguish Annotation and XML once again. Here comes the Annotation version:

Create a class named InitSessionFactory in the package *de.laliluna.hibernate* and add the source code below. This class guaranties that there is only once instance (Singleton pattern) and that the initialisation happens only once when the class is loaded.

```
package de.laliluna.hibernate;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class InitSessionFactory {
   /** The single instance of hibernate SessionFactory */
   private static org.hibernate.SessionFactory sessionFactory;
   private InitSessionFactory() {
   }

   static {
      final Configuration cfg = new Configuration();
      cfg.configure("/hibernate.cfg.xml");
      sessionFactory = cfg.buildSessionFactory();
   }
   public static SessionFactory getInstance() {
      return sessionFactory;
   }
}
```

1.1.7. Configuring Log4J

Hibernate uses log4j as logging output. As you can see above we added the log4j library. This library needs a configuration file in the source directory or it welcomes you with the following error.

```
log4j:WARN No appenders could be found for logger (TestClient).
log4j:WARN Please initialize the log4j system properly.
```

Create a file named log4j.properties in the root directory. We will configure a simple logging output in the Console.

```
# configuration is adapted from www.hibernate.org
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
### set log levels - for more verbose logging change 'info' to 'debug' ###
log4j.rootLogger=debug, stdout
log4j.logger.org.hibernate=info
#log4j.logger.org.hibernate=debug
### log just the SQL
log4j.logger.org.hibernate.SQL=debug
```

1.1.8. Create database and tables

Create a database with MySql or PostgreSQL or anything you like. Call it firsthibernate. This is the same database name, we used in the hibernate.cfg.xml.

We have defined in hibernate.cfg.xml that the tables should be recreated each time when the Hibernate configuration is initialized.

```
<property name="hibernate.hbm2ddl.auto">create</property>
```

The setting create will drop and create any tables. It is only suitable during development. You might use update as well. In this case Hibernate tries to update existing tables or create missing tables. This works in most cases but not always. Therefore, I recommend to disable this for production.

1.1.9. Create a test client

We will create a Java class, which will create, update, delete and query our data.

Create a Java Class *TestExample* in the package *de.laliluna.example*.

Please look through the source code carefully and try it. I will just give you some general informations before:

- you always need a session to access data
- all access read and write happens within a transaction
- when a transaction fails you should role back your transaction (see the discussion below as well). In this first example, we will skip the exception handling.

In the main method we call other methods creating, updating, deleting and querying data.

```
package de.laliluna.example;

import java.util.Iterator;
import java.util.List;
import org.apache.log4j.Logger;
import org.hibernate.Criteria;
import org.hibernate.FetchMode;
import org.hibernate.Hibernate;
import org.hibernate.Session;
import org.hibernate.Transaction;
import de.laliluna.hibernate.InitSessionFactory;

public class TestExample {
   private static Logger log = Logger.getLogger(TestExample.class);
   public static void main(String[] args) {
      /* clean tables */
      clean();

      /* simple create example */
      createHoney();

      /* relation example */
      createRelation();

      /* delete example */
      delete();

      /* update example */
      update();
      /* query example */
      query();
   }
```

The method createHoney creates a new object and saves it in the database by calling session.save.

```
private static Honey createHoney() {
      Honey forestHoney = new Honey();
      forestHoney.setName("forest honey");
      forestHoney.setTaste("very sweet");

      Session session = InitSessionFactory.openSession();
      Transaction tx = session.beginTransaction();
      session.save(forestHoney);
      tx.commit();
      session.close();
      return forestHoney;
   }
```

The method update creates a new object using our last method, changes the name and updates the object using session.update.

```
private static void update() {
      Honey honey = createHoney();
      Session session = InitSessionFactory.openSession();
      Transaction tx = session.beginTransaction();
      honey.setName("Modern style");
      session.update(honey);
      tx.commit();
      session.close();
   }
```

The method delete creates an object and deletes it by calling session.delete.

```
private static void delete() {
      Honey honey = createHoney();
      Session session = InitSessionFactory.openSession();
      Transaction tx = session.beginTransaction();
      session.delete(honey);
      tx.commit();
      session.close();
   }
```

The tables are emptied with the method clean. The method session.createQuery creates a new query and runs it by calling executeUpdate.

```
   private static void clean() {
      Session session = InitSessionFactory.openSession();
      Transaction tx = session.beginTransaction();
      session.createQuery("delete from Bee").executeUpdate();
      session.createQuery("delete from Honey").executeUpdate();
      tx.commit();
      session.close();
   }
```

The method createRelation shows, how to create objects and set an association between these objects. This will write a foreign key relation to the database.

```
private static void createRelation() {
    Session session = InitSessionFactory.openSession();
    Transaction tx = session.beginTransaction();
    Honey honey = new Honey();
    honey.setName("country honey");
    honey.setTaste("Delicious");
    session.save(honey);
    Bee bee = new Bee("Sebastian");
    session.save(bee);

    /* create the relation on both sides */
    bee.setHoney(honey);
    honey.getBees().add(bee);
    tx.commit();
    session.close();
  }
```

The method query shows how to query all honeys in the database. The call to session.createQuery creates the query and list() runs the query and returns a list of Honey objects.

```
  private static void query() {
    Session session = InitSessionFactory.openSession();
    Transaction tx = session.beginTransaction();
    List honeys = session.createQuery("select h from Honey as h").list();
    for (Iterator iter = honeys.iterator(); iter.hasNext();) {
      Honey element = (Honey) iter.next();
      log.debug(element);
    }
    tx.commit();
    session.close();
  }
```

This was a fast example and it is not required to have understood anything.

Lessons learned

- how to create a class, which can be used for mappings (constructor, Serializable, …)
- how to add the mapping
- how to create, update and delete entries
- how to create a simple query

Exception handling was ignored in this example

You should be aware, that in case of an exception, some databases keep resources open, if you do not rollback or commit a transaction. I will speak about transaction handling in detail in chapter Session handling and Transactions Chapter 13, *Session and Transaction Handling*.

Deployment in Jboss Application-Server

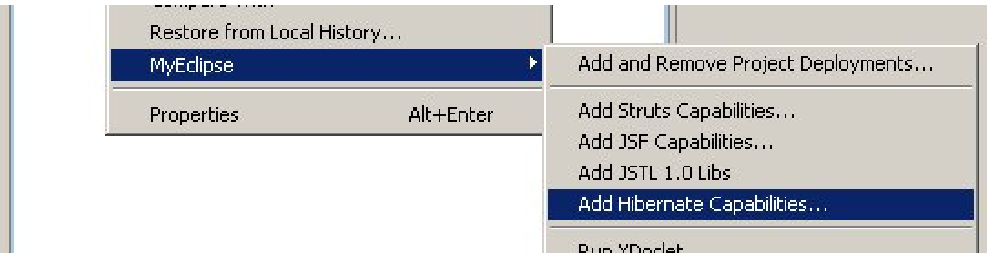
If you try to use the code by a JSP and deploy it to JBoss or deploy any of the later examples to Jboss you might encounter problems. The reason is that Jboss provides already Hibernate libraries and a configuration which conflicts with the Session configuration, I chose for the examples. Either use a servlet engine like Jetty or Tomcat or have a look in chapter Session handling and Transactions Chapter 13, *Session and Transaction Handling* explaining how to configure Hibernate in JTA environments.

1.1.10. Using MyEclipse for Hibernate projects

MyEclipse is a plugin provided by Genuitec on http://www.myeclipseide.com/ . I use it for development of all kinds of Web applications including Hibernate, EJB 2 and EJB3. MyEclipse is very up-do-date with the support of current versions of web frameworks, EJB and of course Hibernate.

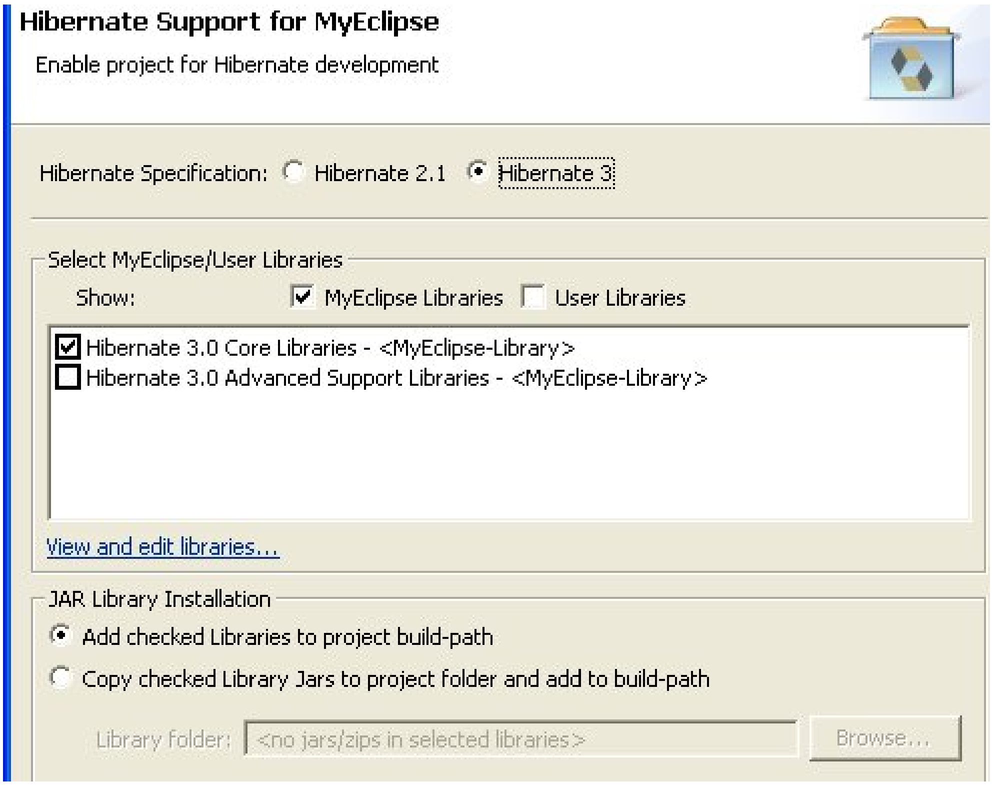
1.1.11. Adding libraries and Hibernate capabilities

Create a webproject first and open the context menu of the project (click with right mouse button on the project in the package view). Select add Hibernate capabilities.


Restore from Local History...
MyEclipse
Properties Alt+Enter
Add and Remove Project Deployments...
Add Struts Capabilities...
Add JSF Capabilities...
Add JSTL 1.0 Libs
Add Hibernate Capabilities...
Run XDoclet

The wizard let you select the Hibernate version and add the needed libraries to your project.

I recommend to add the libraries to the build path and not to copy them to the lib folder of the application. This will allow to set up different configurations for deployment during development and staging.



Hibernate Support for MyEclipse

Enable project for Hibernate development

Hibernate Specification: ○ Hibernate 2.1 ● Hibernate 3

Select MyEclipse/User Libraries

Show: ☑ MyEclipse Libraries □ User Libraries

☑ Hibernate 3.0 Core Libraries - <MyEclipse-Library>
□ Hibernate 3.0 Advanced Support Libraries - <MyEclipse-Library>

View and edit libraries...

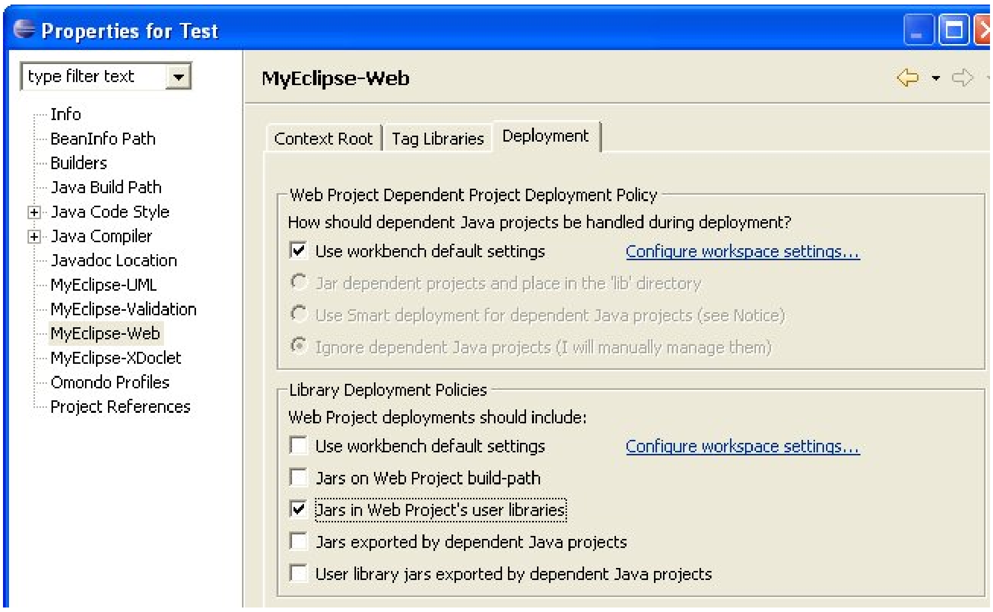
JAR Library Installation

● Add checked Libraries to project build-path
○ Copy checked Library Jars to project folder and add to build-path

Library folder: <no jars/zips in selected libraries>

Browse...

To stage your finished application you should pack all the libraries with your project. You can configure in the project properties to deploy user libraries like the MyEclipse library in your application server.



MyEclipse-Web

Web Project Dependent Project Deployment Policy

How should dependent Java projects be handled during deployment?

☑ Use workbench default settings   Configure workspace settings...

○ Jar dependent projects and place in the 'lib' directory

○ Use Smart deployment for dependent Java projects (see Notice)

● Ignore dependent Java projects (I will manually manage them)

Library Deployment Policies

Web Project deployments should include:

☐ Use workbench default settings   Configure workspace settings...

☐ Jars on Web Project build-path

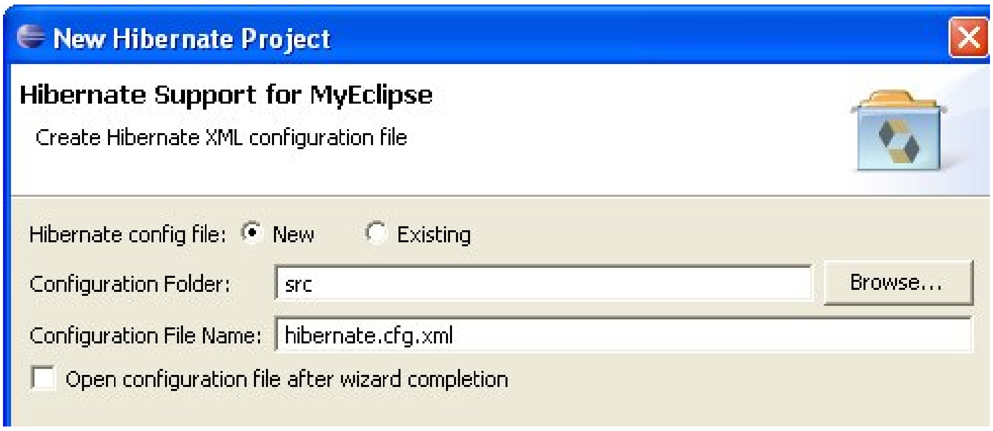
☑ Jars in Web Project's user libraries

☐ Jars exported by dependent Java projects

☐ User library jars exported by dependent Java projects

During development, I prefer to copy all the Hibernate and Struts libraries to the shared library directory of the application server. I configure the MyEclipse configuration in the project properties to not deploy any user libraries. This makes redeployment very fast.

Lets continue with the wizard. You can select an existing Hibernate configuration file hibernate.cfg.xml or you can create a new one.



Hibernate Support for MyEclipse

Create Hibernate XML configuration file

Hibernate config file: ● New ○ Existing

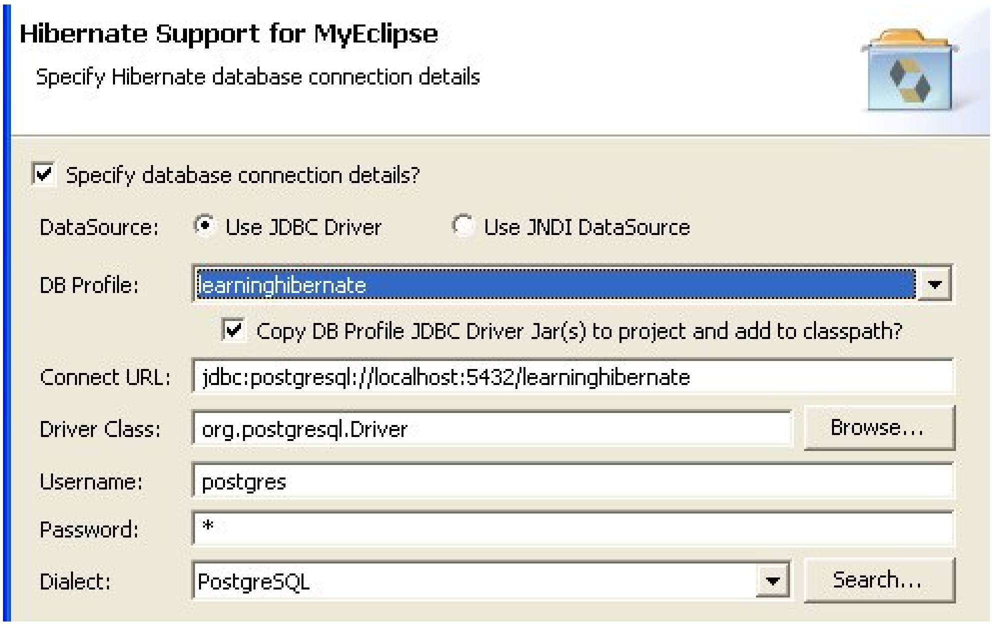
Configuration Folder: src

Configuration File Name: hibernate.cfg.xml

☐ Open configuration file after wizard completion

Then you can select a database connection profile for a JDBC connection or select a JNDI datasource. A connection profile is the complete configuration of a JDBC database with connection string, user name, password and DB driver.

A connection profile can be used for the database browser as well to see tables, triggers, columns. The database browser allows to generate Hibernate mappings from existing tables as well.



Hibernate Support for MyEclipse

Specify Hibernate database connection details

☑ Specify database connection details?

DataSource: ● Use JDBC Driver ○ Use JNDI DataSource

DB Profile: learninghibernate

☑ Copy DB Profile JDBC Driver Jar(s) to project and add to classpath?

Connect URL: jdbc:postgresql://localhost:5432/learninghibernate

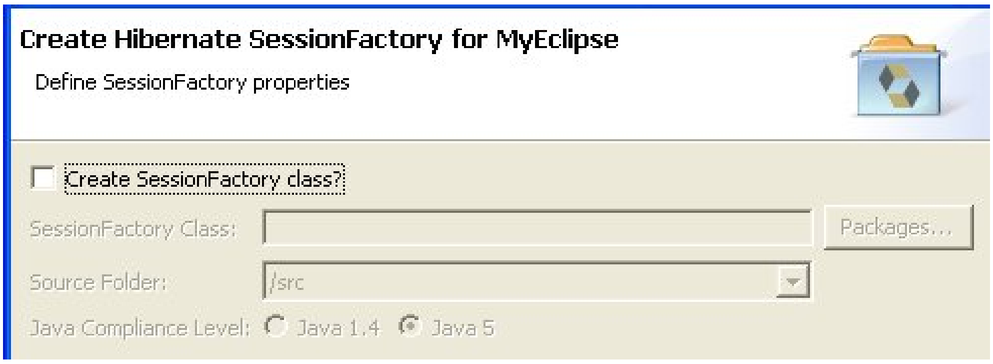
Driver Class: org.postgresql.Driver [Browse...]

Username: postgres

Password: *

Dialect: PostgreSQL [Search...]

Finally you can create a session factory. I prefer to use my own session factory as it is simpler but the MyEclipse session factory works perfectly, as well.



Create Hibernate SessionFactory for MyEclipse

Define SessionFactory properties

□ Create SessionFactory class?

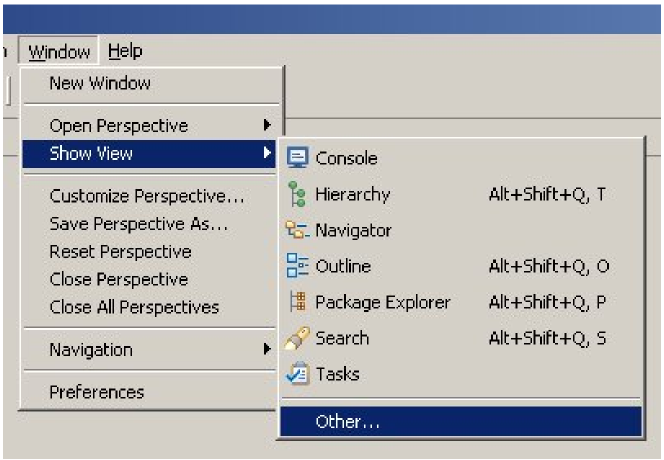
SessionFactory Class: [Packages...]

Source Folder: /src

Java Compliance Level: ○ Java 1.4 ● Java 5

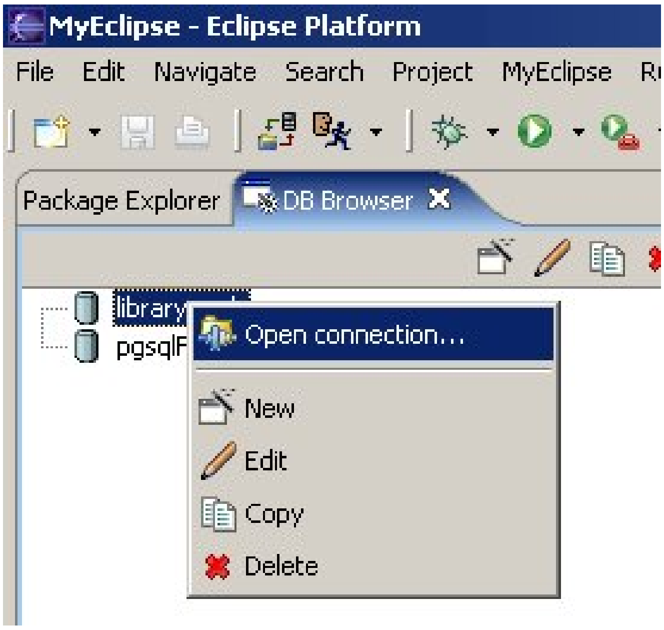
1.1.12. Generate Hibernate mappings from existing db

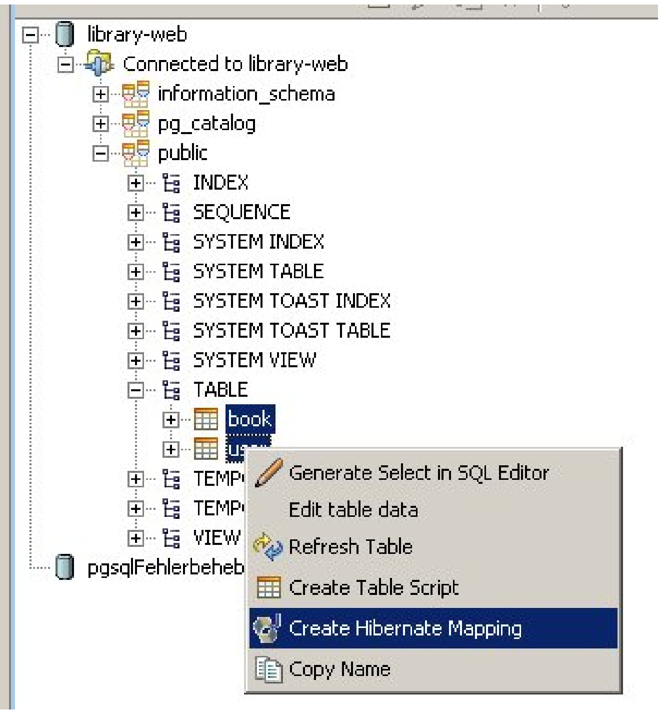
Open the View DB Browser (MyEclipse). If you cannot find it open the „Show View“ Dialog and select in the MyEclipse Enterprise Workbench the DB Browser.



Open the connection profile you specified before.

Select the two tables we have just created. Right click and choose Create Hibernate Mapping.





Follow the wizard to select where to create the mapping files and classes. You can even generate DAOs.

1.2. Hibernate basics

1.2.1. What is Hibernate?

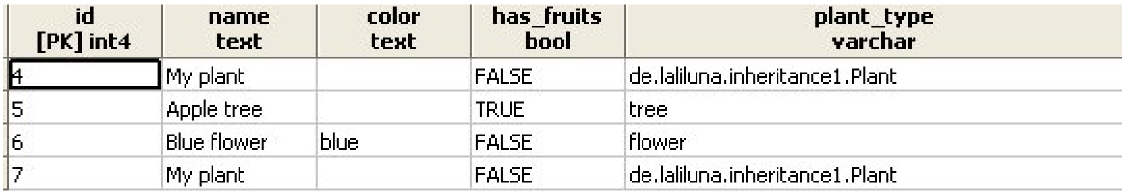
As explained in chapter Basic idea of Hibernate Chapter 1, *Introduction to Hibernate*, Hibernate is an object relational mapping solution. It provides all the features you need to create a powerful persistence layer for your application. This chapter explains some basic features and will explain the Hibernate architecture. You should read this first if you do not want face any problems later.

1.2.2. Powerful mapping

The mapping is not limited to one class or to one table but you can also map a wide range of object-oriented concepts. The following classes represent an inheritance hierarchy.

```
public class Plant {
   private Integer id;
   private String name;
...
public class Tree extends Plant {
   private boolean hasFruits;
...
public class Flower extends Plant {
   private String color;
....
```

They can be mapped to one table containing all columns and having a discriminator column, which distinguish what type a row is. In the picture below plant_type is the discriminator.



id [PK] int4	name text	color text	has_fruits bool	plant_type varchar
4	My plant		FALSE	de.laliluna.inheritance1.Plant
5	Apple tree		TRUE	tree
6	Blue flower	blue	FALSE	flower
7	My plant		FALSE	de.laliluna.inheritance1.Plant

There are other options to map classes and subclasses, e.g. One table per subclass.

Another feature is the mapping of relations 1:n, m:n and 1:1. You can map components. For example you create an Address component which is used in Customer and Supplier.

1.2.3. Powerful query languages

Hibernate provides a powerful object-oriented query language named Hibernate Query Language (HQL).

```
select i from Invoice i inner join fetch i.order
```

If you have to create queries dynamically, you can use the Hibernate Criteria Queries. The following query filters the player by name, but only if the name is not null:

```
Criteria criteria = session.createCriteria(Player.class);
if(name != null)
  criteria.add(Restrictions.eq("name", name));
List players = criteria.list();
```

If you need native SQL or a stored procedure, you can call it from Hibernate as well.

Chapter 2. Hibernate Concepts - State of Objects

States of an entity is a very important concept of Hibernate. An entity can have different states. Using Hibernate is different from using SQL. If you call *session.save(customerObject)* then there is no *insert into customer…* query into the database. Hibernate will set the *id* property (if the *id* is generated) and bind the entity to a persistence context. The persistence context is synchronized with the database when *transaction.commit()* is called.

This approach has various advantages:

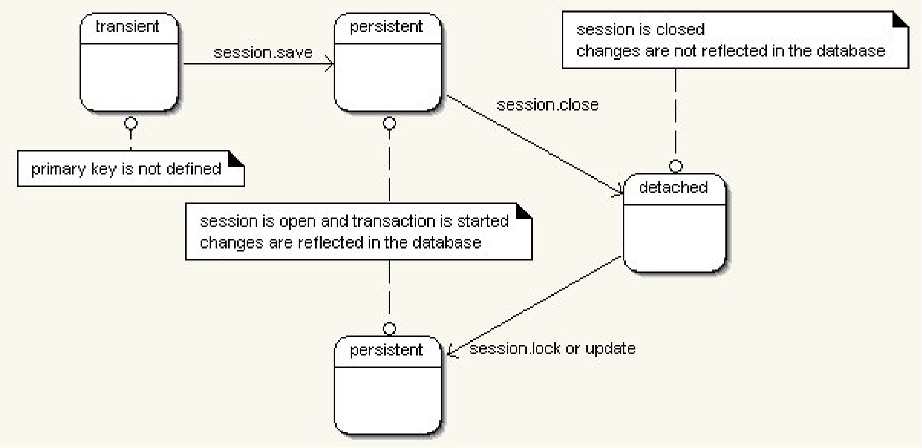
- You can continue to update an entity and all changes to the Java object are persisted with a single database insert/update.

- The update of a database table row causes a row lock. Having a row lock only for a short moment at the end of a transaction, prevents locking of concurrent users or dead lock situations.

2.1. The three states of objects

An object can have three states: transient, persistent and detached.

When it is just created and has no primary key, the state is called transient.


transient
session.save
persistent
session is closed
changes are not reflected in the database
session.close
primary key is not defined
detached
session is open and transaction is started
changes are reflected in the database
persistent
session.lock or update

```
Car car = new Car();
car.setName(“Porsche”);
```

When the session is opened and the object is just saved in or retrieved from the database. This state is called persistent. During this state Hibernate manages the object and saves your changes, if you commit them. Below you can see an example. A car is saved and the name is changed afterwards. As the car is in persistent state, the new name will be saved.

```
Session session = HibernateSessionFactory.currentSession();
tx = session.beginTransaction();
```

```
session.save(car);
car.setName(“Peugeot”);
tx.commit();
```

The following code loads a car by id and changes the name. There is no *session.update* involved. Every object which is loaded by *session.get*, *session.load* or a query is in persistent state. It is stored in the persistence context of the session. When *tx.commit()* is called, Hibernate will flush the persistence context and all not yet written insert, update and delete statements are executed.

```
Session session = HibernateSessionFactory.currentSession();
tx = session.beginTransaction();
Car car = (Car) session.get(Car.class, 4711);
car.setName(“Peugeot”);
tx.commit();
```

When the session was closed, the state changes to detached. The object is detached from its session.

Understanding these states is important in order to know how to deal with instances. Imagine you do a web dialogue

- retrieve an instance of Car from the database
- save instance in the HTTP request and close session
- show a dialog to edit the instance
- user submits form

Now, you would normally save the changes. But you cannot just call

```
transaction = session.beginTransaction();
car.setname(newName);
transaction.commit();
```

If you want to save the changes, you have to reattach your instance. This means the instance must be brought to a persistent state. You can not save an instance when it is not in persistent state, except if it was transient before and has no primary key set.

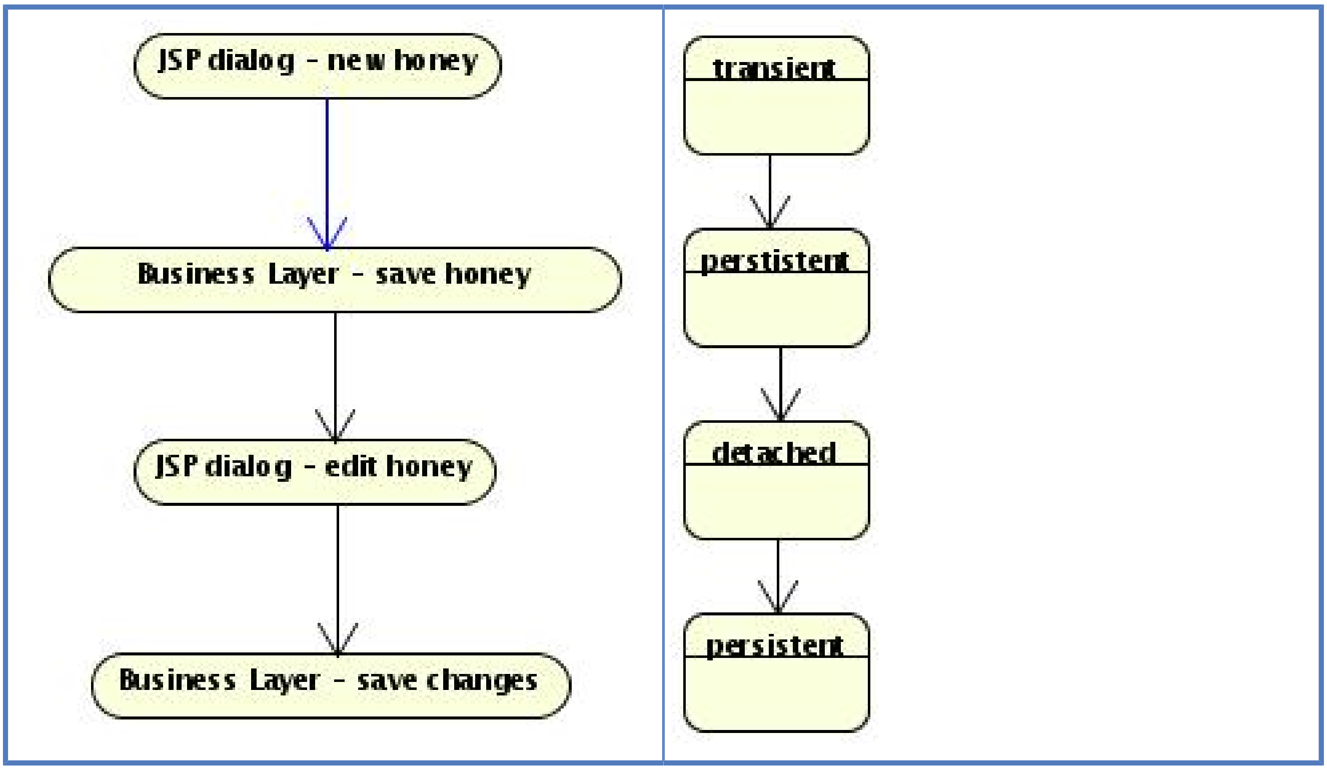
You must reattach an object before you can update it.

```
transaction = session.beginTransaction();
session.buildLockRequest(LockOptions.NONE).lock(car);
car.setname(newName);
transaction.commit();
```

The following picture show the change of status how it could happen in a web application.

Deprecation

You might be aware of the method *session.lock(car)*. It is deprecated since Hibernate 3.6.



How to deal with reattaching and state changes carefully is explained in detail in the chapter Working with Objects Chapter 3, *Working with Objects*.

2.2. Lazy initialization, a Hibernate problem

There is a popular exception which a lot of Hibernate newbees encounter. In order to prevent this exception you need to understand the concept of Lazy Initialization.

1) When Hibernate reads data from the database, the data is hold in the session. You can save references to the data - for example in your HTTP request. Once the transaction is committed and the session is closed you can not load any further data with this session.

In the first example, we have seen that Honey has a collection of Bee. If we load a Honey instance, Hibernate will not load the bees automatically. There is a good reason for this behaviour. Imagine you load a Company and get all orders, order details and articles. Basically you load most of your database just by loading a single Company object. This would result into a memory problem. Therefore Hibernate loads only the first object and replaces collections of other objects by a proxy. If you access the proxy, Hibernate uses the current session to initialize the proxy and load the entries from the database.

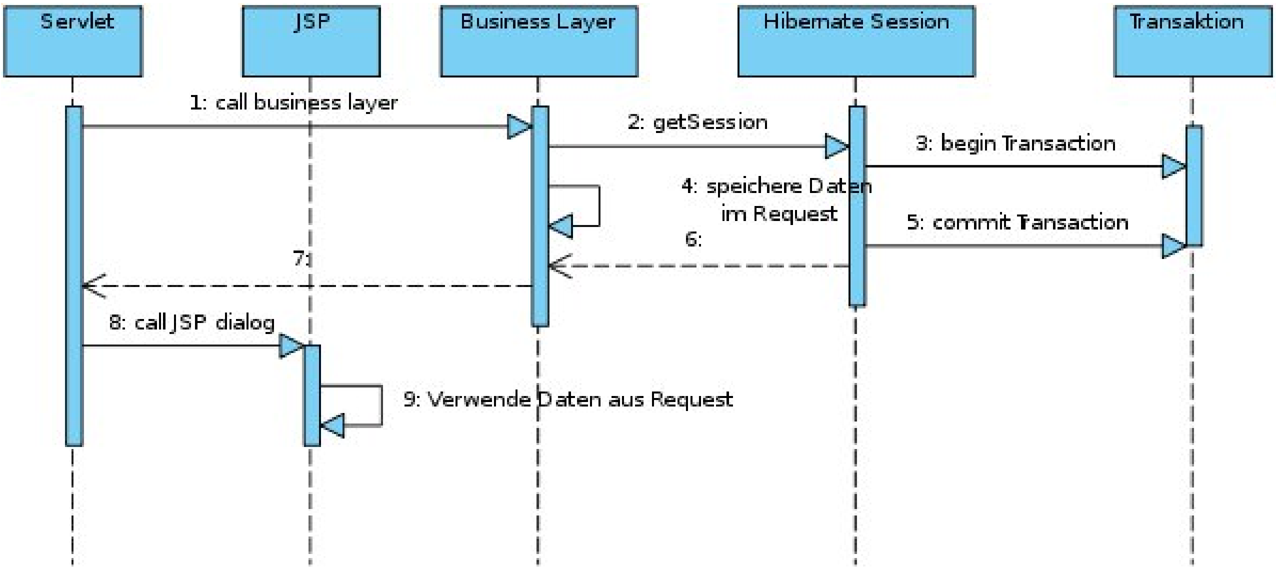
2) Imagine a Struts framework application. If you do not know Struts: it is a framework to develop web applications. When a user requests something, e.g. he has submitted a form on a website, the following happens:

- central servlet is called
- servlet looks up application logic for the request and calls the application logic

- application logic opens a session, saves or retrieves data
- application logic stores retrieved data in the request and closes the session
- control returns to servlet
- servlet calls a JSP to render the dialog
- the JSP uses the data in the request

What is the consequence of 1) and 2)? When you look through the process 2) you can see that the session is already closed, when the dialogue is rendered. Your application logic has finished processing. If you have not initialized any objects while your session is open, you will not be able to display them. Have a look on the following diagram, which explains the situations quite well. \newline

When you access a not initialized object you will get a LazyInitializationException explaining that the session is already closed.



When can this happen? I have mentioned that Hibernate can map relations. Imagine a class department having a number of teams.

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

   private Set teams = new HashSet();
```

If you want to output a list of departments and teams in your JSP, you must not only fetch all instances of department but also all instances of teams which are associated with one of the departments you are retrieving.

I told you that by default all relations are retrieved lazy. This means when you fetch a department, Hibernate will not fetch the teams but create a proxy. When you access a team, the proxy uses the current session to load the team from the database. A proxy can only retrieve data when the session is open.

Having relations in your mapping you must ensure that the object and related objects are initialized as long as the session is open.

There are three solutions to this problem:

- Define lazy=”false” in your mapping
- Explicitly fetch the associated data in your query
- Make use of a trick and postpone the close of the session to a later time, when your JSP is already rendered.

The first solution is dangerous. Imagine a relation like

ApplicationUser → KeyAccounter → Customer → Company → all customers of company → all orders of customers

Every access would load the complete database. Be very careful when you set lazy to false.

The second solution is simple but have some caveats. The trick is named Open-Session-In-View and is explained in chapter Open Session in View Section 13.7.2, “Lifetime until the view is rendered (Open-Session-in-View)”.

The third solution initializes the data before closing the session.

We have two options to initialize data.

Approach a)

```
List honeys = session.createQuery("select h from Honey as h").list();
for (Iterator iter = honeys.iterator(); iter.hasNext();) {
   Honey element = (Honey) iter.next();
   log.debug(element);
   Hibernate.initialize(element.getBees());
```

Approach b)

```
honeys = session.createQuery("select h from Honey as h left join fetch h.bees")
         .list()
```

If you use approach a) you have to call to call *Hibernate.initialize* on each proxy. Each call will generate one query.

Approach b) generates a left join statement. We will only require one query.

Consider to use b) if you query a lot of data.

You must be aware of a draw back of this approach. Left join results in double entries for Invoice when there are multiple orders. Think of the underlying sql query, which leads to a result like

```
invoice 1, joined order line 1
invoice 1, joined order line 2
invoice 2, joined order line 1
.....
```

Hibernate will as well add the invoice 1 multiple times to the result list.

You can use the following approach to get unique invoices (have a look in the HQL and Criteria Query chapter for detailed examples):

```
session.createCriteria(Honey.class).setFetchMode("bees",FetchMode.JOIN)
  .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();

session.createQuery("select h from Honey as h join fetch h.bees")
  .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();
```

Chapter 3. Working with Objects

3.1. Java Persistence versus Hibernate

There are two APIs you can use with your mappings. One is the Hibernate Session and the other one the Java Persistence EntityManager. The latter is standardised as Java Persistence which is the replacement for Entity Beans in EJB 3.

With Java Persistence (JPA) 2.1 the API is good enough to be used instead of the Hibernate Session. Some people argue that it is better to use the Entity Manager as it is already included in any JEE application server like Glassfish, JBoss etc. and it allows to swap the Java Persistence implementation. Well, in theory this is true. In practice you will use implementation specific optimisations for data loading, caching and tuning. In fact, it is easier to deploy a Hibernate application on a different application server.

My opinion

I prefer to use Hibernate for most cases as deployment on different application server or servlet engines is easier and I consider the criteria queries of JPA to be very hard to use as compared to Hibernate criteria queries. But there is an exception as well. If JEE with Session Beans is used as business layer technology, it is very easy to use Java Persistence. It is already provided. If in addition, the team has JPA experience, then I tend to prefer the JPA API.

The EntityManager of Hibernate is based on the Hibernate session. Basically, the EntityManager wraps a Hibernate session and adds some behaviour. The following code, shows how to get the underlying Hibernate session to get access to additional API methods like *session.update*.

```
Session hibernateSession = entityManager.unwrap(Session.class);
hibernateSession.update(country);
```

I will introduce the Hibernate API first. You will find the EntityManager API in the next chapter.

3.2. Hibernate API

3.2.1. Saving

Saving or persisting an object is actually not very complicated. The following code will do it.

```
Transaction tx = session.beginTransaction();
Concert c = new Concert();
c.setName("Peter");
session.save(c);
tx.commit();
```

If you debug the code of the first example, you will see that *session.save* does not create an insert statement immediately but just selects the next sequence value. This is at least true, if you use a sequence as id generator.

It is the *tx.commit()* of the transaction, which leeds to sending the SQL insert statement to the database. Calling *save* makes an object persistent. If the id is configured to be generated then Hibernate will ensure that it is generated and set the value of the id property. In addition the entity is added to the persistence context of the session.

Persistence Context

Simplified: the persistence context contains a map for each entity. The key of the map is the id and the value is a composite of the entity instance itself and the original values of all fields when the object was added to the persistence context. The latter allows Hibernate to determine if a persistent object needs to be updated in the database.

When the persistence context is flushed, Hibernate will send all insert statements to the database, determines all required updates by comparing the entities to the original field values and sends all updates to the database and finally all deletes are sent.

The call to *commit* will cause a flush of the persistence context before the commit is sent to the database.

There are multiple reasons for this behavior:

Less update statements

If you change multiple fields, add a relation while an entity is in persistent state, Hibernate will only send one update statement at the end.

JDBC batching

Hibernate can group updates and make use of JDBC batching. This is more efficient as compared to send statements one by one.

Reduced duration of locks

Sending an insert or update statement will cause a row or page lock depending on the database. Sending such statement at the end just before the transaction commit, will cause the lock to exist only for a short timespan. It reduces the risk of concurrent users waiting for locks or dead lock situations.

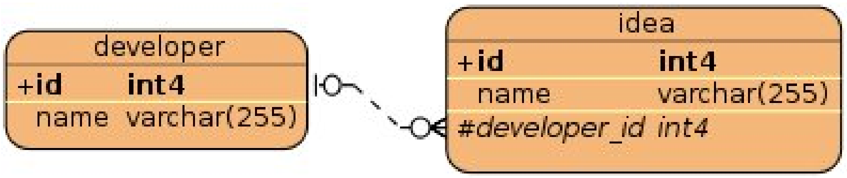
Therefor, Hibernate will do only the minimum required to determine the id of the entity and add the entity to the persistence context of the session. If the id is a generated value and the generator is a sequence or a table, Hibernate will just select the value. But if the id is generated on inserting - for example increment column in MS SQL server - then Hibernate needs to send the SQL insert statement to get an id generated.

Saving becomes more complex, when you start to use relations.

We are having the following tables mapped as 1:n relation. We will define that *developer_id* must not be null, an idea cannot exist without a developer. The database will create a not null constraint for the column *developer_id*.

```
Developer d = new Developer();
Idea idea = new Idea();
d.getIdeas().add(idea);
idea.setDeveloper(d);
```

You must take care that the idea is not saved before the *developer_id* is defined. If you save the idea first, then the *developer_id* is not yet defined and you will receive a constraint violation exception.



Use case: no cascading defined

You must call save in the following order:

```
session.save(developer);
session.save(idea);
```

Use case: cascading from developer to idea

You only need to call

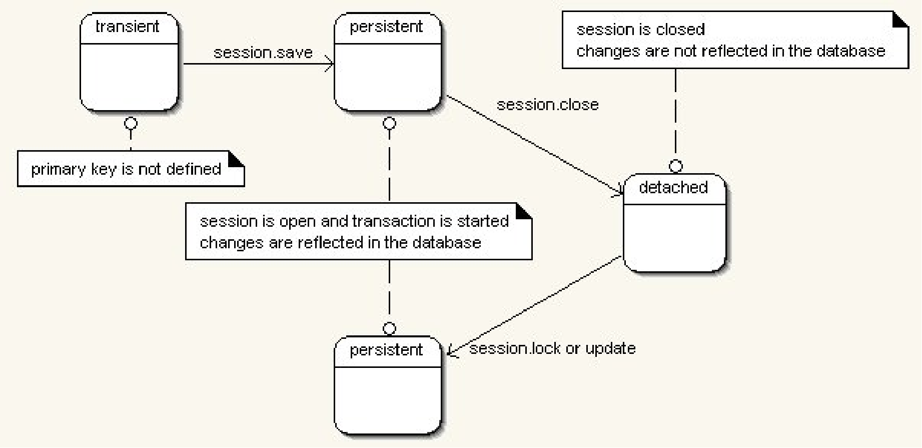
```
session.save(developer);
```

3.2.2. Updating

Considerations

We talked in chapter Hibernate States Chapter 2, *Hibernate Concepts - State of Objects* about the states of an object. Below you can find the figure we used in that chapter.

Updating an entity requires that you get your object in persistent state. One approach is to fetch an object from the database within a transaction, apply your changes and commit the transaction.


transient
session.save
persistent
session is closed
changes are not reflected in the database
session.close
primary key is not defined
detached
session is open and transaction is started
changes are reflected in the database
persistent
session.lock or update

```
session.beginTransaction();
Visitor visitor = session.get(Visitor.class, aVisitorId);
visitor.setName("Edgar");
session.getTransaction().commit();
```

In the code above there is no *session.update*. Working with Hibernate means working with objects and states. This is a big difference compared to JDBC and sending *insert* and *update* SQL statements.

Another approach to change an entity, is to reattach a detached object.

```
session.beginTransaction();
session.buildLockRequest(LockOptions.NONE).lock(visitor); // reattach visitor
visitor.setName("Edgar");
session.getTransaction().commit();
```

Detached objects

What is Reattaching?

Reattaching changes the state of an object from detached to persistent.

When you load an object within a session and close the session, your object is detached from the session and any changes to the object have no influence to the session or the database.

This happens normally if you use the recommended approach of session handling. Read more about session handling in chapter Session Handling Chapter 13, *Session and Transaction Handling*.

If you want to apply any updates you have to reattach the object to a session.

Reattaching

You have different options to reattach an object to a session. The main difference between these approaches are three criteria:

- Can the object be changed before it is reattached.
- A second instance of the same instance (database Id) may exist in the Session.
- Does the object have to exist in the database.

Session.lock

Use session.lock when you expect that the object is unchanged and a new instance is not loaded within the same session.

If you apply any changes before you lock the instance, your session will end in an inconsistent state. To be more precise the changes will exist in the session but they will not be written to the database. Hibernate thinks that it is unchanged.

```
contact1.setFirstName("Peter");
tx = session.beginTransaction();
session.buildLockRequest(LockOptions.NONE).lock(contact1, LockMode.None);
tx.commit();
```

lock(..) throws an *org.hibernate.NonUniqueObjectException* when the instance is already in the session.

This problem can happen in the following situations:

You have methods retrieving new object instances without using the existing detached object.

Your object has a relation to another object X. When X and its relation are loaded then you have two instances as well. For example when you load a Tree class that itself fetches the leafs and then you try to reattach a leaf.

You have different options for the LockOptions:

LockOptions.NONE Tries to get object from cache, if not present read it from the database. It does not create a database lock. *LockOptions.READ* Bypasses cache and reads object from the database. *LockOptions.UPGRADE* Bypasses cache and creates a lock using select for update statement. This might wait for a database timeout. By default the lock request will wait for ever. If supported by your database and your JDBC driver, you can set a timeout. A timeout value of 0 is nowait.

```
session.getTransaction().begin();

Session.LockRequest lockRequest = session.buildLockRequest(LockOptions.UPGRADE);
lockRequest.setTimeOut(10);
lockRequest.lock(john);
session.getTransaction().commit();
session.close();
```

If Hibernate does not find the object in the database it will throw an exception.

```
org.hibernate.StaleObjectStateException: Row was updated or deleted by another
  transaction (or unsaved-value mapping was incorrect): [de.laliluna.hibernate.Concert#1
```

You should handle this Exception as it can happen when multiple users access your application concurrently. You can find an example in the Hibernate Struts chapter.

Session.update

If the object was changed, use *session.update* to save the changes in the database.

```
tx = session.beginTransaction();
contact1.setFirstName("Peter");
session.update(contact1);
tx.commit();
```

The method throws an *org.hibernate.NonUniqueObjectException* if an instance is already in the session. I explained in the last paragraph, in which situation this can happen.

Session.saveOrUpdate

If you do not want to distinguish between inserting or updating an object, you can use *session.saveOrUpdate*. This is a combination of save and update.

The following code will insert the object in the first statement and will automatically update it in the second part.

```
log.debug("saveOrUpdate to insert");
Session session = HibernateSessionFactory.currentSession();
session.beginTransaction();
Concert c = new Concert();
c.setName("Peter");
session.saveOrUpdate(c);
session.getTransaction().commit();

log.debug("saveOrUpdate to reattach and update");

session = HibernateSessionFactory.currentSession();
session.beginTransaction();
session.saveOrUpdate(c);
session.getTransaction().commit();
```

Like lock and update, saveOrUpdate does not like a second instance of an object in the same session. Please, have a look at the explanation in the *session.lock* chapter.

Session.merge

Use *session.merge*, if you consider that a new instance could exist in this session. Hibernate loads the object from the session or the cache or if it is not there from the database. Then it copies the values of your old object to the new object and returns the loaded object. This object is in persistent state. The old object is not attached to the session.

Pit fall using merge

Imagine you have created an object contact.

```
Contact contact = new Contact();
contact.setFirstName("Peter");

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(contact);
tx.commit();
session.close();
```

Later you want to reattach your object to a Hibernate session and change the name to Peter. Do you think you have successfully changed the name?

```
session = sessionFactory.openSession();
tx = session.beginTransaction();
session.merge(contact);
contact.setFirstName("Peter4");
tx.commit();
session.close();
```

You have not! The method merge looks up the object in the session. If it is not found, a new instance is loaded. Then the attributes of contact are copied to the found/created object. If you change contact it has no influence on the object in the session. You must apply your changes to the new object.

The solution is to assign the found/created object to the newInstance variable. Than you can access the variable directly.

```
Contact newInstance =(Contact)   session.merge(contact);
newInstance.setFirstName("Peter4");
```

3.2.3. Deleting

We have already seen how to delete an object. A call to *session.delete* deletes an object.

```
session = sessionFactory.openSession();
session.beginTransaction();
session.delete(concert);
session.getTransaction().commit();
session.close();
```

The object to delete does not have to be in persistent state, it can be detached.

When the object was already deleted from the database before, you will encounter an exception.

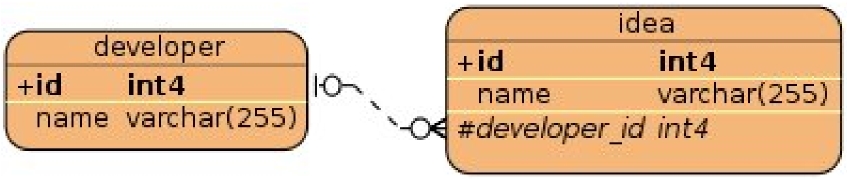
```
org.hibernate.StaleStateException: Batch update returned unexpected row
count from update: 0 actual row count: 0 expected: 1
```

To prevent this you might consider to validate that the object exists before deleting it. You might even consider to lock the object with a UPGRADE{}-lock.

```
Concert freshObject = (Concert) session.get(Concert.class, concert.getId());
 if (freshObject != null)
  session.delete(freshObject);
```

Further attention is needed when relations are used. Let us reuse the example we took above for the update problem.

We will define that an idea can only exist with a developer. This means that developer_id must not be null.



In this case you cannot delete a developer without deleting the ideas first. We have two options to delete the objects.

Deletion without cascading

Reattach your developer, delete all the ideas first, and then delete the developer.

```
session.buildLockRequest(LockOptions.None).lock(developer);
for (Iterator iter = developer.getIdeas().iterator(); iter.hasNext();) {
   Idea element = (Idea) iter.next();
   session.delete(element);
}
session.delete(developer);
```

Have a look at the test class of the relation mapping examples for further examples.

Configure cascading

Cascading is explained in detailed in chapter Cascading Section 7.3, “Cascading”. Here, I present only a short description. If you have set cascading to *delete* or *all*, then you just need to delete the developer and Hibernate will delete the ideas for you.

3.2.4. Additional commands

The Hibernate session provides more commands. I will explain the most important in this chapter.

If a column of a table is calculated by a database trigger, it might be required to reload the object. A call to refresh will reread the object from the database.

```
session.refresh(developer);
```

An object in persistent state is stored in the persistence context and cannot be garbage collected even if it is no longer referenced from your code. Sometimes you want to limit the size of the session for example to keep memory consumption low. The command *evict* removes one object from the session. If the object was changed, the SQL update statement is most likely not yet send to the database. If you do not want to loose your changes, then you should call flush to send all open changes to your database.

```
session.flush();
session.evict(developer);
```

The manual call to flush is only required in use cases as the one just described. By default flush is called when you commit a transaction.

3.3. EntityManager API

3.3.1. Saving

Saving or persisting an object is actually not very complicated. The following code will do it.

```
EntityTransaction tx = entityManager.getTransaction();
tx.begin();
Concert c = new Concert("Peter");
entityManager.persist(c);
tx.commit();
entityManager.close();
```

If you debug the code of the first example, you will see that *entityManager.persist* does not create an insert statement immediately but just selects the next sequence value. This is at least true, if you use a sequence as id generator.

It is the *tx.commit()* of the transaction, which leeds to sending the SQL insert statement to the database. Calling *persist* makes an object persistent. If the id is configured to be generated then Hibernate will ensure that it is generated and set the value of the id property. In addition the entity is added to the persistence context of the session.

Persistence Context

Simplified: the persistence context contains a map for each entity. The key of the map is the id and the value is a composite of the entity instance itself and the original values of all fields when the object was added to the persistence context. The latter allows Hibernate to determine if a persistent object needs to be updated in the database.

When the persistence context is flushed, Hibernate will send all insert statements to the database, determines all required updates by comparing the entities to the original field values and sends all updates to the database and finally all deletes are sent.

The call to *commit* will cause a flush of the persistence context before the commit is sent to the database.

There are multiple reasons for this behavior:

Less update statements

If you change multiple fields, add a relation while an entity is in persistent state, Hibernate will only send one update statement at the end.

JDBC batching

Hibernate can group updates and make use of JDBC batching. This is more efficient as compared to send statements one by one.

Reduced duration of locks

Sending an insert or update statement will cause a row or page lock depending on the database. Sending such statement at the end just before the transaction commit, will cause the lock to exist only for a short timespan. It reduces the risk of concurrent users waiting for locks or dead lock situations.

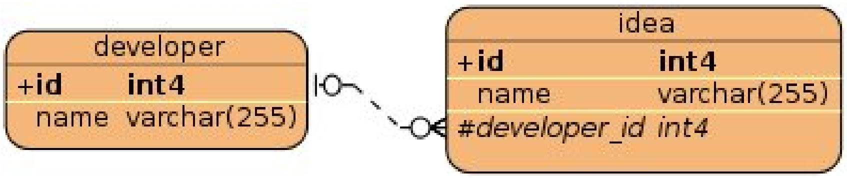
Therefor, Hibernate will do only the minimum required to determine the id of the entity and add the entity to the persistence context of the session. If the id is a generated value and the generator is a sequence or a table, Hibernate will just select the value. But if the id is generated on inserting - for example increment column in MS SQL server - then Hibernate needs to send the insert table to get an id generated.

Saving becomes more complex, when you start to use relations.

We are having the following tables mapped as 1:n relation. We will define that *developer_id* must not be null, an idea cannot exist without a developer. The database will create a not null constraint for the column *developer_id*.

```
Developer d = new Developer();
Idea idea = new Idea();
d.getIdeas().add(idea);
idea.setDeveloper(d);
```

You must take care that the idea is not saved before the *developer_id* is defined. If you save the idea first, then the *developer_id* is not yet defined and you will receive a constraint violation exception.



Use case: no cascading defined

You must call save in the following order:

```
entityManager.persist(developer);
entityManager.persist(idea);
```

Use case: cascading from developer to idea

You only need to call

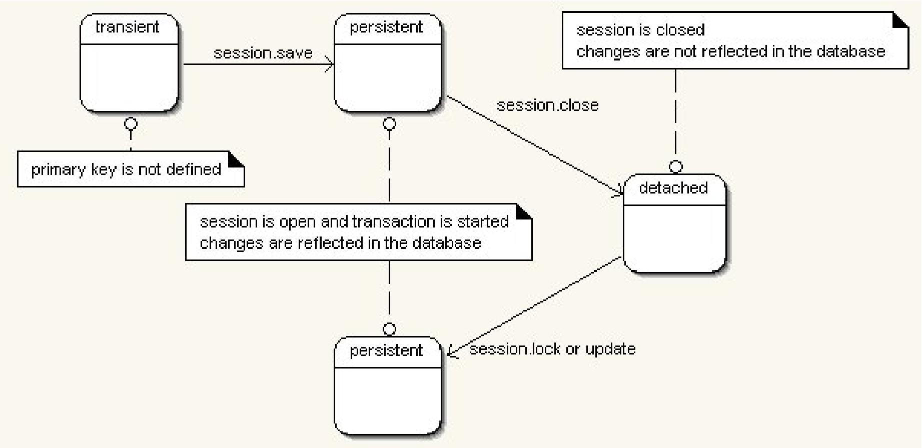
```
entityManager.persist(developer);
```

3.3.2. Updating

Considerations

We talked in chapter Hibernate States Chapter 2, *Hibernate Concepts - State of Objects* about the states of an object. Below you can find the figure we used in that chapter.

Updating an entity requires that you get your object in persistent state. One approach is to fetch an object from the database within a transaction, apply your changes and commit the transaction.


transient
session.save
persistent
session is closed
changes are not reflected in the database
session.close
primary key is not defined
detached
session is open and transaction is started
changes are reflected in the database
persistent
session.lock or update

```
EntityTransaction tx = entityManager.getTransaction();
tx.begin();
Country country = entityManager.find(Country.class, 4711);
country.setName("United Kingdom");
tx.commit();
entityManager.close();
```

In the code above there is no *merge*. Working with Hibernate means working with objects and states. It is quite different from JDBC and sending *insert* or *update* SQL statements to the database.

Java Persistence has only one method to make an existing object persistent: merge.

Hibernate loads the object from the session or the cache or if it is not there from the database. Then it copies the values of your old object to the new object and returns the loaded object. This object is in persistent state. The old object is not attached to the session.

Pit fall using merge

Imagine you have a reference to a detached object. You want to reattach your object to a EntityManager and change the name to Peter. Do you think you have successfully changed the name?

```
EntityTransaction tx = entityManager.getTransaction();
tx.begin();
entityManager.merge(contact);
contact.setFirstName("Peter4");
tx.commit();
```

You have not! The method merge looks up the object in the session. If it is not found, a new instance is loaded. Then the attributes of contact are copied to the found/created object. If you change contact it has no influence on the object in the session. You must apply your changes to the new object.

The solution is to assign the found/created object to the newInstance variable. Than you can access the variable directly.

```
Contact newInstance = entityManager.merge(contact);
newInstance.setFirstName("Peter4");
```

3.3.3. Deleting

An object can be removed by calling *entityManager.remove*.

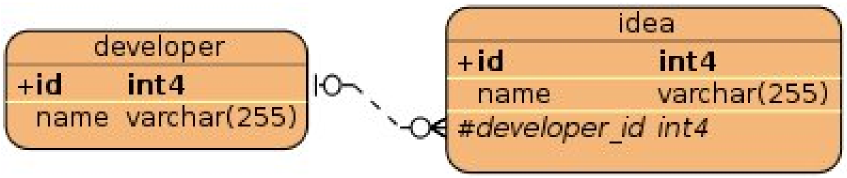
```
EntityTransaction tx = entityManager.getTransaction();
tx.begin();
Contact merged = entityManager.merge(contact);
entityManager.remove(merged);
tx.commit();
entityManager.close();
```

There is a difference compared to Hibernate Session API. You need to make the object persistent before removing it. When the object was already deleted from the database before, you will encounter an exception.

```
org.hibernate.StaleStateException: Batch update returned unexpected row
count from update: 0 actual row count: 0 expected: 1
```

Further attention is needed when relations are used. Let us reuse the example we took above for the update problem.

We will define that an idea can only exist with a developer. This means that developer_id must not be null.



In this case you cannot delete a developer without deleting the ideas first. We have two options to delete the objects.

Deletion without cascading

Reattach your developer, delete all the ideas first, and then delete the developer.

```
em = factory.createEntityManager();
em.getTransaction().begin();
paolo = em.merge(paolo);
for (Idea idea : paolo.getIdeas()) {
    em.remove(idea);
}
em.remove(paolo);
em.getTransaction().commit();
em.close();
```

Have a look at the test class of the relation mapping examples for further examples.

Configure cascading

Cascading is explained in detailed in chapter Cascading Section 7.3, “Cascading”. Here, I present only a short description. If you have set cascading to delete or all you can just delete the developer and Hibernate will delete the ideas for you.

3.3.4. Additional commands

In this section, I will describe further commands provided by the EntityManager.

EntityManager.lock

Use entityManager.lock can be used to place a lock on an object in persistent state. This method behaves differently as Hibernate’s session, which makes an object persistent as well.

```
EntityManager em = factory.createEntityManager();
em.getTransaction().begin();
player = em.merge(player);
em.lock(player, LockModeType.PESSIMISTIC_READ);
em.getTransaction().commit();
em.close();
```

There are different options for the LockModeType:

LockModeType.NONE This is the default behaviour. If entites are cached, the cache is used. If a version column is present, the version is incremented and verified during the update SQL statement. You will find more details about *@Version* in chapter Optimistic Locking Section 13.8.1, “Optimistic Locking”. *LockModeType.OPTIMISTIC* The object is read from the database and not from the cache. If it was already read from the database, it will not be read again. If a version column is defined and the object was changed, then the version is incremented and verified during the update. This is the same as the JPA 1 option: *LockModeType.READ* *LockModeType.OPTIMISTIC_FORCE_INCREMENT* Same as *OPTIMISTIC* but will increment the version even if the object was unchanged. This is the same as the JPA 1 option: *LockModeType.WRITE* *LockModeType.PESSIMISTIC_READ* Reads the object from the database for shared usage. For a PostgreSQL database the SQL looks like *select id from Player where id =? for share*. *LockModeType.PESSIMISTIC_WRITE* Tries to create an update lock and throws an exception if this is not immediately possible. This is not supported by all databases. The object is read from the database for shared usage. For a PostgreSQL database the SQL looks like *select id from Player where id =? for update*. If a version column exists and the object was changed, the version is incremented and verified during the update.

I could not find a lock with no wait option as supported by the Hibernate Session API. The query hint to timeout the lock seems to be database dependent. It does not work with PostgreSQL, though it is supported by the database.

```
Map<String,Object> map = new HashMap<String, Object>();
map.put("javax.persistence.lock.timeout", 2);

em.find(Player.class, 1, LockModeType.PESSIMISTIC_WRITE, map);
```

There are bugs related to locking which were fixed with version 3.6 Final See https://hibernate.onjira.com/browse/HHH-5032

Reloading an entity

If a column of a table is calculated by a database trigger, it might be required to reload the object. A call to refresh will reread the object from the database.

```
em.refresh(developer);
```

An object in persistent state is stored in the persistence context and cannot be garbage collected even if it is no longer referenced from your code. Sometimes you want to limit the size of the session for example to keep memory consumption low. The command *evict* removes one object from the session. If the object was changed, the update statement is most likely not yet send to the database. If you do not want to loose your changes, then you should call flush to send all open changes to your database.

```
em.flush();
em.detach(developer);
```

We have to consider Hibernate behaviour when using detach. Hibernate does not write all changes to an object immediately to the database but tries to optimise the insert/update statements. All collected changes are written to the database, before you execute a query or if you commit a transaction.

The manual call to flush is only required in special use cases. By default the persistence context is for example flushed when you call *commit* and before the commit is actually sent to the database.

Chapter 4. A more complex example – web application

I would like to demonstrate a real world web application using Hibernate. Step by step, you can create an application which allows to edit books.

We will implement

- a sortable grid displaying books
- a create page
- an edit page

This should take you about 15 minutes, if everything runs fine and assuming that you have Maven installed and that you have some experience to deploy a web application to a servlet engine.

What is Maven?

Maven is a dependency management solution and project build tool.

- It allows to define libraries which are required by your application.
- It downloads the libraries and libraries required by those libraries from Maven repositiories on the Internet.
- It can setup an Eclipse, Netbeans or IntelliJ project.
- It can package your own projects for production deployment as well.
- The central Maven repositiories provide nearly all Java Open Source libraries.

I would not replace my IDE build during development but it is a great help to manage dependencies and build a project for production deployment.

Setup the project (5 minute)

You need an Internet connection and you need to have Maven installed. http://maven.apache.org/ I have used Maven version 3.

In a shell input the following command. This needs to be on one line.

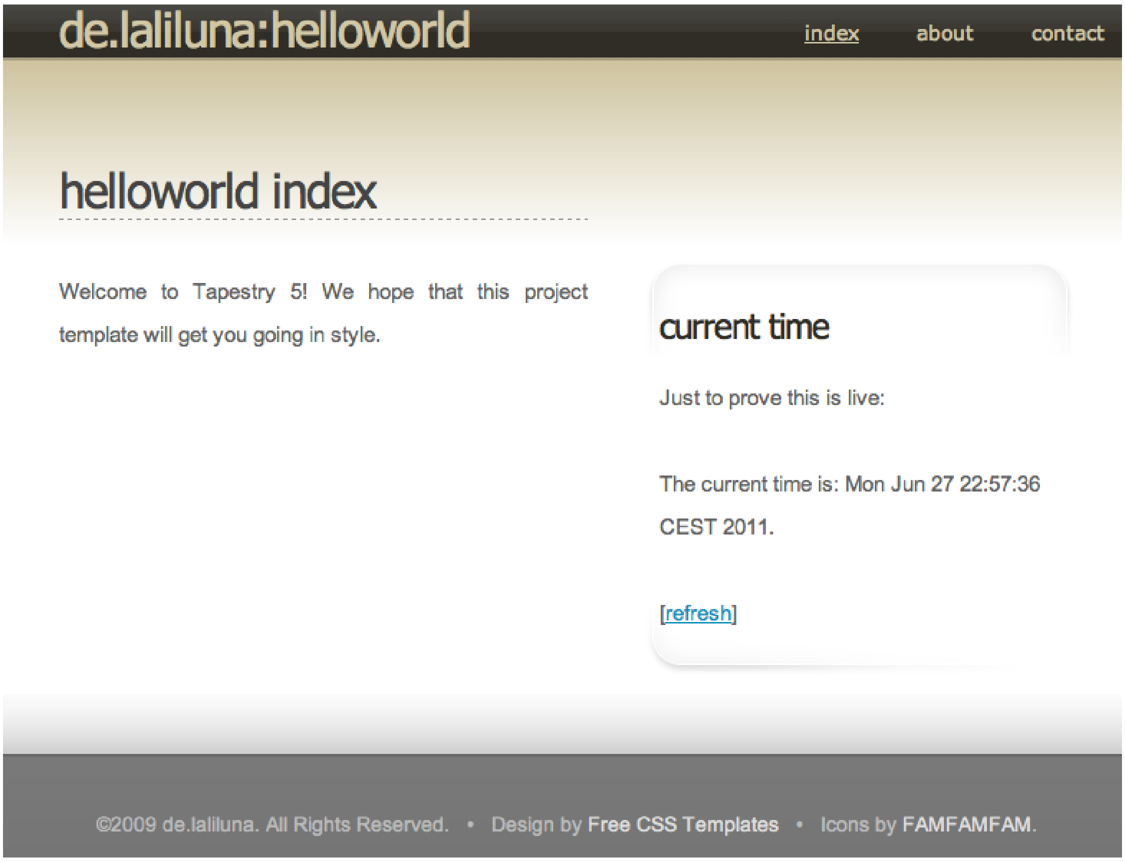
```
mvn -DarchetypeVersion=5.2.5 -Darchetype.interactive=false -DarchetypeArtifactId=quicksta
 -Dversion=1.0-SNAPSHOT  -DarchetypeGroupId=org.apache.tapestry  -DgroupId=de.laliluna \
 -Dpackage=de.laliluna.helloworld -DartifactId=helloworld --batch-mode \
 -DarchetypeRepository=http://tapestry.apache.org archetype:generate
```

It will setup a new quickstart project. If you prefer to setup the project manually, have a look in the Tapestry articles on my website http://www.laliluna.de

To have a first look at your application, tell Maven to download the dependencies. Type the following command in a shell.

```
cd helloworld
mvn dependency:resolve

# Start an internal web server
mvn jetty:run
```

Visit http://localhost:8080/helloworld in your browser.



helloworld index

Welcome to Tapestry 5! We hope that this project template will get you going in style.

current time

Just to prove this is live:

The current time is: Mon Jun 27 22:57:36 CEST 2011.

[refresh]

©2009 de.laliluna. All Rights Reserved. • Design by Free CSS Templates • Icons by FAMFAMFAM.

We are going to use the *Tapestry Hibernate Module*. In addition we need to add the JDBC driver for the database.

Edit the Maven build file *myPathToTheProject/helloworld/pom.xml*

In the section <*dependencies*> add the following dependency:

Maven pom.xml.

```
<dependencies>

    <dependency>
        <groupId>org.apache.tapestry</groupId>
        <artifactId>tapestry-hibernate</artifactId>
        <version>${tapestry-release-version}</version>
    </dependency>

    <dependency>
        <groupId>postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>8.3-603.jdbc3</version>
    </dependency>
```

```
<!-- more dependencies -->
```

Other JDBC driver

If you use another database, you need to replace the postgresql dependency. You can find dependencies in Maven repository using search engines. For google use the search term: site:ibiblio.org mysql.

Change into the directory *helloworld* and let Maven resolve the libraries.

```
cd helloworld
mvn dependency:resolve
```

Import and run the project

IntelliJ and Netbeans support Maven out of the box. Eclipse requires the M2Eclipse plugin.

Alternatively using Eclipse you may type the following to create normal eclipse project files.

```
mvn eclipse:eclipse
```

Inside of your IDE deploy the project to a webserver like Tomcat or Jetty.

Add a Hibernate configuration (1 minute)

src/main/resources/hibernate.cfg.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>

  <session-factory>
      <property name="hbm2ddl.auto">create-drop</property>

    <property name="connection.url">
      jdbc:postgresql://localhost:5432/play
    </property>
    <property name="connection.username">postgres</property>
    <property name="connection.password">p</property>
    <property name="connection.driver_class">
      org.postgresql.Driver
    </property>
    <property name="dialect">
      org.hibernate.dialect.PostgreSQLDialect
    </property>
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>


  <mapping class="de.laliluna.helloworld.domain.Book"/>

  </session-factory>

</hibernate-configuration>
```

Create the book entity (1 minute)

src/main/java/de/laliluna/helloworld/Book.java.

```
@Entity
public class Book {

    @GeneratedValue
    @Id
    private Integer id;
    private String title;

    @Temporal(TemporalType.DATE)
    private Date published;

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder();
        sb.append("Book");
        sb.append("{id=").append(id);
        sb.append(", title='").append(title).append('\'');
        sb.append(", published=").append(published);
        sb.append('}');
        return sb.toString();
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Date getPublished() {
        return published;
    }

    public void setPublished(Date published) {
        this.published = published;
    }

}
```

Create a sortable data grid (3 minutes)

In the *Index.tml* page, we are going to add the grid.

Index.tml.

```
<html t:type="layout" title="newapp Index"
      t:sidebarTitle="Current Time"
      xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
      xmlns:p="tapestry:parameter">

<t:grid source="allBooks" />
... snip ...
```

Tapestry has a concept of a template and a corresponding Java class. The template can access properties of the Java class. As the grid uses *allBooks*, we need to add a *getAllBooks* method to the *Index.java* class.

Index.java.

```
import org.apache.tapestry5.ioc.annotations.Inject;
import org.hibernate.Session;

import java.util.*;

/**
 * Start page of application newapp.
 */
public class Index
{

    @Inject
    private Session session;

   public List<Book> getAllBooks(){
      return session.createQuery("select b from Book b").list();
   }

... snip ...
```

Redeploy your application and have a look at your sortable data grid.

As it is annoying, to have an empty grid, we will add an *action link* to create random books. An action link calls an action method on the Java class.

Index.tml.

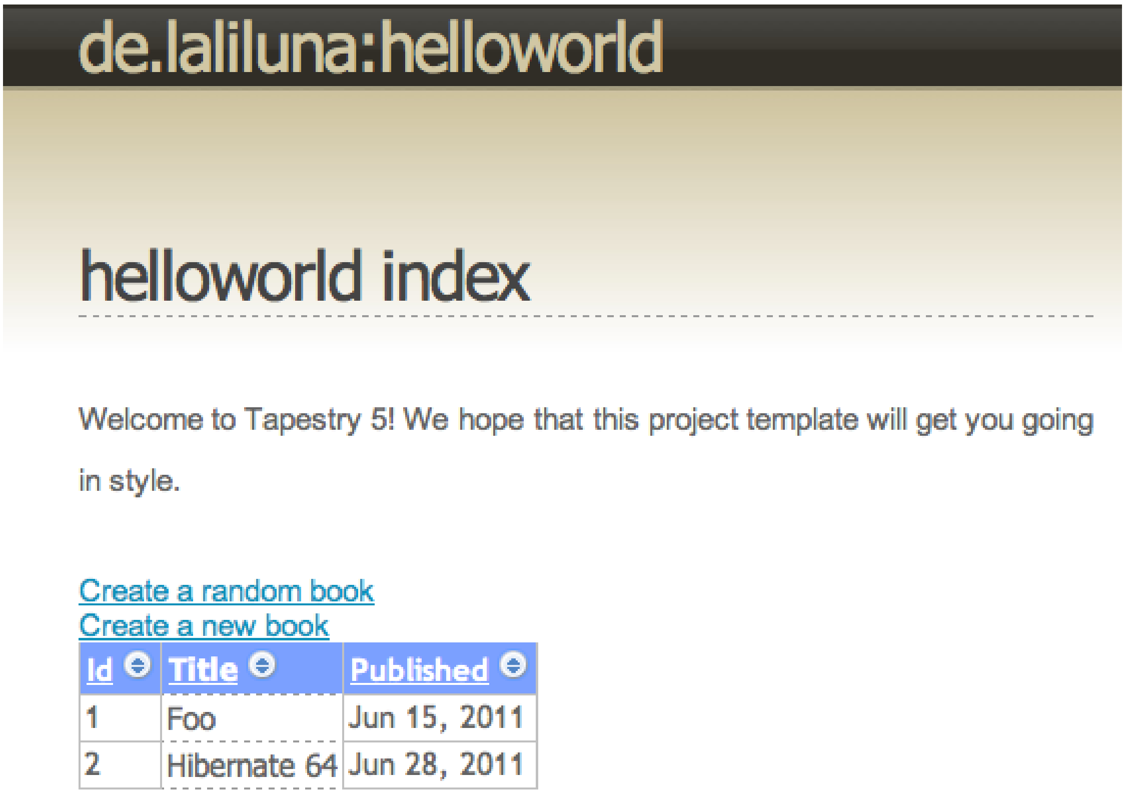
```
<html t:type="layout" title="helloworld Index"
      t:sidebarTitle="Current Time"
      xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
      xmlns:p="tapestry:parameter">

        <div>
      <t:actionlink t:id="createRandomBook">Create a random book</t:actionlink>
      </div>
        <t:grid source="allBooks" />
... snip ...
```

The corresponding action method in the *Index.java* file creates a book. Please take care that the method name corresponds to the *t:id* of the action link. Tapestry uses a lot of conventions like that. As the method returns null, you will stay on the same page.

Index.java.

```
public class Index{

    @Inject
    private Session session;
    @CommitAfter
    public Object onActionFromCreateRandomBook(){
        session.save(new Book("Hibernate " + new Random().nextInt(100),
         new Date()));
        return null;
    }
... snip ...
```

You should be able to create books, see them in the table and sort them by now.



de.laliluna:helloworld

helloworld index

Welcome to Tapestry 5! We hope that this project template will get you going in style.

Id	Title	Published
1	Foo	Jun 15, 2011
2	Hibernate 64	Jun 28, 2011

Dialog to create a book (4 minutes)

Of course, you need to create real books as well and input the data.

Create a new template for the dialog.

/src/main/resources/de/laliluna/helloworld/pages/CreateBook.tml.

```
<html t:type="layout" title="Create a book"
      t:sidebarTitle="Current Time"
      xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
      xmlns:p="tapestry:parameter">

<t:beaneditform exclude="id" object="book"/>

</html>
```

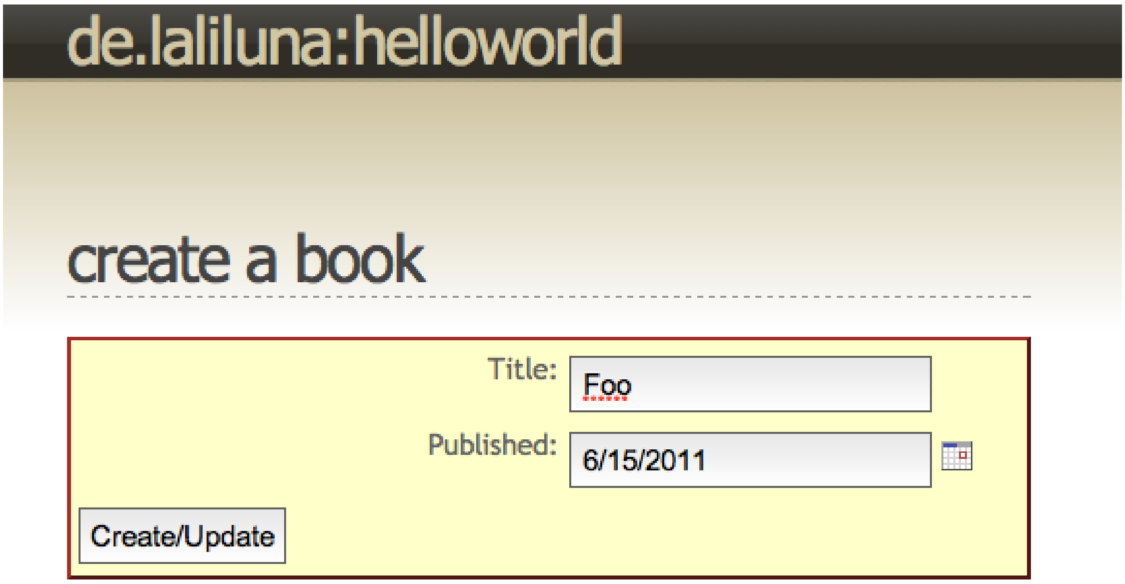
The *t:beaneditform* is a powerful Tapestry component which creates a dialog directly from your model. The model must be provided in the corresponding Java class. In addition, the *CreateBook.java* class has a *onSuccess* method being called, when the form is submitted.

/src/main/java/de/laliluna/helloworld/pages/CreateBook.java.

```
public class CreateBook {

    /*
    @Property makes the book available to the EditBook.tml page.
     */
    @Property
    private Book book;

    @Inject
    private Session session;

    /*
    Inject a page. It is used later for navigation purpose.
     */
    @InjectPage
    private Index index;

    /**
     * Commit the transaction using {@code session.getTransaction().commit()}
     * right after themethod was executed.
     */
    @CommitAfter
    Object onSuccess() {
        session.saveOrUpdate(book);
        /*
        Return the injected page to navigate to the page.
         */
        return index;
    }
}
```

Finally, add a link to your *index.tml* to be able to navigate to the new page.

```
<div>
    <t:pagelink page="CreateBook">Create a new book</t:pagelink>
</div>
```



Title: Foo

Published: 6/15/2011

The *t:beaneditform* component is powerful because it is flexible. You can override each input elements and adapt it to your needs. Let’s add a character LOB to the book and a <*textarea*> input to the dialog.

Book.java.

```
@Entity
public class Book {

    @Lob
    private String description;

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
... snip ...
```

We need to change the *CreateBook.tml* to override the description to make use of an textarea.

CreateBook.tml.

```
<html t:type="layout" title="Create a book"
      t:sidebarTitle="Current Time"
      xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
      xmlns:p="tapestry:parameter">

<t:beaneditform exclude="id" object="book">
    <t:parameter name="description">
        <t:label for="description">My description</t:label>
        <t:textarea t:id="description" value="book.description"/>
    </t:parameter>
</t:beaneditform>
</html>
```

Just check that everything is working.

Order of properties

The order of getter and setter in the *Book class* is relevant for the order of the form properties. You might consider to add the getter and setter of the *description* field at the end. Alternatively, there is an @Reorder annotation, which allows to define the order in the grid and in the *t:beaneditform*.

Editing a book (3 minutes)

In order to open an edit book dialog, we will modify the table displaying all books. Clicking on the book title in the grid, should allow you to edit a book. We need to modify how the title is rendered in the grid, add a new template for the edit dialog and a corresponding Java class with the Hibernate code to update the book.

The link first:

Modify the *t:grid* component. We override the cell for the title.

Index.tml.

```
<t:grid source="allBooks" row="book">
    <p:titleCell>
        <t:pagelink page="EditBook" context="book.id">${book.title}</t:pagelink>
    </p:titleCell>

</t:grid>
```

The grid makes use of a *book* property to store the current book while iterating. Therefor the corresponding Java class *Index.java* needs to provide such a property. Add the following code.

Index.java.

```
@Property
private Book book;
```

Create a new template.

/src/main/resources/de/laliluna/helloworld/pages/EditBook.tml.

```
<html t:type="layout" title="Edit a book"
      t:sidebarTitle="Current Time"
      xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
      xmlns:p="tapestry:parameter">

<t:beaneditform exclude="id" object="book">
    <t:parameter name="description">
        <t:label for="description">Description</t:label>
        <t:textarea t:id="description" value="book.description"/>
    </t:parameter>
</t:beaneditform>

</html>
```

Create the corresponding Java class.

/src/main/java/de/laliluna/helloworld/pages/EditBook.java.

```
public class EditBook {

    /*
    @Property makes the book available to the EditBook.tml page.
     */
    @Property
    private Book book;

    @Inject
    private Session session;

    /*
    Inject a page. It is used later for navigation purpose.
     */
    @InjectPage
    private Index index;

    /**
     * Is called before the page is rendered. A value encoder provided by the
     * Tapestry Hibernate module, knows how to convert the id at the end of the
     *  URL localhost:8080/editbook/2 into an instance of {@link Book}
     * @param book the book to edit
     */
    void onActivate(Book book){
        this.book = book;
    }

    /**
     * On redirecting to the same page for example, if validation fails, it
     * is required to make Tapestry aware of the page context object.
     *
     * Otherwise you will loose the information which book was edited.
     * @return the edited book
     */
    Book onPassivate(){
        return book;
    }

    /**
     * Commit the transaction using {@code session.getTransaction().commit()}
     * right after the method was executed.
     */
    @CommitAfter
    Object onSuccess() {
        session.saveOrUpdate(book);
        /*
        Return the injected page to navigate to the page.
         */
        return index;
    }
}
```

You have completed a first web application using Hibernate.

4.1. Summary

How does it work behind the scenes?

Tapestry provides it own Dependency Injection Framework. It is used internally to deal with the configuration, page and component handling. You can use it for your own code as well or make use of the integration with Spring, Google Guice, or EJB3.

Dependency injection inject all what you need into a page. The *@Inject* annotation in the following code let Tapestry inject a Hibernate Session.

```
public class EditBook {

    @Inject
    private Session session;
```

Where does the session come from?

The Tapestry Hibernate module makes use of distributed configuration supported by Tapestry. When Tapestry starts the module, it builds a Hibernate *SessionFactory* and registers a factory for the Hibernate session. Whenever you inject a Hibernate session it is created by the factory using the normal *sessionFactory.openSession()* you have already used in the first example.

The module registers a method interceptor as well. It is called whenever the Tapestry Dependency Injection framework finds a *@CommitAfter* annotation. It is pretty common to let the transaction be handled by dependency injection frameworks. You might have come across the term Aspect Oriented Programming (AOP). Transaction handling is an aspect, which can be provided by method interceptors and there is no need to bloat your code with it.

In fact many technologies (EJB, Spring, Google Guice, Pico-Container) makes use of this approach.

I hope you got a good impression, how Hibernate can be integrated into a web framework. Component based frameworks are great as you can use and develop powerful reusable components. I can only recommend to try out Tapestry. It is one of the best frameworks, I am aware of. Have you seen the live class reloading of Tapestry? You only need to save a file and reload the page in your browser to see your pages.

By the way, you will find a powerful Hibernate integration in the Wicket Framework as well.

Part II. Mapping, Queries

Chapter 5. Basic Mappings

This chapter contains a large selection of examples. The complete examples including mapping, classes and a test class showing how to insert, update, delete and query the mapped objects can be found in the example Java project *mapping-examples-xml* for XML mappings and *mapping-examples-annotation* for annotation mapping.

5.1. Annotation versus XML

There are two approaches to mapping: XML and Annotation. The latter is standardised as Java Persistence and Hibernate supports it since version 3.x. Annotation mapping requires Java 1.5 alias 5 and is the preferred mapping approach by many people.

Annotation mappings are defined directly in the class.

This book contains many but not all existing mappings. I recommend two sources to look up further mappings. The hibernate reference and the JUnit test cases of Hibernate. The source code provided with the Hibernate download has a test folder containing many complicated and inspiring variations.

Class mapped with annotations.

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
@Entity
@Table(name="computer")
@SequenceGenerator(name="computer_seq", sequenceName="computer_id_seq")
public class Computer implements Serializable{
   private static final long serialVersionUID = 3322530785985822682L;
   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="computer_seq")
   private Integer id;
   private String name;
   public Computer() {
   }
   public Integer getId() {
      return id;
   }
   public void setId(Integer id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String toString() {
      return "Computer: " + getId() + " Name: " + getName();
   }
}
```

Annotations start always with an @ character. *@Entity* defines that this class is a mapped class. *@Table* specifies that the entity is mapped to the table computer *@SequenceGenerator* specifies the Hibernate name and the database name of a sequence, which can be used to generate unique values. *@Id* defines the primary key of the class and @GeneratedValue specifies that the id is generated by the sequence, we defined at the beginning of the class.

Careful chosen default behaviour. Annotation mapping has careful chosen default behaviour. Hopefully in most cases you do not have to write anything to get the wanted behaviour.

Advantages of annotation mapping over XML mapping files

- Mapping is included directly in the class, what I consider clearer
- Less definitions to type, because of well chosen default values (→ I did not need to put any annotations in front of the name attribute in the former example.
- Faster to develop
- Careful chosen default values

Disadvantages

- Missing features, e.g. natural-id, some rare mapping types
- Java 1.5 required
- If you serialize the class and send it to another system (e.g. JEE) those system needs to have the Jar-files containing the annotation as well. This could be an annoyance, in case you connect to various backend systems with different Hibernate versions.

Opinion: I think that annotation mapping is clearer, faster and represents the future approach to mapping. You might consider to use it as well. In the next chapters, I will explain both approaches to you.

5.2. Annotation mapping

What are annotations?

An annotation is a tag which can be added to the source code. Annotations are used to add special behaviour, suppress warnings or to define Hibernate mappings. It starts with an @ and can have parameters. The following source code is having an annotation indicating a one-to-many relation:

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "developer_id")
public Set<Hobby> getHobbies() {
   return hobbies;
}
```

A annotation resembles an interface declaration. Here is the source code of the *@SequenceGenerator* annotation.

```
/*
 * The contents of this file are subject to the terms
 * of the Common Development and Distribution License
 * (the License).  You may not use this file except in
 * compliance with the License.
 *
 * You can obtain a copy of the license at
 * https://glassfish.dev.java.net/public/CDDLv1.0.html or
 * glassfish/bootstrap/legal/CDDLv1.0.txt.
 * See the License for the specific language governing
 * permissions and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL
 * Header Notice in each file and include the License file
 * at glassfish/bootstrap/legal/CDDLv1.0.txt.
 * If applicable, add the following below the CDDL Header,
 * with the fields enclosed by brackets [] replaced by
 * you own identifying information:
 * "Portions Copyrighted [year] [name of copyright owner]"
 *
 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
 */
package javax.persistence;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface SequenceGenerator {
    String name();
    String sequenceName() default "";
    int initialValue() default 1;
    int allocationSize() default 50;
}
```

Field, Method, Class annotations

Annotations can be assigned to a class, method or a field.

```
@Entity
public class BoxTurtle implements Serializable {
    @EmbeddedId
```

This is not an option but defined in the annotation. In the source code of an annotation you can see where you can add the annotation. The following code shows an annotation which can be added to a field or a method.

```
@Target({METHOD, FIELD})
```

Pitfall: You must choose if you add your annotations to the field or the methods. If you add an @Id annotation to your id field, Hibernate ignores any annotations to methods and vice versa.

Requirements to use annotations

In order to use annotation mapping you need the following things:

- J2SDK 1.5 / 5 or greater for support of annotation
- Libraries from http://annotations.hibernate.org (ejb3-persistence.jar and hibernate-annotation.jar)

Further information

In addition to the samples we provide in this chapter, you can find a complete reference of annotations in this book. Have a look in chapter Annotation reference Section A.1, “Annotation Reference”.

5.2.1. Mapping fields

The following is the shortest possible mapping. It marks the class as Hibernate entity and the id field with *@Id*. The entity will be stored in a table player having two columns: id and name. Annotation mapping treats all fields as mapped by default. Inherited fields are not mapped by default.

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Player {

   @Id
   private Integer id;

   private String name;


    public Player() {
    }

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

    public Integer getId() {
      return id;
   }

   public String getName() {
      return name;
   }

}
```

If you do not want to map a field, you need to mark it with *@Transient*.

```
@Entity
public class Player {

    @Transient
    private int value;
```

```
.....
```

If you want to map fields of parent classes you need to mark the parent class with *@MappedSuperclass*.

```
@MappedSuperclass
public class AbstractPlayer {
    @Id
    @GeneratedValue
    protected Integer id;
    protected String name;
}
```

If you want to be more explicit, you can use the *@Basic* and the *@Column* annotation. *@Basic* allows to define that a single field is not loaded immediately but lazy when the getter is called for this field. This is used extremely rare for single fields and in order to get this to work, you need to use byte code instrumentation. In addition *@Basic* allows to define that a field is not nullable what is possible with *@Column* as well.

```
@Column(name = "player_name", unique = true, nullable = false,
   updatable = false, length = 5)
private String name;
```

The sample maps the field *name* to the database column *player_name*. *unique, nullable and length* are hints for the schema generation. If you let Hibernate generate your tables, unique or not null constraints will be created. Length corresponds to the length of the varchar column.

Furthermore Hibernate will check **nullable=false** constraints itself, without sending a query to the database.

updatable allows to disable update statements to a field. You should use it for immutable fields. An immutable field can be saved once but an update is not possible. The field will not be included in the fields of an update statement. There is an *insertable* option available as well, which can turn a field into a read only value.

java.util.Date

A *java.util.Date* could be a date, a time or a date and time value. You can specify the type for a field using *@Temporal*

```
@Temporal(TemporalType.DATE)   // TIMESTAMP, TIME or DATE
private Date birthDay;
```

Enum types

Annotation mapping supports enum types without using a user defined type.

```
public class Player {
   public enum Experience {BASIC, MEDIUM, PROFESSIONAL}

   @Enumerated
   private Experience experience;
```

The value are stored with the enum type’s ordinal value, which is 0 for the first enum value (BASIC), 1 for the second and so on.

If you prefer a text representation (for example BASIC for Experience.BASIC), you can use *@Enumerated(EnumType.STRING)*. I tend to use a text representations for all tables which are not very big, as there are more readable.

Components

A component is a class containing multiple fields. It is a good object oriented development practise to compose a class of other classes instead of having a single class with a lot of fields. Imagine the frequent use case of addresses. We want to store street and city of the player. Instead of adding individual fields, a new class *Address* is created containing the address fields. You need to mark it as *@Embeddable*. We will see components in greater detail later.

```
@Embeddable
public class Address {
    private String street;
    private String city;

    public String getStreet() {
        return street;
    }

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

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}
```

Inside of the player class you just need to add the field and add the *@Embedded* annotation. The fields of the address are by default stored in the same table as the player.

```
public class Player {
    @Embedded
    private Address address;
//....
```

5.2.2. Where to put annotations

You can add annotations to your fields and to your getter methods. First, of all you need to choose one approach. Inside of a class Hibernate looks for the *@Id* annotation. If it is found in front of a field, Hibernate will only consider annotations in front of fields. It the *@Id* is found in front of a getter method it will only look for annotations in front of getters.

Player with getter annotations.

```
@Entity
public class Player {
    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }

    @Column(name = "player_name")
    public String getName() {
        return name;
    }
```

The placement of the annotation influences the default behaviour how Hibernate is writing values from a database row to a class. Annotation in front of getters let Hibernate use the setter and getter methods, whereas annotations in front of fields let Hibernate use direct field access.

Use the *@AccessType* annotation to override the default behaviour.

```
// the name is accessed using the getter and setter method

@Access(AccessType.PROPERTY)
protected String name;
```

5.3. XML Mapping

Overview

Each mapping document starts with the tag

```
<hibernate-mapping ... >
```

This tag can include multiple mappings. Though I think that it is cleaner to have a separate XML per entity.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="de.laliluna.cachetest">
  <class name="Computer" table="tcomputer">
........ snip .......
  </class>
  <class name="Developer" table="tdeveloper">
........ snip .......
  </class>
</hibernate-mapping>
```

5.3.1. Field mapping

Each field, which should be stored, need to be added to the XML. You need to specify the field name and the type if it cannot be determined by Hibernate.

```
<class name="Player">
    <id name="id">
        <generator class="native"/>
    </id>
    <property name="birthDay" type="date"/>
    <property name="name" />

</class>
```

The field *birthDay* is of type *java.util.Date*. Hibernate cannot determine if you want to store a date, time or timestamp value. Therefor you need to specify the type. For a date field the following types are available: date, time, timestamp.

Java’s date time is mutable which renders it useless for many use cases. In addition it provides you with features like January is 0. For many years know there are plans to replace date in Java but since Java 7 does not appear, the date will not appear as well. You might consider to use a better date implementation like Joda Time.

If you treat your date as immutable, you can take advantage of Hibernate optimisations. Change your type to imm_date, imm_time or imm_timestamp and Hibernate will treat it as immutable as well. You need to take car that in your code, you will never change individual fields of a date but create a new instance and assign it to your entity.

```
// Good immutable usage
Calendar c = new GregorianCalendar();
c.setTime(john.getBirthDay());
c.add(Calendar.HOUR_OF_DAY, 1);
john.setBirthDay(c.getTime());

// Bad usage
john.getBirthDay().setHours(22);
```

There are additional attributes which allow to specify field length, not null or unique constraints. The access type can be specified as well. By default XML based mappings use the getter and setter to write fields, but you can change the approach per field.

```
<property name="name" length="20" column="player_name" not-null="true" unique="true" acc
```

Enum types cannot be persisted out of the box. You need to implement a UserType. Have a look in the Hibernate Reference for a description on writing user types.

The following table shows attributes you can set. The default behaviour is displayed as bold text. Very useful or important settings are marked with color. You only have to define properties which change the default behaviour. Most of the following is rarely needed.

<hibernate-mapping>

schema="schemaName"
The default database schema used for all mappings. You can overwrite the value per class. PostgreSql supports schemas.

catalog="catalogName"
The default database catalog used for all mappings. You can overwrite the value per class. Informix and MSSQL supports catalogs

default-cascade="none | all | save-update | delete | all-delete-orphan | delete-orphan"
Default cascading behaviour for relations. I prefer to leave this value and to define this explicitly in the relations.

default-access="field | **property** | myPropertyAccessorClass"
Default strategy used for accessing properties. Standard is property and you should keep this normally. This will use getters and setters to access fields. Field will access a property directly by its name. So your variables must be public. You can invent further methods with your own implementation of the interface org.hibernate.property.PropertyAccessor

default-lazy="true|false"
Default behaviour for loading of relations. You should leave this to true. More information on lazy loading can be found in chapter Lazy Initialization Lazy initialization

auto-import="true|false"
Allows use of unqualified class names in the query language for classes in this mapping. In most cases you appreciate this behaviour. Instead of *from de.laliluna.example.Car c where c.color='blue'* you can write *from Car c where c.color='blue'*

package="package.name"
Defines a package prefix for all class names in this mapping. This saves a lot of typing work when your classes are all in the same package.

5.3.2. Class mapping

The following table gives an overview of all tags available. Most of them are rarely needed. I have marked the most common tags with colour.

<class>

name="ClassName"
Name of the class or entity, Optional if you don’t want to change the entity name,

table="tableName"
Optional, default value is the class name. This is the name of mapped database table.

discriminator-value="discriminator_value"
This is used for inheritance mapping. Have a look at chapter *Inheritance Mapping* to read more about this. The value is used to distinguish sub classes. Acceptable values include null.

mutable="true|false"
Allows to define readonly classes. It can be used in combination with readonly cache which gives a lightning fast mapping. For performance tuning you could think of individual mappings for read only access.

schema="owner"
The database schema used for this class. This overwrites the default value. PostgreSql and other databases support schemas.

catalog="catalog"
The database catalog used for this class. This overwrites the default value. Informix, MSSQL and other databases support catalogs

proxy="ProxyInterface"
This is in my opinion only needed in special cases. You can define an interface used for lazy initialising proxies. Read more about this in the Hibernate reference. Search for proxy.

dynamic-update="true|false"

Normally update queries are prepared during the initialisation of Hibernate. If set to true, Hibernate will generate the SQL at runtime and include only changed properties. This is by far slower but useful when you use blobs fields in your mapping or if you want to allow concurrent updates to the same row. Concurrent updates = Two threads update the same row. This can work when different fields are changed. It can be used in combination with a special optimistic locking configuration.

dynamic-insert="true|false"

Same as for dynamic-update. Only difference is that in the insert statements only not null fields are included.

select-before-update="true|false"

Slows down performance and should not be used without a good reason. If true, Hibernate should never perform an SQL update unless it is certain that an object is actually modified. This is useful in case when you reattach objects and you do not want an update statement to be called in order to prevent that a trigger is called.

polymorphism="implicit|explicit"

Should only be changed with a good reason. Implicit means that a query of a super class of the mapped class will return this class. Explicit is useful when you have the normal class and a lightweight class not including all the fields. Then a query of the super class would not include the lightweight class when it is set to explicit.

where="arbitrary sql where condition"

Defines a condition always added when you retrieve objects of this class. It is useful when you want to hide data. For example the status 0 defines deleted data then you could use *where="status <> 0"*

persister="myPersisterClass"

Should only be changed with a good reason. A persister is responsible for writing the data. You could create a custom persister to save data to LDAP, a flat file, … In the Hibernate download you can find an example in the test directory *org.hibernate.test.legacy.CustomPersister*

batch-size="1"

Very interesting for tuning in special situations. Can be any number but should be reasonable. When you iterate through a list of 10 Book objects and call book.getAuthor than Hibernate will generate ten queries to fetch the author. A batch size of 4 would lead to 3 queries fetching 4,4 and 2 authors. If you use older Oracle versions and the class contains Blob/Clob you may be forced to use a batch-size of 1.

optimistic-lock="none | version | dirty | all"

Defines the optimistic locking strategy. The hibernate reference recommends strongly version. All would check all columns, dirty would check only changed columns. This would allow concurrent updates when the fields are not the same. Read more about optimistic locking in chapter TODO: Referenz nicht gefunden.

lazy="true|false"

You can disable lazy fetching of relations. Use this with care. Read more about this in chapter *Hibernate Architecture*

entity-name="EntityName"

Entity name is the name used in all queries. The default value is the class name. You do not have to set it. You can map the same class to multiple tables using different entity names. A special use case is the mapping to XML or Maps instead of classes. Read more about XML mapping and dynamic mapping in the Hibernate reference in the chapter dynamic models and XML mapping.

check="arbitrary sql check condition"

Can be used to check that a column only contains special values. For example you have an address table containing billing and delivery addresses. A type column contains either billing or delivery An example can be found in chapter *Typed relation*

rowid="rowid"

A rowID is the physical position of data. This is used for example in Orcale and speeds up update performance. Oracle only.

subselect="SQL expression"

Allows to generate views even if your database does not support it. You can define a complex select statement and map the resulting columns to the class. This can only be used with readonly mappings ⇒ mutable="false"

abstract="true|false"

Abstract means that in an inheritance mapping using union-subclass this class needs no corresponding table. Only subclasses objects are generated. You will understand this, when you read more about inheritance mapping. Read more about this in chapter *Inheritance Mapping*

node="element-name"

Only needed for XML mapping. Read more about this in the Hibernate Reference in chapter XML Mapping.

Other mapping tags

This book does include a reference of Hibernate annotations but not yet a complete reference of XML mappings. We have examples for the most common mapping situations in the next chapters. If you need a overview of all mapping tags, have a look in the Hibernate Reference provided with the Hibernate download. The chapter Mapping declaration describes the XML mappings completely.

Chapter 6. Primary key mapping

A primary key in a database is a unique identifier, you can use as key to a data row. A primary key precisely identifies a row of a table.

Table 6.1. Example of a table having a unique Integer as primary key

Primary key	Name
1	Jenny
2	Stephan
3	Sebastian

A primary key can consist of multiple column as well. In this case the combination of the columns must be unique.

Table 6.2. Example of a table having name and surname as primary key

Primary key - name	Primary key – surname	employedDate
Hennebrueder	Sebastian	March, 2nd 2005
Hennebrueder	Michael	April, 5th 2006
Jones	Jim	Mai, 8th 2007

Hibernate id

A primary key is used in Hibernate as well. It is called identifier (id) and identifies an object. Once a id is specified you cannot update it in Hibernate. Hibernate supports simple and composite ids. You can assign an id or use a generator to create an id. For example, a sequence generator, generates a unique Integer or Long value using a database sequence.

6.1. Natural versus Surrogate Ids

Natural Ids are ids, which uses real data to define a unique id. A id of a house can be city + street + house number. A surrogate id has nothing to do with the real data. It is artificial. It can be a unique number or a unique string. Do only use a natural id, if you are absolutely sure, that the columns included in this id, will never change. If in the example above the name of the street changes, you cannot update the name using Hibernate. Using SQL you will have to update the house table and all tables having a foreign relation to this table. A suitable case for a natural id could be a table holding ISO country codes:

Table 6.3. Example of a table having a natural primary key

Primary key – iso-code	German name	English name
en	Englisch	English
de	Deutsch	German

Keep in mind that even ISO codes do change over time.

See the discussion in the wikipedia as well: http://en.wikipedia.org/wiki/Surrogate_key

Source code. Source code for examples can be found in the projects *mapping-examples-xml* and *mapping-examples-annotation* in the Java package *de.laliluna.primarykey*.

6.2. Assigned Id

Useful for natural ids or in case you have your own strategy to create a unique id:

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Lion implements Serializable {
    @Id
    private Integer id;
```

Same as XML mapping:

```
   <class name="Lion" table="lion">
      <id name="id">
         <generator class="assigned"></generator>
      </id>
....... snip .......
```

6.3. Generated with Auto Strategy

The AUTO strategy will select a generator depending on your database. This is a good choice, if you need to support multiple databases.

If your database supports sequences, they will be used. By default Hibernate uses a sequence named hibernate_id_seq. Make sure that it exists if you create your database manually.

You must add the *@GeneratedValue* annotation to your primary key.

```
@Entity
public class Cheetah implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
```

Same as XML mapping:

```
  <class name="Cheetah" table="cheetah">
    <id name="id" type="integer">
      <column name="id" />
      <generator class="native">
      </generator>
    </id>
..... snip .....
```

6.4. Other Annotation Strategies

GenerationType.SEQUENCE

The sequence strategy uses a database sequence to generate the id values. The following sample uses a database sequence named *puma_id_seq* to generate the id values. For every insert the sequence is called and the value is set as id value.

Code sample.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "my_gen")
@SequenceGenerator(name = "my_gen", sequenceName = "puma_id_seq",
allocationSize = 1)
private Integer id;
```

If we change the allocation size to 50, Hibernate will apply a so-called high-low-algorithm to create the id values. A database sequence of 10 will let Hibernate generate ids from 10x50=500 to 549, a sequence value of 11 from 11x50=550 to 549. This is very useful, because Hibernate only needs to select the next sequence value for every 50teenth insert statement.

Code sample.

```
@Id
@TableGenerator(name = "puma_gen", table = "primary_keys")
@GeneratedValue(strategy = GenerationType.TABLE, generator = "puma_gen")
private Integer id;
```

Table 6.4. Id Generator Strategies

Strategy	Description
GenerationType.SEQUENCE	Uses a sequence, default name is *hibernate_seq*, (Oracle PostgreSql and other)
GenerationType.TABLE	Uses a table to store the latest primary key value (all databases)
GenerationType.IDENTITY	Special column type (MS SQL and other)

All these generators are supported by JPA as well. Hibernate adds an extension called GenericGenerator. It allows to use all kind of XML ID generators with annotation as well. Further information to these can be found in our Annotation reference in chapter Annotation Reference Section A.1, “Annotation Reference”.

6.5. Composite Id

A composite Id consists of multiple database columns mapped to a id class. There are two options to map a composite Id:

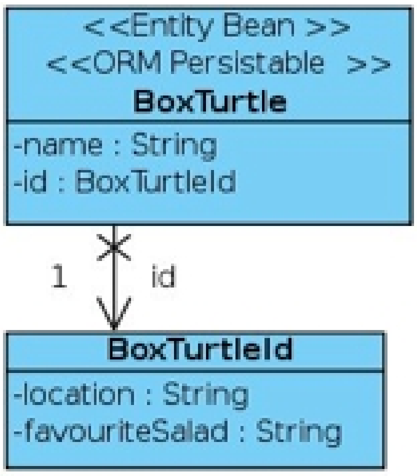
- @IdClass
- @EmbeddedId

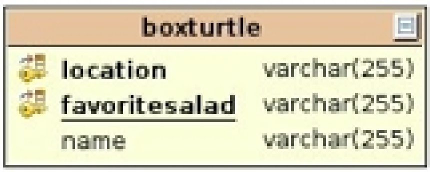
Important: You must overwrite equals and hashCode for composite id classes. If not Hibernate will think that the following id classes are different.

```
BoxTurtleId b1 = new BoxTurtleId("Bad Vilbel", "Roman salad");
BoxTurtleId b2 = new BoxTurtleId("Bad Vilbel", "Roman salad");
b1.equals(b2);
```

Eclipse and probably most IDEs provides a generator function for this. In Eclipse it is right click on the source → source → Generate hashCode, equals. You will find a detailed explanation about equals and hashCode in the next chapter.

Let’s have a look at an example: The famous box turtle for example can be clearly identified by its locations and the favourite salad. We will map it with an *@EmbeddedId*.





BoxTurtle class.

```
import java.io.Serializable;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
@Entity
public class BoxTurtle implements Serializable {

    @EmbeddedId
    private BoxTurtleId id;
```

Id class.

```
import java.io.Serializable;
public class BoxTurtleId implements Serializable {
    private String location;
    private String favoriteSalad;
```

The primary key fields are included in the Id class. The BoxTurtle class only references the Id class.

Same as XML mapping:

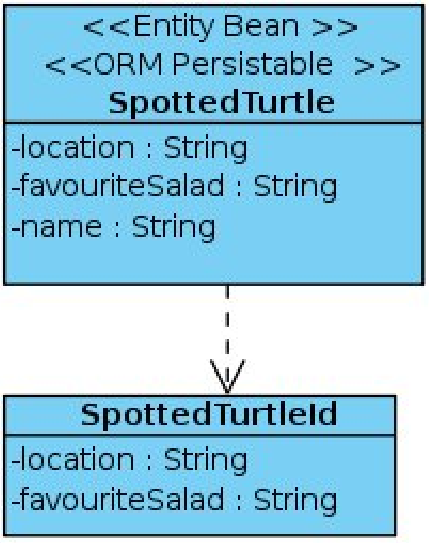
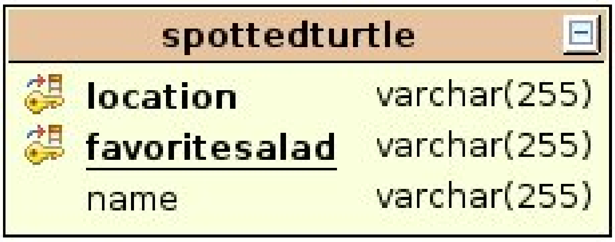
BoxTurtle.hbm.xml.

```
   <class name="BoxTurtle" table="boxturtle">
      <composite-id class="BoxTurtleId" name="id">
         <key-property name="favouriteSalad"></key-property>
         <key-property name="location"></key-property>
      </composite-id>
......... snip .........
```

Usage examples.

```
/* save/create a box turtle */
 BoxTurtleId boxTurtleId = new BoxTurtleId("Bad Vilbel", "Roman salad");
BoxTurtle boxTurtle = new BoxTurtle();
boxTurtle.setId(boxTurtleId);
session.save(boxTurtle);

/* get a box turtle from db */
BoxTurtleId boxTurtleId = new BoxTurtleId("Bad Vilbel", "Roman salad");
BoxTurtle boxTurtleReloaded = (BoxTurtle) session.get(
  BoxTurtle.class, boxTurtleId);
/* find a box turtle */
List<BoxTurtle> turtles = session.createQuery(
  "from BoxTurtle b where b.id.favouriteSalad = :salad")
  .setString("salad", "Roman salad").list();
```

The SpottetTurtle is totally different from its outlook but can identified by its location and its favourite salad as well. We will map it as @IdClass. The main difference is that the fields *location* and *favoriteSalad* are included in the *Turtle* class and the Id class. I recommend the first approach, as it provides less redundancy and is clearer in the class model.

Classes	Tables
 <<Entity Bean >> <<ORM Persistable >> SpottedTurtle -location : String -favouriteSalad : String -name : String SpottedTurtleId -location : String -favouriteSalad : String	 spottedturtle location varchar(255) favoritesalad varchar(255) name varchar(255)

SpottedTurtle class.

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;
@Entity
@IdClass(SpottedTurtleId.class)
public class SpottedTurtle implements Serializable {

    @Id
    private String location;
    @Id
    private String favoriteSalad;
```

SpottedTurtleId.

```
import java.io.Serializable;
import de.laliluna.utils.ClassUtils;

public class SpottedTurtleId implements Serializable {
    private String location;
    private String favoriteSalad;
```

The same as XML mapping:

```
   <class name="SpottedTurtle" table="spottedturtle">
      <composite-id class="SpottedTurtleId" mapped="true">
         <key-property name="favouriteSalad"></key-property>
         <key-property name="location"></key-property>
      </composite-id>
........ snip ..........
```

Usage examples.

```
/* create or save a turtle */
SpottedTurtle spottedTurtle = new SpottedTurtle("Leipzig",
  "Greek salad", "Daniel");
session.save(spottedTurtle);

/* get a box turtle from db */
SpottedTurtleId spottedTurtleId = new SpottedTurtleId("Leipzig",
  "Greek salad");
SpottedTurtle spottedTurtleReloaded = (SpottedTurtle) session.get(
  SpottedTurtle.class, spottedTurtleId);

/* find a box turtle */
List<SpottedTurtle> turtles = session.createQuery(
  "from SpottedTurtle b where b.favouriteSalad = :salad")
  .setString("salad", "Roman salad").list();
```

6.6. Equals and Hashcode

Equals and hashCode are two methods which are very important for various data structure and for Hibernate as well. You will find a lot of nonsense about equals and hashCode, therefor I will take greatest care to provide you with a good explanation.

In Java you can compare two objects using ==. The comparison returns true, if both objects have the same memory address.

```
if(foo == bar)
  log.debug("Foo has the same memory address as bar");
```

The equals method

The *equals* method is implemented by java.lang.Object and, if not overridden, will behave as ==. The intention of *equals* is to compare objects from a business point of view. For example a class *Country* has a field *isocode* like *DE* for Germany or *FR* for France etc. We could consider to compare two countries using the isocode.

```
public class Country {
    private String isocode;
    private String description;
    private int population;
    public String getIsocode() {
        return isocode;
    }

    @Override
    public boolean equals(Object o) {
        if(o instanceof Country){
            Country other = (Country) o;
            return isocode.equals(other.getIsocode());
        }

        return false;
    }
}
```

Good candidates for *equals* are fields which could be used as natural primary key or alternatively are having a unique key constraint in the database.

There are the following rules to respect when implementing the *equals* method:

Transitiv
if a.equals(b) and b.equals(c) then a.equals(c) as well.

Reflexiv
a.equals(a) should return true

Symmetric
if a.equals(b) then b.equals(a) as well.

Consistent
If you call a.equals(b) multiple times then it should always return the same value

Comparing with null
if b is null then a.equals(b) should return false.

The hashCode method

If you use your class with structures calling *equals* and *hashCode* - for example a *HashSet* or a *HashMap* - then you should override both methods. Their behaviour depend on each other. The hashCode computes a unique int value for your class. It does not need to be perfectly unique but should be well distributed. Let’s have a look at an example.

A java.util.HashSet is a structure guarantying that only one instance of an object is included. It consists of buckets (simplified!). The following code will add a new country instance to a *HashSet*. Internally, the HashSet will compute the hash code and find a bucket using a modulo operator. The *country* is added to the bucket under some conditions.

```
Set<Country> countries = new HashSet<Country>();
countries.add(new Country("DE", "Germany"));
```

If the HashSet has 16 buckets then the bucket is calculated *hashCode % 16*. If there are already entries in a bucket, then hashCode will compare all entries using the *equals* method. If no existing entry is equal, then the item will be added to the bucket. Therefor, if we implement *hashCode*, we should implement *equals* as well.

The relation between *equals* and *hashCode* is: If *equals* returns *true*, then *hashCode* must return the same value. Otherwise a structure like *java.util.HashSet* won't find the same bucket and could not guaranty that there are no two equal objects in a *HashSet*.

This relation is not equivalent. If the hash code of two objects are the same, then the two objects don't have to be equal but they can be equal.

Country with equals and hashCode.

```
public class Country {
    private String isocode;
    private String description;
    private int population;

    public String getIsocode() {
        return isocode;
    }

    @Override
    public boolean equals(Object o) {
        if(o instanceof Country){
            Country other = (Country) o;
            return isocode.equals(other.getIsocode());
        }

        return false;
    }

    @Override
    public int hashCode() {
        return isocode.hashCode();
    }
```

Further more, the computed hash code should not change while a structure contains the object. Have a look at the following code. Though we added the country object to the *HashSet*, we cannot find it any more, as the hashCode changes if we set the isocode to a different value.

```
Set<Country> countries = new HashSet<Country>();
Country country = new Country("DE", "Germany");
countries.add(country);
country.setIsocode("FR");
System.out.println(countries.contains(country)); // prints most likely false
```

HashCode rules

The hashCode should be well distributed to let structures distribute the objects on buckets well distributed.

You can change an object such that the hashCode changes but not while the object is in a *Set* or *Map* structure and you or Hibernate is using the structure.

Always implement *equals* if you implement *hashCode*.

Consider to let your IDE generate *equals* and hashCode for you. Eclipse, IntelliJ and Netbeans are capable to generate it. You just need to know which fields to use → unique keys or business keys or natural primary key candidates.

Read the book *Effective Java* from Joshua Bloch for further details.

Hibernate and equals

To answer the most important question first: Do I have to implement *equals* and *hashCode*?

Answer: It depends.

A composite id must implement both methods or the id will not work properly. Hibernate must be able to compare ids. A composite id will most likely use all its id fields in the *equals* and *hashCode* methods.

An entity does not need to implement *equals* and *hashCode* under some conditions. The persistence context guaranties that an entity with a given id exists only once. You cannot let two objects with the same id become persistent. Assuming that a class *Country* uses the isocode as ID, the following code will cause a non unique object exception.

```
Country first = new Country();
first.setIsocode("DE");
session.save(first);
Country second = new Country();
second.setIsocode("DE");
session.save(second); // NonUniqueObject exception
```

Therefor, if you save or update or merge your objects before adding them to a *Set* or *Map*, then **you do not need** to implement *equals* and *hashCode*.

But if you want to compare your objects or add them to a *Set* or *Map*, when the session is closed and the object is detached, **then you must** implement *equals* and *hashCode*. If you **do not compare** your objects with each other or do not use a *Set* or *Map*, then you **do not have to** implement *equals* and *hashCode*.

If you implement *equals*, then you should use all fields which are either business keys, unique key constraints in the database or natural primary key candidates. In many cases all three conditions are the same.

If you do not have a unique key, then you could use the id as business key, if you are careful. The simple rule is: do always save your objects before adding them to a *Set*. Saving an object will generate the id (if generated).

A sample implementation.

```
class Country{

 @Id @GeneratedValue
 private Integer id;

 @Override
 public boolean equals(Object o) {
    if(o instanceof Country){
        Country other = (Country) o;
        return id.equals(other.getId());
    }

    return false;
  }

  @Override
  public int hashCode() {
    return id.hashCode();
  }
}
```

Hibernate and equal

When implementing equals you should use *instanceof* to allow comparing with subclasses. If Hibernate lazy loads a one to one or many to one relation, you will have a proxy for the class instead of the plain class. A proxy is a subclass. Comparing the class names would fail.

More technically: You should follow the *Liskows Substitution Principle* and ignore *symmetricity*.

The next pitfall is using something like *name.equals(that.name)* instead of *name.equals(that.getName())*. The first will fail, if *that* is a proxy.

That should be all you need to know, I hope.

6.7. Other XML Id tags

<id>

name="propertyName"
Name of the class property. This should correspond to a field. name="id" when the class contains private Integer id; public Integer getId() { return id;} public void setId(Integer id) { this.id = id;}

type="typename"
A Java type like java.lang.Integer or a Hibernate type. I recommend using Hibernate types as they allow to distinguish between date, time and timestamp. Java.lang.Date cannot do this!

column="column_name"
Name of the database column. Default is taken from name attribute. But you could define one if you like.

unsaved-value="null|any|none|undefined|id_value"
Specify the value used when the object is not yet saved. This attribute is rarely needed in Hibernate 3.

access="field|property|ClassName"
Default strategy used for accessing properties. Standard is property and you should keep this normally. This will use getters and setters to access fields. Field will access a property directly by its name. So your variables must be public. You can invent further methods with your own implementation of the interface *org.hibernate.property.PropertyAccessor*

node="element-name|@attribute-name|element/@attribute|."
: Only needed for XML mapping. Read more about this in the Hibernate Reference in chapter XML Mapping.

Defines the primary key generator. You can find more details below.

Primary key generators

The generators can be subdivided into two groups. The first depends on database specific features and can only be used with the correct database. The second group is database independent. I recommend sequence or identity if supported by your database. When you need unique ids across databases you can uses uuid or guid if supported by your database. An alternative is to use composite ids. The first column is an identifier for the database. The second can be any kind of generator.

Database independent

hilo
: gets the next high value from a configured database table and generates a low value.

assigned
: Lets the application specify the primary key. Useful for natural unique keys.

foreign
: Primary key is taken from a one-to-one related class

uuid
: 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.

increment
: Do not use it. Generates only unique keys when no other thread is writing data at the same moment.

org.hibernate.id.MultipleHiLoPerTableGenerator
: Supports multiple hilo generators in a single table, defined in the EJB3 spec

Database dependent

identity
: Uses identity columns which are supported at least by DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The type depends on the database and the column. It can be long, short or integer.

sequence
: uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int.

seqhilo

The advance of seqhilo over sequence is that Hibernate can generate more than one id from one database request. It uses a hi/lo algorithm to generate identifiers of the types long, short or int. You must specify a database sequence for the high part.

guid

It uses a database-generated GUID string on MS SQL Server and MySQL. This is unique across databases.

native

Selects a generator depending on the database capabilities. Chooses between identity, sequence or hilo.

select

Retrieves a primary key assigned by a database trigger.

Composite Id

```
<class name="BoxTurtle" table="boxturtle">
  <composite-id class="BoxTurtleId" name="id">
    <key-property name="favouriteSalad"></key-property>
    <key-property name="location"></key-property>
</composite-id>
```

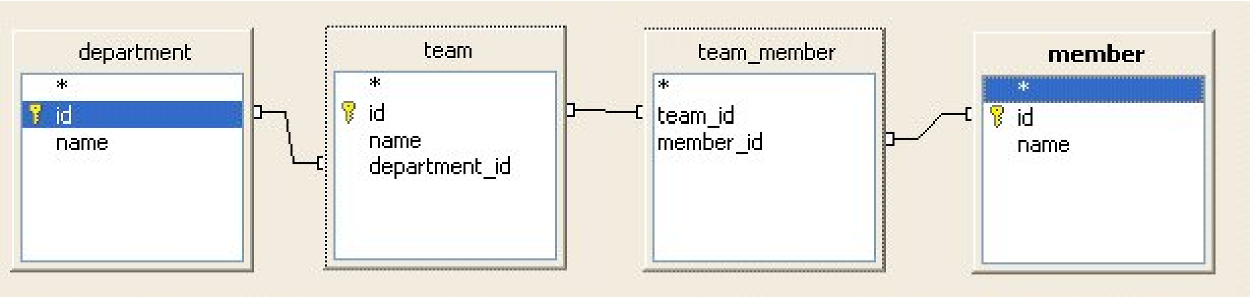
Chapter 7. Relation mapping

Relation mapping is the mapping of classes which have a relation. For example Team has a relation to a department and to a Set of Member.

Department 1:n Teams m:n Members.

```
public class Team {
   private Integer id;

   private String name;

   private Department department;

   private Set members = new HashSet();
```

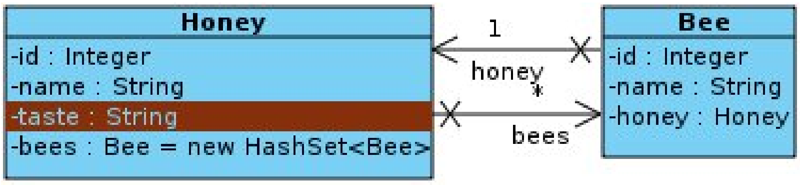
Figure 7.1. Table structure



Team can be initialized by Hibernate so that team.getDepartment will give you the Department and team.getMembers will get you a Set of Members.

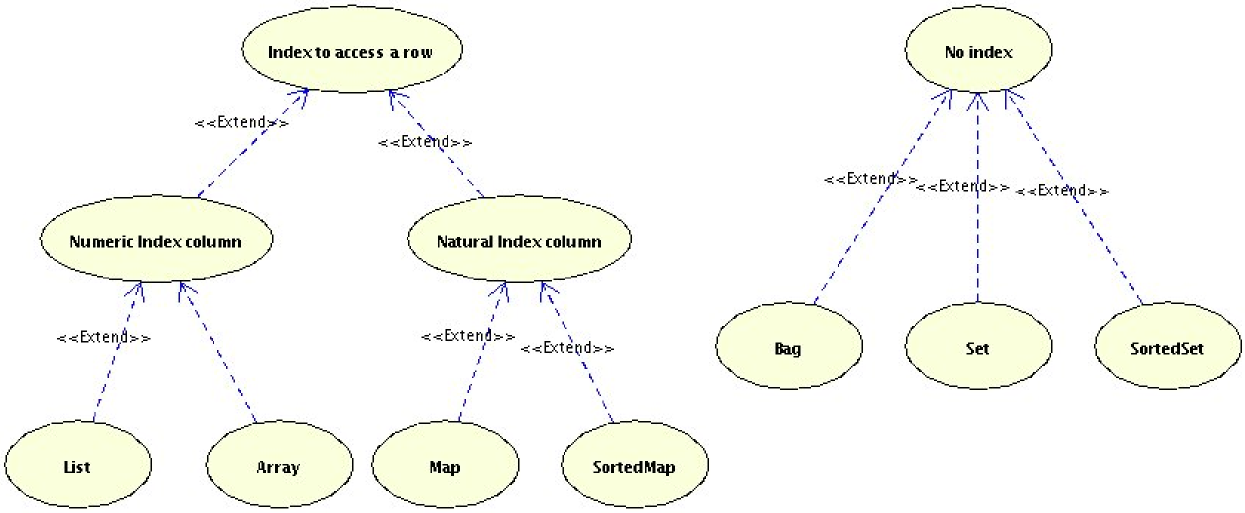
7.1. Selecting between List, Set, Map or array to hold many side

If you map a 1:n or a m:n relation you will have a class holding references to many of the other class. Here is the class diagram of our first example:



The field bees in Honey was of type *java.util.Set*. Hibernate supports other types as well. In general you can distinguish the following types:

Figure 7.2. Supported Collections and Maps



The main difference between these approaches is the presence or lack of an index column. A mapping with an index column is very fast to update. The numeric index is a column starting with a 0 for the first element, a 1 for the second and so on. If you delete an entry, all following index columns must be updated. This is of course slower. The following lines gives further tips for selection an option. Although it is more oriented to XML mapping, have a look at it.

Selecting XML mappings

Each option fit different requirements. The provided source code Developer Guide package *de.laliluna.relation.overview* holds a simple example of all types of mappings 1:n The following table gives an overview of XML mapping options.

Annotation mapping	XML mapping	Corresponding java type	Pros and cons
	set	java.util.Set	Fast, does not need an index
	map	java.util.Map	Fast to update.Needs an index column equivalent to map key, to access the entry.
	map	java.util.SortedMap	Same as map, In addition, sorting by a comporator is supported
	list	java.util.List	Fast to update. Needs an index column
	bag	java.util.List	Quite slow to update, because the whole bag has to be searched. A bag can have double entries
	array	Array of any mapped object	Needs an index column

Annotation mapping	XML mapping	Corresponding java type	Pros and cons
	primitive-array	Array of Integer, String,	Needs an index column

What do I use?

I use a Set in most cases. If I need a sort order, I make use of SortedSet. If I need some kind of natural order or I have to update single entries of a relation very often, I use List. The use of Map is rare. It is useful, if your data has some kind of natural map. In the sample below, we have languages, where the iso country code was used as key. The rest may have some rare use cases, but in my opinion, you will need them in very rare situation.

Selecting annotation mapping

Annotations do not provide the same structure. You are not forced to use *java.util.List* with the list mapping or a *java.util.Set* with a set. There is only have a annotation and you are allowed to use a List, a Set or a Collection. Of course, in case you want to use a map you need a type of *java.util.Map*. Source code for the following samples can be found in package *de.laliluna.relation.overview*.

Non indexed

```
Set hobbies = new HashSet();
```

Simplified and not perfectly correct: a set works like a map having a hash as a key and the object as value. A hash is a artificial number generated from the data. The number should be unique. Java accesses an entry in a HashSet by the generated hash. The *java.lang.Object* class implements a *hashCode* method.

Table 7.1. Sample hash

Hash	Value
7043360	a object of type Developer
8812347	an other developer
1234536	and a third one

Updates to an entry might be slower than an update to an indexed row, but if you delete a row there are no side effects (e.g. index update) to other elements.

Annotation mapping.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "developer_id")
private Set<Hobby> hobbies = new HashSet<Hobby>();
```

The *@JoinColumn* is actually not necessary. By default the foreign key column would be named *developer_id* anyway. If you use annotations, you can use a List as well.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "developer_id")
private List<Hobby> moreHobbies = new ArrayList<Hobby>();
```

XML mapping.

```
<set name="hobbies" cascade="all">
 <key column="developer_id" not-null="true"></key>
 <one-to-many class="Hobby" />
</set>
```

This mapping does not need an index column as map or list. The definition will create a table like

```
CREATE TABLE thobby
(
  id int4 NOT NULL,
  name varchar(255),
  developer_id int4,
  PRIMARY KEY (id),
... snipped away the foreign key constraints ...
)
```

Non indexed but sorted

```
   private SortedSet<Sport> sports = new TreeSet<Sport>(new SportComparator());
```

The characteristics of a sorted set are identical to a *Set*. In addition, a sortedSet can be sorted. This is not made by the database but in memory. Have a look at the TreeSet documentation of Java to find out more. One option to sort is to pass a comparator to the TreeSet. The comparator below sorts Sport items by name.

```
import java.util.Comparator;

public class SportComparator implements Comparator<Sport> {

   public int compare(Sport o1, Sport o2) {
      if (o1 = null || o1.getName()= null)
         return 1;
      if (o2 = null || o2.getName()= null)
         return -1;
      return o1.getName().compareTo(o2.getName());
   }
}
```

Another option is to implement the Comparable interface in the Sport class. This kind of sorting is called natural sort. In this case you have to apply the *SortType.NATURAL*.

Annotation mapping.

```
@OneToMany(cascade = CascadeType.ALL)
   @JoinColumn(name = "developer_id")
   @Sort(type = SortType.COMPARATOR, comparator = SportComparator.class)
   private SortedSet<Sport> sports = new TreeSet<Sport>(new SportComparator());
```

XML mapping.

```
<set name="sports" cascade="all" sort="de.laliluna.example1.SportComparator">
```

```
  <key column="developer_id" not-null="true"></key>
  <one-to-many class="Sport" />
</set>
```

It is probably faster to sort large sets by the database.

Table structure.

```
CREATE TABLE tsport
(
  id int4 NOT NULL,
  name varchar(255),
  developer_id int4 NOT NULL,
  PRIMARY KEY (id),
... snipped away the foreign key constraints ...
)
```

Non indexed with a bag mapping

```
private List ideas = new ArrayList();
```

A bag has no index. So access to an element always needs to traverse the complete bag until the element is found. This is only an issue when your bag can be large. Another disadvantage is that one element can be in a bag more than once. So you must be careful when adding entries to a bag. This kind of mapping is only supported by XML mappings

XML mapping.

```
    <bag name="ideas" cascade="all">
      <key column="developer_id" not-null="true"></key>
      <one-to-many class="Idea" />
    </bag>
```

Table structure.

```
CREATE TABLE tidea
(
  id int4 NOT NULL,
  name varchar(255),
  developer_id int4,
  PRIMARY KEY (id),
... snipped away the foreign key constraints ...
)
```

There is one situation when a bag can be faster than a set. Having a bi-directional one-to-many relation where *inverse="true"*, i.e. the relation is managed on the one-side as opposed to our example.

```
Developer d = session.get(Developer.class, 4711);
Idea idea = new Idea();
idea.setDeveloper(d);
d.getIdeas().add(idea);
```

In this case the *getIdeas* method does not need to initialize the ideas list. When there are many ideas this can be a great speed advantage. **Indexed using a java.util.Map**

```
   private Map developmentLanguages = new HashMap();
```

A map is a data structure where each value is accessed by a key. The value can be any kind of object, starting from primitives (String, Integer, …) to normal objects. When your data is similar to the following table you might consider using a map.

Short name	**Country**
de	Germany
us	United States
fr	France

A map is quite fast, as the key is used as an index to the data.

Annotation mapping.

```
   @CollectionOfElements
   @JoinTable(name = "development_languages", joinColumns =
      @JoinColumn(name = "developer_id"))
   @Column(name = "name", nullable = false)
   private Map<String, String> developmentLanguages = new HashMap<String, String>();
```

If we want to change the column of the key, we can overwrite the default column name *mapkey* in front of the class with the following annotation:

```
@AttributeOverrides( {
 @AttributeOverride(name = "developmentLanguages.key",
    column = @Column(name = "short_name"))
})
public class Developer implements Serializable {
```

XML mapping.

```
<map name="developmentLanguages" table="tdevelopmentlanguage" cascade="all">
 <key column="developer_id" not-null="true"></key>
 <map-key type="string" column="shortname"></map-key>
 <element column="name" type="string"></element>
</map>
```

Table structure.

```
CREATE TABLE tdevelopmentlanguage
(
  developer_id int4 NOT NULL,
  name varchar(255),
  shortname varchar(255) NOT NULL,
  PRIMARY KEY (developer_id, shortname),
... snipped away the foreign key constraints ...
)
```

Below you can see a map mapping to an object

```
   @OneToMany(cascade=CascadeType.ALL)
   @JoinColumn(name="developer_id")
   @MapKey(name="isocode")
   private Map<String, LovedCountry> lovedCountries =
      new HashMap<String, LovedCountry>();
```

```
CREATE TABLE lovedcountry
(
  isocode varchar(255) NOT NULL,
  name varchar(255),
  developer_id int4,
   PRIMARY KEY (isocode),
... snipped away the foreign key constraints ...
);
```

Indexed and sorted using java.util.SortedMap

```
private SortedMap lovedCountries = new TreeMap();
```

A sorted map shares the features of the *java.util.Map* and can be sorted like the *SortedSet*. The sort is not made by the database but in memory. Have a look at the TreeMap documentation of Java to find out more. It is probably faster to sort large maps by the database. This is currently not supported with annotations but you might mix in a XML mapping. indexterm:[<composite-element>}

XML mapping.

```
    <map name="lovedCountries" cascade="all" sort="natural">
      <key column="developer_id" not-null="true"></key>
      <map-key type="string">
        <column name="isocode"></column>
      </map-key>
      <composite-element class="LovedCountry">
        <property name="name" type="string"></property>
      </composite-element>
    </map>
```

Table structure.

```
CREATE TABLE tlovedcountries
(
  developer_id int4 NOT NULL,
  name varchar(255),
  isocode varchar(255) NOT NULL,
  PRIMARY KEY (developer_id, isocode),
... snipped away the foreign key constraints ...
)
```

Indexed with numeric index using java.util.List

```
private List computers = new ArrayList();
```

A list mapping always has an index column, if you use XML mapping. This allows a fast access to an element by the index. To remove an entry in a list is slower as compared to a Set. The reason is that all the following entries need to get an updated index value. Another advantage of an indexed List is that the entries will keep their sort order.

Annotation mapping.

```
@OneToMany(cascade = CascadeType.ALL)
   @JoinColumn(name = "developer_id")
   @IndexColumn(name = "listindex")
  private List<Computer> computers = new ArrayList<Computer>();
```

XML mapping.

```
    <list name="computers" cascade="all">
      <key column="developer_id" not-null="true"></key>
      <list-index column="listindex"></list-index>
      <one-to-many class="Computer" />
    </list>
```

Table structure.

```
CREATE TABLE tcomputer
(
  id int4 NOT NULL,
  name varchar(255),
  developer_id int4 NOT NULL,
  listindex int4,
  PRIMARY KEY (id),
... snipped away the foreign key constraints ...
)
```

Indexed with an idbag mapping

```
private List dreams = new ArrayList();
```

An idbag mapping is only suitable for many-to-many mappings. It has an index, it is defined by the tag collection-id. So an update is as fast as a set, list or map. Note: the primary key generator of type native is not supported at the moment. Once again this kind of mapping is not supported by annotations. Closest is probably a List mapping with an index column.

XML mapping.

```
<idbag name="dreams" cascade="all" table="developer_dream">
 <collection-id type="integer" column="id">
 <generator class="sequence">
  <param name="sequence">developer_dream_id_seq</param>
 </generator>
 </collection-id>
 <key column="developer_id" not-null="true"></key>
 <many-to-many column="dream_id" class="Dream"></many-to-many>
</idbag>
```

Table structure.

```
CREATE TABLE developer_dream
(
  developer_id int4 NOT NULL,
  dream_id int4 NOT NULL,
  id int4 NOT NULL,
  PRIMARY KEY (id),
... snipped away the foreign key constraints ...
)
```

Indexed - array of objects

```
private JuneBeetle juneBeetles[];
```

In my opinion, an array is only useful if you do not have to add or remove items from your array. An array always needs an index column.

Annotation mapping.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "developer_id")
@IndexColumn(name = "listindex")
private JuneBeetle juneBeetles[];
```

XML mapping.

```
<array name="juneBeetles" cascade="all">
  <key column="developer_id" not-null="true"></key>
  <list-index column="listindex"></list-index>
  <one-to-many class="JuneBeetle" />
</array>
```

Table structure.

```
CREATE TABLE tjunebeetle
(
  id int4 NOT NULL,
  name varchar(255),
  developer_id int4,
  listindex int4,
  CONSTRAINT tjunebeetle_pkey PRIMARY KEY (id),
  ... snipped away the foreign key constraints ...
)
```

Indexed - array of primitives

```
private Integer[] favouriteNumbers;
```

You will probably need this mapping only in rare cases, i.e. if you need access to primitives. An array always needs an index column. This kind of mapping is not EJB3 compliant and only possible with Hibernate extensions (CollectionOfElements).

Annotation mapping.

```
@CollectionOfElements
@IndexColumn(name = "listindex")
private int[] favouriteNumbers;
```

XML mapping.

```
<primitive-array name="favouriteNumbers" table="tfavouritenumber">
```

```
  <key column="developer_id"></key>
  <list-index column="listindex" base="0"></list-index>
  <element type="integer" column="number"></element>
</primitive-array>
```

Table structure.

```
CREATE TABLE tfavouritenumber
(
  developer_id int4 NOT NULL,
  number int4,
  listindex int4 NOT NULL,
  CONSTRAINT tfavouritenumber_pkey PRIMARY KEY (developer_id, listindex),
  ... snipped away the foreign key constraints ...
)
```

7.2. Uni- and Bi-directional relations

We have to distinguish between uni-directional and bi-directional relations.

Uni-directional

Uni-directional is a relation where one side does not know about the relation.

```
public class Computer {
   private Integer id;
   private String name;
... snip ....
public class Developer  {
   private Integer id;
   private String name;
   private List computers = new ArrayList();
.... snip ....
```

In this case you can only set or delete the relation on one side

```
developer.getComputers().add(computer);
```

or

```
developer.getComputers().remove(computer);
```

Foreign key constraints

When there is a foreign key constraint on the relation, you must specify nullable=false in the JoinColumn annotation or add not-null="true" in the key tag.

Annotation mapping.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "developer_id", nullable=false)
@IndexColumn(name = "listindex")
private List<Computer> computers = new ArrayList<Computer>();
```

XML mapping.

```
<set name="computers" table="tcomputer" >
  <key column="developer_id" not-null="true"></key>
  <one-to-many class="Computer"/>
</set>
```

Bi-directional

In a bi-directional relation both sides know about the other side.

```
public class Computer {
   private Integer id;
   private String name;
 private Developer developer;
... snip ....
public class Developer  {
   private Integer id;
   private String name;
   private List computers = new ArrayList();
.... snip ....
```

In this case you must always set the relation on both sides. If you do not do this your session will contain false data!

```
developer.getComputers().add(computer);
computer.setDeveloper(developer);
```

or

```
developer.getComputers().remove(computer);
computer.setDeveloper(null);
```

When a computer cannot exist without a developer, i.e. the foreign key has a not null constraint, then the following will delete the relation.

```
developer.getComputers().remove(computer);
session.delete(computer);
```

Foreign key constraints In a bi-directional relation Hibernate can cause exceptions of type foreign key violations. To prevent this you must use *inverse="true"* on the many side. Inverse=true defines that a side does not manage a relation. In our example, the department side below is not responsible for the relation and team will not be saved, if only the department is saved. You must save the team to have the relation set or use cascade on the department side.

class department.

```
    <set name="teams" table="tteam" inverse="true">
      <key column="department_id"></key>
      <one-to-many class="Team"/>
    </set>
... snip ....
```

class team.

```
    <many-to-one name="department" class="Department">
    <column name="department_id" not-null="true"></column>
    </many-to-one>
```

7.3. Cascading

Cascading means that if you insert, update or delete an object, related objects are inserted, updated or deleted as well. If you do not use cascade you would have to save both objects independently. If you initially create objects and you do not cascade then you must save each object explicitly.

```
Department d = new Department();
Team t1 = new Team();
Team t2 = new Team();
d.getTeams().add(t1);
d.getTeams().add(t2);
t1.setDepartment(d);
t2.setDepartment(d);
session.save(d);
session.save(t1);
session.save(t1);
```

If you configure cascade on the department side

Annotation.

```
@OneToMany(cascade = {CascadeType.ALL})
private Set<Team> teams;
```

XML.

```
<set name="teams" table="tteam" inverse="false" cascade="all">
  <key column="department_id"></key>
  <one-to-many class="Team"/>
</set>
```

then you only need to call

```
session.save(d);
```

and the rest will be automatically cascaded. You can combine options, as well.

JPA Standard Annotation.

```
@OneToMany(cascade = {CascadeType.MERGE, CascadeType.PERSIST})
```

Hibernate Annotation.

```
@OneToMany
@Cascade({CascadeType.MERGE, CascadeType.PERSIST})
```

XML.

```
<set name="teams" table="tteam" inverse="false"
  cascade="persist,lock,replicate,save-update,delete,delete-orphan,refresh">
```

The following tables explains the different cascade types available. You will see that they are linked to methods from the session like session.persist(), session.delete(), session.buildLockRequest(LockOptions.NONE).lock(), ….

Java Persistence and Hibernate provides both a way to configure cascading. If you use the Hibernate API (the session), then I recommend to use *@Cascade(…)*. If you use the Java Persistence API (EntityManager), then you should use the cascading options inside of the relation *@OneToMany(cascade = {…})*. The simple reason: the JPA misses options to cascade session API methods like save, update or replicate.

Table 7.2. JPA Annotation Cascade Types

Type	**Description**
ALL	Cascades all but not the deletion of orphan members.
PERSIST	session.persist()
MERGE	session.merge()
REMOVE	session.delete(), does not delete orphan members
REFRESH	session.refresh() rereads object from the datbase (useful after trigger execution)

Table 7.3. Hibernate Annotation and XML cascade types

Type	**Description**
none	Default style, do not cascade
all	Cascades all but not the deletion of orphan members.
all-delete-orphan	All + delete-orphan
persist	session.persist()
save-update	session.saveOrUpdate()
save()	update()
lock	session.buildLockRequest(LockOptions.NONE).lock()
delete	Session.delete()
does not delete orphan members	delete-orphan
Deprecated, Deletes orphan members, for example you delete the department and the teams must be deleted as well.	refresh
session.refresh() rereads object from the database (useful after trigger execution)	evict or detach
Session.evict() removes an object from the session cache	replicate

Orphan Removal

If you remove for example an invoice position from the collection of an invoice, it is called an orphan entit. If you configure *orphan removal* it will be deleted just by fact that it was removed from the collection.

Since Java Persistence 2.0 orphan removal, the Hibernate cascade type is deprecated. Here is the correct way to use it.

```
@OneToMany (cascade = CascadeType.PERSIST, orphanRemoval = true)
private List<InvoicePosition> positions;
```

Usage.

```
Invoice invoice = (Invoice) session.get(Invoice.class, 4711);
invoice.getPositions().remove(1);
```

Understanding the examples

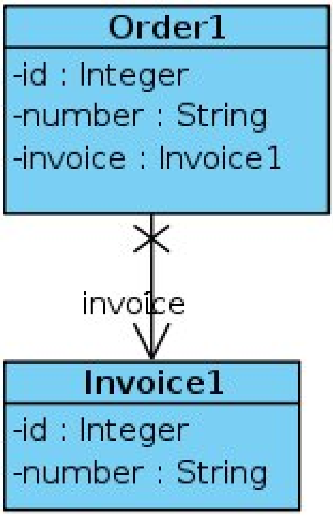
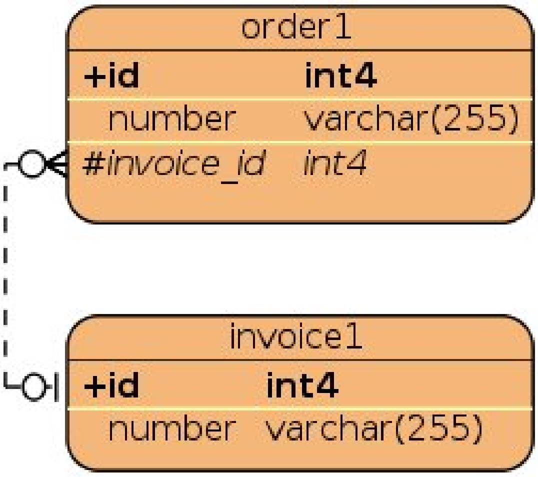
You can find the source code for annotation mapping in the *mapping-examples-annotation* project. The sourcecode of the XML mappings is in the *mapping-examples-xml* project. Each example references a package in the provided source code. We will explain the relevant part of the mapping, resulting tables and some more information in the book. You can find classes and a test class showing examples how to insert, update, delete and query the mapped objects in the source code.

7.4. 1:1 relation

Full source code is provided in package: *de.laliluna.relation.one2one* If a class has a relation to another class then you have the option to map this as a relation or as a component. A relation is more useful if both class are used for themselves. Having a relation between order and invoice, you will probably have business methods dealing only with the order or only with the invoice, so a relation is the better choice.

Uni-directional

Order1 has a relation to invoice. Invoice1 does not have any notion of the relation.

Classes	Tables
 Order1 -id : Integer -number : String -invoice : Invoice1 invoice Invoice1 -id : Integer -number : String	 order1 +id int4 number varchar(255) #invoice_id int4 invoice1 +id int4 number varchar(255)

Annotation mapping.

```
import javax.persistence.CascadeType;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
....... snip ..........
   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "invoice_id")
   private Invoice1 invoice;
```

@One2One specifies the relation. Cascade is explained in chapter xref:cascading. *@JoinColumn* defines the column in the order table having a foreign key relation to the invoice table. This annotation is optional. By default the column name targetClassName_id would be chosen.

XML mapping. The used tag is many-to-one in combination with unique set to true. This might be disturbing but it is the only way to define a uni-directional one-to-one relation. The one-to-one tag would not allow to configure the foreign key in the order table.

```
<hibernate-mapping package="de.laliluna.relation.one2one">
  <class name="Order1" table="torder">
....... snip .......
    <many-to-one name="invoice" class="Invoice1" cascade="all" unique="true">
      <column name="invoice_id"></column>
    </many-to-one>
  </class>
</hibernate-mapping>
```

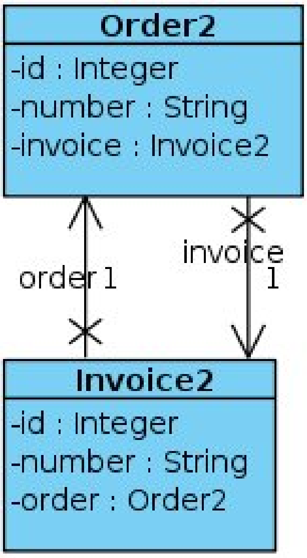
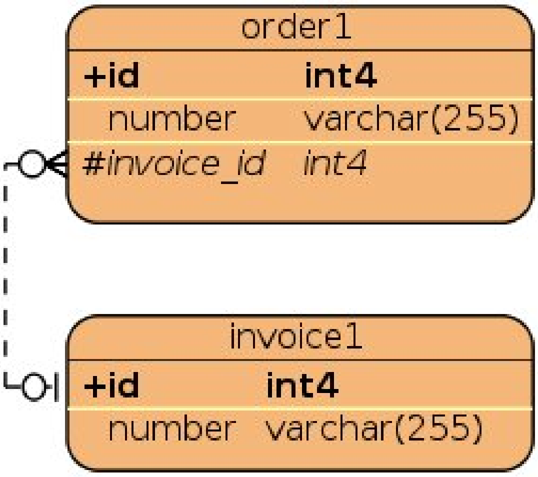
The Invoice mapping file does not contain any tags related to this relation. It is uni-directional. Resulting tables:

```
CREATE TABLE tinvoice
(
  id int4 NOT NULL,
  number varchar(255),
  CONSTRAINT tinvoice_pkey PRIMARY KEY (id)
) ;
CREATE TABLE torder
(
  id int4 NOT NULL,
  number varchar(255),
  invoice_id int4,
  PRIMARY KEY (id),
  FOREIGN KEY (invoice_fk)
      REFERENCES tinvoice (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
```

Below you can find some usage samples:

```
/* create entries and set relation */
Order1 order = new Order1(null, "123");
Invoice1 invoice = new Invoice1(null, "456");
order.setInvoice(invoice);
session.save(order); // cascade will save invoice as well

/* delete an invoice*/
// reattach order (update) to bind it to the current session
session.update(order1); // cascade also updates the invoice
session.delete(order1.getInvoice());
order1.setInvoice(null);

/* select all order and initialize the invoices with one join (very fast) */
List<Order1> list = session.createQuery("from Order1 o left join fetch o.invoice")
   .list();

/*select orders where invoice number starts with 2 */
List<Order1> list = session.createQuery(
   "from Order1 o where o.invoice.number like '2%' ").list();

/*select some invoices where order number starts with 1 */
List<Invoice1> invoices = session.createQuery(
   "select o.invoice from Order1 o where o.number like '1%' ").list();
```

Bi-directional

Classes	Tables
 Order2 -id : Integer -number : String -invoice : Invoice2 order 1 invoice 1 Invoice2 -id : Integer -number : String -order : Order2	 order1 +id int4 number varchar(255) #invoice_id int4 invoice1 +id int4 number varchar(255)

Annotation mapping.

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
....... snip ......

@Entity
public class Order2 implements Serializable {

@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "invoice_id")
private Invoice2 invoice;
```

@One2One specifies the relation. Cascade is explained in chapter xref:cascading. *@JoinColumn* defines the column in the order table having a foreign key relation to the invoice table. This annotation is optional. By default the column name targetClassName_id would be chosen.

```
import javax.persistence.Entity;
import javax.persistence.OneToOne;
....... snip ......

public class Invoice2 implements Serializable {
   @OneToOne(mappedBy="invoice")
   private Order2 order;
```

@OneToOne(mappedBy="invoice") defines that the relation is managed by the Order2 class and not by the Invoice2 class. Only if you assign the invoice in the Order2 class, Hibernate will reflect the relation in the database. Keep in mind that you have to set bi-directional relations on both sides, if you do not want to render your Hibernate session in an inconsistent state.

XML mapping.

```
<hibernate-mapping package="de.laliluna.relation.one2one">
  <class name="Order2" table="torder2">
.......snip ........
    <many-to-one name="invoice" class="Invoice2" cascade="all" unique="true">
      <column name="invoice_id"></column>
    </many-to-one>
  </class>
</hibernate-mapping>
```

```
<hibernate-mapping package="de.laliluna.example2" >
  <class name="Invoice2" table="tinvoice2">
......... snip ........
   <one-to-one name="order" property-ref="invoice"/>
  </class>
</hibernate-mapping>
```

The resulting tables are the same as mentioned above. I found that the following works as well. The Hibernate reference proposes the foreign key reference.

```
<one-to-one name="order" class="Order2"/>
```

Now, let’s have a look at some samples of use. It is important that you always set the relations on both sides. If not, you will render your Hibernate session into an inconsistent state.

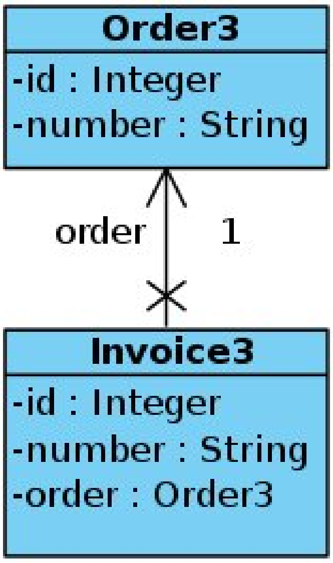
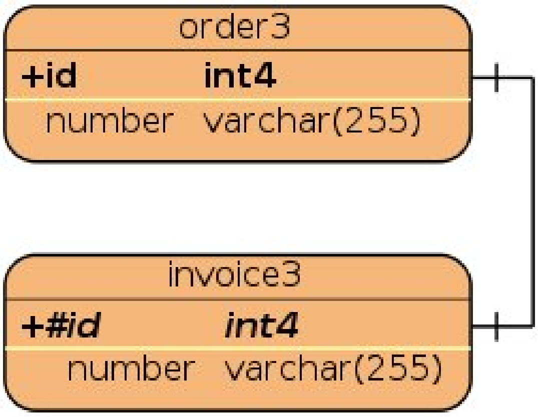
```
order.setInvoice(invoice);
invoice.setOrder(order);
session.save(order);
```

```
/* create and set relation */
Order2 order = new Order2(null, "123");
      Invoice2 invoice = new Invoice2(null, "456");
      // bi-directional set on both sides !!!
      order.setInvoice(invoice);
      invoice.setOrder(order);
      session.save(order); // cascade will save order as well

/* delete an invoice
* order is detached because the old session is closed, so reattach it using
* update */
      session.update(order); // cascade also updates the invoice
      session.delete(order.getInvoice());
      order.setInvoice(null);

/* find orders where invoice number starts with 2 */
```

```
List<Order2> list = session.createQuery(
            "from Order2 o where o.invoice.number like '2%' ").list();

/* find invoices where order number starts with 1 ");
      List<Invoice2> invoices = session.createQuery(
            "select i from Invoice2 i where i.order.number like '1%' ").list();
```

Relation to a primary key

The samples above use a foreign key relation. Sometimes both tables should share the same primary key. In this case we need a relation to the primary key. The primary key of the second table is set to the same value as the table of the first table. This is managed by Hibernated.

Classes	Tables
 Order3 -id : Integer -number : String order 1 Invoice3 -id : Integer -number : String -order : Order3	 order3 +id int4 number varchar(255) invoice3 +#id int4 number varchar(255)

Annotation mapping.

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;
....... snip ......

@Entity
public class Invoice3 implements Serializable {

   @Id
   @GeneratedValue(generator = "foreign_id")
   @GenericGenerator(name = "foreign_id", strategy = "foreign", parameters = {
      @Parameter(name = "property", value = "order") })
   private Integer id;
   @OneToOne(cascade = CascadeType.ALL,optional=false)
   @PrimaryKeyJoinColumn
   private Order3 order;
```

The Order3 class does not contain any relation specific annotation.

XML mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="de.laliluna.example2" >
  <class name="Invoice3" table="tinvoice3">
    <id name="id" >
      <generator class="foreign">
       <param name="property" >order</param>
      </generator>
    </id>

     <one-to-one name="order" class="Order3"  />
........ snip ........

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

You can add a constraint as well:

```
<one-to-one name="order" class="Order3" constrained="true" />
```

Table structure.

```
CREATE TABLE tinvoice3
(
  id int4 NOT NULL,
  number varchar(255),
  PRIMARY KEY (id),
  FOREIGN KEY (id)
      REFERENCES torder3 (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
CREATE TABLE torder3
(
  id int4 NOT NULL,
  number varchar(255),
  PRIMARY KEY (id)
) ;
```

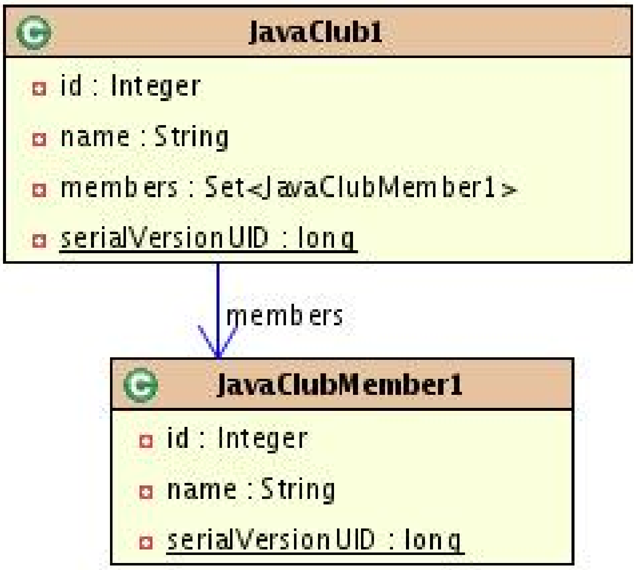
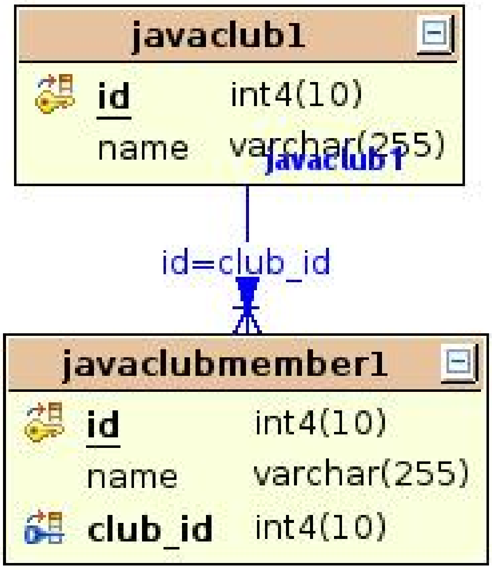
Samples of use:

```
/* create and set relation */
Order3 order = new Order3(null, "123");
Invoice3 invoice3 = new Invoice3(null, "456");
invoice3.setOrder(order);
session.save(order);
session.save(invoice3);

/* delete an invoice */
session.buildLockRequest(LockOptions.NONE).lock(invoice3); // reattach using lock
session.delete(invoice3);
```

7.5. 1:n

Full source code is provided in package: *de.laliluna.relation.one2many* **Uni-directional**

We have a class JavaClub1 having a set of JavaClubMember1. The member does not know about the relation.

Classes	Tables
	

As a consequence, the relation is managed on the one-side of the relation. This kind of relationship is not very efficient. An insert of a club with two members leads to 5 queries. If this relation is created or updated frequently, you should consider to create a bi-directional relation where the many-side manages the relation. This can be achieved with *inverse="true"* or the mappedBy attribute. Generated queries:

```
Hibernate: insert into tjavaclub (name, id) values (?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
Hibernate: update tjavaclubmember set club_id=? where id=?
Hibernate: update tjavaclubmember set club_id=? where id=?
```

Annotation mapping.

```
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.Entity;
........ snip ........
@Entity
public class JavaClub1 implements Serializable {
   @OneToMany
   @JoinColumn(name="club_id", nullable=false)
   private Set<JavaClubMember1> members = new HashSet<JavaClubMember1>();
```

The class *Club1* has no annotations related to the relation.

XML mapping.

```
public class JavaClub1 implements Serializable {
   private Set members = new HashSet();

   public Set getMembers() {
      return members;
   }

   public void setMembers(Set members) {
      this.members = members;
   }
........ snip .......
```

```
<hibernate-mapping package="de.laliluna.relation.one2many">
  <class name="JavaClub1" table="tjavaclub">
...... snip ..........
    <set name="members">
      <key column="club_id" not-null="true"></key>
      <one-to-many class="JavaClubMember1"/>
    </set>
  </class>
</hibernate-mapping>
```

Table structure.

```
CREATE TABLE tjavaclub
(
  id int4 NOT NULL,
  name varchar(255),
  CONSTRAINT tjavaclub_pkey PRIMARY KEY (id)
) ;
CREATE TABLE tjavaclubmember
(
  id int4 NOT NULL,
  name varchar(255),
  club_id int4 NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY (club_id)
      REFERENCES tjavaclub (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
```

Some examples of use:

```
/* create and set relation */
JavaClub1 club1 = new JavaClub1("Hib club");
JavaClubMember1 member1 = new JavaClubMember1("Eric");
JavaClubMember1 member2 = new JavaClubMember1("Peter");
// relation is uni-directional => we can only set the relation on one
// side
club1.getMembers().add(member1);
club1.getMembers().add(member2);
session.save(club1);
session.save(member1);
session.save(member2);

/* delete a member */
session.update(clubMember1);
JavaClub1 club1 = (JavaClub1) session.createQuery(
      "from JavaClub1 c where ? in elements(c.members) ").setEntity(
      0, clubMember1).uniqueResult();
// first take away the member from the club, than delete it.
club1.getMembers().remove(clubMember1);
session.delete(clubMember1);

/* simple select which initializes the club only, further queries are issued,
 * if you access the members*/
List list = session.createQuery("from JavaClub1").list();

/* select using fetch to initialize everything with one query and remove
 * double entries from the result */
list = session.createQuery(
  "from JavaClub1 c left join fetch c.members")
 setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();

/* same query using criteria instead of HQL */
list = session.createCriteria(JavaClub1.class)
  .setFetchMode("members", FetchMode.JOIN)
  .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();
```

Hibernate creates one instance for each line of a result set.\newline If a club has two members, you will receive two lines when you select the club and left join the members. Your list would have double entries of clubs. You had to use a *HashSet* in former times or you received double entries.

```
Set set = new HashSet(session.createCriteria(JavaClub1.class)
  .setFetchMode("members", FetchMode.JOIN).list());
```

Now there is a better approach:

```
List result = new DistinctRootEntityResultTransformer()
  .transformList(session.createQuery("from JavaClub1 c left join fetch c.members")
  .list());
```

If you use criteria queries, take the following approach.

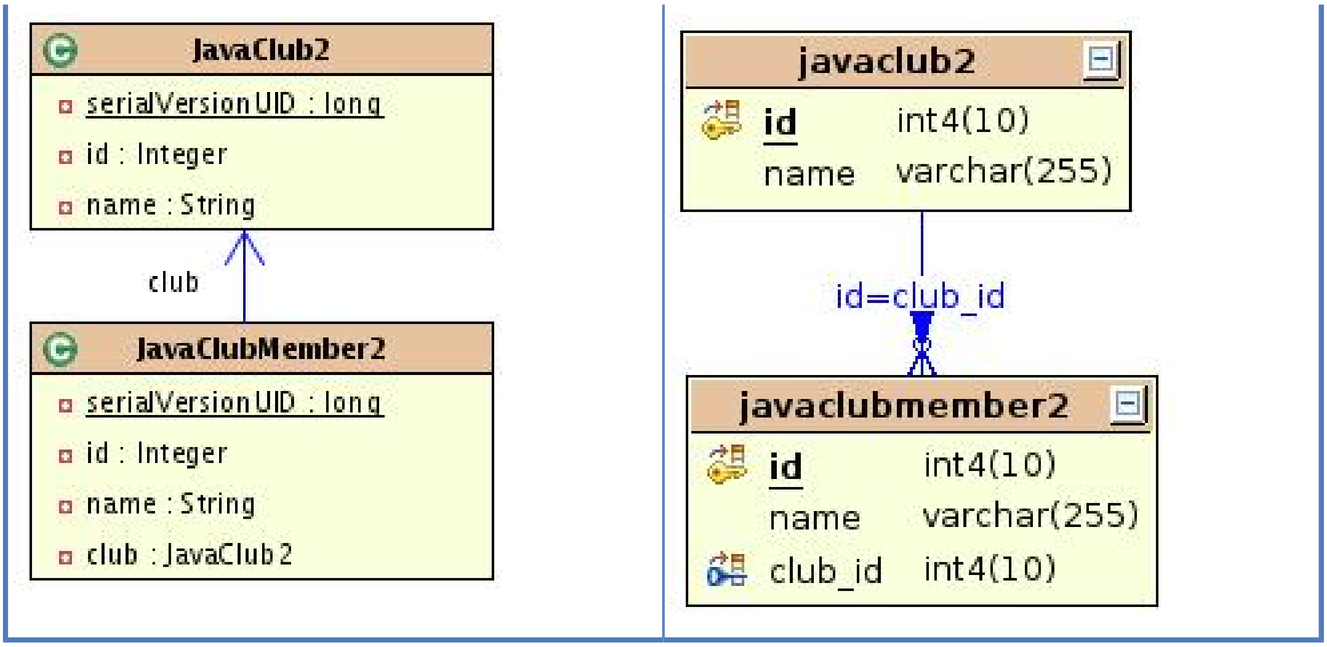
```
List list = results = session.createCriteria(JavaClub3.class)
  .addOrder(Order.desc("name"))
  .setFetchMode("members", FetchMode.JOIN)
  .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)).list();
```

Uni-directional (other side)

We have a class *JavaClub2* and *JavaClubMember2* where the club does not know about the relation. This kind of relation is more efficient than the one before. When you create a club with two members only three queries are issued.

```
Hibernate: insert into tjavaclub (name, id) values (?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
```

Classes	Tables



Annotation mapping.

```
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
..... snip ......
@Entity
public class JavaClubMember2  implements Serializable{

   @ManyToOne
   @JoinColumn(name="club_id")
   private JavaClub2 club;
```

The *@ManyToOne* annotation specifies the relation. *@JoinColumn(name="club_id")* specifies how the tables are joined. It is optional and you may rely on the default values.

XML mapping.

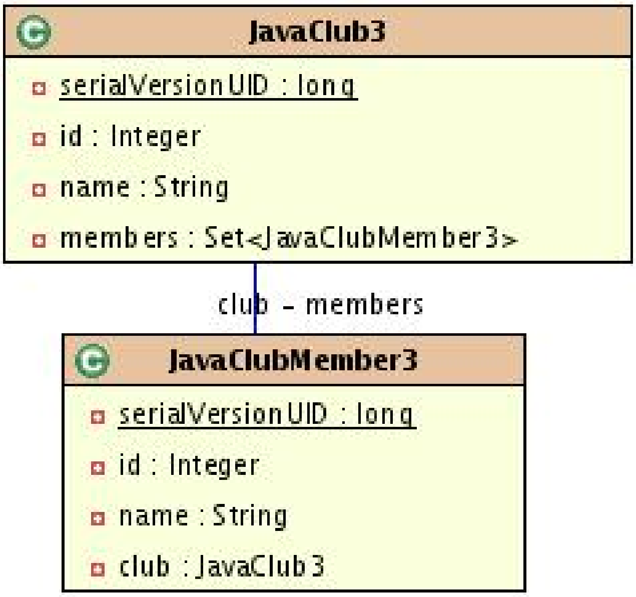
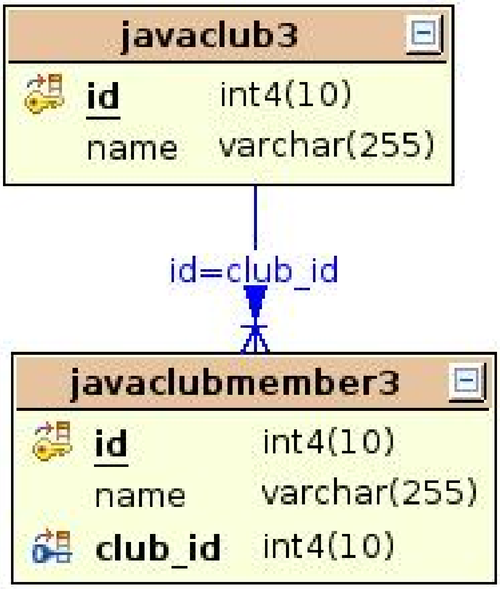
```
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

public class JavaClub1 implements Serializable {
   private Set members = new HashSet();
public Set getMembers() {
      return members;
   }
   public void setMembers(Set members) {
      this.members = members;
   }
...... snip .....
```

```
<hibernate-mapping package="de.laliluna.example3">
  <class name="JavaClubMember2" table="tjavaclubmember">
...... snip ........
    <many-to-one name="club" class="JavaClub2">
      <column name="club_id" not-null="true"></column>
    </many-to-one>
  </class>
</hibernate-mapping>
```

The resulting tables are of course the same as in our first example. Examples of use:

```
/* create and set relation */
JavaClub2 club2 = new JavaClub2("Hib club");
JavaClubMember2 member1 = new JavaClubMember2("Eric");
JavaClubMember2 member2 = new JavaClubMember2("Peter");
member1.setClub(club2);
member2.setClub(club2);
// we did not configure any cascadeType, so we have to save any of the objects
session.save(club2);
session.save(member1);
session.save(member2);

/* delete */
// just delete, we do not have to update or reconnect
session.delete(clubMember2);

/* select JavaClubMember but do not initialize the Club */
List list = session.createQuery("from JavaClubMember2").list();

/* select JavaClubMembers and initialize the club directly using a join */
List list = session.createQuery(
   "from JavaClubMember2 m left join fetch m.club").list();

/* same using criteria instead of HQL */
List list = session.createCriteria(JavaClubMember2.class).setFetchMode(
     "club", FetchMode.JOIN).list();
```

Bi-directional

Classes	Tables
	

In a 1 to n relation you should consider to manage the relation on the many-side (= JavaClubmember), as this leads to less queries, if you set a relation. Queries, if the relation is managed on the one-side:

```
Hibernate: insert into tjavaclub (name, id) values (?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
Hibernate: update tjavaclubmember set club_id=? where id=?
Hibernate: update tjavaclubmember set club_id=? where id=?
```

Queries, if the relation is managed on the many-side:

```
Hibernate: insert into tjavaclub (name, id) values (?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
Hibernate: insert into tjavaclubmember (name, club_id, id) values (?, ?, ?)
```

Annotation mapping.

```
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
..... snip .....
@Entity
public class JavaClubMember3  implements Serializable{

   @ManyToOne
   @JoinColumn(name = "club_id", nullable = false)
   private JavaClub3 club;
```

@ManyToOne specifies the relation. *@JoinColumn(name = "club_id", nullable = false)* specifies how the table is joined and that a member cannot exist without a club, *club_id* cannot be null.

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.OneToMany;
...... snip ........
@Entity
public class JavaClub3 implements Serializable {

   @OneToMany(mappedBy="club")
   private Set<JavaClubMember3> members = new HashSet<JavaClubMember3>();
```

@OneToMany(mappedBy="club") specifies that the relation is managed by the *club* property of the *JavaClubMember3*.

XML mapping. Although you might choose, which side manages the relation, you must manage the relation on the many-side, if your foreign key (club_id) cannot be null. In this case set, you have to use inverse="true". See the discussion in xref:RefUsingrelationsandcascading.

```
import java.util.HashSet;
import java.util.Set;
......... snip ......
public class JavaClub3 implements Serializable {
   private Set members = new HashSet();
   public Set getMembers() {
      return members;
   }

   public void setMembers(Set members) {
      this.members = members;
   }.
......... snip ......
```

```
public class JavaClubMember3 implements Serializable {
   private JavaClub3 club;

   public JavaClub3 getClub() {
      return club;
   }

   public void setClub(JavaClub3 club) {
      this.club = club;
   }
......... snip ......
```

```
<hibernate-mapping package="de.laliluna.example3">
  <class name="JavaClub3" table="tjavaclub">
..... snip ........
    <set name="members" inverse="true">
 <!-- we have a set and the relation is managed on the other side -->
      <key column="club_id" not-null="true"></key>
<!-- defines how the tables are joined. -->
      <one-to-many class="JavaClubMember3"/>
<!-- target class of the relation -->
    </set>
  </class>
</hibernate-mapping>
```

```
<hibernate-mapping package="de.laliluna.example3">
  <class name="JavaClubMember3" table="tjavaclubmember">
......... snip .........
    <many-to-one name="club" class="JavaClub3">
<!-- specifies property and target class -->
      <column name="club_id" not-null="true"></column> <!-- join column-->
    </many-to-one>
  </class>
</hibernate-mapping>
```

The resulting tables are once again the same. Do not forget to set and delete the relations on both sides.

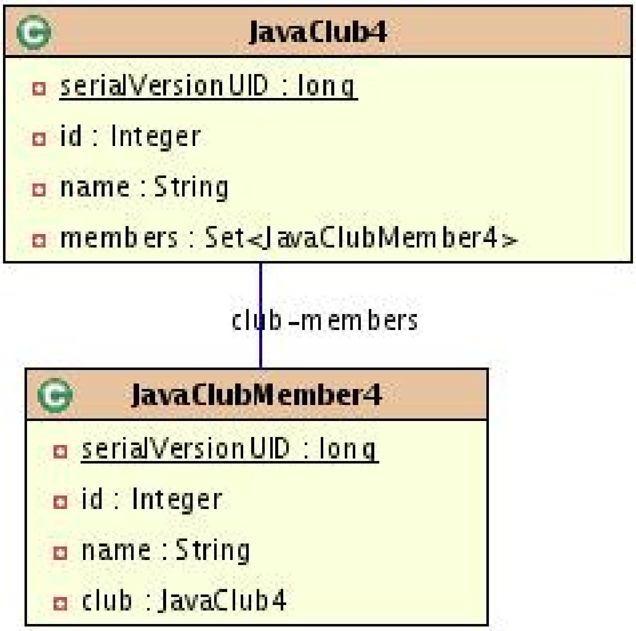
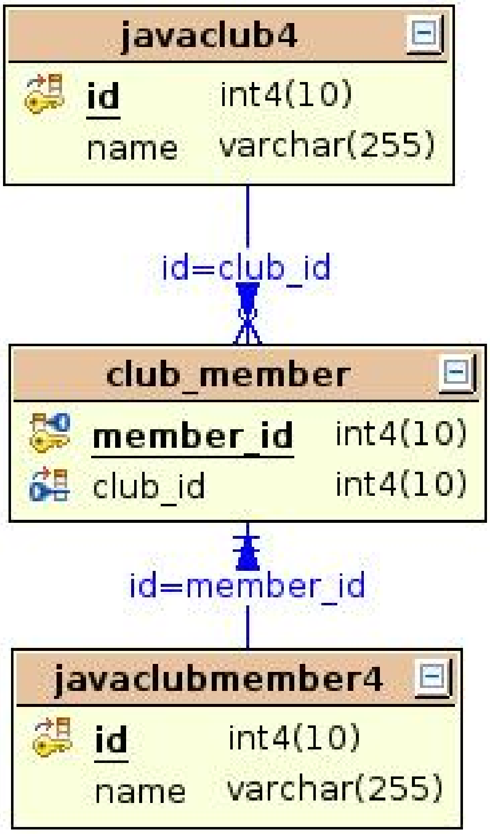
```
member1.setClub(club);
club.getMembers().add(member1);
member2.setClub(club);
club.getMembers().add(member2);
```

Examples of use:

```
/* create and set relation */
JavaClub3 club = new JavaClub3("Hib club");
JavaClubMember3 member1 = new JavaClubMember3("Eric");
JavaClubMember3 member2 = new JavaClubMember3("Peter");
// relation is bi-directional => we must set the relation on both sides
member1.setClub(club);
member2.setClub(club);
club.getMembers().add(member1);
club.getMembers().add(member2);
// no cascade configured so save everything
session.save(club);
session.save(member1);
session.save(member2);

/* delete */
// we must reattach the member to the new session
session.update(clubMember3);
clubMember3.getClub().getMembers().remove(clubMember3);
session.delete(clubMember3);

/* have a look in the uni-directional cases for query samples */
```

Bi-directional with join table

Sometimes you do not want to have a foreign key in the table of the many side but define the relation in a separate join table.

Classes	Tables
 JavaClub4 serialVersionUID : long id : Integer name : String members : Set<JavaClubMember4> club-members JavaClubMember4 serialVersionUID : long id : Integer name : String club : JavaClub4	 javaclub4 id int4(10) name varchar(255) id=club_id club_member member_id int4(10) club_id int4(10) id=member_id javaclubmember4 id int4(10) name varchar(255)

Annotation mapping.

```
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.OneToMany;
..... snip ......
@Entity
public class JavaClub4 implements Serializable {

   @OneToMany(mappedBy="club")
   private Set<JavaClubMember4> members = new HashSet<JavaClubMember4>();
```

@OneToMany(mappedBy="club") defines the relation and that it is managed by the property club of JavaClubMember.

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToOne;
... snip .......
@Entity
public class JavaClubMember4 implements Serializable {

   @ManyToOne
   @JoinTable(name = "club_member",
      joinColumns = { @JoinColumn(name = "member_id") },
      inverseJoinColumns = { @JoinColumn(name = "club_id") }
      )
   private JavaClub4 club;
```

@JoinTable(name = "club_member".., specifies the join table. joinColumns specifies which columns reference the JavaClubMember primary key. inverseJoinColumns specifies which columns reference the JavaClub primary key.

XML mapping. On the JavaClub4 side, we define a many-to-many relation and set the JavaClubMember4 to unique. This might be confusing but is the correct approach. Annotation mapping is somewhat clearer for this kind of mapping. I set inverse to true, to have more efficient updates.

```
public class JavaClubMember4 implements Serializable {
   private JavaClub4 club;

   public JavaClub4 getClub() {
      return club;
   }

   public void setClub(JavaClub4 club) {
      this.club = club;
   }
```

```
public class JavaClub4 implements Serializable {
   private Set<JavaClubMember4> members = new HashSet<JavaClubMember4>();

   public Set<JavaClubMember4> getMembers() {
      return members;
   }

   public void setMembers(Set<JavaClubMember4> members) {
      this.members = members;
   }
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
```

```
<hibernate-mapping package="de.laliluna.example3">
  <class name="JavaClub4" table="tjavaclub4">
........ snip .......
    <set name="members" inverse="true" table="club_member">
      <key column="club_id"></key>
      <many-to-many class="JavaClubMember4" column="member_id" unique="true"/>
    </set>
  </class>
</hibernate-mapping>
```

The JavaClubMember4 (many side of relation) defines the join.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="de.laliluna.example3">
   <class name="JavaClubMember4" table="tjavaclubmember4">
........ snip ...........
      <join table="club_member" >
        <key column="member_id" ></key>
         <many-to-one name="club" class="JavaClub4">
            <column name="club_id" not-null="true"></column>
         </many-to-one>
      </join>
   </class>
</hibernate-mapping>
```

The resulting tables are:

```
CREATE TABLE javaclub4
(
  id int4 NOT NULL,
  name varchar(255),
  PRIMARY KEY (id)
) ;
CREATE TABLE javaclubmember4
(
  id int4 NOT NULL,
  name varchar(255),
  PRIMARY KEY (id)
) ;
CREATE TABLE club_member
(
  member_id int4 NOT NULL,
  club_id int4 NOT NULL,
  PRIMARY KEY (member_id),
  FOREIGN KEY (member_id)
      REFERENCES javaclubmember4 (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  FOREIGN KEY (club_id)
      REFERENCES javaclub4 (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
);
```

Do not forget to set and delete the relations on both sides.

```
member1.setClub(club);
club.getMembers().add(member1);
member2.setClub(club);
```

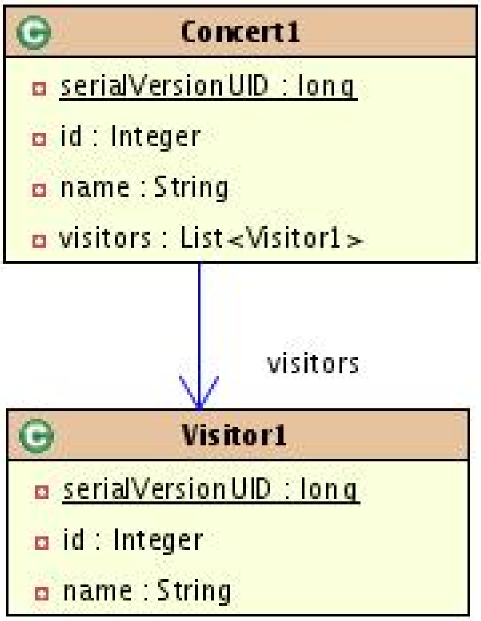
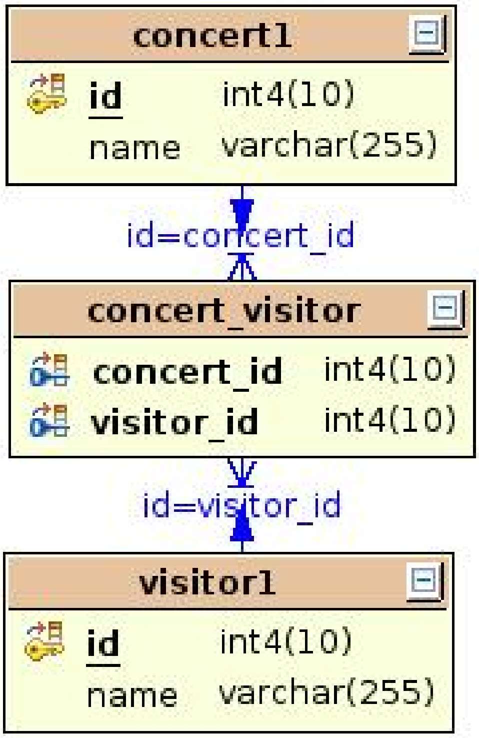
```
club.getMembers().add(member2);
```

Samples of use:

```
/* create and set relation */
session.beginTransaction();
JavaClub4 club = new JavaClub4("Hib club");
JavaClubMember4 member1 = new JavaClubMember4("Eric");
JavaClubMember4 member2 = new JavaClubMember4("Peter");
// relation is bi-directional => we must set the relation on both sides
member1.setClub(club);
member2.setClub(club);
club.getMembers().add(member1);
club.getMembers().add(member2);
// no cascade configured so save everything
session.save(club);
session.save(member1);
session.save(member2);

/* delete */
// we must reattach the member (our session is closed)
session.buildLockRequest(LockOptions.NONE).lock(clubMember4);
clubMember4.getClub().getMembers().remove(clubMember4);
session.delete(clubMember4);

/* query samples can be found in the previous samples */
```

7.6. m:n

Full source code is provided in package: *de.laliluna.relation.many2many* **Uni-directional**

Classes	Tables
	

Annotation mapping.

```
import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
......... snip .........
@Entity
public class Concert1 implements Serializable {

   @ManyToMany(cascade = CascadeType.ALL)
   @JoinTable(name = "concert_visitor",
         joinColumns = { @JoinColumn(name = "concert_id") },
         inverseJoinColumns = { @JoinColumn(name = "visitor_id") })
   private List<Visitor1> visitors = new ArrayList<Visitor1>();
```

@ManyToMany(cascade = CascadeType.ALL) specifies the relation. *@JoinTable(name = "concert_visitor"*, specifies the join table. *joinColumns* specifies which columns reference the Concert primary key. *inverseJoinColumns* specifies which columns reference the Visitor primary key. The *Visitor1* class has no relation related annotations.

XML Mapping. The relation is completely defined on the concert side. I used the tag

```
foreign-key
```

only to define the name of the foreign key as I reuse the tables for multiple mapping examples. Not naming the foreign keys would lead to random key names and double foreign key generation. Usually you do not need this property.

```
<hibernate-mapping package="de.laliluna.relation.many2many">
  <class name="Concert1" table="tconcert">
........ snip ........
    <list name="visitors" table="visitor_concert" >
      <key column="concert_id" foreign-key="visitor_concert_concert_id_fkey"/>
      <list-index column="list_index"></list-index>
      <many-to-many class="Visitor1"
         foreign-key="visitor_concert_visitor_id_fkey">
        <column name="visitor_id"  ></column>
      </many-to-many>
    </list>
  </class>
</hibernate-mapping>
```

The created tables are shown below:

```
CREATE TABLE tconcert
(
  id int4 NOT NULL,
  name varchar(255),
 PRIMARY KEY (id)
) ;
CREATE TABLE tvisitor
```

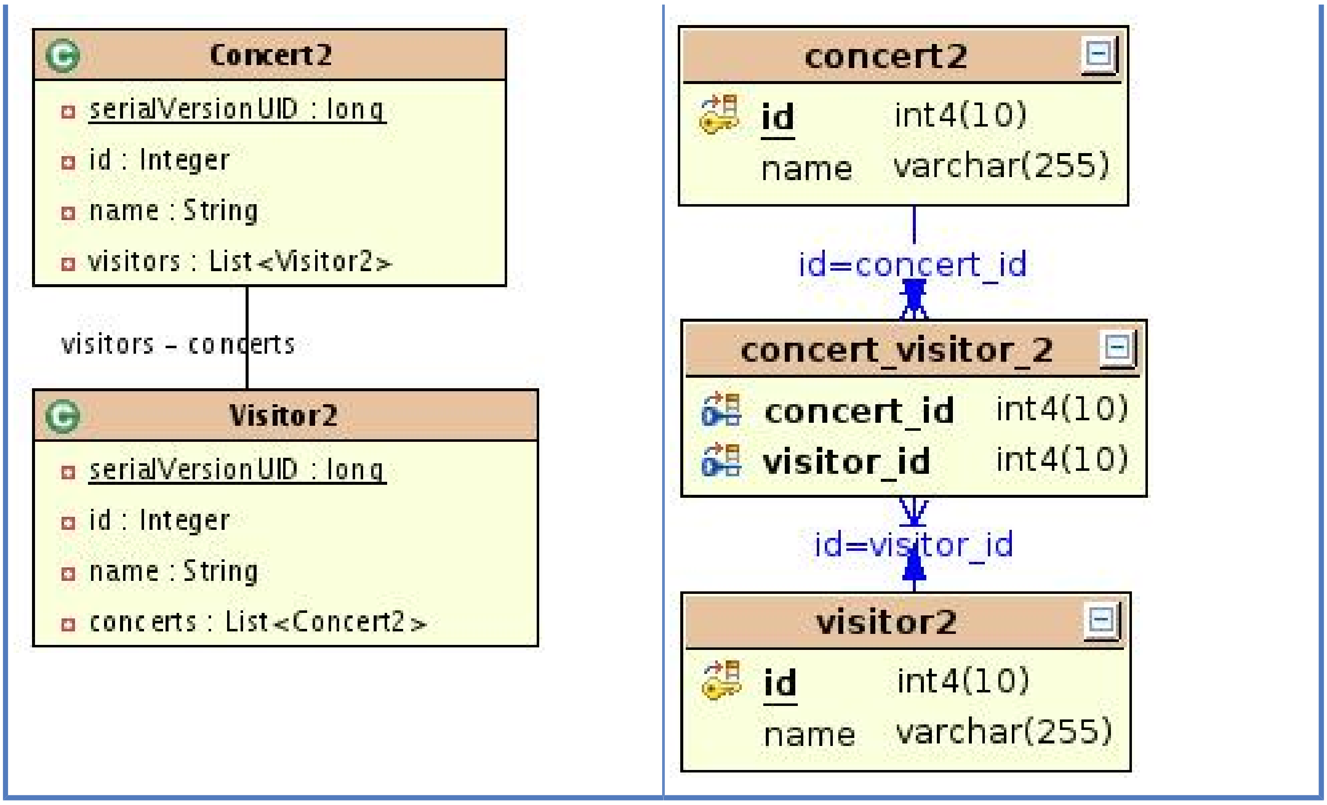
```
(
  id int4 NOT NULL,
  name varchar(255),
  PRIMARY KEY (id)
) ;
CREATE TABLE visitor_concert
(
  concert_id int4 NOT NULL,
  visitor_id int4 NOT NULL,
  list_index int4 NOT NULL,
  PRIMARY KEY (visitor_id, list_index),
  FOREIGN KEY (concert_id)
      REFERENCES tconcert (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  FOREIGN KEY (visitor_id)
      REFERENCES tvisitor (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
```

Samples of use:

```
/* create and set relation */
      Visitor1 visitor1 = new Visitor1(null, "Edgar");
      Concert1 concert1 = new Concert1(null, "Udo Juergens");
      concert1.getVisitors().add(visitor1);
      session.save(visitor1);
      session.save(concert1);

/* delete */
// reattach the visitor to our new session
      session.buildLockRequest(LockOptions.NONE).lock(visitor1);
      // remove visitor from all concerts
      List list = session.createQuery(
            "from Concert1 c where ? in elements(c.visitors)").setEntity(0,
            visitor1).list();
      for (Iterator iter = list.iterator(); iter.hasNext();)
      {
         Concert1 element = (Concert1) iter.next();
         element.getVisitors().remove(visitor1);
      }
      // delete visitor
      session.delete(visitor1);

/* select all concerts having a visitor named Carmen */
      List list = session
            .createQuery(
                "select distinct c from Concert1 c left join c.visitors v "
            +"where v.name like 'Carmen%' order by c.id").list();
```

Bi-directional

Classes	Tables



Annotation mapping.

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
........ snip .........
@Entity
public class Concert2 implements Serializable {

   @ManyToMany(cascade = CascadeType.ALL)
   @JoinTable(name = "concert_visitor_2",
      joinColumns = { @JoinColumn(name = "concert_id") },
      inverseJoinColumns = { @JoinColumn(name = "visitor_id") })
   private List<Visitor2> visitors = new ArrayList<Visitor2>();
```

@ManyToMany(cascade = CascadeType.ALL) specifies the relation. *@JoinTable(name = "concert_visitor"*, specifies the join table. *joinColumns* specifies which columns reference the Concert primary key. *inverseJoinColumns* specifies which columns reference the Visitor primary key.

```
import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.ManyToMany;
........... snip ..........

@Entity
public class Visitor2 implements Serializable {

   @ManyToMany(mappedBy="visitors", cascade=CascadeType.ALL)
   private List<Concert2> concerts = new ArrayList<Concert2>();
```

@ManyToMany(mappedBy="visitors", cascade=CascadeType.ALL) specifies that the relation is managed by the property visitors of the Concert class. If both sides manage a relation, they will both try to write the relation into the database and cause a primary key violation.

XML Mapping.

```
<hibernate-mapping package="de.laliluna.relation.many2many ">
  <class name="Concert2" table="tconcert">
......... snip ........
    <list name="visitors" table="visitor_concert" >
      <key column="concert_id" foreign-key="visitor_concert_concert_id_fkey"/>
      <list-index column="list_index"></list-index>
      <many-to-many class="Visitor2" column="visitor_id"
         foreign-key="visitor_concert_visitor_id_fkey"></many-to-many>
    </list>
  </class>
</hibernate-mapping>
```

```
<hibernate-mapping package="de.laliluna.relation.many2many ">
  <class name="Visitor2" table="tvisitor">
......... snip ........
    <list name="concerts" table="visitor_concert" inverse="true">
      <key column="visitor_id" foreign-key="visitor_concert_visitor_id_fkey"/>
      <list-index column="list_index"></list-index>
      <many-to-many class="Concert2" column="concert_id"
      foreign-key="visitor_concert_concert_id_fkey"></many-to-many>
    </list>
  </class>
</hibernate-mapping>
```

The created tables are the same as in our previous example. Examples of use can be found in the provided source code.

There are some important aspects you have to consider: column names If you do not set the foreign key column name of the visitor on the Concert side,

```
<many-to-many class="Visitor2" column="visitor_id"
```

then there will be a column created, even though the column is already defined on the visitor side with

```
 <list name="concerts" table="visitor_concert" inverse="true">
      <key column="visitor_id"
```

The column name used in this case is ELT. I don’t know why but I only warn you. Foreign-key Usually you will not need this tag. I needed it because I have two mappings (Concert1, Concert2) to the same table. When you map two classes to the same table you should name the foreign-keys or you will get the foreign keys created twice. *Inverse="true"* Inverse=true defines that this side will not manage the relation. You must set one side to inverse="true". If not, both sides try to write the relation into the database and cause a primary key violation. Correct lock usage We set *inverse="true"* on the visitor side or *mappedBy* on the visitor side without any cascades. As the visitor does not manage the relation, the deletion of a visitor will not remove the relation. You receive a foreign key exception. The reason is that if you reattach the visitor side, this does not reattach the concert side as well. The following code does not work as long as you do not uncomment the explicit lock for the concert.

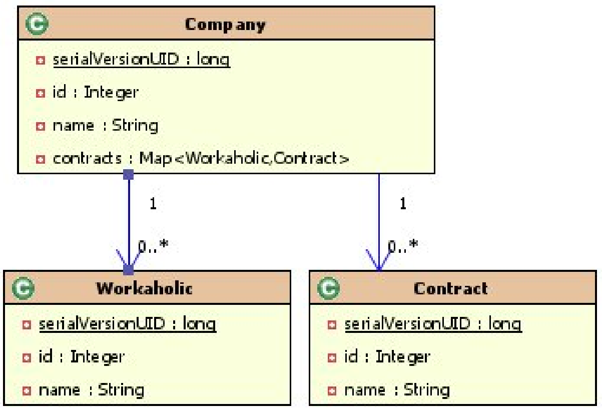
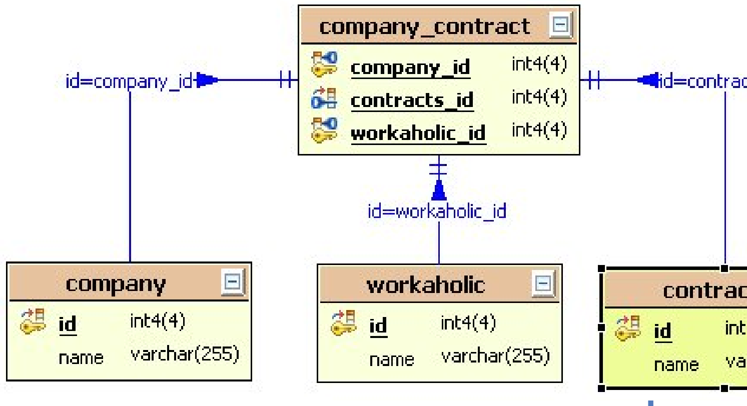
```
      session.buildLockRequest(LockOptions.NONE).lock(visitor);
      List list = visitor.getConcerts();
      Concert2 concert = (Concert2) list.iterator().next();
      //session.buildLockRequest(LockOptions.NONE).lock(concert);
      concert.getVisitors().remove(visitor);
      visitor.getConcerts().remove(concert);
```

An alternative approach is to define Cascading on the visitor side.

```
    <list name="concerts" table="visitor_concert" inverse="true" cascade="lock">
```

7.7. 1:n:1

Typical examples for this kind of relations are company → contract → employee, Order → Orderposition → Article. The Hibernate reference uses the term ternary association for this kind of relation. There are three approaches to map this relation.

Classes	Tables
	

Simple way

Use a 1:n relation and a second n:1 relation. You can use the examples we provided before. **Map-key-many-to-many**

A nice way is a special kind of mapping using a map. We will use company → employee → contract as an example. Employee will be used as a key to get the contract from a *java.util.Map*. Full source code is provided in the package: *de.laliluna.relation.ternary*

Annotation mapping.

```
import java.util.HashMap;
import java.util.Map;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import org.hibernate.annotations.MapKeyManyToMany;
.......... snip ..........
@Entity
public class Company implements Serializable{

   @OneToMany(cascade=CascadeType.ALL)
   @JoinTable(name="company_contract",
      joinColumns={@JoinColumn(name="company_id")})
   @MapKeyManyToMany(joinColumns={@JoinColumn(name="workaholic_id")})
   private Map<Workaholic,Contract> contracts =
      new HashMap<Workaholic,Contract>();
```

Neither Workaholic, nor Contract have any annotation related to the relation.
The @JoinTable is in fact obsolete, as it describes the default values.
@MapKeyManyToMany(joinColumns={@JoinColumn(name="workaholic_id")}) is in fact the magic bringing the ternary relation. It defines that the key of the map is referenced by the *workaholic_id* column. The key of hour map is workaholic.

XML mapping of Company.

```
<hibernate-mapping package="de.laliluna.relation.ternary">
  <class name="Company" table="tcompany" >
...... snip .....
    <map name="contracts">
    <key column="company_id"></key>
    <map-key-many-to-many column="workaholic_id" class="Workaholic"/>
    <one-to-many class="Contract"/>
    </map>
  </class>
</hibernate-mapping>
```

Neither Contract nor Workaholic have any relation specific tags in their mapping file. The created tables differ slightly, as the annotation requires a join table.

```
CREATE TABLE tcompany
(
  id int4 NOT NULL,
  name varchar(255),
  PRIMARY KEY (id)
) ;
CREATE TABLE tworkaholic
(
  id int4 NOT NULL,
  name varchar(255),
  PRIMARY KEY (id)
) ;
Annotation version:
CREATE TABLE annotation.contract
(
  id int4 NOT NULL,
  name varchar(255),
  PRIMARY KEY (id)
) ;
CREATE TABLE company_contract
(
  company_id int4 NOT NULL,
  contracts_id int4 NOT NULL,
  workaholic_id int4 NOT NULL,
  CONSTRAINT company_contract_pkey PRIMARY KEY (company_id, workaholic_id),
  CONSTRAINT fkc6d16ad43c5add7b FOREIGN KEY (contracts_id)
      REFERENCES contract (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fkc6d16ad48c60df4a FOREIGN KEY (workaholic_id)
      REFERENCES workaholic (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fkc6d16ad4da4faaaa FOREIGN KEY (company_id)
      REFERENCES company (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT company_contract_contracts_id_key UNIQUE (contracts_id)
)
XML version:
CREATE TABLE tcontract
(
  id int4 NOT NULL,
  name varchar(255),
  company_id int4,
  workaholic_id int4,
 PRIMARY KEY (id),
  FOREIGN KEY (workaholic_id)
      REFERENCES tworkaholic (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  FOREIGN KEY (company_id)
      REFERENCES tcompany (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
```

Usage samples:

```
/* create and set relation */
Workaholic workaholic1 = new Workaholic("Karl");
Workaholic workaholic2 = new Workaholic("Susi");

Company company = new Company("exploiter international");
Contract contract1 = new Contract("slave 123");
Contract contract2 = new Contract("no holiday");
session.save(workaholic1);
session.save(contract1);
session.save(workaholic2);
session.save(contract2);
company.getContracts().put(workaholic1, contract1);
company.getContracts().put(workaholic2, contract2);
session.save(company);

/* find company of a contract */
Company company = (Company) session.createQuery
   ("select c from Company c left join c.contracts cr where cr.id = ?“)
      .setInteger(0,id).uniqueResult();
```

However, there are however some aspects you have to consider. If you change a field of the Workaholic and try to access the contract directly after your change, you will not be lucky.

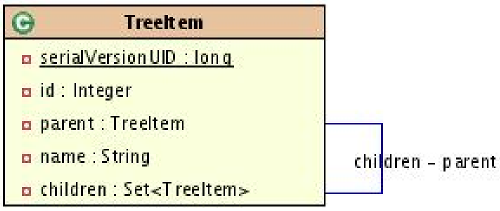
```
Workaholic.setName("Udo");
Contract c = (Contract) company.getContracts().get(workaholic);
```

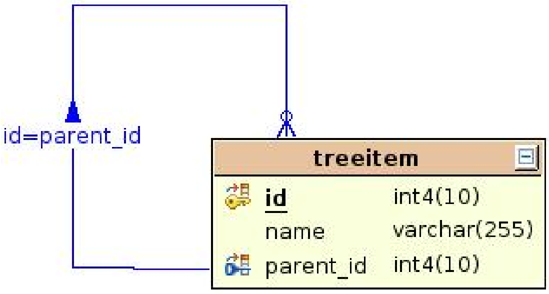
Keep in mind that you are working with a map. Changing a field affects the hashCode. So do not change the map key! **Component**

You can use a component mapping to achieve this. The example for this kind of mapping can be found in chapter xref:RefComposition13An3A1

7.8. Recursive relation

Recursive relations are possible. Typical example are Trees where each node can have a subtree. We will create a Tree where each TreeItem knows its parent and its children. Full source code is provided in the package: *de.laliluna.relation.recursive*

<table>
<tr><th>Classes</th><th>Tables</th></tr>
<tr><td>


</td><td>


</td></tr>
</table>

Annotation mapping.

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
....... snip ......

@Entity
public class TreeItem implements Serializable {

   @ManyToOne
   @JoinColumn(name="parent_id")
   private TreeItem parent;

   @OneToMany(mappedBy="parent", cascade = CascadeType.ALL)
   private Set<TreeItem> children = new HashSet<TreeItem>();
```

The relation is in fact a simple one-to-many relation. *@OneToMany(mappedBy="parent", cascade = CascadeType.ALL)* specifies that the relation is managed by the parent property. *@ManyToOne* specifies the relation *@JoinColumn(name="parent_id")* specifies which column is the foreign key column to the TreeItem primary key.

XML mapping.

```
<hibernate-mapping package="de.laliluna.example6">
  <class name="TreeItem" table="ttreeitem" >
........ snip ......
    <many-to-one name="parent"  class="TreeItem"  unique="true" not-null="false" >
     <column name="parent_id" ></column>
    </many-to-one>
    <set name="children" inverse="true" cascade="all,delete-orphan">
      <key column="parent_id"></key>
      <one-to-many class="TreeItem" />
    </set>
  </class>
</hibernate-mapping>
```

What is interesting in the mapping is that I have set a cascade.

```
cascade="all,delete-orphan"
```

This is convenient in a twofold sense: First, when you create a tree, you only have to save the top item. All other items will be saved automatically. Second, when you delete an item the whole subtree will be deleted. The created table is:

```
CREATE TABLE ttreeitem
(
  id int4 NOT NULL,
  name varchar(255),
  parent_id int4,
  PRIMARY KEY (id),
  FOREIGN KEY (parent_id)
      REFERENCES ttreeitem (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
```

Samples of use:

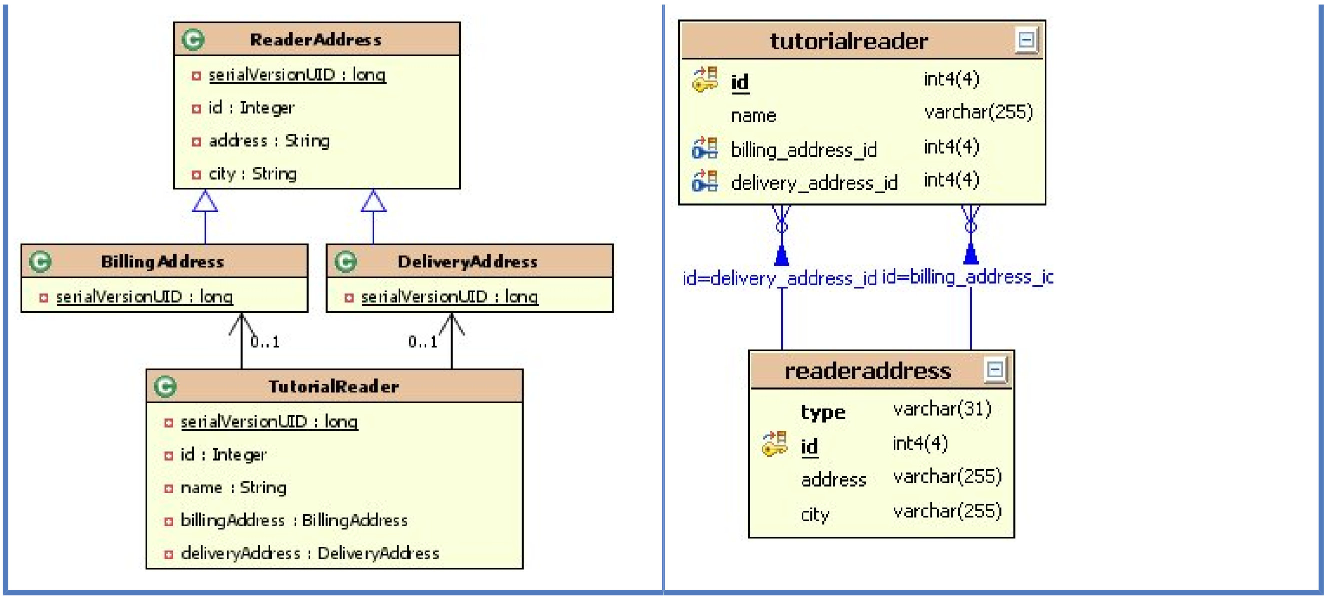
```
/* create and set relationship */
TreeItem main = new TreeItem(null, "main");
      TreeItem sub1 = new TreeItem(null, "go swimming");
      sub1.setParent(main);
      main.getChildren().add(sub1);
      TreeItem sub11 = new TreeItem(null, "lake");
      sub11.setParent(sub1);
      sub1.getChildren().add(sub11);
      session.save(main);
      // cascade will save all the children

/* delete a sub tree */
// reattach subTree to the new session using lock
      session.buildLockRequest(LockOptions.NONE).lock(sub1);
      /* remove the children from the parent or it will be resaved
      when the parent is saved.
     */
      sub1.getParent().getChildren().remove(sub1);
      session.delete(sub1);
```

Warning: You can create recursive relations with bag or set notation. If you want to use other like an indexed list, you must create some “dirty" work arounds. Have a look in the Hibernate reference for more information.

7.9. Typed relation (XML only)

A tutorial reader has two addresses, a billing and a delivery address. Only these address types are allowed. Full source code is provided in the package: *de.laliluna.relation.typed*

Classes	Tables



```
  <class name="TutorialReader" table="ttutorialreader">
......... snip .........
    <many-to-one name="billingAddress" entity-name="BillingReaderAddress" cascade="all"
      <column name="billingaddress_fk"></column>
      <formula>'billing'</formula>
    </many-to-one>
    <many-to-one  name="deliveryAddress" entity-name="DeliveryReaderAddress"  cascade="a
      <column name="deliveryaddress_fk"></column>
      <formula>'delivery'</formula>
    </many-to-one>
  </class>

  <class entity-name="BillingReaderAddress" name="ReaderAddress"
   table="treaderaddress" where="type='billing'"
    check="type in('billing','delivery')">
    <composite-id name="id" class="ReaderAddressId">
      <key-property name="readerId" column="reader_id"/>
      <key-property name="type"/>
    </composite-id>
    <property name="address" type="string"></property>
    <property name="city" type="string"></property>
  </class>
  <class entity-name="DeliveryReaderAddress" name="ReaderAddress"
   table="treaderaddress" where="type='delivery'"
    check="type in('billing','delivery')">
    <composite-id name="id" class="ReaderAddressId">
      <key-property name="readerId" column="reader_id"/>
      <key-property name="type"/>
    </composite-id>
    <property name="address" type="string"></property>
    <property name="city" type="string"></property>
  </class>
```

The following tables are generated:

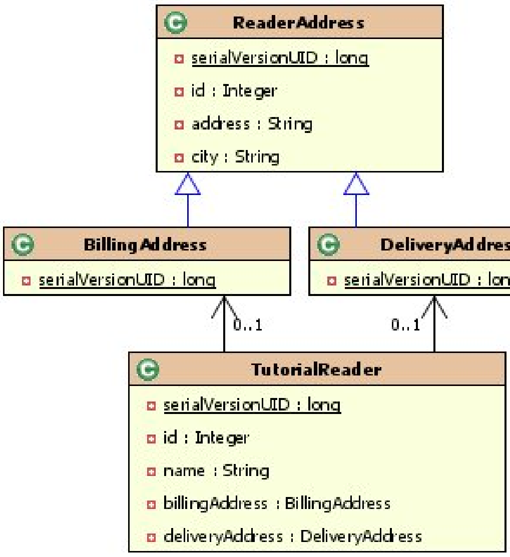
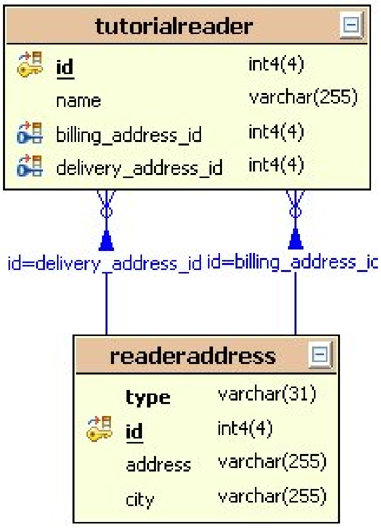
```
CREATE TABLE ttutorialreader
(
  id int4 NOT NULL,
  name varchar(255),
  billingaddress_fk int4,
  deliveryaddress_fk int4,
  CONSTRAINT ttutorialreader_pkey PRIMARY KEY (id)
) ;
CREATE TABLE treaderaddress
(
  reader_id int4 NOT NULL,
  "type" varchar(255) NOT NULL,
  address varchar(255),
  city varchar(255),
  CONSTRAINT treaderaddress_pkey PRIMARY KEY (reader_id, "type"),
  CONSTRAINT treaderaddress_type_check CHECK ("type"::text = 'billing'::text OR "type"::t
  CONSTRAINT treaderaddress_type_check1 CHECK ("type"::text = 'billing'::text OR "type":
)
```

Samples of use:

```
/* create and set relation */
TutorialReader reader = new TutorialReader();
reader.setName("Sebastian");
ReaderAddress billing = new ReaderAddress(new ReaderAddressId(reader
      .getId(), ReaderAddressId.BILLING), "Alte Landstrasse",
      "Frankfurt");
ReaderAddress delivery = new ReaderAddress(new ReaderAddressId(reader
      .getId(), ReaderAddressId.DELIVERY), "Neue Landstrasse",
      "Frankfurt");
reader.setBillingAddress(billing);
reader.setDeliveryAddress(delivery);
session.save(reader);

/* select all billingReaderAddresses */
      List list = session.createQuery("from BillingReaderAddress").list();

/* select tutorial reader with billing address in Bad Vilbel */
List list = session.createQuery("from TutorialReader r where r.billingAddress.city='Bad V
```

7.10. Typed relation (annotation workaround)

In chapter Section 7.9, “Typed relation (XML only)” we explained a typed relation. This approach is not possible with annotation mapping. There are two alternatives: First, TutorialReader has two fields billing_address_id, delivery_address_id. If you add a billing address to the reader, you set the type in ReaderAddress manually to “billing". Second, you map BillingAddress, DeliveryAddress and ReaderAddress as a inheritance structure. This approach is shown below: A tutorial reader has two addresses, a billing and a delivery address. Only these address types are allowed. Full source code is provided in the package: *de.laliluna.relation.typed*

Classes	Tables
	

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
...... snip ..........
@Entity
public class TutorialReader implements Serializable{

   @OneToOne(cascade=CascadeType.ALL)
   @JoinColumn(name="billing_address_id")
   private BillingAddress billingAddress;

   @OneToOne(cascade=CascadeType.ALL)
   @JoinColumn(name="delivery_address_id")
   private DeliveryAddress deliveryAddress;
```

The ReaderAddress includes all common properties:

```
import java.io.Serializable;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.SequenceGenerator;

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
public class ReaderAddress implements Serializable {

   @Id
   @SequenceGenerator(name = "readeraddress_seq",
      sequenceName = "readeraddress_id_seq")
   @GeneratedValue(strategy=GenerationType.SEQUENCE,
      generator="readeraddress_seq")
   private Integer id;

   private String address;

   private String city;
```

@Inheritance(strategy=InheritanceType.SINGLE_TABLE) defines the inheritance strategy. All addresses will be kept in one table. The different addresses can be identified by a discriminator column. A discriminator column holds the type of the address. *@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)*

Deliveryaddress and BillingAddress are fairly short.

```
@Entity
public class BillingAddress extends ReaderAddress {
   public BillingAddress() {
      super();
   }
   public BillingAddress(Integer id, String address, String city) {
      super(id,address,city);
   }
   private static final long serialVersionUID = 3313063223421102585L;
}
```

```
import javax.persistence.Entity;

@Entity
public class DeliveryAddress extends ReaderAddress {
   private static final long serialVersionUID = 8902940839248062796L;
   public DeliveryAddress(){
      super();
   }
   public DeliveryAddress(Integer id, String address, String city) {
      super(id,address,city);
   }
}
```

The following tables are generated:

```
CREATE TABLE ttutorialreader
(
  id int4 NOT NULL,
  name varchar(255),
  billingaddress_fk int4,
  deliveryaddress_fk int4,
  PRIMARY KEY (id),
  FOREIGN KEY (billing_address_id)
      REFERENCES annotation.readeraddress (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  FOREIGN KEY (delivery_address_id)
      REFERENCES annotation.readeraddress (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
) ;
CREATE TABLE readeraddress
(
  "type" varchar(31) NOT NULL,
  id int4 NOT NULL,
  address varchar(255),
  city varchar(255),
  PRIMARY KEY (id)
)
```

Samples of use:

```
/* create and set relation */
TutorialReader reader = new TutorialReader();
reader.setName("Sebastian");
BillingAddress billing = new BillingAddress(null, "Alte Landstrasse",
      "Frankfurt");
DeliveryAddress delivery = new DeliveryAddress(null, "Neue Landstrasse",
      "Frankfurt");
reader.setBillingAddress(billing);
reader.setDeliveryAddress(delivery);
session.save(reader);

/* select all billing addresses */
   List list = session.createQuery("from BillingAddress").list();

/* select tutorial reader with billing address in Frankfurt */
List    list = session.createQuery("from TutorialReader r where
r.billingAddress.address='Alte Landstrasse'").list();
```

Chapter 8. Components = Composition mapping

Components can be used to implement the object-oriented concept of composition.

This kind of relation can of course also be designed using a relation to an entity. So the question is, how to choose between entity relations versus and composition?

Let’s have a look at a first example and then we will work out the criteria to choose the correct approach.

A person has an address component.

Person class.

```
import javax.persistence.Entity;
@Entity
public class Person{

   private Address address;
```

Address class.

```
import javax.persistence.Embeddable;

@Embeddable
public class Address {
    private String street;
```

Collections of components are supported as well. A person might have a collection of former addresses.

Person class.

```
import javax.persistence.Entity;
@Entity
public class Person{
   private Address address;

    @ElementCollection
    @CollectionTable(name = "person_former_addresses",
      joinColumns = @JoinColumn(name = "person_fk"))
   private Set<Address> formerAddresses = new HashSet<Address>();
```

8.1. Composition versus entity relations

There are two characteristics of components:

- Dependent lifecycle
- Shared references are not supported

If an object has a **dependent lifecycle**, it will be deleted when the parent object is deleted.

Some examples:

- Shop order and order position
- Person and address
- Product and product details

Some examples for independent lifecycles:

- Customer and key accounter
- Order and article
- Football team and player.

The last case could be discussed. If the team needs to be shut down, do you really want to keep the player in the league?

A **shared reference** is a case of two entities having a relation to the same entity instance. For example two shop products might have a reference to the same shop category.

Recommendation

If a class does not need to support shared references and has a dependent lifecycle, you should map it as component, as you get the saving, updating and deleting for free.

Keep in mind

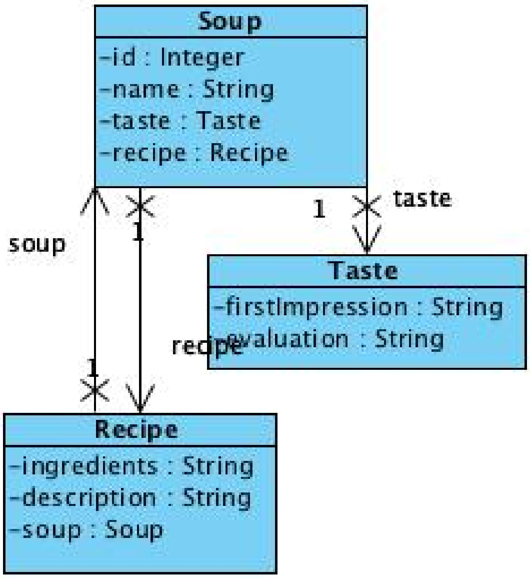
You can always live and develop without using components. It is a just cleaner mapping.

Finally, using *Cascade.ALL* and *orphan removal* you can achieve component like behaviour with entities.

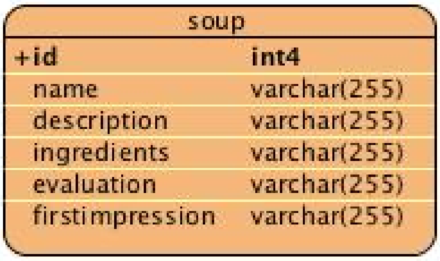
The *Hibernate Reference* uses the term *entity relation* but I believe that the term *entity agregation* is more precise. When speaking about composition the *Hibernate Reference* uses *component collections*. Alternatively you may speak about composition .

8.2. Composed class in one table

Full source code is provided in the package: *de.laliluna.component.simple*

<table>
<tr><th>Classes</th><th>Tables</th></tr>
<tr><td>

Soup
-id : Integer
-name : String
-taste : Taste
-recipe : Recipe
soup
1
1
taste
Taste
-firstImpression : String
-evaluation : String
recipe
1
Recipe
-ingredients : String
-description : String
-soup : Soup

</td><td>soup
+id int4
name varchar(255)
description varchar(255)
ingredients varchar(255)
evaluation varchar(255)
firstimpression varchar(255)</td></tr>
</table>

In this example a soup has two components: recipe and taste. All information is kept in one table. We declare Recipe and Taste as simple properties. In their mapping we have to define that they are embeddable.

Annotation mapping.

```
import java.io.Serializable;
import javax.persistence.Entity;
....... snip .......
@Entity
public class Soup  implements Serializable{

   private Taste taste;

   private Recipe recipe;
```

As you can see there are no @Embedded annotations. We only need them, if we want to overwrite the default values.

```
import java.io.Serializable;
import javax.persistence.Embeddable;
import org.hibernate.annotations.Parent;

@Embeddable
public class Recipe implements Serializable {
    private String ingredients;

    private String description;

    @Parent
    private Soup soup;
```

@Embeddable specifies that this class can be embedded into other entities. @Parent defines that this is a reference back to the embedding class. This is the soup in our case.

```
import java.io.Serializable;
import javax.persistence.Embeddable;

@Embeddable
public class Taste implements Serializable {
   private String firstImpression;

   private String evaluation;
```

XML mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="de.laliluna.component.simple">
  <class name="Soup" table="tsoup" >
........... snip ..............
    <component name="taste" class="Taste">
      <property name="firstImpression" column="first_impression"></property>
      <property name="evaluation" column="evaluation"></property>
    </component>

    <component name="recipe" class="Recipe" unique="true">
      <parent name="soup"/>
      <property name="ingredients" column="ingredients"></property>
      <property name="description" column="description"></property>
    </component>
  </class>
</hibernate-mapping>
```

The *unique="true"* causes a unique key for the database table. Ingredients and description are of course unique. If you need to access the soup from the recipe, than you can add a parent tag.

```
<parent name="soup"/>
```

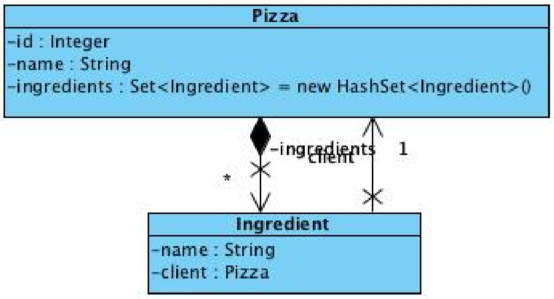
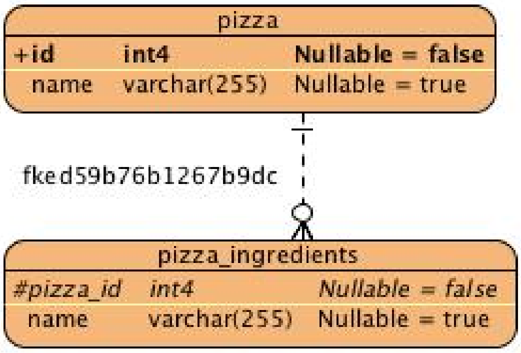
In this case your recipe class must provide a soup attribute.

Samples of use:

```
/* create a soup */
Soup soup = new Soup("Vegetable soup");
Taste taste = new Taste();
taste.setEvaluation("best eaten so far");
taste.setFirstImpression("incredible");
soup.setTaste(taste);
Recipe recipe = new Recipe();
recipe.setDescription("wash and cut vegetables\nadd water\ncook");
recipe.setIngredients("choice of vegetables you like");
soup.setRecipe(recipe);

session.save(soup);

/* select soups where attribute evaluation of component taste
is „best eaten so far“ */
List<Soup> list = session.createQuery("from Soup s where s.taste.evaluation = ?")
   .setString(0,"best eaten so far")
   .list();
```

8.3. Composition as set of many classes

Full source code is provided in the package: *de.laliluna.component.collection2*

Classes	Tables
 Pizza -id : Integer -name : String -ingredients : Set<Ingredient> = new HashSet<Ingredient>() -ingredients client 1 * Ingredient -name : String -client : Pizza	 pizza +id int4 Nullable = false name varchar(255) Nullable = true fked59b76b1267b9dc pizza_ingredients #pizza_id int4 Nullable = false name varchar(255) Nullable = true

Annotation mapping.

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.*;
....... snip ........
@Entity
public class Pizza {
// ... snip ...

   @ElementCollection
   @CollectionTable(name="pizza_ingredients", joinColumns =
      @JoinColumn(name="pizza_id"))
   private Set<Ingredient> ingredients = new HashSet<Ingredient>();
```

@ElementCollection defines the component mapping. *@JoinTable* is optional. It specifies the name of the ingredient table and the foreign key column.

Deprecated Hibernate extension

Before Java Persistence 2 only Hibernate supported this kind of mapping. The annotation was named slightly different *@CollectionOfElements* and is now deprecated.

```
import org.hibernate.annotations.Parent;
import javax.persistence.Embeddable;

@Embeddable
public class Ingredient {

   private String name;

   @Parent
   private Pizza client;
```

@Parent specifies the property to be a reference back to the embedding class, i.e. PizzaClient in our case.

XML mapping.

```
<hibernate-mapping package="de.laliluna.component.collection2">
<class name="Pizza" table="tclient" >
  <!-- .. snip .. ->
  <set name="ingredients" table="pizza_ingredients">
    <key column="client_fk"></key>
    <composite-element class="de.laliluna.component.collection2.Ingredient"  >
      <property name="name"/>
    </composite-element>
  </set>
</class>
</hibernate-mapping>
```

Samples of use:

```
/* create and set component */
Pizza pizza = new Pizza("Speciale");
Ingredient cheese = new Ingredient("Cheese");
Ingredient salami = new Ingredient("Salami");
Ingredient tomatoes = new Ingredient("Tomatoes");
pizza.getIngredients().add(salami);
pizza.getIngredients().add(cheese);
pizza.getIngredients().add(tomatoes);
session.save(pizza);

/* select pizza clients having an address in London */
List<PizzaClient> list = session
    .createQuery(
   "from Pizza c left join c.ingredients a where a.name = :ingr")
   .setString("ingr", "Tomato").list();
```

8.4. Equals implementation

If you try out this mapping, you will be surprised about the amount of queries generated if you update or delete an element of the collection.

If your pizza has 3 ingredients and you will delete one, you will find the following queries.

```
delete from Pizza_ingredients where Pizza_id=? and name=?
delete from Pizza_ingredients where Pizza_id=? and name=?
delete from Pizza_ingredients where Pizza_id=? and name=?
insert into Pizza_ingredients (Pizza_id, name) values (?, ?)
insert into Pizza_ingredients (Pizza_id, name) values (?, ?)
```

Hibernate is deleting all ingredients and re-inserts the 2 which persist. The reason is simple. If you do not implement equals and hashcode, Hibernate has no way to detect which element you have removed.

Once you have added an equals method like the following, removing an ingredient will only cause a single delete.

Extract of Ingredient class.

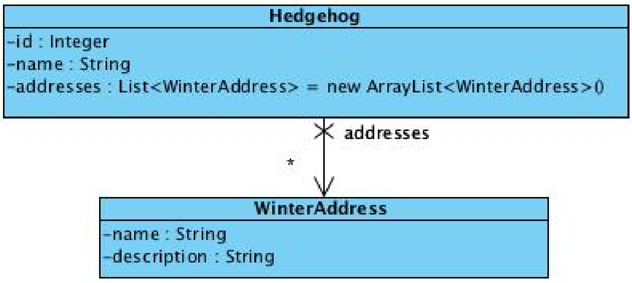
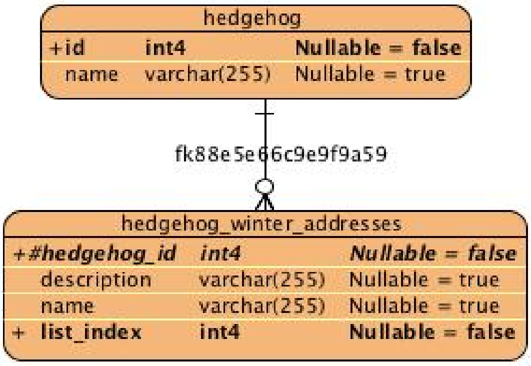
```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Ingredient)) return false;
    Ingredient that = (Ingredient) o;
    if (!name.equals(that.getName())) return false;
    return true;
}

@Override
public int hashCode() {
    return name.hashCode();
}
```

You need to take greatest care, when implementing *equals* and *hashCode*. Hibernate has additional requirements. Please have a look in the chapter Equals and HashCode Section 6.6, “Equals and Hashcode”.

8.5. Composition as list of many classes

Full source code is provided in the package: *de.laliluna.component.collection1*

In this example the component will have a defined order, which is guarantied by adding a position column to the database.

Classes	Tables
 Hedgehog -id : Integer -name : String -addresses : List<WinterAddress> = new ArrayList<WinterAddress>() addresses * WinterAddress -name : String -description : String	 hedgehog +id int4 Nullable = false name varchar(255) Nullable = true fk88e5e66c9e9f9a59 hedgehog_winter_addresses +#hedgehog_id int4 Nullable = false description varchar(255) Nullable = true name varchar(255) Nullable = true + list_index int4 Nullable = false

A hedgehog which is successful in life has of course many winter addresses.

Annotation mapping.

```
import java.util.ArrayList;
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.IndexColumn;

@Entity
public class Hedgehog {

   @ElementCollection
   @CollectionTable(name = "hedgehog_winter_addresses",
      joinColumns = @JoinColumn(name = "hedgehog_id"))
    @OrderColumn(name = "list_index")
   //@IndexColumn(name = "list_index")
   private List<WinterAddress> addresses = new ArrayList<WinterAddress>();
```

@ElementCollection defines the component mapping. *@CollectionTable is optional. It specifies the name of the table and the foreign key column.* _*@IndexColumn* (Hibernate API) is optional and defines this relation as an indexed relation.

IndexColumn

Hibernate has always offered the *@IndexColumn* but since JPA 2 there is a JPA alternative as well. *@OrderColumn*

The WinterAddress should be made embeddable. If you do not make it embeddable, then the class will be serialized and written to a blob field. You can still read and write the objects, but they are not selectable in the database.

```
import java.io.Serializable;
import javax.persistence.Embeddable;

@Embeddable
public class WinterAddress implements Serializable{
   private String name;

   private String description;
```

XML mapping.

```
<hibernate-mapping package="de.laliluna.component2">
  <class name="Hedgehog" table="thedgehog">
   ....... snip ........
    <list name="addresses" table="twinteraddress">
      <key column="hedgehog_id" not-null="true"></key>
      <list-index column="list_index"></list-index>
      <composite-element class="WinterAddress">
        <property name="name" type="string"></property>
        <property name="description" type="string"></property>
      </composite-element>
    </list>
  </class>
</hibernate-mapping>
```

Samples of use:

```
/* create and set components */
Hedgehog hedgehog = new Hedgehog("Peter");
WinterAddress address1 = new WinterAddress("stack of wood",
   "close to the apple tree");
WinterAddress address2 = new WinterAddress("shelter",
   "old shelter of the neighbour");
hedgehog.getAddresses().add(address1);
hedgehog.getAddresses().add(address2);
session.save(hedgehog);

/* select hedhehogs having a address named „first class hotel“*/
List<Hedgehog> list = session.createQuery
   ("from Hedgehog h left join h.addresses a where a.name = ?")
   .setString(0, "first class hotel").list();
```

IndexColumn and equals implementation

Using a *Set* required implementation of *equals* and *hashcode*. Using an index column this is not necessary. Hibernate uses the index column to identify an element of the collection.

Using a *List* and omitting the index column will always lead to the inefficient deletes and reinserts when updating an element. Implementing *equals* does not help with *List*.

8.6. Advanced details

Changing column names

By default, the column names are taken from the component. For the *recipe* examples a column *firstImpression* will be added to the *recipe* table. If you want to change column names of a component in an entity, you can use @AttributeOverrides in the annotation mapping, have a look at the Sheep and Pullover example in the same package as the previous example.

Sheep class.

```
@Entity
public class Sheep  {

    @Embedded
    // we would like the color field of pullover to be mapped to a different
    // column
    @AttributeOverride(name = "color", column = @Column(name = "pullover_column"))
    private Pullover pullover;
```

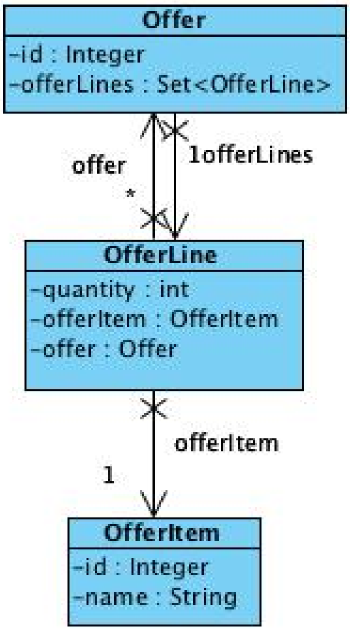
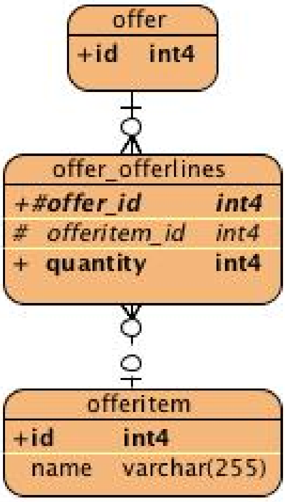
Querying components

You cannot load a component by id or write a query like *select a from Address a*, but it is still possible to query a list of all addresses. You just have to make use of the owning entity.

```
List<Address> addresses = session.createQuery
      ("select p.address from Person p")
      .list();
```

8.7. Composition 1:n:1

Full source code is provided in the package: *de.laliluna.component.ternary* A typical relation for this kind of mapping is: Offer → OfferLine → OfferItem

Classes	Tables
 Offer -id : Integer -offerLines : Set<OfferLine> offer * 1offerLines OfferLine -quantity : int -offerItem : OfferItem -offer : Offer offerItem 1 OfferItem -id : Integer -name : String	 offer +id int4 offer_offerlines +#offer_id int4 # offeritem_id int4 + quantity int4 offeritem +id int4 name varchar(255)

Of course you could map this without using components as a simple 1:n + 1:1 relation. You may choose the component mapping if your n-class is not an entity that you will deal with directly. Your application logic will probably always access an offerLine by selecting the offer first. In this case you will probably not need an entity offerLine. In case of a relation Customer → Order → Invoice would be probably better mapped as 1:n + 1:1 as you will probably access all three entities. Equals and hashcode As already explained in example component3 you must implement equals and hashcode when you want to use a set. Implementing is not at all easy, so you might consider to prefer list, map, idbag for this kind of mapping. Nevertheless, I used a set here.

Annotation mapping.

```
import java.io.Serializable;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import javax.persistence.JoinColumn;
import org.hibernate.annotations.CollectionOfElements;
import javax.persistence.Entity;
...... snip ........

@Entity
@SequenceGenerator(name = "offer_seq", sequenceName = "offer_id_seq")
public class Offer implements Serializable {

   @ElementCollection
   @CollectionTable(name="offer_line", joinColumns = @JoinColumn(name = "offer_id"))
   private Set<OfferLine> offerLines = new HashSet<OfferLine>();
```

```
   import java.io.Serializable;
import javax.persistence.Embeddable;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import org.hibernate.annotations.Parent;
```

```
@Embeddable  // class can be embedded into other classes
public class OfferLine  implements Serializable{

   @OneToOne  // one to one relation
   @JoinColumn(name="offeritem_id") // optional column name of the foreign key
   private OfferItem offerItem;

   @Parent  // offer is specified as reference back to the embedding class
   private Offer offer;
```

The OfferLine needs to be embeddable and has a relation to OfferItem. In addition, I added a relation back to the parent class (Offer). OfferItem has no component specific annotations

XML mapping.

```
<hibernate-mapping package="de.laliluna.component.ternary " >
  <class name="Offer" table="toffer" >
......... snip .......
    <set name="offerLines" table="toffer_line" >
    <key column="offer_fk" ></key>
      <composite-element class="OfferLine" >
      <parent name="offer"/>
        <property name="quantity" type="integer" ></property>
        <many-to-one name="offerItem" class="OfferItem" >
          <column name="item_fk"></column>
        </many-to-one>
      </composite-element>
    </set>
  </class>
</hibernate-mapping>
```

OfferItem has no component related tags. It is a simple mapping.

Samples of use:

```
/* create */
OfferItem item1 = new OfferItem(null, "Red flowers");
session.save(item1);

Offer offer = new Offer();
OfferLine line1 = new OfferLine();
line1.setQuantity(5);
line1.setOfferItem(item1);
offer.getOfferLines().add(line1);
session.save(offer);

/* select offers where quantity of red flowers are 5 */
List<Offer> list = session.createQuery(
   "from Offer o left join o.offerLines l where l.quantity = :q " +
      "and l.offerItem.name = :n")
    .setInteger("q", 5).setString("n", "Red flowers").list();
```

8.8. Not included mappings

I decided not to show the use of some more complex mappings as I want to focus on the most common tasks. If you are interested in this mappings have a look at the test cases coming with the Hibernate download.

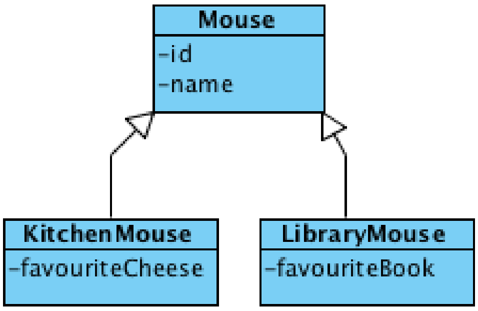
- composite-map-key
- dynamic components

Chapter 9. Inheritance

We will explore mapping Java inheritance structures to the database.

9.1. Use Case

Our use case is about mice. There is a parent class *Mouse* and two sub classes *LibraryMouse*, *KitchenMouse*. As the name indicates one lives in a library and the other one in the kitchen.



Hibernate allows to map inheritance structures to database table and it provides a number of approaches to do this. Not all approaches are support by XML mappings and not all by annotations. Source code for the samples can be found in the package *de.laliluna.inheritance* in the project *mapping-examples-xml* and *mapping-examples-annotation*.

9.2. Overview on mapping approaches

There are five approaches to map class hierarchies. I will give you a quick overview before the next chapters describes all the details.

Mapped super class

If the parent class is not an entity but only provides some common attributes and methods, then you have to annotate it with @MappedSuperclass. Once it is annotated, the attributes will be stored in the tables of the sub classes *LibraryMouse* and *KitchenMouse*.

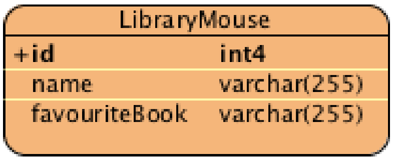
```
@MappedSuperclass
public class Mouse {
...
```

You cannot have a relation from a class to a *Mouse* as it is no entity but you are still able to query for the Mouse class.

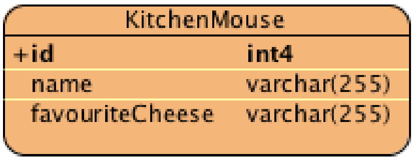
```
session.createQuery("select m from Mouse m").list();
```

This will sent two queries one for *KitchenMouse* and one for the *LibraryMouse* table. Then Hibernate will add both results to the result list.

By the way you can query *java.lang.Object* as well.



LibraryMouse	
+id	int4
name	varchar(255)
favouriteBook	varchar(255)



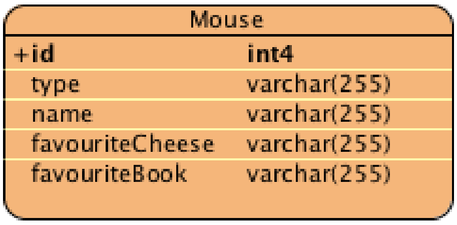
KitchenMouse	
+id	int4
name	varchar(255)
favouriteCheese	varchar(255)

Single table

Package : …inheritance.singletable

```
// Annotation
@Inheritance(
strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="mouse_type",discriminatorType=
DiscriminatorType.STRING)

// XML
<subclass name="KitchenMouse" discriminator-value="kitchen">
```

One table for the class hierarchy. A column identifies the type of a table row. In the picture below it is the *type* column.



Mouse	
+id	int4
type	varchar(255)
name	varchar(255)
favouriteCheese	varchar(255)
favouriteBook	varchar(255)

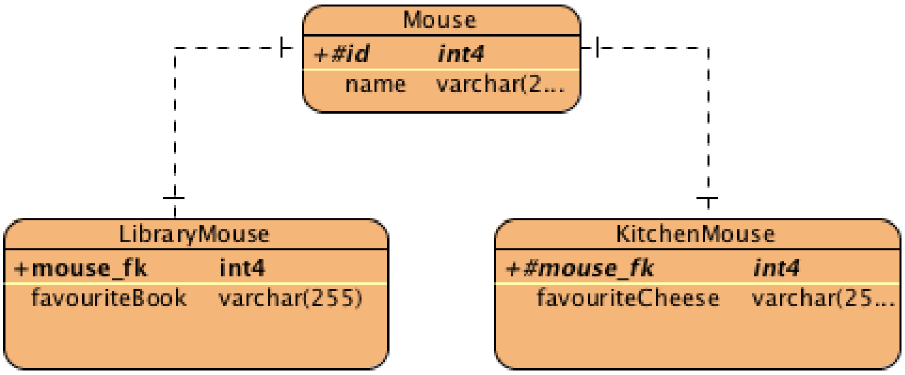
Queries are very simple without any needs for joins or unions. Performance is good: Only one statement needed for inserts and updates. No joins when data is selected All individual attributes of a subclass must allow null values, as other subclasses will not set them.

Joined subclass

Package: …inheritance.joined

```
// Annotation
@Inheritance(strategy=InheritanceType.JOINED)

// XML
<joined-subclass name="LibraryMouse" table="LibraryMouse">
```

One table for each class of the hierarchy including parent and subclasses.



Common fields in common table, individual fields in subclass table. Two statements are needed for inserts or updates. Join needed when parent or subclass is selected.

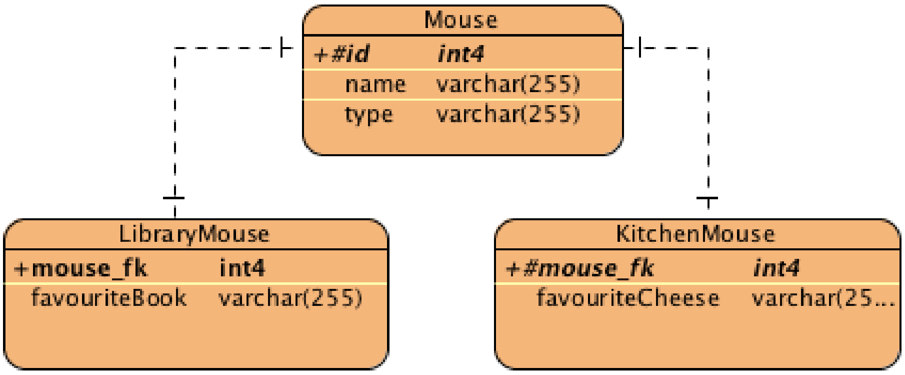
Mixing Joined subclass with a discriminator

XML only!

Package: …inheritance.joineddiscriminator

```
<subclass name="LibraryMouse" discriminator-value="LibraryMouse">
<join table="LibraryMouse">
```

One table for each class of the hierarchy including parent and subclasses. Discriminator column to identify object type



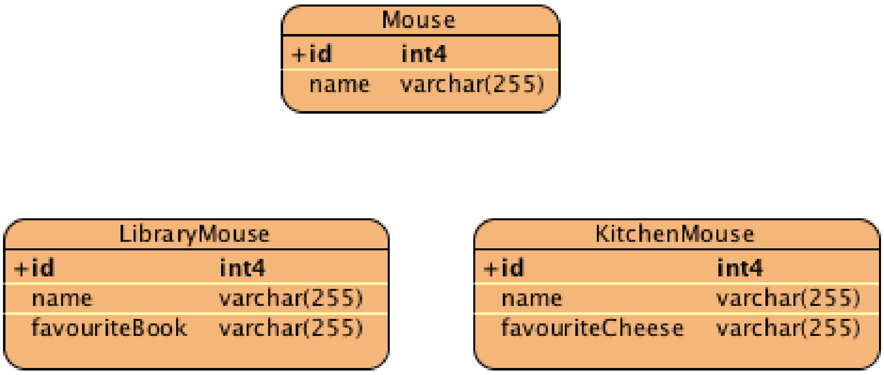
Queries are not always easy when relations have to be queried. Common fields in common table, individual fields in subclass table. Two statements needed for inserts and updates. Join needed when parent or subclass is selected but join is faster as compared to joined-subclass.

Table per class

Package: …inheritance.union (XML) and .inheritance.tableperclass (annotation)

```
// Annotation
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

// XML
<union-subclass name="GirlGroup" table="tgirlgroup">
```

One table for each subclass and optional for the parent class.



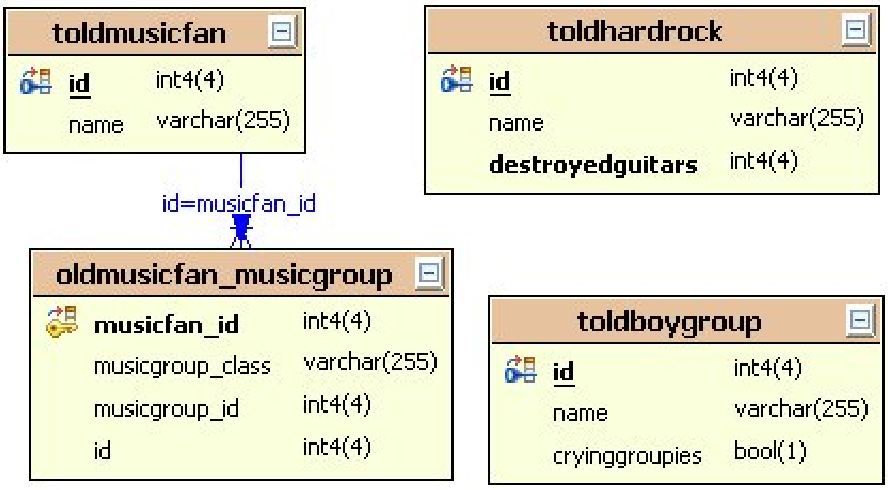
A sub class instance is stored in one table. Only one query needed for inserts. Big union including all subclasses when parent is called. No union when subclass is called. Id generator limitations foreign key relations are not possible

XML includes

XML only Package: inheritance.xmlinclude

```
<class name="OldBoyGroup" table= "toldboygroup"> \&allproperties;
```

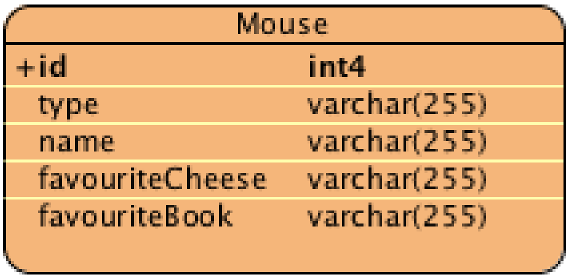
One table for each subclass . XML inclusion is used



Relations to parent class are not easy to query. Only one query needed for inserts. Query to parent class needs one select for each subclass.

9.3. Single Table

The first approach will only use one table. From the class diagram below you can see that a plant has some fields and that each subclass adds a field, for example flower adds the color. This results can be found in one big table holding all the fields. The disadvantage of this approach is that fields in the subclasses Tree and Flower must accept null values. A flower will not set the field has_fruits. So has_fruits must allow null values. Full source code is provided in the package: *de.laliluna.inheritance.singletable*



Mouse	
+id	int4
type	varchar(255)
name	varchar(255)
favouriteCheese	varchar(255)
favouriteBook	varchar(255)

Hibernate distinguishes the different classes using a discriminator column. When the column *type* contains KitchenMouse then the row will be treated as KitchenMouse.

Annotation mapping.

```
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
.......... snip .......
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type")
public class Mouse {
    @Id @GeneratedValue
    private Integer id;
    private String name;
........
```

The *LibraryMouse* and *KitchenMouse* class are fairly simple. They inherit from Mouse and add only their specific attributes.

```
@Entity
public class KitchenMouse extends Mouse{

    private String favouriteCheese;
......
```

Optionally you can define a different discriminator value. By default the class name is used.

```
@Entity
@DiscriminatorValue("lm")
public class LibraryMouse extends Mouse{

    private String favouriteBook;
.....
```

Other classes can have relations to the subclass (Flower) as well as to the parent class (Plant).

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
........ snip ........
@Entity
```

```
public class House implements Serializable {
.... snip .....
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "garden_plant_id")
    private Set<Mouse> allMice = new HashSet<Mouse>();

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "garden_flower_id")
    private Set<KitchenMouse> kitchenMice = new HashSet<KitchenMouse>();
```

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

Required Is set in the parent class Defines the inheritance strategy.

@DiscriminatorColumn(name="plant_type",discriminatorType=DiscriminatorType.STRING)

Optional Is set in the parent class Can be used to explicitly specify a discriminator column

@DiscriminatorValue(value="kitchen_mouse")

Optional Can be set in the parent and subclasses Can be used to explicitly define what is written into the discriminator column for the specific class.

DiscriminatorType.CHAR | .INTEGER | .STRING

Optional Is set in the parent class Is part of the discriminator column definition. Defines which type the discriminator column has.

XML mapping.

```
<hibernate-mapping package="de.laliluna.inheritance.singletable" >
  <class name="Mouse" >
......... snip .......
    <discriminator column="plant_type" type="string"></discriminator>
    <subclass name="KitchenMouse" discriminator-value="KitchenMouse">
      <property name="favouriteCheese" />
    </subclass>
    <subclass name="LibraryMouse"  discriminator-value="LibraryMouse">
      <property name="favouriteBook"/>
    </subclass>
  </class>
</hibernate-mapping>
```

Other classes can have relations to the subclass (KitchenMouse) as well as to the parent class (Mouse).

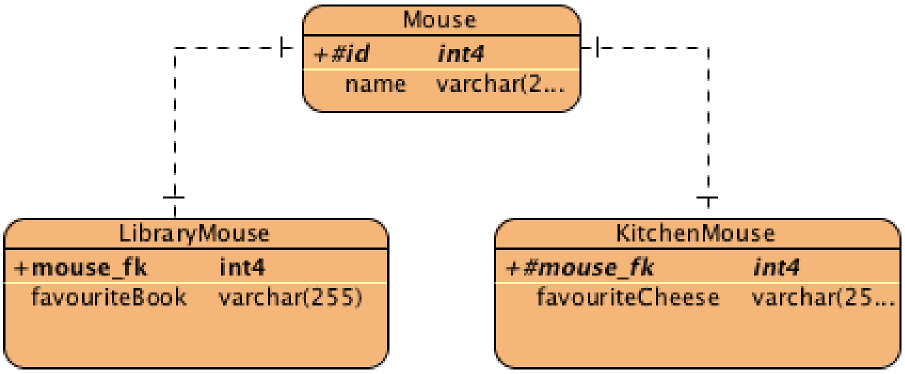
```
<hibernate-mapping package="de.laliluna.inheritance.singletable" >
  <class name="House" >
...... snip ........
    <set name="allMice"  table="house_mouse">
      <key column="house_id"></key>
      <many-to-many class="Mouse">
       <column name="mouse_id"></column>
      </many-to-many>
    </set>

    <set name="kitchenMice" table="house_kitchen_mice" >
      <key column="house_id"></key>
      <many-to-many class="KitchenMouse">
       <column name="kitchen_mouse_id"></column>
      </many-to-many>
    </set>
  </class>
</hibernate-mapping>
```

Samples of use:

```
/* create and set relation */
House house = new House();
Mouse bea = new Mouse("Bea");
house.getMice().add(bea);
KitchenMouse john = new KitchenMouse("John");
house.getMice().add(john);
LibraryMouse tim = new LibraryMouse("Tim");
house.getMice().add(tim);
session.save(bea);
session.save(john);
session.save(tim);
session.save(house);

/* get all kind of mice*/
List<Mouse> result = session.createQuery("select m from Mouse m")
   .list();

/* select all kitchen mice who like Gauda cheese blue flowers */
List<KitchenMouse result = session
   .createQuery("select m from KitchenMouse m where m.favouriteCheese ='Gauda'")
   .list();

/* select all mice of type LibraryMouse */
List<LibraryMouse> result = session
   .createQuery("select m from Mouse m where type(m) = LibraryMouse ")
   .list();
```

9.4. Joined Inheritance

Full source code is provided in the package: *de.laliluna.inheritance.joined*



The parent class holding common attributes (id, name) is saved in the mouse table. The individual tables share the same primary key with the parent class table.

This approach is fully normalized. We do not need a discriminator column. Hibernate works out the type by clever SQL queries.

Annotation mapping.

```
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
....... snip  ....

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Mouse {
    @Id @GeneratedValue
    private Integer id;
    private String name;
```

@Inheritance(strategy = InheritanceType.JOINED) specifies the inheritance strategy. The subclasses do not have any inheritance related annotations.

```
@Entity
public class KitchenMouse extends Mouse{

    private String favouriteCheese;
......
```

```
@Entity
public class LibraryMouse extends Mouse{

    private String favouriteBook;
.....
```

XML mapping.

```
<hibernate-mapping package="de.laliluna.inheritance.joined">
  <class name="Mouse">
....... snip ...........
    <joined-subclass name="KitchenMouse">
      <key column="mouse_id"></key>
      <property name="favouriteBook" column="favourite_book"/>
    </joined-subclass>
    <joined-subclass name="LibraryMouse" table="library_mouse">
      <key column="mouse_id"></key>
      <property name="favouriteCheese" column="favourite_cheese"/>
    </joined-subclass>
  </class>
</hibernate-mapping>
```

To be aware of possible performance issues, I will explain you the behaviour of this mapping. When we insert an object of the subclass LibraryMouse, Hibernate will generate two inserts. Common attributes are saved in the table mouse. Subclass specific attributes are saved in the table of the subclass.

```
insert into Mouse (name, id) values (?, ?)
insert into LibraryMouse (favouriteBook, id) values (?, ?)
```

When we select data from a subclass, we always need a join.

```
session.createQuery("from LibraryMouse m ").list();
```

Resulting SQL query:

```
select
    kitchenmou0_.id as id5_,
    kitchenmou0_1_.name as name5_,
    kitchenmou0_.favouriteCheese as favourit1_6_
from
    KitchenMouse kitchenmou0_
inner join
    Mouse kitchenmou0_1_
        on kitchenmou0_.id=kitchenmou0_1_.id
```

Selecting the parent class will result in a big join of all subclasses.

```
session.createQuery("from Mouse ").list();
```

Resulting SQL query:

```
select
    mouse0_.id as id5_,
    mouse0_.name as name5_,
    mouse0_1_.favouriteCheese as favourit1_6_,
    mouse0_2_.favouriteBook as favourit1_7_,
    case
        when mouse0_1_.id is not null then 1
        when mouse0_2_.id is not null then 2
        when mouse0_.id is not null then 0
    end as clazz_
from
    Mouse mouse0_
left outer join
    KitchenMouse mouse0_1_
        on mouse0_.id=mouse0_1_.id
left outer join
    LibraryMouse mouse0_2_
        on mouse0_.id=mouse0_2_.id
```

As in our previous example, other classes can have a relation to the parent class or to one of the sub classes.

Samples of use.

```
/* create and set relation */
House house = new House();
Mouse bea = new Mouse("Bea");
house.getMice().add(bea);
KitchenMouse john = new KitchenMouse("John");
house.getMice().add(john);
LibraryMouse tim = new LibraryMouse("Tim");
house.getMice().add(tim);
session.save(bea);
session.save(john);
session.save(tim);
session.save(house);

/* get all kind of mice*/
List<Mouse> result = session.createQuery("select m from Mouse m")
   .list();

/* select all kitchen mice who like Gauda cheese blue flowers */
List<KitchenMouse result = session
   .createQuery("select m from KitchenMouse m where m.favouriteCheese ='Gauda'")
   .list();

/* select all mice of type LibraryMouse */
List<LibraryMouse> result = session
   .createQuery("select m from Mouse m where type(m) = LibraryMouse ")
   .list();
```

9.5. Joined Inheritance with Discriminator

Full source code is provided in the package: *de.laliluna.inheritance.joineddiscriminator* This mapping is only support, if you use XML mappings. It has the same class hierarchy as our last example. This approach is a combination of the two former examples. We have a discriminator as in the single table example and a table structure as in the last example. We combine a subclass with a join.

```
<hibernate-mapping package="de.laliluna.inheritance.joineddiscriminator">
  <class name="MusicFan" table="tmusicfan">
........ snip ..........
    <set name="musicGroups" table="musicfan_musicgroup">
      <key column="musicfan_id"></key>
      <many-to-many class="MusicGroup">
        <column name="musicgroup_id"></column>
      </many-to-many>
    </set>
  </class>
  <class name="MusicGroup" table="tmusicgroup">
........ snip ..........
    <discriminator column="discriminator"></discriminator>
    <property name="name" type="string"></property>

    <subclass name="BoyGroup" discriminator-value="boygroup">
     <join table="tboygroup">
        <key column="musicgroup_id"></key>
        <property name="cryingGroupies" type="boolean"></property>
      </join>

    </subclass>
    <subclass name="HardrockGroup" discriminator-value="hardrock">
      <join table="thardrock">
        <key column="musicgroup_id"></key>
        <property name="destroyedGuitars" type="integer" not-null="true"/>
      </join>
    </subclass>
  </class>
</hibernate-mapping>
```

To be aware of possible performance issues, I will explain you the behaviour of this mapping. When we insert an object of the subclass boygroup, Hibernate will generate two inserts. Common attributes are saved in the table tmusicgroup. Subclass specific attributes are saved in the table of the subclass.

```
insert into tmusicgroup (name, discriminator, id) values (?, 'boygroup', ?)
insert into tboygroup (cryingGroupies, musicgroup_id) values (?, ?)
```

When we select data from a subclass, we always need a join.

```
session.createQuery("from HardrockGroup").list();
```

Resulting query:

```
select hardrockgr0_.id as id45_,
hardrockgr0_.name as name45_,
hardrockgr0_1_.destroyedGuitars as destroye2_47_
from
tmusicgroup hardrockgr0_
inner join thardrock hardrockgr0_1_ on
   hardrockgr0_.id=hardrockgr0_1_.musicgroup_id
where hardrockgr0_.discriminator='hardrock'
```

When we select data form the parent class all tables are joined.

```
session.createQuery("from MusicGroup").list();
```

Resulting query:

```
select
musicgroup0_.id as id45_,
musicgroup0_.name as name45_,
musicgroup0_1_.cryingGroupies as cryingGr2_46_,
musicgroup0_2_.destroyedGuitars as destroye2_47_,
musicgroup0_.discriminator as discrimi2_45_
from
tmusicgroup musicgroup0_
left outer join tboygroup musicgroup0_1_
   on musicgroup0_.id=musicgroup0_1_.musicgroup_id
left outer join thardrock musicgroup0_2_
   on musicgroup0_.id=musicgroup0_2_.musicgroup_id
```

The key difference between this approach and xref:inheritanceonetableperclass1 is the use of the discriminator column in queries.

```
session.createQuery("from MusicGroup mg where  mg.destroyedGuitars>150")
   .list();
```

would result in the following query for the current approach.

```
select musicgroup0_.id as id45_, musicgroup0_.name as name45_,
musicgroup0_1_.cryingGroupies as cryingGr2_46_,
musicgroup0_2_.destroyedGuitars as destroye2_47_,
musicgroup0_.discriminator as discrimi2_45_
from
tmusicgroup musicgroup0_
left outer join tboygroup musicgroup0_1_
   on musicgroup0_.id=musicgroup0_1_.musicgroup_id
left outer join thardrock musicgroup0_2_
   on musicgroup0_.id=musicgroup0_2_.musicgroup_id
where musicgroup0_2_.destroyedGuitars>150
```

The query is considerably faster as compared to the normal joined apporach. For a test I run with about 20.000 music groups, equally divided into boygroups and hardrock groups. It becomes even faster when you explicitly specify the class.

```
list = session.createQuery(
 "from MusicGroup mg where mg.class = HardrockGroup and mg.destroyedGuitars>150")
 .list();
```

You might consider to use this approach instead of if select performance is important.

9.6. Mixing Single table and Joined

Mixing xref:inheritanceonetableperclass2 and xref:inheritancesingletable I just want to mention that mixing these two approaches is possible.

```
    <subclass name="Flower" discriminator-value="flower">
    <property name="color" type="string"></property>
    </subclass>
    <subclass name="HardrockGroup" discriminator-value="hardrock">
      <join table="thardrock">
        <key column="musicgroup_id"></key>
        <property name="destroyedGuitars" type="integer" not-null="true"/>
      </join>
    </subclass>
```

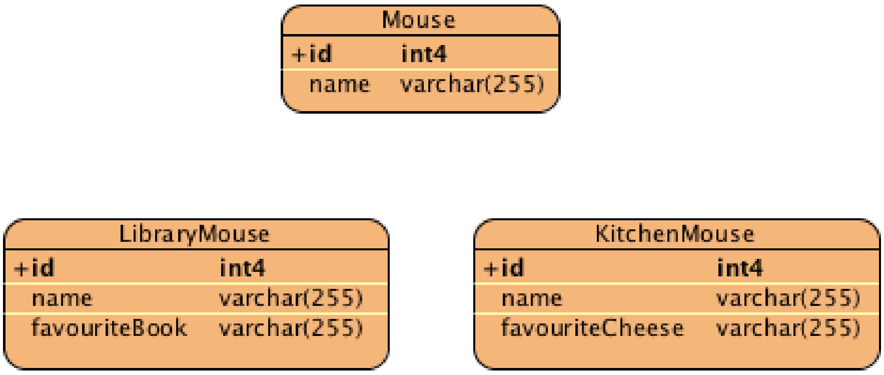
9.7. Union Inheritance

Full source code is provided in the package: *de.laliluna.inheritance.union* for the XML mapping and *de.laliluna.inheritance.tableperclass* for the annotation mapping. The class hierarchy does not differ from our former examples. The difference is in the tables. Parent class objects will be saved in a parent class table and each sub class objects in a separate table.

An instance of *Mouse* is saved in table *mouse*, a *KitchenMouse* in the *kitchen_mouse* table and an instance of *LibraryMouse* in the *library_mouse* table.

Imagine a relation from the class *House*. The relation is persisted in a join table *house_mouse* containing two columns *house_fk* and *mouse_fk*. The *house_fk* references the house table and *mouse_fk* one entry either in the table *mouse*, *kitchen_mouse* or *library_mouse*. As the table is not known before hand, we cannot impose a foreign key reference constraint.

A further limitation exists with id generators. The primary key of *mouse*, *kitchen_mouse* or *library_mouse* must be shared across all these tables. You cannot use IDENTITY or AUTO as strategy to generate primary keys. Choose a shared SEQUENCE if supported by your database, or another strategy that guaranties unique primary keys across these tables. The parent class can be abstract.



Annotation mapping.

```
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
...... snip ....
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Mouse {
    @Id @GeneratedValue
    private Integer id;
    private String name;
.......
```

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) defines the inheritance strategy. The sub classes are quite simple and do not contain inheritance specific annotations.

```
@Entity
public class KitchenMouse extends Mouse{

    private String favouriteCheese;
......
```

```
@Entity
public class LibraryMouse extends Mouse{

    private String favouriteBook;
.....
```

The class *House* contains a normal one to many relation to *Mouse*. In this case a relation table is used but you could use a simple foreign key column in the *mouse*, *kitchen_mouse* and *library_mouse* tables as well.

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToMany;
..... snip ........
@Entity
public class House {

    @Id @GeneratedValue
    private Integer id;

   @OneToMany
   @JoinTable(name = "house_mouse", joinColumns = @JoinColumn(name = "house_fk"),
        inverseJoinColumns = @JoinColumn(name = "mouse_fk"))
    private Set<Mouse> mice = new HashSet<Mouse>();
```

XML mapping

We use the mapping union-subclass.

```
<hibernate-mapping package="de.laliluna.inheritance.union">
  <class name="House">
....... snip .....
    <set name="mice" table="house_mouse">
      <key column="house_fk"></key>
      <many-to-many class="Mouse">
```

```
        <column name="mouse_fk"></column>
      </many-to-many>
    </set>
  </class>
  <class name="Mouse" table="mouse">
....... snip .....
    <union-subclass name="KitchenMouse" table="kitchen_mouse">
     <property name="favouriteCheese"/>
    </union-subclass>
    <union-subclass name="LibraryMouse" table="library_mouse">
      <property name="favouriteBook" />
    </union-subclass>
  </class>
</hibernate-mapping>
```

Union subclass mapping does build a union from the parent class and all subclasses. Columns that do not exist in a subclass are set to null. This has no influence on your classes but is needed to make union work. If your Mouse class is abstract or if you do not need a table mouse you can replace

```
  <class name="Mouse" table="mouse">
```

with

```
  <class name="Mouse" abstract="true">
```

Now, let’s have a look at the behaviour of this mapping. A query like

```
session.createQuery("from Mouse m where type(m) = LibraryMouse").list();
```

will create a select from a subselect, where the subselect does union all the subclasses and the parent class table.

```
select
    mouse0_.id as id5_,
    mouse0_.name as name5_,
    mouse0_.favouriteCheese as favourit1_6_,
    mouse0_.favouriteBook as favourit1_7_,
    mouse0_.clazz_ as clazz_
from
    ( select
        id,
        name,
        null::varchar as favouriteCheese,
        null::varchar as favouriteBook,
        0 as clazz_
    from
        Mouse
    union
    all select
        id,
        name,
        favouriteCheese,
        null::varchar as favouriteBook,
        1 as clazz_
    from
        KitchenMouse
    union
    all select
        id,
        name,
        null::varchar as favouriteCheese,
        favouriteBook,
        2 as clazz_
    from
        LibraryMouse
) mouse0_
    where
        clazz_=2
```

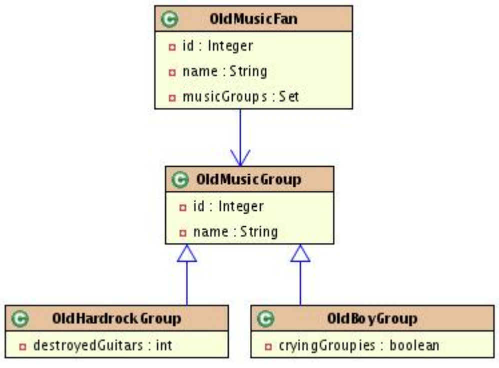
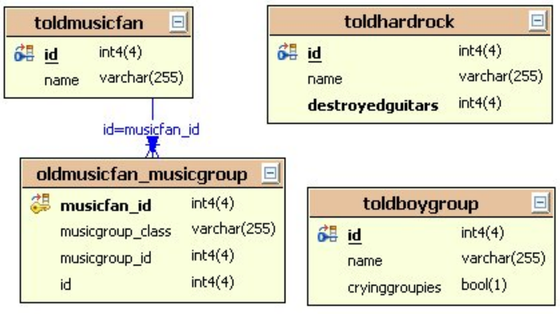
To build a union like this takes some time. An advantage of the union approach is that an insert of a class *Mouse* will only happen in its table. The parent table is not touched.

Samples of use.

```
/* create and set relation */
House house = new House();
Mouse bea = new Mouse("Bea");
house.getMice().add(bea);
KitchenMouse john = new KitchenMouse("John");
house.getMice().add(john);
LibraryMouse tim = new LibraryMouse("Tim");
house.getMice().add(tim);
session.save(bea);
session.save(john);
session.save(tim);
session.save(house);

/* get all kind of mice*/
List<Mouse> result = session.createQuery("select m from Mouse m").list();

/* select all kitchen mice who like Gauda cheese blue flowers */
List<KitchenMouse result = session
   .createQuery("select m from KitchenMouse m where m.favouriteCheese ='Gauda'").list();

/* select all mice of type LibraryMouse */
List<LibraryMouse> result = session
   .createQuery("select m from Mouse m where type(m) = LibraryMouse ").list();
```

9.8. XML Includes

Full source code is provided in the package: *de.laliluna.inheritance.xmlinclude* This mapping is only possible with XML. It uses implicit polymorphism. We just do not specify any mapping for the parent class OldMusicGroup but use the inherited properties in our sub classes.

Classes	Tables
	

The consequence of this approach is that we have to specify all common fields (id, name) for all mappings. We can reduce the effort using a XML include.

OldMusicFan mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[<!ENTITY allproperties SYSTEM "bin/de/laliluna/example11/common.xml"> ] >

<hibernate-mapping package="de.laliluna.example11">
  <class name="OldMusicFan" table="toldmusicfan">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">tmusicfan_id_seq</param>
      </generator>
      <!--  MySQL generator for a increment field
        <generator class="increment"/>
      -->
    </id>
    <property name="name" type="string"></property>
    <set name="musicGroups" table="oldmusicfan_musicgroup">
      <key column="musicfan_id"></key>
     <many-to-any meta-type="string" id-type="integer">
      <meta-value class="OldBoyGroup" value="oldboy"/>
      <meta-value class="OldHardrockGroup" value="oldhardrock"/>
      <column name="musicgroup_class"></column>
      <column name="id"></column>
      </many-to-any>
    </set>
  </class>
  <class name="OldBoyGroup" table="toldboygroup">
    &allproperties;
    <property name="cryingGroupies" type="boolean"></property>
  </class>
  <class name="OldHardrockGroup" table="toldhardrock">
    &allproperties;
    <property name="destroyedGuitars" type="integer" not-null="true"></property>
  </class>
</hibernate-mapping>
```

common.xml.

```
<id name="id">
  <generator class="sequence">
    <param name="sequence">tmusicfan_id_seq</param>
  </generator>
  <!--  MySQL generator for a increment field
    <generator class="increment"/>
  -->
</id>

<property name="name" type="string"></property>
```

The command [<!ENTITY allproperties SYSTEM "bin/de/laliluna/example11/common.xml">] will replace all occurrences of &allproperties; with the context of the common.xml file. An insert into a subclass will only result into an insert in one table. A query of all old music groups causes two queries to be issued, one for each subclass.

```
session.createQuery("from de.laliluna.inheritance.xmlinclude.OldMusicGroup")
   .list();
```

Very special is also the relation to music fan. We need to use a special kind of any mapping to allow Hibernate to retrieve the proper class. I recommend naming the second column with the same name as both field name and column name of the subclass.

```
<column name="musicgroup_class"></column>
<column name="id"></column>
```

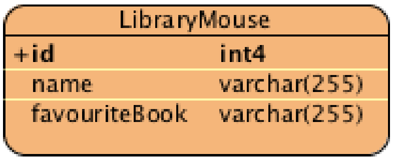
If you do not follow this, you can not issue a query like the following.

```
session.createQuery("select mf from OldMusicFan mf  where  mf.musicGroups.id  in "+
   "(select hg.id from OldHardrockGroup  hg where hg.destroyedGuitars > 150) ")
   .list();
```

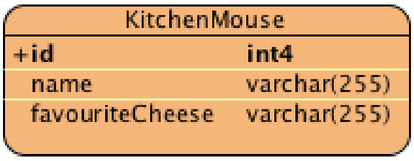
I consider this to be a Hibernate problem. My personal impression is that this is not at all a very beautiful mapping approach, especially when it comes to relations.

9.9. Mapped Super Class

This mapping is the annotation approach to map inherited fields. Full source code is provided in the package: *de.laliluna.inheritance.mappedsuperclass*



LibraryMouse	
+id	int4
name	varchar(255)
favouriteBook	varchar(255)



KitchenMouse	
+id	int4
name	varchar(255)
favouriteCheese	varchar(255)

This example has the same class hierarchy compared to former examples. This mapping is not a typical inheritance mapping. The parent class itself is not mapped and cannot be used in queries or relations. This approach is useful, if you want to include properties of the parent class in the subclass tables, but do not want to use the parent class directly. It could be abstract as well.

Mouse parent class.

```
import javax.persistence.Entity;
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.MappedSuperclass;
...... snip .......
@MappedSuperclass
public class Mouse implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "mouse_seq")
    private Integer id;
    private String name;
```

The sub classes do not have any inheritance specific annotations.

KitchenMouse sub class.

```
@Entity
public class KitchenMouse extends Mouse{

    private String favouriteCheese;
......
```

LibraryMouse sub class.

```
@Entity
public class LibraryMouse extends Mouse{

    private String favouriteBook;
.....
```

As already mentioned, another class can not have a relation to the parent class *Mouse*.

House.

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
....... snip .....
@Entity
public class House implements Serializable {
   @Id
   @GeneratedValue
   private Integer id;
   private String name;

   @ManyToMany(cascade=CascadeType.ALL)
   @JoinTable(name = "house_kitchenmouse",
   joinColumns = { @JoinColumn(name = "house_id") },
   inverseJoinColumns = {@JoinColumn(name = "kitchenmouse_id") })
   private Set<KitchenMouse> kitchenMice = new HashSet<KitchenMouse>();

   @ManyToMany(cascade=CascadeType.ALL)
   @JoinTable(name = "house_librarymouse",
      joinColumns = { @JoinColumn(name = "mouse_id") },
      inverseJoinColumns = { @JoinColumn(name = "librarymouse_id") })
   private Set<LibraryMouse> libraryMice = new HashSet<LibraryMouse>();
```

The query behaviour is the same as a normal 1:n mapping. If you do not need to access the parent class, than this is the fastest choice for inheritance mapping. You have to set the GeneratedValue in the parent class. The Generator itself has to be defined in the sub classes. I consider this a funny behaviour of Hibernate. I expected to define both in the parent class.

Samples of use.

```
/* create and set relation */
House house = new House("Edwin");
LibraryMouse libraryMouse = new LibraryMouse();
libraryMouse.setName("sensitive");
house.getLibraryMice().add(libraryMouse);
KitchenMouse kitchenMouse = new KitchenMouse();
kitchenMouse.setName("Mouse from Amsterdam");
```

```
house.getKitchenMice().add(kitchenMouse);
session.save(kitchenMouse);
session.save(libraryMouse);
session.save(house);

/* select houses having kitchen mice */
List list = session.createQuery(
      "select h from House h where size(h.kitchenMice) > 0").list();

/* select all houses with a kitchenMice having gouda as favourite cheese */
List list = session.createQuery(
      "select h from House h left join h.kitchenMice  k "+
   "where  k.favouriteCheese = :cheese)")
    .setParameter("cheese", "Gauda").list();
```

Chapter 10. Lob with Oracle and PostgreSQL

A lob is a large object. Lob columns can be used to store very long texts or binary files. There are two kind of lobs: CLOB and BLOB. The first one is a character lob and can be used to store texts. It is an alternative to varchar, which is limited in most databases. The second is a binary lob and can be used to store binary files.

10.1. PostgreSQL

You can find the source code in the project BlobPostgreSQL. I used PostgreSQL 8 with the drivers from jdbc.postgresql.org. If you use older versions or other drivers you might encounter different behaviour. **Character lob**

A character lob is very simple, if you use PostgreSQL. In the table it is represented as text and can take any size of characters:

```
CREATE TABLE annotation.document
(
  largeText text,
...... snip
```

The class has a simple String field:

```
   private String text;
```

Annotation mapping.

```
import javax.persistence.Entity;
import javax.persistence.Lob;
..... snip ......
@Entity
public class Document implements Serializable {
...... snip .......
   @Lob
   private String text;
```

Just add a *@Lob* annotation to a string field. That’s all.

XML mapping. There is one caveat, if we use XML. You must specify the type or Hibernate will generate a varchar column instead of text, if you let Hibernate generate your tables.

```
<property name="text" type="text"></property>
```

Binary lob (blob)

There are two options to store blobs in PostgreSQL. The first approach stores the file directly in the column. The type of such a column is bytea, which is a short form of byte array. The second approach is to store a OID in the column which references a file. PostgreSQL keeps the file separately from your table. This type of such a column is blob or binary object. You can store large blobs in both column types. The simpler way is to use bytea, but PostgreSQL needs a lot of memory, if you use the bytea column PostgreSQL and select a lot of \ rows having large bytea columns. Columns of type blob or binaryobject can read the lob as stream, once you access the data. The disadvantage of blob/binaryobject is that if you delete a table row the file is not deleted automatically. The following examples will show a work around for this problem. So you may freely select any of this approaches.

The PostgreSQL table is having a *bytea* and a *blob* column.

```
CREATE TABLE annotation.image
(
  imageasblob oid,
  imageasbytea bytea,
......
```

The bytea approach needs a byte array field in the class. I used *imageasBytea[]* Annotation mapping: It is important that you specify the type. You will get a blob column, if you don’t.

```
import javax.persistence.Entity;
import org.hibernate.annotations.Type;
...... snip ......
@Entity
public class Image implements Serializable {

   @Type(type = "org.hibernate.type.BinaryType")
   private byte imageAsBytea[];
........
```

XML mapping. If your field is of type byte[] you don’t have to specify a type. We added it optionally.

```
    <property name="imageAsBytea" type="org.hibernate.type.BinaryType"/>
```

There is nothing special in using this column. You can set byte arrays and get byte arrays.

Samples of use.

```
/* create a byte array and set it */
byte byteArray[] = new byte[10000000];
for (int i = 0; i < byteArray.length; i++) {
   byteArray[i] = '1';
}
Image image = new Image();
image.setImageAsBytea(byteArray);
/* write a field to a file */
FileOutputStream outputStream =
   new FileOutputStream(new File("image_file_bytea"));
outputStream.write(image.getImageAsBytea());
```

The blob approach requires some additional code. If we delete or update a blob image, the large object will not be deleted as well. Therefore we add a rule to the database, which will provide this for us. Hibernate will not see any of this code, but can simply rely on the fact that, if it deletes an entry. the lob will be deleted as well. Tip: The large object file will not be deleted automatically, if you use a blob column. In the psql client or pgadmin issue the following two statements. They will create rules. The first rule is called when a row is deleted. It deletes the corresponding lob. The second rule is called, when a row is updated. It deletes the old image, if the image has changed.

```
CREATE RULE droppicture AS ON DELETE TO annotation.image
  DO SELECT lo_unlink( OLD.imageasblob );
CREATE RULE reppicture AS ON UPDATE TO annotation.image
```

```
  DO SELECT lo_unlink( OLD.imageasblob )
   where OLD.imageasblob <> NEW.imageasblob;
```

We have to options in the class: a java.sql.Blob field and a byte array. We can use a byte array to map a lob to a blob column as well.

Annotation mapping.

```
import java.sql.Blob;
import javax.persistence.Entity;
import javax.persistence.Lob;
...... snip .........
import org.hibernate.annotations.Type;
import org.hibernate.type.BlobType;@Entity
public class Image implements Serializable {
   @Lob
   private byte imageAsBlob[];
   private Blob imageAsBlob2;
```

XML mapping: A field of type java.sql.Blob can be mapped with the following code. \ The type is optionally:

```
    <property name="imageAsBlob2" type="java.sql.Blob"></property>
```

The byte array approach does not work for XML. Either convert your byte arrays from and to java.sql.Blob or create a custom type that provides this feature. You can find further information about custom types in the Hibernate wiki.

Samples of use.

```
/* creating a blob */
byte byteArray[] = new byte[10000000];
for (int i = 0; i < byteArray.length; i++) {
   byteArray[i] = '1';
}
Image image = new Image();
image.setImageAsBlob(byteArray);  // a blob as byte array
image.setImageAsBlob2(Hibernate.createBlob(byteArray)); // a blob as blob
/* reading */
// read blob from a byte array is as simple as from a bytea
FileOutputStream outputStream =
   new FileOutputStream(new File("image_file_blob_array"));
outputStream.write(image.getImageAsBlob());
outputStream.close();
// reading of a blob from a blob is in fact a inputstream
outputStream = new FileOutputStream(new File("image_file_blob_blob"));
outputStream.write(image.getImageAsBlob2()
   .getBytes(1,(int)image.getImageAsBlob2().length()));
outputStream.close();
```

Tip: You can only access the length field if your transaction is open.

```
image.getImageAsBlob2().length()
```

10.2. Oracle

Character lob

A character lob is very simple, if you use can use Oracle 10 or Oracle XE (Express Edition). Oracle the setString() method of the prepared statement had a limitation of 32756 bytes = about 4000 characters. You can find additional documentation on the Oracle website. http://www.oracle.com/technology/sample_code/tech/java/codesnippet/jdbc/clob10g/handlingclobsinoraclejdbc10g.html The class has a simple String field:

```
   private String text;
```

Annotation mapping: Just add a *@Lob* annotation to a string field. That’s all.

```
import javax.persistence.Entity;
import javax.persistence.Lob;
..... snip ......
@Entity
public class Document implements Serializable {
...... snip .......
   @Lob
   private String text;
```

XML mapping

There is one caveat, if we use XML. You must specify the type or Hibernate will generate a varchar column instead of text, if you let Hibernate generate your tables.

```
    <property name="text" type="text"></property>
```

Oracle 9 work around If your field can be larger than 4000 characters, we need the following work around. We have to change the field type to Clob.

Annotation mapping.

```
import java.sql.Clob;
import javax.persistence.Lob;
........ snip .....
@Lob
   private Clob textWorkaround;
```

XML mapping.

```
    <property name="textWorkaround" type="clob"></property>
```

When you save data, use the following code. The code is not portable to Oracle 10.

```
/* writing a cblob */
/* initialize with short blob */
document.setTextWorkaround(Hibernate.createClob(" "));
/* save before we continue */
session.save(document);
/* get a oracle clob to have access to outputstream */
SerializableClob sc = (org.hibernate.lob.SerializableClob) document
.getTextWorkaround();
oracle.sql.CLOB clob = (oracle.sql.CLOB) sc.getWrappedClob();
/* write the text to the clob outputstream */
try {
   java.io.Writer pw = clob.getCharacterOutputStream();
   pw.write(buffer.toString());
   pw.close();
   session.getTransaction().commit();
} catch (SQLException e) {
   throw new RuntimeException(
         "Datenbankfehler beim Speichern des Lobs",e);
} catch (IOException e) {
   throw new RuntimeException(
         "Datenbankfehler beim Speichern des Lobs",e);
}
/* reading a cblob */
StringBuffer textFromWorkaround = new StringBuffer();
try {
    BufferedReader bufferedClobReader = new BufferedReader(documentReloaded.
      getTextWorkaround().getCharacterStream());
    String line = null;
    while((line = bufferedClobReader.readLine()) != null) {
        textFromWorkaround.append(line);
    }
    bufferedClobReader.close();
} catch (IOException e) {
    throw new RuntimeException("Fehler beim Lesen des Lobs",e);
} catch (SQLException e) {
    throw new RuntimeException("Fehler beim Lesen des Lobs",e);
}
```

Instead of using a Clob field in your class, we could hide this code as well. We could add a second property providing the clob as java.util.String, create a CustomType.

Binary lob (blob)

A binary lob is very simple, if you use can use Oracle 10 or Oracle XE (Express Edition). We have to options in the class: a java.sql.Blob field and a byte array. The corresponding column is always a blob. Tip: Only annotation mapping does support mapping a byte array to a blob out of the box. You could create a workaround and create a ArrayOutputStream and write this to a blob.

Annotation mapping.

```
import java.sql.Blob;
import javax.persistence.Entity;
import javax.persistence.Lob;
...... snip .........
import org.hibernate.annotations.Type;
import org.hibernate.type.BlobType;@Entity
public class Image implements Serializable {
   @Lob
   private byte imageAsBlob[];

   private Blob imageAsBlob2;
```

XML mapping. A field of type java.sql.Blob can be mapped with the following code. The type is optionally:

```
    <property name="imageAsBlob2" type="java.sql.Blob"></property>
```

The byte array approach does not work for XML. Either convert your byte arrays from and to java.sql.Blob or create a custom type that provides this feature. You can find further information about custom types in the Hibernate wiki.

Samples of use.

```
/* creating a blob */
byte byteArray[] = new byte[10000000];
for (int i = 0; i < byteArray.length; i++) {
   byteArray[i] = '1';
}
Image image = new Image();

image.setImageAsBlob(byteArray);  // a blob as byte array
image.setImageAsBlob2(Hibernate.createBlob(byteArray)); // a blob as blob

/* reading */
// read blob from a byte array is as simple as from a bytea
FileOutputStream outputStream =
   new FileOutputStream(new File("image_file_blob_array"));
outputStream.write(image.getImageAsBlob());
outputStream.close();
// reading of a blob from a blob is in fact a inputstream
outputStream = new FileOutputStream(new File("image_file_blob_blob"));
outputStream.write(image.getImageAsBlob2()
   .getBytes(1,(int)image.getImageAsBlob2().length()));
outputStream.close();
```

Tip: You can only access the length field if your transaction is open.

```
image.getImageAsBlob2().length()
```

Further discussions of blob mapping for older Oracle versions can be found here: http://www.hibernate.org/56.html http://forum.hibernate.org/viewtopic.php?t=931155

Chapter 11. Querying data

You have got three options to query data: HQL, criteria queries and SQL.

HQL is a object-oriented query language of Hibernate and supports all relation and inheritance mappings.

```
List orders = session.createQuery(
   "from Order o where o.invoice.paid=true order by o.datePurchased desc")
   .list();
```

Criteria queries are useful to dynamically generate queries. They support relation and inheritance mappings as well. Of course you can replace HQL completely with criteria query. The only disadvantage is that nearly everybody knowing SQL will be able to understand HQL easily. Criteria queries are completely different. The following method receives an array of payment stati and creates a query dynamically. This is very simple, isn’t it?

```
public List findOrderByPaymentstatus(Integer paymentstatus[])
         throws DBLayerException {

   Session session = InitSessionFactory.getInstance().getCurrentSession();
   session.beginTransaction();
   Criteria criteria = session.createCriteria(Order.class)
      .add(Property.forName("orderStatus").in(paymentstatus))
      .addOrder(Order.desc("datePurchased"));
   List orders = criteria.list();
   session.getTransaction().commit();

   return orders;
   }
```

I recommend to use HQL for all known queries and criteria queries when you need to generate complex queries dynamically. SQL does not know any relations or inheritance mappings, so it is by far more narrative to type SQL queries as compared to HQL. Only in rare cases when you need some special optimisation you could think of using SQL.

11.1. Useful tools

When starting to learn HQL it can be annoying to test queries. Every time, you want to execute a query, you need to startup a Hibernate configuration, which can take a couple of seconds.

There are a number of tools you might use.

11.1.1. Beam me into the code

Did you ever want to beam yourself into your code. Don’t search any more. You can achieve this using JRuby. JRuby has an interactive shell, which allows to execute any kind of code. JRuby can import and execute Java classes.

Start the Jruby shell by typing *jirb*. Below you can see the commands I have typed in the shell to start up Hibernate and to execute queries.

```
include Java
```

```
Dir["/path/toallLibraries/\*.jar"].each { |jar| require jar }

c =  Java::OrgHibernateCfg::Configuration.new
c.configure
sf = c.buildSessionFactory
s = sf.openSession

s.createQuery('select h from Hedgehog h').list
```

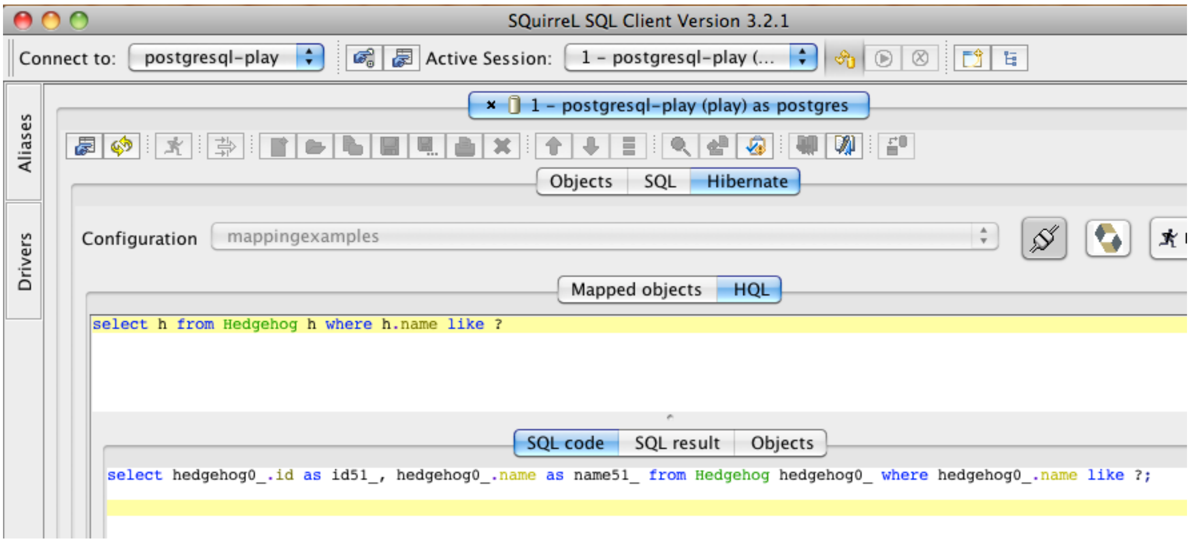
11.1.2. JBoss Tools

JBoss tools is a plugin for the Eclipse IDE. It provides an HQL editor which allows to execute instant HQL queries.

http://www.jboss.org/tools

11.1.3. Squirrel SQL Client

Squirrel is a well known SQL client which supports many databases. There is a Hibernate plugin which allows to execute HQL, see the resulting SQL and the result set.



Configuration: mappingexamples

```
select h from Hedgehog h where h.name like ?
```

```
select hedgehog0_.id as id51_, hedgehog0_.name as name51_ from Hedgehog hedgehog0_ where hedgehog0_.name like ?;
```

http://squirrel-sql.sourceforge.net/

11.2. HQL

The chapter HQL of the Hibernate reference is really good. This is why I keep this chapter fairly short. Further query examples can be found in the source code of the mapping examples. The queries can be found in the example project DeveloperGuide in the class *test.de.laliluna.example.TestQuery*

11.2.1. Select objects, scalars, etc

With HQL you can select

- one single class object (one JavaClub)

- a list of class objects (all JavaClubs)
- a list of an array of class objects(a list of an array, the array contains JavaClub and , multiple objects, object.children, data touple, create new objects (report item)

11.2.2. Simple select

```
   List results = session.createQuery("from JavaClub3 ").list();
      for (Iterator iter = results.iterator(); iter.hasNext();) {
         JavaClub3 club3 = (JavaClub3) iter.next();
         log.debug(club3);
      }
```

You do not need to write

```
createQuery("select c from JavaClub3 c").list()
```

This is only needed when your want to specify a specific object from joined object or a specific property. See below for further examples.

11.2.3. Select with a unique result

If you expect a unique result, you can call uniqueResult instead of list to get one object only.

```
JavaClub3 aClub = (JavaClub3) session.createQuery("from JavaClub3 where id=5")
   .uniqueResult();
log.debug("one single club: "+aClub);
```

11.2.4. Select with join returning multiple objects

If you use a join, you will receive multiple objects. The next example will show how to select only one of the objects.

```
results = session.createQuery("from JavaClub3 c left join c.members").list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   Object object[] = (Object[]) iter.next();
   log.debug("Club:  "+object[0]+ " Member: "+object[1]);
}
```

11.2.5. Select with join returning one object

If you use a join, you will receive multiple objects. If you name one object with select c you will get only one of them. This query will return multiple lines of the same Javaclub3 if multiple members exist.

```
results = session.createQuery(
   "select c from JavaClub3 c left join fetch c.members where "+
   "c.members.name='Peter'")
   .list();
   for (Iterator iter = results.iterator(); iter.hasNext();) {
      JavaClub3 javaClub3 = (JavaClub3) iter.next();
      log.debug("Club:  "+javaClub3);
   }
```

11.2.6. Select with join returning distinct results

The former example returned multiple entries of *Javaclub3* if multiple members exist. You can get distinct results if you create a HashSet but you will loose your sort order. This approach was recommended in former times.

```
Set setResults = new HashSet(session.createQuery(
      "select c from JavaClub3 c left join c.members  order by c.name desc")
   .list());
for (Iterator iter = setResults.iterator(); iter.hasNext();)
{
   JavaClub3 club3 = (JavaClub3) iter.next();
   log.debug("Club:  " + club3 );
}
```

The new class *DistinctRootEntityResultTransformer* provides a better approach. It will keep your sort order.

```
results = session.createQuery(
      "select c from JavaClub3 c left join c.members  order by c.name desc")
   .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 club3 = (JavaClub3) iter.next();
   log.debug("Club:  " + club3 );
}
```

11.2.7. Selecting a single column (scalar values)

Simple types of Integer, String are called scalar values. You can select them explicitly.

```
results = session.createQuery("select c.id from JavaClub3 c").list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   Integer clubId = (Integer) iter.next();
   log.debug("Club Id:  "+clubId);
}
```

11.2.8. Selecting multiple columns (scalar values)

When you select multiple scalar you will get an array of objects.

```
results = session.createQuery("select c.id, c.name from JavaClub3 c").list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   Object object[] = (Object[]) iter.next();
   log.debug("Club Id:  "+object[0]+ " name: "+object[1]);
}
```

11.2.9. Selecting objects and scalar values

You can mix scalar and objects of course. You only have to keep in mind that you get an array holding different kind of classes.

```
results = session.createQuery(
   "select c, m.name from JavaClub3 c left join c.members m").list();
```

```
for (Iterator iter = results.iterator(); iter.hasNext();) {
   Object object[] = (Object[]) iter.next();
   JavaClub3 club3 = (JavaClub3) object[0];
   String name = (String) object[1];
   log.debug("Club:  "+club3+ " Member name: "+name);
}
```

11.2.10. Selecting selective properties of a class

If you have a class with a lot of properties but need only a few of them you can create an object with selective properties. This is only useful when reading data. You can not persist such an object.

```
results = session.createQuery(
      "select new JavaClub3(c.name) from JavaClub3 c").list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 element = (JavaClub3) iter.next();
   log.debug(element.getName());
}
```

Do not forget that you need a corresponding constructor in your class.

11.2.11. Simple where condition

If your where condition will limit the result to one object you can use the method uniqueResult() instead of list().

```
aClub = (JavaClub3) session.createQuery("from JavaClub3 c where c.id=5")
   .uniqueResult();
log.debug("one single club: "+aClub);
```

You are not obliged to use alias names but it is recommended to use them.

```
JavaClub3 aClub = (JavaClub3) session.createQuery("from JavaClub3 where id=5")
   .uniqueResult();
log.debug("one single club: "+aClub);
```

11.2.12. Walking through relations

Once your application becomes more complex, you will have to walk through deeper relations. The following picture shows a room having multiple cupboards with one lock each with one key each.

images:images/c_queries_walking_relations.jpg[]

You can easily walk through one to one relations:

```
results = session.createQuery("from Cupboard c where c.lock.key.name = ?")
   .setString(0, "old key")
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   Cupboard   cupboard=  (Cupboard) iter.next();
   log.debug(cupboard);
}
```

If you want to walk through 1:n relations you cannot use something like:

```
session.createQuery("from Room r where r.cupboards.lock.key.name = ?")
   .setString(0, "old key")
   .list();
```

The cupboards attribute of room is a *java.util.List* and has no attribute lock. Instead you must use alias names for each many attribute.

```
/* walking through relations with 1:n */
results = session.createQuery(
   "select r from Room r left join r.cupboards c where c.lock.key.name = ?")
   .setString(0, "old key")
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   Room   room=  (Room) iter.next();
   log.debug(room);
}
```

11.2.13. Where condition in a related object

We want all clubs where there is a member named Peter, so the where condition is in a related object. We do not need a join, Hibernate will join implicitly.

```
List results = session.createQuery(
    "select c from JavaClub3 c left join c.members m where m.name='Peter' ")
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   JavaClub3 club3 = (JavaClub3) iter.next();
   log.debug("Club with member named Peter: "+club3);
}
```

11.2.14. Where condition with parameters

You can pass any kind of parameters using the session.set…. methods. There are special methods for all kinds of Java types. Mapped classes can be set with setEntity

```
Query query = session
   .createQuery("select c from JavaClub3 c left join c.members m " +
                  "where c.id > ? and m.name like ?");
results = query.setInteger(0,5).setString(1, "Peter%").list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   JavaClub3 javaClub3 = (JavaClub3) iter.next();
   log.debug("Some clubs: "+javaClub3);
}
```

11.2.15. Conditions on collections

If you want to select all authors having written more than 2 books, you can use the *size* condition on a collection.

```
List<Author> authors = session.createQuery(
   "from Author a where size(a.books) > 2").list();
for (Author author : authors) {
    log.info(author);
}
```

To select entities with empty collections, you may use *empty*.

```
List<Author> authors = session.createQuery(
   "from Author a where a.books is empty").list();
for (Author author : authors) {
    log.info(author);
}
```

The collections can be searched as well for a specific entity. The example selects all books belonging to the crime story category.

```
BookCategory crimeStory = (BookCategory) session.createQuery(
   "from BookCategory b where b.name = ?")
   .setString(0, "Crime story")
   .uniqueResult();
List<Book> books = session.createQuery(
    "from Book b where ? in elements(b.bookCategories)")
    .setParameter(0, crimeStory).list();
for (Book book : books) {
    log.info(book);
}
```

An alternative solution is using *member of*.

```
List<Book> books = session.createQuery(
   "from Book b where :x member of b.bookCategories")
   .setParameter("x", crimeStory)
   .list();
```

The functions *empty*, *elements* or *size* are a comfortable short cut to avoid subqueries.

11.2.16. Where condition with mapped class

We use a mapped class as parameter in this query.

```
results = session.createQuery("from JavaClubMember3 m where m.club = ?")
   .setEntity(0,club3)
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   JavaClubMember3 member = (JavaClubMember3) iter.next();
   log.debug("Member in the club: "+club3+" is: "+member);
}
```

11.2.17. Where condition with mapped class on the multiple side

When the attribute you want to compare is a collection like JavaClub3.members then you must use in elements(…).

```
results = session.createQuery("from JavaClub3 c where ? in elements (c.members)" )
   .setEntity(0,member3)
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
 JavaClub3 javaClub3 = (JavaClub3) iter.next();
   log.debug("Member: "+member3+" is in the club: "+javaClub3);
}
```

11.2.18. All, In, Some, Exists, Any elements queries

These element can be used to compare a field to values of a subselect. The following samples will explain the usage.

This query selects all pools not being in Frankfurt which size is at least bigger than one pool in Frankfurt.

```
pools = session.createQuery(
   "from SwimmingPool p where p.city <> 'Frankfurt' and p.size > "+
   "any (select p2.size from SwimmingPool p2 where p2.city like 'Frankfurt')")
   .list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   SwimmingPool pool = (SwimmingPool) iter.next();
   log.debug(pool);
}
```

Some is a synonym for Any.

This query selects all pools not being in Frankfurt which size is bigger than all pools in Frankfurt.

```
pools = session.createQuery(
   "from SwimmingPool p where p.city <> 'Frankfurt' and p.size "+
   "> all (select p2.size from SwimmingPool p2 where p2.city like 'Frankfurt')")
    .list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   SwimmingPool pool = (SwimmingPool) iter.next();
   log.debug(pool);
}
```

This query selects all pools being the same size as a pool in Frankfurt.

```
pools = session.createQuery(
   "select p from SwimmingPool p where p.city <> 'Frankfurt' and p.size in "+
   "(select p2.size from SwimmingPool p2 where p2.city like 'Frankfurt')")
    .list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   SwimmingPool pool = (SwimmingPool) iter.next();
   log.debug(pool);
}
```

Elements can be used to refer to single elements of the n side of a relation. The next query selects all pool owners having a swimming pool.

```
pools = session.createQuery(
      "select o from PoolOwner o where exists elements(o.swimmingPools)")
      .list();
```

Finally, we select all PoolOwner having a swimming pool greater than 20.

```
pools = session.createQuery(
    "select o from PoolOwner o where elements(o.swimmingPools) in "+
   "(select p from SwimmingPool p where p.size > 20)")
    .list();
```

11.2.19. NamedQuery

A named query is defined once and reused multiple times. It has a name to be identified. Named queries are useful, if you need to reuse a complex query at various places. In addition they provide a slight performance advantage.

Sample code in *mapping-examples-annotation* package *de.laliluna.other.namedquery*.

The example below shows a named SQL query using a result set mapping.

```
@NamedQueries({
   @NamedQuery(name = "bookQuery", query =
   "from ComputerBook b where b.id > :minId and b.name = :name",
    hints = {@QueryHint(name = "org.hibernate.readOnly", value = "false")})
})
@Entity
public class ComputerBook {...
```

Using a named query is pretty simple.

```
List<ComputerBook> books  = session.getNamedQuery("bookQuery").list();
for (ComputerBook report : books) {
    System.out.println(report);
}
```

org.hibernate.annotation.NamedQuery and javax.persistence.NamedQuery

Both Hibernate and Java Persistence includes a *@NamedQuery* annotation. If you use the Hibernate API (= Session) you might consider to use the Hibernate version. It is easier to configure, as it provides variables for configurations like *fetchSize*, *timeOut* or the caching behaviour *cacheMode* of the query. JPA requires to use string constants for query hints.

Hibernate named query.

```
@org.hibernate.annotations.NamedQueries{
   @org.hibernate.annotations.NamedQuery(
   name = "foo", query =
   "from ComputerBook b where b.id > :minId and b.name = :name",
   flushMode = FlushModeType.AUTO,
    cacheable = true, cacheRegion = "", fetchSize = 20, timeout = 5000,
    comment = "A comment", cacheMode = CacheModeType.NORMAL,
    readOnly = true)})
```

Java Persistence named query.

```
@javax.persistence.NamedQueries({@javax.persistence.NamedQuery(
   name = "bookQuery", query =
   "from ComputerBook b where b.id > :minId and b.name = :name",
    hints = {
        @QueryHint(name = "org.hibernate.readOnly", value = "false"),
        @QueryHint(name = "org.hibernate.timeout", value = "5000")})
    })
```

Opinion on Named Queries

I personally do not like them, as I prefer to see the HQL or SQL code right in the code where it occurs without looking around for the definition of the named query.

11.3. Criteria Queries

A criteria query can include multiple criterias. Each criteria can include multiple criterions, which can include other criterions as well. See the following figure to understand the structure.

- Criteria(a class e.g. JavaClub)
- Criterion: or condition
- Criterion: name = *xy*
- Criterion: id = 5
- Criterion: city = *Frankfurt*
- Criteria(a related class, e.g. JavaClub.members)
- Criterion: name=*Peter*

11.3.1. Simple select

```
List results = session.createCriteria(JavaClub3.class).list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 club3 = (JavaClub3) iter.next();
   log.debug(club3);
}
```

11.3.2. Select with a unique result

If you expect a unique result, you can call uniqueResult instead of list to get one object only.

```
JavaClub3 aClub = (JavaClub3) session.createCriteria(JavaClub3.class)
   .add(Restrictions.eq("id", 5))
   .uniqueResult();
log.debug("one single club: " + aClub);
```

11.3.3. Select with join returning objects multiple times

If you fetch the members of a JavaClub3 you will receive multiple entries for a JavaClub3 if it has more than one member.

```
log.debug("Selecting with fetching does not result in an array of objects");
results = session.createCriteria(JavaClub3.class)
   .setFetchMode("members", FetchMode.JOIN)
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 javaClub3 = (JavaClub3) iter.next();
   log.debug("Club:  " + javaClub3);
}
```

If you only want to initialize the members you can use a result transformer.

11.3.4. Select with join returning distinct objects

The former sample explained that a join can result into multiple lines for an object. In the HQL samples we used an approach like

```
Set set = new HashSet(getSession().createQuery(
   "select i from Invoice i inner join fetch i.orders").list());
```

The main disadvantage is that we loose any kind of ordering. Criteria queries provide an alternative: a ResultTransformer which keeps the ordering as well.

```
log.debug("Select with fetching with a result transformer");
results = session.createCriteria(JavaClub3.class)
   .addOrder(Order.desc("name"))
   .setFetchMode("members", FetchMode.JOIN)
   .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 javaClub3 = (JavaClub3) iter.next();
   log.debug("Club:  " + javaClub3);
}
```

11.3.5. Select with a where condition on a related object

Simply add a further criteria for the related object. In this case we select all clubs with a member named Peter.

```
log.debug("Selecting an object with a where condition on a related object");
results = session.createCriteria(JavaClub3.class).createCriteria("members")
      .add(Restrictions.eq("name", "Peter")).list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 javaClub3 = (JavaClub3) iter.next();
   log.debug("Club:  " + javaClub3);
}
```

11.3.6. Selecting a single column (scalar values, projections)

Simple types of Integer, String are called scalar values. You can select them explicitly.

```
results = session.createCriteria(JavaClub3.class).setProjection(
   Projections.projectionList()
      .add(Projections.property("id"))
      .add(Projections.property("name")))
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   Object object[] = (Object[]) iter.next();
   log.debug("Club Id:  " + object[0] + " name: " + object[1]);
}
```

Selecting multiple columns (scalar values, projections)

When you select multiple scalar you will get an array of objects.

```
results = session.createQuery("select c.id, c.name from JavaClub3 c")
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   Object object[] = (Object[]) iter.next();
   log.debug("Club Id:  "+object[0]+ " name: "+object[1]);
}
```

Selecting selective properties of a class

If you have a class with a lot of properties but need only a few of them you can create an object with selective properties. This is only useful when reading data. You can not persist such an object. With criteria queries you use a transformer to create the object.

```
results = session.createCriteria(JavaClub3.class)
   .setProjection(Projections.property("name"))
   .setResultTransformer(
      new AliasToBeanResultTransformer(JavaClub3.class))
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   JavaClub3 element = (JavaClub3) iter.next();
   log.debug(element.getName());
}
```

Walking through relations

Once your application becomes more complex, you will have to walk through deeper relations. The following picture shows a room having multiple cupboards with one lock each with one key each.

images:images/c_queries_walking_relations.jpg[]

You can easily walk through 1:1 and 1:n relations using criteria queries:

```
results = session.createCriteria(Cupboard.class).createCriteria("lock")
   .createCriteria("key")
   .add(Restrictions.eq("name", "old key"))
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
{
   Cupboard cupboard = (Cupboard) iter.next();
   log.debug(cupboard);
}
```

```
results = session.createCriteria(Room.class).createCriteria("cupboards")
   .createCriteria("lock").createCriteria("key")
   .add(Restrictions.eq("name", "old key"))
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();)
```

```
{
   Room room = (Room) iter.next();
   log.debug(room);
}
```

11.3.7. Simple where condition

The result set is a single object, so we return it with uniqueResult().

```
JavaClub3 aClub = (JavaClub3) session.createCriteria(JavaClub3.class)
   .add(Restrictions.eq("id", 5))
   .uniqueResult();
log.debug("one single club: " + aClub);
```

11.3.8. Where condition in relation

We need to create a second criteria for the property.

```
List results = session.createCriteria(JavaClub3.class)
   .createCriteria("members")
   .add(Restrictions.eq("name", "Peter"))
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   JavaClub3 club3 = (JavaClub3) iter.next();
   log.debug("Club with member named Peter: " + club3);
}
```

11.3.9. Where with or condition

We will select all clubs with id > than 5 or with a name \ starting with Java where there is a member named Peter.

Hint: disjunction is the same as or in HQL

```
// TODO when you use Java 1.4 or older you must replace the number 5
results = session.createCriteria(JavaClub3.class)
   .add(Restrictions.disjunction()
      .add(Restrictions.eq("name", "Java%"))
      .add(Restrictions.gt("id", 5)))
   .createCriteria("members")
      .add(Restrictions.eq("name", "Peter"))
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
   JavaClub3 club3 = (JavaClub3) iter.next();
   log.debug("Club with member named Peter: " + club3);
}
```

11.3.10. Conditions on collections

If you want to select all authors having written more than 2 books, you can use the *size* condition on a collection.

```
List<Author> authors = session.createCriteria(Author.class)
   .add(Restrictions.sizeGt("books", 2)).list();
```

```
for (Author author : authors) {
    log.info(author);
}
```

To select entities with empty collections, you may use *empty*.

```
List<Author> authors = session.createCriteria(Author.class)
   .add(Restrictions.isEmpty("books")).list();
```

11.3.11. All, In, Some, Any elements queries

These element can be used to compare a field to values of a subselect. The following samples will explain the usage.

This query selects all pools not being in Frankfurt which size is at least bigger than one pool in Frankfurt. Some is a synonym for Any.

```
DetachedCriteria detachedCriteria = DetachedCriteria.forClass(
      SwimmingPool.class).setProjection(Projections.property("size"))
   .add(Restrictions.eq("city", "Frankfurt"));
pools = session.createCriteria(SwimmingPool.class)
   .add(Restrictions.ne("city", "Frankfurt"))
   .add(Property.forName("size").gtSome(detachedCriteria))
   .list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   SwimmingPool pool = (SwimmingPool) iter.next();
   log.debug(pool);
}
```

This query selects all pools not being in Frankfurt which size is bigger than all pools in Frankfurt.

```
DetachedCriteria detachedCriteria = DetachedCriteria.forClass(
      SwimmingPool.class).setProjection(Projections.property("size")).add(
      Restrictions.eq("city", "Frankfurt"));
pools = session.createCriteria(SwimmingPool.class).add(
      Restrictions.ne("city", "Frankfurt")).add(
      Property.forName("size").gtAll(detachedCriteria)).list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   SwimmingPool pool = (SwimmingPool) iter.next();
   log.debug(pool);
}
```

This query selects all pools being the same size as a pool in Frankfurt.

```
detachedCriteria = DetachedCriteria.forClass(SwimmingPool.class)
   .setProjection(Projections.property("size"))
      .add(Restrictions.eq("city", "Frankfurt"));
pools = session.createCriteria(SwimmingPool.class)
   .add(Restrictions.ne("city", "Frankfurt"))
   .add(Property.forName("size").in(detachedCriteria))
   .list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   SwimmingPool pool = (SwimmingPool) iter.next();
   log.debug(pool);
}
```

Finally, we select all PoolOwner having a swimming pool outside of Frankfurt which size is not greater than 20.

```
detachedCriteria = DetachedCriteria.forClass(SwimmingPool.class)
   .add(Restrictions.gt("size", 20))
   .add(Restrictions.ne("city", "Frankfurt"))
   .setProjection(Projections.property("id"));
pools = session.createCriteria(PoolOwner.class).createAlias("swimmingPools", "s")
   .add(Subqueries.propertyIn("s.id", detachedCriteria))
   .list();
for (Iterator iter = pools.iterator(); iter.hasNext();)
{
   PoolOwner pool = (PoolOwner) iter.next();
   log.debug(pool);
}
```

Important: Frequently, it is better to use a projection in subqueries to select an id, instead of an object like SwimmingPools. I found that working on objects does not work as expected in some situations.

11.4. Native SQL

I recommend not to use native SQL but still there are some border cases requiring direct access to SQL. Hibernate even provides help to convert SQL result sets into entities.

By default SQL result set rows are returned as object array. Here is a simple example query.

```
List results = session.createSQLQuery(
    "select c.id, c.name from  tjavaclub c left join tjavaclubmember m "+
   "on c.id=m.club_id where m.name='Peter' ")
   .list();
for (Iterator iter = results.iterator(); iter.hasNext();) {
     Object[] objects = (Object[]) iter.next();
     Integer id = (Integer) objects[0];
     String name = (String) objects[1];
     log.debug("Club Id: " + id + " Name: " + name);
  }
```

Update or insert queries are not possible. The following leads to an exception.

```
session.createSQLQuery(
      "update tjavaclub  set name='new name' where id =5")
      .executeUpdate();
```

```
Exception in thread "main" java.lang.UnsupportedOperationException:
Update queries only supported through HQL
   at org.hibernate.impl.AbstractQueryImpl.executeUpdate(
      AbstractQueryImpl.java:753)
   at test.de.laliluna.example1.TestQuery.nativeSql(TestQuery.java:69)
```

11.4.1. SQL to Entity

Hibernate helps you to transform SQL results into entity classes.

Sample code in *mapping-examples-annotation* package *de.laliluna.other.namedquery*.

Assuming that SQL column names and class properties are the same, you can write the following code to get a list of computer books.

```
List<ComputerBook> reports = session.createSQLQuery
   ("select * from computerbook")
    .addEntity(ComputerBook.class)
   .list();
```

The column names of the SQL result must match the expected columns. We will slightly modify the example. The book has two attributes. The name of the book is mapped to *book_name*

```
@Entity
public class ComputerBook {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Column(name = "book_name")
    private String name;
// ...
```

Therefor the SQL must return at least two columns named *id* and *book_name*. The following SQL query will fail with a *column book_name not found exception*.

```
List<ComputerBook> reports = session.createSQLQuery
   ("select id, name from computerbook")
   .addEntity(ComputerBook.class)
   .list();
```

To fix the query you can make use of alias in your SQL query or use SQL result set mapping. The latter is explained in the next chapter.

Using an alias.

```
List<ComputerBook> reports = session.createSQLQuery
   ("select id, name as book_name from computerbook")
   .addEntity(ComputerBook.class)
   .list();
```

Modifying entities

Your entities are real entities. Their state is persistent after the query, you can modify the entities and they will be saved in the database with the next transaction commit.

```
Session session = InitSessionFactory.getInstance().getCurrentSession();
session.beginTransaction();
Lab aLab = (Lab) session.createSQLQuery("select l.* from lab l limit 1")
    .addEntity(Lab.class)
    .uniqueResult();
aLab.setTitle("My first Hibernate Lab");
session.getTransaction().commit();
```

SQL debug output.

```
DEBUG SQL:111 - /* update de.laliluna.other.namedquery.Lab */
   update Lab set title=? where id=?
```

Multiple entities

If you want to select multiple entities or join a collection, there is a good chance to have naming conflicts of the result set columns (multiple columns with the same name). Hibernate cannot find out which column to use to create an entity.

There for there is a special syntax *{alias.*}* or *{alias.propertyName}*. If this syntax is used, Hibernate will modify the SQL to create distinct alias names.

The next query will load an instance of lab and the related book.

```
List<Object[]> result = session.createSQLQuery(
    "select {c.*}, {l.*} from computerbook c join lab l on l.computerbook_fk=c.id")
    .addEntity("c", ComputerBook.class)
    .addEntity("l", Lab.class)
    .list();

for (Object report[] : result) {
    ComputerBook book = (ComputerBook) report[0];
    Lab lab = (Lab) report[1];
    System.out.println(book+ "-"+lab);
}
```

The "c" in *addEntity(..)* corresponds to the *{c.*}* alias. It tells Hibernate to fill the entity *Computerbook* with the values of *{c.*}*

Below you can see that the SQL query has been modified by Hibernate before being executed.

```
select c.id as id73_0_, c.book_name as book2_73_0_, l.id as id74_1_,
   l.title as title74_1_
   from computerbook c join lab l on l.computerbook_fk=c.id
```

SQL results with Java Persistence

The *{alias.*}* notation is a Hibernate extension. If you want to stick with the JPA standard and prefer a more verbose approach have a look in the SQL result set mapping chapter below.

Instead of using wildcards, you can be explicit as well with the alias names.

```
List<Object[]> result = session.createSQLQuery(
    "select c.id as {c.id}, c.book_name as {c.name}, " +
        "l.id as {l.id}, l.title as {l.title} from computerbook c " +
        "join lab l on l.computerbook_fk=c.id")
    .addEntity("c", ComputerBook.class)
    .addEntity("l", Lab.class)
    .list();

for (Object report[] : result) {
    ComputerBook book = (ComputerBook) report[0];
    Lab lab = (Lab) report[1];
    System.out.println(book+ "-"+lab);
}
```

Initializing collections

The class *Computerbook* has a property *Set<Lab> labs*. Using the *{..}* syntax we can eager load the collection in the same query. The next query will load a computerbook and will initialize the collection *labs* of the book in one query.

```
List<Object[]> result = session.createSQLQuery(
    "select {c.*}, {l.*} from computerbook c join lab l on l.computerbook_fk=c.id")
    .addEntity("c", ComputerBook.class)
   .addJoin("l", "c.labs")
   .list();

for (Object report[] : result) {
    ComputerBook book = (ComputerBook) report[0];
    Lab lab = (Lab) report[1];
    System.out.println(book+ "-"+lab);
}
```

The method *addJoin* defines that the collection *labs* of the computer book is initialized and filled with the values of *{l.*}* .

11.4.2. SQL Resultset Mapping

SQL result set mapping is an alternative to map JDBC columns and class properties. As JPA does not support the *{alias.*}* notation, result set mapping is at the same time a more portable way.

Sample code in *mapping-examples-annotation* package *de.laliluna.other.namedquery*.

The following annotation defines a resultset mapping with the name *bookReport2*.

SQL result set mapping.

```
@SqlResultSetMapping(name = "bookReport",  entities = {
   @EntityResult(entityClass = ComputerBook.class,
    fields = {
      @FieldResult(name = "id", column = "id"),
        @FieldResult(name = "name", column = "book_name")
   })
})

@Entity
public class ComputerBook { ...
```

Now, we can use the result set mapping in a SQL query.

Using an SQL result set mapping.

```
List<ComputerBook> books = session.createSQLQuery
    ("select id, book_name from computerbook")
    .setResultSetMapping("bookReport")
   .list();
for (ComputerBook report : books) {
    System.out.println(report);
}
```

A result set mapping may contain as well scalar values (ie. single columns).

The next example returns a list of object arrays. The first element of the array is an *ComputerBook* and the second the number of books with the same name.

```
@SqlResultSetMappings({
    @SqlResultSetMapping(name = "bookReport2", entities = {@EntityResult
      (entityClass = ComputerBook.class,
      fields = {
         @FieldResult(name = "id", column = "id"),
         @FieldResult(name = "name", column = "name")})},
      columns = {@ColumnResult(name = "count_group")})
})
```

Using the result set mapping.

```
List<Object[]> list = session.createSQLQuery(
    "select b.id, b.book_name, (select count(1) as count_group " +
    "from computerbook where book_name = b.book_name) as count_group " +
    "from computerbook b")
    .setResultSetMapping("bookReport2")
    .list();
for (Object o[] : list) {
    ComputerBook computerBook = (ComputerBook) o[0];
    long total = ((BigInteger) o[1]).longValue();
    log.debug(computerBook + "-- Books with same name: " + total);
}
```

11.4.3. Named SQL Queries

A named native query is defined once and reused multiple times. It has a name to be identified. Named queries are useful, if you need to reuse a complex query at various places. In addition they provide a slight performance advantage.

Sample code in *mapping-examples-annotation* package *de.laliluna.other.namedquery*.

The example below shows a named SQL query using a result set mapping.

```
@NamedNativeQueries({
    @NamedNativeQuery(name = "reportSql",
        query = "select b.id, b.book_name, (select count(1) as count_group " +
            "from " +
            "computerbook where book_name = b.book_name) as count_group " +
            "from computerbook b", resultSetMapping = "bookReport2")})
@Entity
public class ComputerBook {...
```

Using a named query is pretty simple.

```
List<Object[]> list = session.getNamedQuery("reportSql").list();
for (Object o[] : list) {
    ComputerBook computerBook = (ComputerBook) o[0];
    long total = ((BigInteger) o[1]).longValue();
    log.debug(computerBook + "-- Books with same name: " + total);
}
```

You might have noticed that *@NamedNativeQuery* and *@NamedQuery* can both be executed by calling the *getNamedQuery* method. This can help if you need to migrate an existing application with a lot of SQL queries. First, you transform the SQL in named queries and later on you change the code to HQL.

11.4.4. JDBC Connection

Sometimes you need plain access to the JDBC connection. There are two approaches.

The first provides access to the currently used connection and in addition provides exception handling. It will close the connection if an exception happens. I would consider it the preferred approach. The *session.doWork(..)* method will inject the current session.

session.doWork.

```
session.doWork(new Work(){
    @Override
    public void execute(Connection connection) throws SQLException {
        ResultSet resultSet = connection.prepareCall(
         "select * from hello_world").getResultSet();
    }
});
```

Alternatively, you can get a new connection from the connection provider. This connection will not have any relation to Hibernate.

ConnectionProvider.

```
ConnectionProvider connectionProvider = ((SessionFactoryImplementor) sf)
   .getConnectionProvider();

Connection connection = null;
try {
    connectionProvider.getConnection();
} catch (SQLException e) {
    e.printStackTrace();

}
finally {
    if(connection!=null)
        try {
            connection.close();
        } catch (SQLException e) {
            // ignore or warn (as you like)
        }
}
```

Deprecated connection method

session.connection() is now deprecated.

Part III. Building applications and Architecture

Chapter 12. Data Access Objects

12.1. Best practices and DAO

This chapter will explain two approaches you might use as a structure for your Hibernate application. Both are based on DAO (Data Access Objects). First, I will explain the DAO as defined in the Java blueprints. Then we will discover the general requirements for a Hibernate application.

Finally, we will continue with the examples showing implementations for Java 1.5 with generics and an implementation for older Java versions. I will try to explain the ideas behind each approach to make you better understand what you are doing.

12.2. Data Access Objects DAO

Applications need to access any kind of persistent storage. As you are learning Hibernate you probably intend to use a database as persistent storage.

But persistent storage is not limited to databases. Any kind of LDAP directories holding user and user roles, external messaging systems etc. are persistent storages.

A DAO class encapsulates data sources and the used query language and provides a well defined set of methods for your application.

Imagine a userDao class providing an interface as shown in the next figure. When your application only deals with the interface userDao when accessing user information, you can easily replace it with an implementation using a different database or a LDAP directory. This is the main intention of the Dao pattern.

```
public interface UserDao {
   public User findById(Integer id);
   public List findAll();
   public void save(User user);
   public void delete(User user);
}
```

get a PostgreSQL user Dao

```
UserDao userDao = new UserDaoPostgreSQL();
```

or a LDAP userDao

```
UserDao userDao = new UserDaoLdap();
```

Hibernate does slightly change the look of this pattern. If you use plain JDBC you would create a Dao for each different database you want support. When using Hibernate this is no longer needed. Hibernate supports multiple database dialects just by changing the configuration.

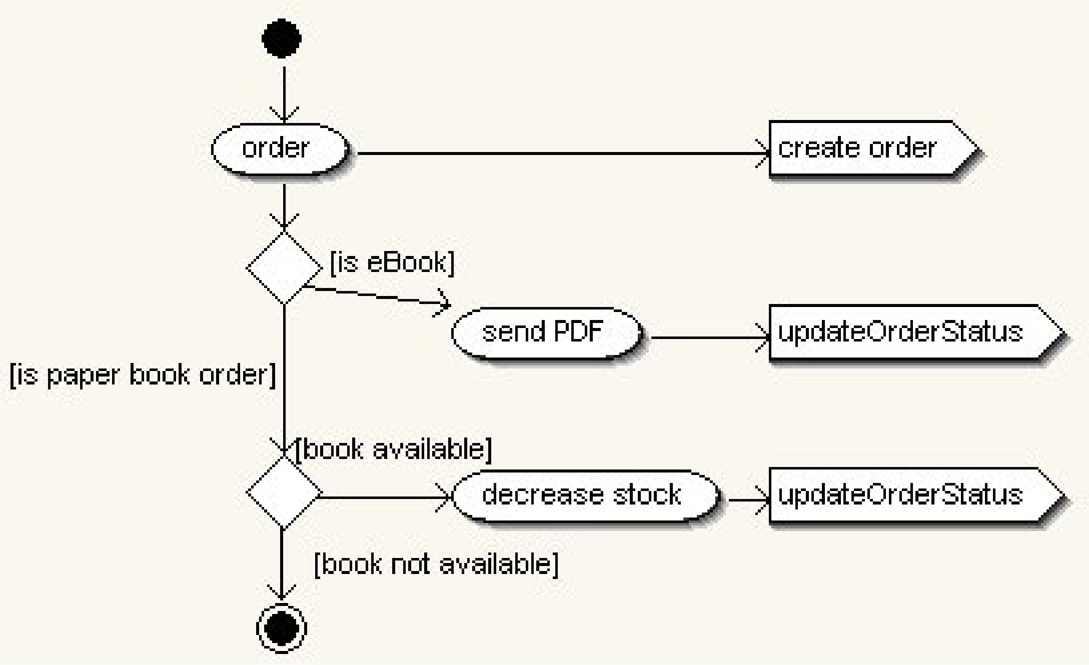
Therefore, the content of a Hibernate DAO can be quite simple.

```
.... snip ....
   public User findById(Integer id) {
      return (User) getSession().get(User.class, id);
   }

   public List findAll() {
      return getSession().createQuery("from User").list();
   }

   public void save(User user) {
         session.save(user);
   }
.... snip ....
```

12.3. Weaving the application structure

In order to illustrate the process of designing we will take a use case as underlying requirement.

We have two kinds of products: paper books and eBooks. Paper books are taken from stock, when they are ordered by the customer. Ebooks are PDF documents, which can be provided immediately to the customer. There is not stock available.



12.3.1. Paper book order

A customer orders a book. The system checks the stock. If the book is available it is delivered and the stock is reduced. If it is not available, the order remains undelivered. A use case which issues a command to a supplier will not be treated here.

The question is how to handle exceptions happening during the access of Hibernate and where the transaction demarcations can be found.

I would propose the following transaction demarcations:

The first transaction is around the initial creation of an order. The later change of the order status should have no influence on the order creation.

The process of checking the stock level, the reduction of the stock and the update should happen in one transaction.

To protect the application against parallel access we should consider to lock the stock level of the product. If we do not do this, we might have a situation with two clients doing the following:

- client A checks stock level
- client B checks stock level
- client A decreases stock and updates order
- client B decreases stock and updates order

The exceptions should not be shown to the customer but handled in a proper way.

12.3.2. eBook order

A customer orders a book. The order is created and the applications sends the eBook as PDF to the customer. The order status is set to delivered. We will keep things simple and allow only the order of one book which is a paper or an eBook. Where are the transaction demarcations and how will we treat exceptions?

I propose the following transaction demarcations:

The first transaction is around the initial creation of an order. The later change of the order status should have no influence on the order creation.

The update of the order status should only happen when the PDF has been successfully sent. This is our second transaction.

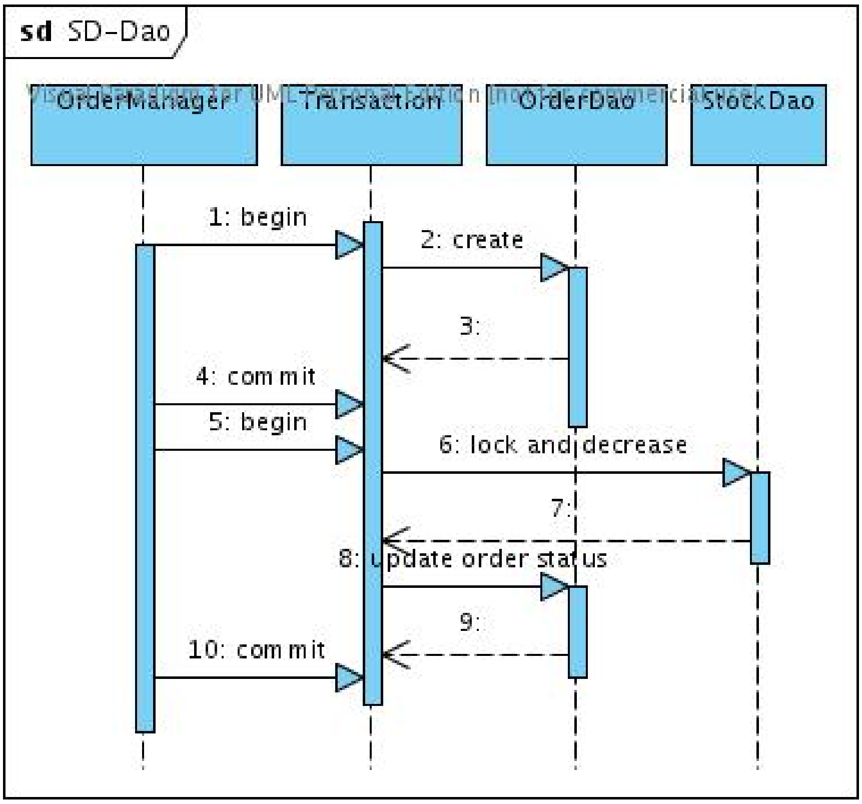
The exceptions should not be shown to the customer but handled in a proper way.

12.3.3. Defining the application layers

We have defined our use cases. The next step is to set up a proper structure for our application.

Choosing a suitable structure is a question of personal preferences. I propose the following approach:

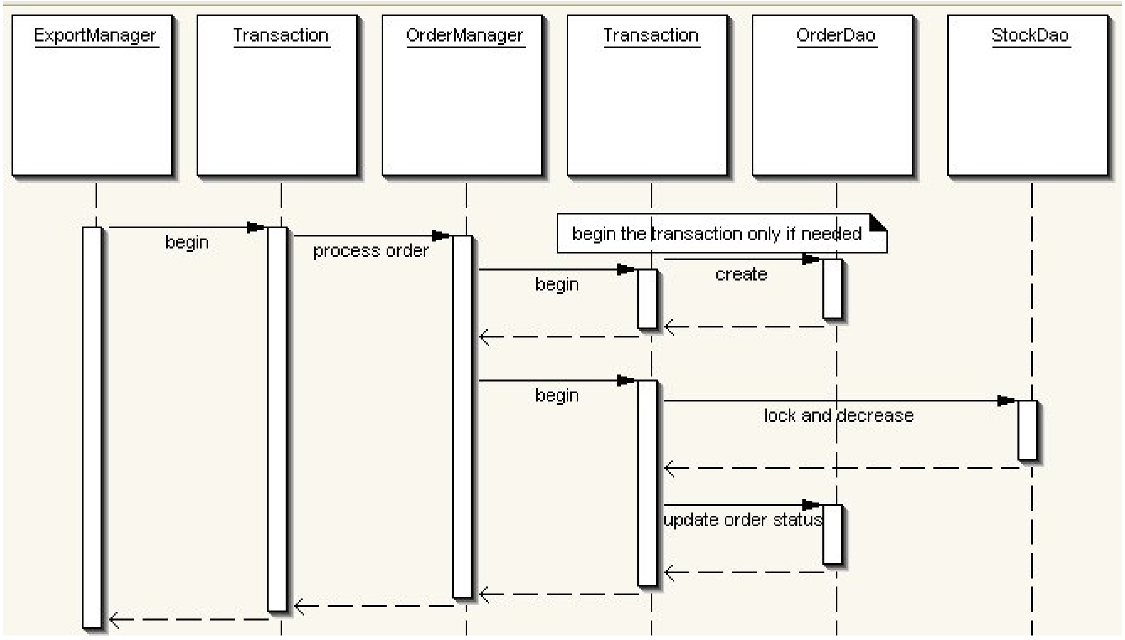
We will set up a business logic layer containing the use cases. (OrderManger in the figure below) This layer is responsible for the transaction management as well. This layer uses DAOs to access the data. A DAO does not manage any transactions at all. It receives a Hibernate session and just updates or returns data.



The Problem of reusing business logic

There is one situation we must think about. The class OrderManager will have a method called processOrder which will manage all the transactions. What will happen if we want to use this method in a larger context with its own transaction? For example another class called ExportManager verifies if we need an export permission. This class starts a transaction itself, requests the permission and continues to processOrder of our class OrderManager. In this case the OrderManager should not create a transaction of its own but should run in the transaction of our ExportManager.

Once approach to solve this problem is to improve our processOrder method. We can make it check for an existing transaction context. If one exist the method does not handle the transaction, else it does.



```
public void processOrder(Order order) {
   // check if this method is responsibe for the transaction
   boolean handleTransaction = true;
   if (getSession().getTransaction().isActive())
      handleTransaction = false;
   if (handleTransaction)
      getSession().beginTransaction();

   // do something important here

   if (handleTransaction)
      getSession().getTransaction().commit();
}
```

Approach for reuse of business logic

Our Business methods should be clever enough to check if they run within a transaction. If this is allowed, they should not open a new transaction. If it is not allowed they should throw an exception.

Those using EJB 2 or container managed transaction (CMT) will know the different transaction types we are simulating here.

CMT supports a wider range of transactions like

Requires-new: to have a new transaction for this method, open transactions are paused.

Requires: will run in an existing transaction if exist. If there is no transaction a new one will be started.

But this is not our topic here. I just wanted to mention it.

Reattaching problems

A use case frequently reattaches objects and updates them. Most reattachment strategies do not allow that an object is already attached. If you allow that use cases can call other use cases you must be careful that you do not try to reattach an object which is already in the session.

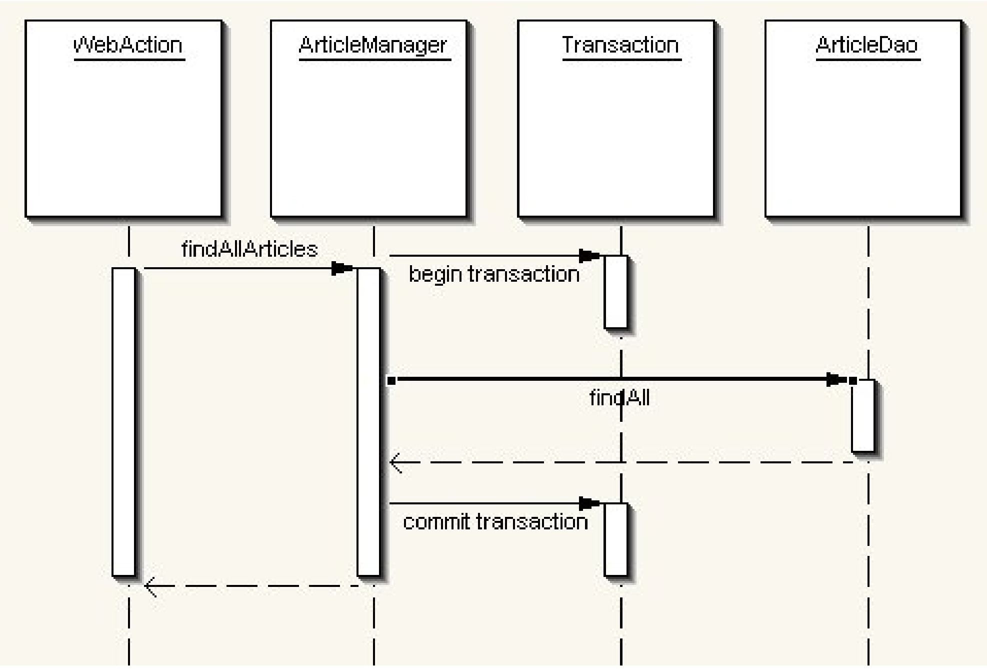
Simple use cases

Let’s think about simple use cases that happen very often in applications:

- Displaying a list of articles
- Displaying one article

The methods are already defined in the *ArticleDao*. The ArticleManager would begin the transaction, call the Dao and close the transaction. We would need a method in the business class only adding the transaction.

Alternatively we could add transaction handling to the DAO methods. When it is not called within an transaction, the DAO saves the information that is responsible for the transaction and starts and commits the transaction. In this case the DAO must be clever enough to start a new transaction itself. The following method could be used in a DAO. It ensures that a transaction is used.



```
   private boolean transactionHandled;

protected void ensureTransaction() {
      if (!transactionHandled) {
         if (getSession() == null
               || !getSession().getTransaction().isActive()) {
            transactionHandled = true;
            // get a current or new session, we are responsible so we must
            // be aware that the session could not exist.
            factory.getCurrentSession().beginTransaction();
         }
      } else {
         transactionHandled = false;
         try {
            getSession().getTransaction().commit();
            getSession().close(); // only useful when auto close in hibernate.cfg.xml is
         } catch (HibernateException e) {
            // do not print stacktrace as we will rethow the exception
            // e.printStackTrace();
            // clean up the session and transaction when an exception
            // happens
            try {
               getSession().getTransaction().rollback();
               getSession().close();// only useful when auto close in hibernate.cfg.xml
            } catch (HibernateException e1) {

            }
            if (e instanceof StaleObjectStateException) // normaly a
               // version
               // problem
               throw e;

            else
               throw new HibernateException(e);
         }
      }
   }
```

When we use version tracking a StaleObjectException can happen. The method above will rethrow the exception to allow that our web action can handle this information.

There are also reasons not to use this kind of “clever” DAO. First, the structure is not very consistent. Your DAOs become some kind of business objects. You will need to handle Hibernate exceptions directly in your web layer.

I wanted to show you both approaches. I think that not using “clever” DAOs is by far a cleaner approach but using them leads to less redundant work.

12.3.4. DAO and DaoFactory

A DAO should not have any code configuring a specific session, but it should get the session or a session factory from the outside. The reason is that you reduce relations and dependencies in your code and get an application that is easier to maintain or debug. You could even think of a situation where you have multiple databases and a DAO should receive the session of a specific database.

As a consequence you will need a DAOFactory to create the DAO classes. This factory initialises the DAO object with a session or a session factory.

One issue I do differently compared to Spring/Hibernate’s examples or the CaveatEmptor application from Hibernate in Action, is that I do not set a session in a DAO but only a sessionFactory. The reason becomes clear when you look at the following case.

```
OrderDao orderDao = DaoFactory.getOrderDao();
      deliverEbooks(order);
      deliverPaperBook(order);
      getSession().beginTransaction();
// here is the problem
      orderDao.reattach(order);
```

If we initialise the orderDao with a session there is a fair chance that a deliverEbooks method is closing the current session. We started a new session with the call to getSession() but the orderDao still has the old abandoned session. This is why I prefer to save only a session factory in the DAO that returns always the session currently open.

12.3.5. Creating DAOs with generics

As one example you may use the project *DaoExample*.

Take a look at the source code now and research the following things

Dao Creation

The class *OrderDao* extends the class *BasicDaoImp*. Common methods like save, update, lock, delete are implemented in the *BasicDaoImp*. *OrderDao* just adds some special methods needed to access an order. The *orderDao* is created by the *DaoFactory*.

```
...
public T findById(Integer id) {
    return (T) getSession().get(type, id);
}

public void reattach(T entity) {
    getSession().buildLockRequest(LockOptions.READ).lock(entity);
}

public void save(T entity) {
    getSession().saveOrUpdate(entity);
}

public void update(T entity) {
    getSession().update(entity);

}


public class ArticleDaoImp extends BasicDaoImp implements ArticleDao {


    public ArticleDaoImp(SessionFactory factory) {
        super(factory, Article.class);
    }

    public boolean lockAndDecrease(Article article, int quantity) {
        getSession().buildLockRequest(LockOptions.UPGRADE).lock(article);
        return article.decreaseStock(quantity);
    }

}
```

Chapter 13. Session and Transaction Handling

Before we discuss advantages and disadvantages in detail, I would state a general recommendation.

I recommend to use optimistic locking in combination with a short running session for most web applications. If you want to understand why then continue to read this chapter.

13.1. Hibernate Session

A session provides all methods to access the database and is at the same time a cache holding all data being used in a transaction. If you load objects using a query or if you save an object or just attach it to the session, than the object will be added to the Persistence Context of the session.

Persistence Context

Objects are continuously added to the persistence context of the session until you close the session. If you call session.get and the object already exists in the persistence context, then the object will not be retrieved from the database but from the context. Therefore, the session is called a first level cache.

There is one caveat with this behaviour. If you query a lot of objects, you might run out of memory if your session became to large. We will handle strategies for this caveat in the chapter performance.

All database access including read and write access should happen inside a transaction.

A session is not thread safe and should only be used by one thread. Usually this is guaranteed by the SessionFactory.

To conclude: A session is used by one user in his thread to write and read data inside of transactions.

Typical Session usage.

```
Session session = HibernateSessionFactoryUtil.openSession();
session.beginTransaction();
// do something
session.save(myObject);
session.getTransaction().commit();
session.beginTransaction();
// do something else
List result = session.createQuery(
   "from Order o where o.invoice.number like '2%' ").list();
session.getTransaction().commit();
session.close();
```

Write behind behaviour

The session is not sending the SQL statements directly to the database, if you call session.save. There are a number of reasons for this behaviour:

- keep write locks in the database as short as possible

- more efficient sending of SQL statements
- allow optimisations like JDBC batching

By the default configuration the SQL statements are flushed when

- tx.commit is called
- a query is executed for an Entity which has a insert/update/delete statement in the queue
- session.flush is called explicitly

This is the reason why you might see no SQL statements when calling session.save.

There is an exception to this rule as well. If you use select or identity as id generator, it is required that Hibernate sends the insert immediately.

13.2. JTA versus JDBC Transactions

Starting from Hibernate 3 the implementation of the session factory has changed. The creation of a session factory is much easier and the factories you will find through out this book are quite different from the ones you can find in older examples.

Concerning the transaction handling, there are different options for session factories.

One option is to use the transaction management of your application server. Every JEE application server provides the Java Transaction API (JTA), for example Orion, Jonas, Bea Weblogic, Websphere, Jboss. JTA transactions are more flexible as JDBC transaction handling. You can enlist other transactional resources into the same transaction or run a transaction across multiple databases. This requires that the transactional resources supports two-phase-commit transactions.

A pure Servlet application server like Tomcat, Jetty does not include a JTA transaction manager. You may plug in a JTA solution (e.g. http://jotm.objectweb.org/ or JTA of Jboss) or use the other option JDBC transactions.

JDBC transactions can be used in servlet engines or in all other kind of Java applications. Christian Bauer and Gavin King argue in Java Persistence in Action that JTA transactions provide a higher quality transaction management. I do not agree with this opinion. JTA sooner or later has to call a JDBC transaction, this is why it cannot have a *higher* quality. If you run a application against a single database and don’t need distributed transactions or transaction monitoring – which might be provided by a JEE application server as well – then it is fine to use JDBC transactions.

13.3. Transaction handling – default pattern

I have explained before that if you call the Hibernate session, you have to handle exceptions. You don’t have to code the exception handling everywhere. The Struts Integration in chapter Hibernate and Struts Section 14.2, “Hibernate and Struts” example shows how to do this in a single place. In the following, I will show you all of the required steps.

Default pattern for transaction/exception handling.

```
SessionFactory factory = InitSessionFactory.getInstance();
Session session = factory.openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   tx.commit();
   } catch (RuntimeException e) {
      try {
         if (tx != null)
            tx.rollback();
      } catch (HibernateException e1) {
      log.error("Transaction roleback not succesful");
   }
   throw e;
} finally {
   session.close();
}
```

We open a session, then we start a transaction calling *session.beginTransaction*. The transaction is automatically bound to the session. If we get no exception then we call *tx.commit* to commit the transaction. Before Hibernate sends the commit to the database it will call flush internally to synchronize the persistence context with the database. All open insert/delete/update statements will be sent to the database.

If we get an exception, the Hibernate session becomes inconsistent. We must close it. In the code above, this is ensured with the finally block.

In order to prevent a transaction leak in the database we must rollback the transaction. This is done calling *tx.rollBack*. Imagine your database is unavailable. The rollback may fail as well, this is the reason why I wrapped it into another try-catch block. After this we throw the first exception again.

We can simplify the pattern but I wanted to explain all requirements we have to implement first.

Your responsibilities when using Hibernate

- Wrap access to Hibernate in a transaction

- Roll back the transaction in case of an exception

- close the Hibernate after an exception, it is in an inconsistent state

13.4. JDBC transactions with ThreadLocal

You can find the full source code in the example project DeploymentJdbc.

This configuration was used in most of the sample projects.

If we want to use a JDBC transaction factory, we don’t have to configure anything. It is the default behaviour. If you like you can define it explicitly in the Hibernate configuration file hibernate.cfg.xml

```
<property name="transaction.factory_class">
 org.hibernate.transaction.JDBCTransactionFactory
</property>
```

We can define a session context. This is a place where Hibernate will store the session. One option is a ThreadLocal session context. It uses a Java ThreadLocal which guaranties that every thread can only see its own session.

```
    <!--  thread is the short name for
      org.hibernate.context.ThreadLocalSessionContext
      and let Hibernate bind the session automatically to the thread
    -->
    <property name="current_session_context_class">thread</property>
```

In addition, the thread context of Hibernate will close the session automatically, if we call commit or rollback. In contrast to the default pattern, we need to call *sessionFactory.getCurrentSession* to pick a session from the context. If there is no session in the context, Hibernate will create one for us.

The Hibernate pattern is slightly shorter as we can omit the closing of the session:

```
SessionFactory factory = InitSessionFactory.getInstance();
Session session = factory.getCurrentSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
// alternativly:  factory.getCurrentSession().beginTransaction();
   // do some work
   tx.commit();
   } catch (RuntimeException e) {
      try {
         if (tx != null)
            tx.rollback();
      } catch (HibernateException e1) {
      log.error("Transaction roleback not succesful");
   }
   throw e;
}
```

Please be aware, that if you use factory.openSession, the session is not placed in the ThreadLocal context.

Hibernate in former times

In the old times you had to implement the ThreadLocal session context on your own. This is no longer required, at least if you are happy with the default contexts available.

But you will still find a lot of old examples in the web.

13.5. JTA transaction with a single database

You can find sample source code in the project DeploymentJBossJTA.

In order to use JTA you need to configure a transaction manager. This is specific for every application server. For the JBoss application server, you have to add the following properties to the *hibernate.cfg.xml*.

```
<property name="hibernate.transaction.factory_class">
  org.hibernate.transaction.JTATransactionFactory
</property>
<property name="hibernate.transaction.manager_lookup_class">
  org.hibernate.transaction.JBossTransactionManagerLookup
</property>
<!--  jta is the short name for org.hibernate.context.JTASessionContext
and let Hibernate bind the session automatically to the JTA transaction.
This implies that Hibernate will close the session after the transaction.
<property name="hibernate.current_session_context_class">jta</property>
```

Add the following property, if you want Hibernate to close your session automatically, when you call commit or rollback.

```
<property name="transaction.auto_close_session">true</property>
```

Let’s have a look at the Hibernate pattern we get:

```
      Transaction tx = null;
      try {
         Session session = InitSessionFactory.getInstance().openSession();
         tx = session.beginTransaction();
         Invoice invoice = new Invoice();
//  This line is just to show that you can use getCurrentSession from now on
         InitSessionFactory.getInstance().getCurrentSession().save(invoice);
         tx.commit();
      } catch (RuntimeException e) {
         if (tx != null)
         {
            try {
               tx.rollback();
            } catch (HibernateException e1) {
               log.error("Error rolling back");
            }
         }
         throw e;
      }
```

There is a subtle difference. We call session.openSession instead of getCurrentSession. In a application server Hibernate needs to bind the session to a transaction before it is stored in a session context. The transaction is not available before having called session.beginTransaction. Therefore Hibernate places the session in the context after beginTransaction was called. Inside of the transaction you can use getCurrentSession.

Connection and JTA

If you use JTA you must use a datasource provided by your application server. This is normally a JNDI datasource. You can find a JNDI datasource in the sample project of this chapter.

Another option is to bind the factory to JNDI. You just have to specify the following in your hibernate.cfg.xml. This is useful to share a sessionFactory.

```
<property name="hibernate.session_factory_name">HibernateTest1</property>
```

13.6. JDBC or JTA with the Spring framework

The Spring framework like any other framework offering aspect oriented programming and inversion of control, let you externalize the transaction handling. Inside your code you will just have an annotation.

```
@Transactional
public void hello(){
...
}
```

This is not a special feature of Spring framework. Lighter and easier to learn frameworks like Google Guice or the Nano Container (http://nanocontainer.codehaus.org/) offer the same functionality considering transaction handling. You can find an example for Spring integrating in chapter Hibernate and Spring Section 14.1, “Hibernate and Spring”.

13.7. Conversations and Session Lifetime

A conversation consists of one or multiple dialogs. Think about a dialog to edit a product which let you go through a couple of JSP pages.

In the former examples we always have closed the session directly after a commit or rollback of the transaction. Most of the time Hibernate has done this for us automatically as we have configured a ThreadLocal session context.

```
<property name="current_session_context_class">thread</property>
```

But is this really the best approach or do we have alternative options. We could keep the session open all the time, but how to deal with the transaction in that case. The user might expect that either all or none of the changes to the product is saved. We might hold the transaction open all the time or not save at all.

In chapter Lazy Initialization Lazy initialization we have seen the problem of lazy initialization. We named the option to extend the lifetime of a session from the beginning of the request until the JSP was rendered. This allows to fetch data during the rendering of the JSP. This approach is called Open-Session-in-View. This is another option as well.

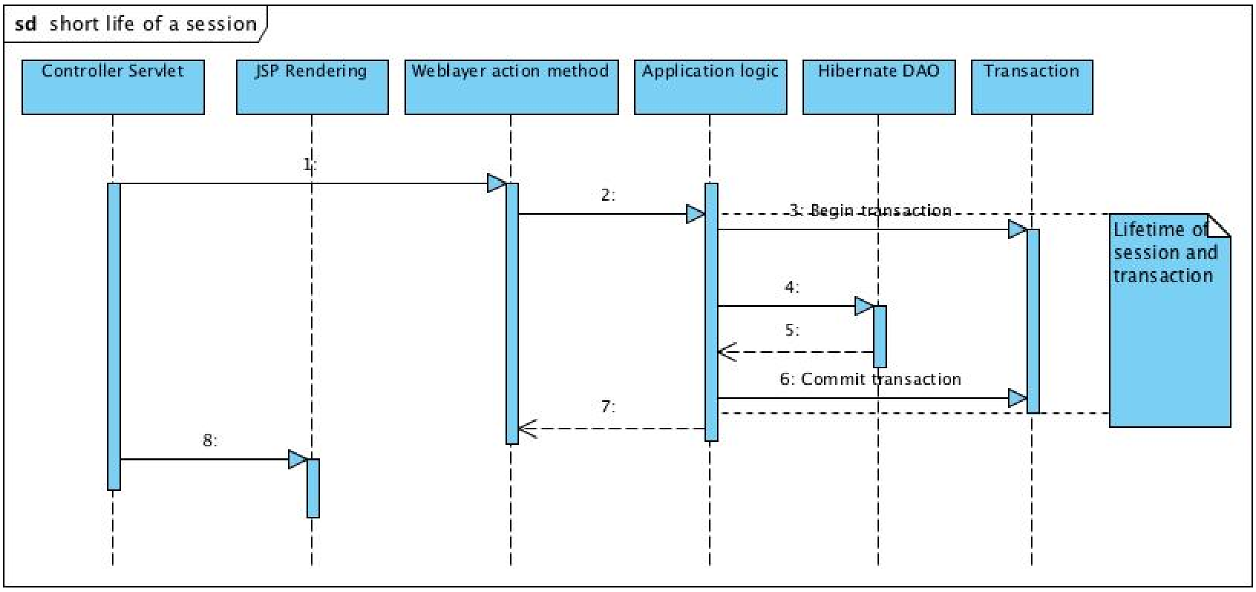
Let’s have a look at the different approaches available.

13.7.1. Short life of a session

For every request we create a new and clean session inside of the application logic. It is closed at the end of the application logic. The advantage is that it is unlikely to get the NonUniqueObject exception, we talked about in chapter 1.6 Working with Objects.

We might get a *LazyInitialization* exception during rendering. This happens if we have not initialized collections, we want to print out during the rendering phase. There is an advantage of the exception as well: we see immediately what we need to initialize and can do this efficiently.

Another advantage is a very clean application structure. It is simple to handle exceptions from our application logic and redirect the user to an error page.



A consequence of this approach is that the objects change their state to detached between each request/dialog and they need to be reattached using session.update, session.merge or session.lock.

Let’s have a look at sample code.

Dialog 1

We enter a dialog to edit a turtle.

```
   Session session = factory.getCurrentSession();
   session.beginTransaction();
   Turtle turtle = (Turtle) session.get(Turtle.class, "4711");
   session.getTransaction().commit();
```

Dialog 2

The user has selected a name for the turtle and clicks on the save button.

```
      session = factory.getCurrentSession();
      session.beginTransaction();
      turtle.setName("Alexander");
      session.update(turtle);
      session.getTransaction().commit();
```

Dialog 3

On the next screen the user can input the size of the turtle. He inputs the size and presses the save button again.

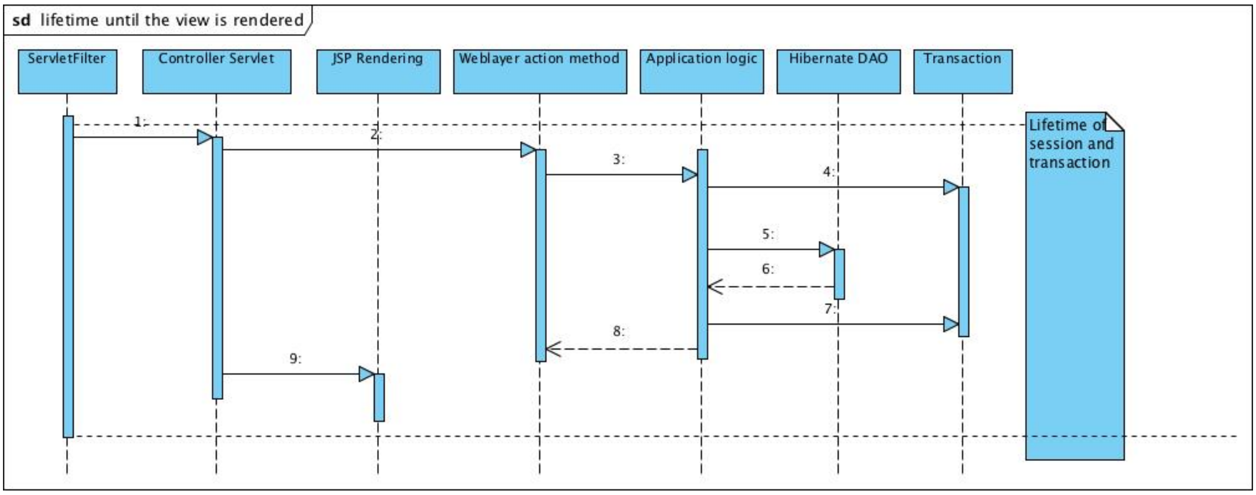
```
      session = factory.getCurrentSession();
      session.beginTransaction();
      turtle.setSize(TurtleSize.MEDIUM);
      session.update(turtle);
```

```
        session.getTransaction().commit();
```

This approach is simple but we will write the changes to the database in every step. If you want a acid transaction saving either all or none of the changes to turtle, we would have store the turtle in the HTTP session for example and save it only in the last step.

13.7.2. Lifetime until the view is rendered (Open-Session-in-View)

The pattern Open-Session-in-View extends the lifetime of the session. You can find an example in the project OpenSessionInView. In this project a Servletfilter opens a session and begins a transaction. After this the application logic is called, then the JSP is called to render a view. At the end the transaction is committed and the session is closed.



This is a pretty old approach to do this and will talk about a better approach later one. But let’s have a lock at the advantages and disadvantages. Basically it is the opposite of the last approach. We probably don’t get the LazyInitializationException any more but it might happen that data is inefficiently fetched during the rendering of the view.

Another problem we will face is exception handling. The commit of the transaction happens in the servlet filter. With the commit Hibernate flushes all open SQL statements. If we get any exception there, the view is already rendered and we can’t even show an error page. The page is already partly rendered and the browser will not accept a redirect at this point.

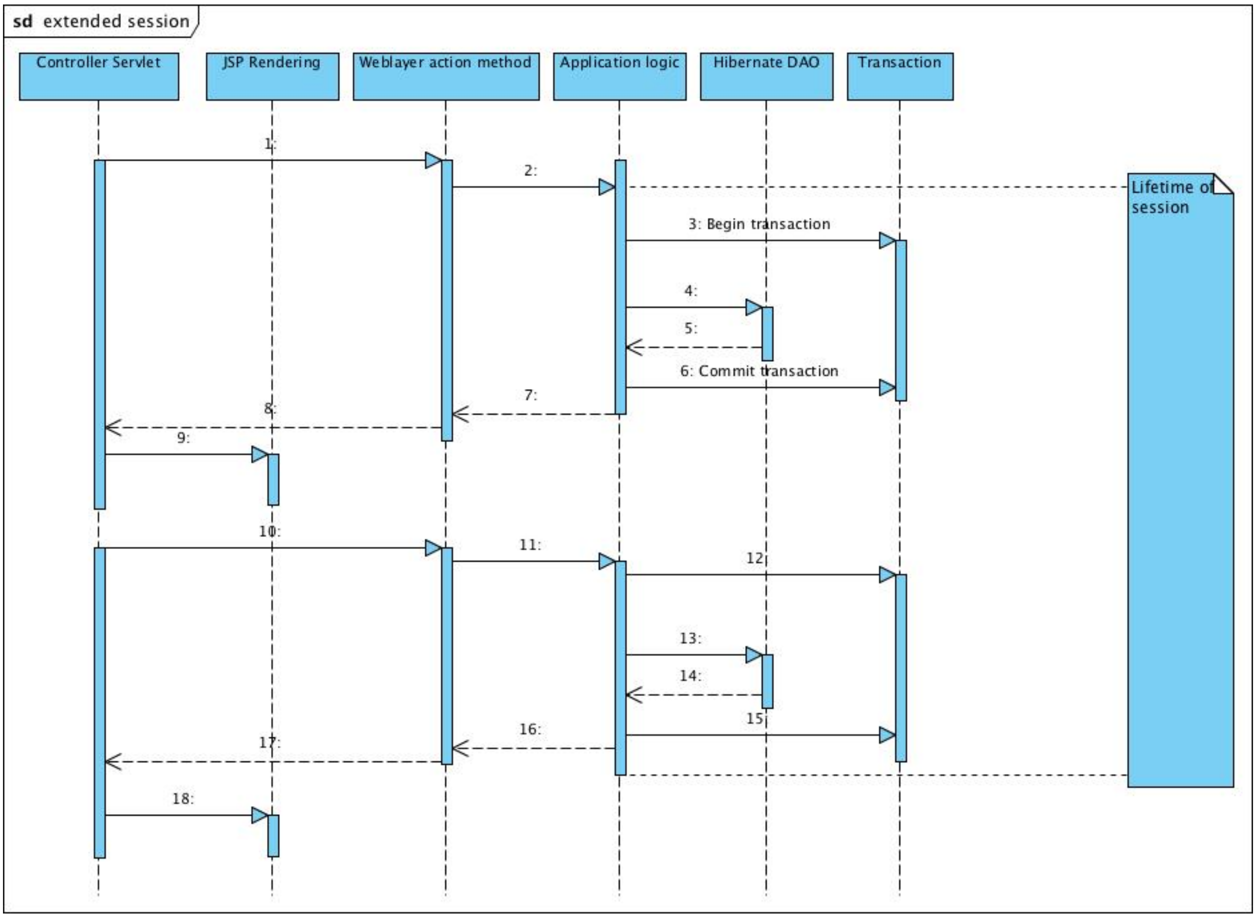
At a minimum measure you should call session.flush in the application logic to enforce that open statements are inserted. This will probably reduce the number of exceptions but still a commit might fail.

The best approach in my opinion is to commit the transaction inside of the application logic and to open a second transaction for rendering. You will need some coding and might use the ManagedSessionContext class of Hibernate in that case.

Concerning the conversation, we will have the same code as in the last example. Between requests, all entities will be in detached state and need to be reattached, if we want to change them.

13.7.3. Long life of a session

A session with a long lifetime is called Extended Session. The session is opened when a conversation starts and closed when it ends.



Between the requests we need to store the session somewhere. One option is for example the HTTP Session of a web application.

In every dialog happens a transaction, but we tell Hibernate not to flush any SQL statements. Normally a commit would flush all open statements. If we set the FlushMode of the session to manual, then we take over the responsibility and can do the flushing at the end. Let’s have a look at the code.

Dialog 1

We show a dialog to create a new turtle. The first thing we do after we have called \ openSession is to change the FlushMode to manual. It is not shown in the code but we will save the Hibernate session in the HTTP session.

```
Session session = factory.openSession();
session.setFlushMode(FlushMode.MANUAL);
session.beginTransaction();
Turtle turtle = new Turtle();
session.save(turtle);
session.getTransaction().commit();
```

Dialog 2

The user selects a beautiful name for the turtle and press save. We fetch the Hibernate session from the HTTP session and continue to work with it.

```
// Hibernate Session is stored in HTTP-Session
session.beginTransaction();
turtle.setName("Alexander");
session.getTransaction().commit();
```

Dialog 3

Finally the user measures the size of the turtle and inputs it in the last dialog. The Hibernate session is fetched from the HTTP session. At the end session.flush is called to send all open statements to the database. As we haven’t configured automatic closing for the session in the hibernate.cfg.xml, we have to call \ session.close.

```
// Hibernate Session is stored in HTTP-Session
session.beginTransaction();
turtle.setSize(TurtleSize.MEDIUM);
session.flush();
session.getTransaction().commit();
session.close();
```

The extended session approach has an attractive advantage: the session is never closed and the entities in the session stay in persistent state. You just have to change them and call session.flush at the end. There is no need to call session.update and you may prevent exceptions like the NonUniqueObject exception. There is a trade off as well. The session might grow as it contains all objects you have loaded with queries. In addition you are storing this large Hibernate session in the HTTP session. The former approaches close the session and release all unused entities to be garbage collected. The size of the session can be managed. Session.evict \ removes entities from the session.

It is up to you to decide what you want to do.

Pitfalls

Sometimes the manual flush doesn’t work. Hibernate just ignores what you want and flushes.

The reason is the chosen id generator. If you use sequence as generator, you will see a select statement for the id when calling session.save. Even in flush mode manual Hibernate figures out the id. There are id generators which requires an insert statement, ie: identity and select. If we use them we will get an immediate insert statements in the first dialog.

Open Session in View

We might combine the extended session with open session in view. In this case we need to set a kind of marker in the application logic to tell a servlet filter that it can close the session.

13.8. Concurrent Access

Let us think about the following scenario. User Foo selects a turtle to edit it. User Bar selects the same turtle to edit it. User Foo press the save button, then user Bar press the save button. Well, if two user edit the same data, we need to think of concurrent access. What should happen? Basically we have four options:

- We could prevent that Bar – the later one to edit – can edit the turtle.

- The changes of user Bar – the last one to press save – could overwrite the changes of Foo?But we have to keep in mind, Bar did not know that Foo was already editing and the changes of Foo might be more important.

- Should the changes of Foo – the first one to press save – be successful and Bar gets an error message and needs to restart editing?

- Should Foo be successful while Bar is being informed about the changes and he can decide if we wants to overwrite, merge or cancel his changes?

The default behaviour of Hibernate is option two. The last change will silently overwrite the changes before. Although it sounds bad, it is perfectly fine for a lot of application. If you don't expect that two user edit the same data in parallel then you don't have to care.

The first approach requires pessimistic locking, the third approach can be easily done with the optimistic locking functionality of Hibernate. The last approach is basically the third approach plus manual coding to craft the merge/overwrite/cancel dialog. Let's have a look at the different approaches.

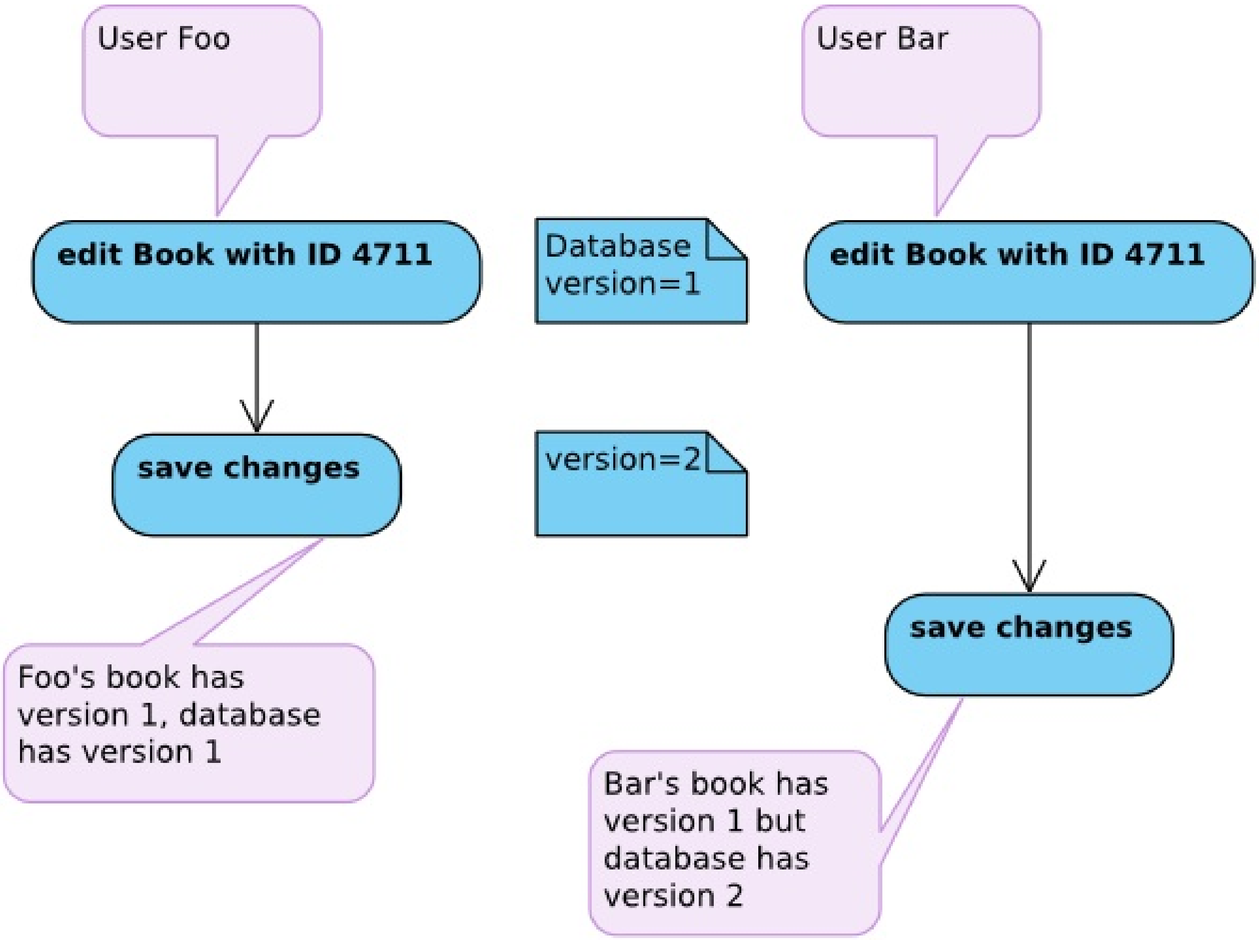
13.8.1. Optimistic Locking

Optimistic locking means that in fact we don't lock the database at all. This is not perfectly true, because for a short moment during the insert or update statement, we will see a short write lock in the database. To prevent the caveats of concurrent access a version column is used.

The database table has a column indicating the current version of the row. The column can contain an integer number which is increased by each change or a timestamp showing the change time.

User Foo (left side in the diagram) and user Bar (right side) load data to edit the same employee. The version column has the value *1*. User Foo saves his changes. Hibernate will verify which version of the object he has and which version is in the database. Only if he has the current version, he can update the row. In this case, he has version *1*, which is the same as in the database and his changes will be saved. At the same time Hibernate increments the version to *2*

Now user Bar tries to save his changes. Hibernate compares the version numbers. Bar has version *1* but the database has already version *2*. Hibernate will throw an exception.



```
 org.hibernate.StaleObjectStateException: Row was updated or deleted
by another transaction
```

We can catch the exception and tell Bar that he has to restart. You can find sample code for this approach in chapter Hibernate and Struts Section 14.2, “Hibernate and Struts”.

Version column impact on performance

The impact of a version column is neglect able. Don’t expect that you will get a separate select statement for the version. Hibernate will slightly change the update statement.

update employee set name=?, version=2 where id = ? and version = 1

An SQL update statement returns the number of changed rows. If the SQL result is 1, Hibernate knows that the entry was successfully updated and that the version is incremented. If it is 0, Hibernate will throw an exception.

Finally let’s have a look how to do this.

Version column The first important thing is to add a version column to your mapping. The type can be any of integer, long, short, timestamp, dbtimestamp or calendar. The version will be tracked by a integer value starting at zero or with a change timestamp depending on the type you selected. I recommend not to use timestamps. It is possible to create the same timestamp in the same moment and in addition modern machines quite often run on virtualized server. They suffer of larger time delays. If the time is readjusted, you might see new version conflicts.

Annotation

```
@Version
   private int version;
```

XML

```
<version name="version"  type="integer"></version>
```

Optimistic locking without a version column

If you are not allowed to add a version column – a common problem with legacy systems – you can still use optimistic locking. Hibernate can compare all columns of the old version to the current columns in the database. This is of course far less efficient but still better than nothing.

Annotation

We need to set optimisticLock to DIRTY or ALL and enable dynamic updates. This allows Hibernate to construct the update statements for every update. Normally it reuses the same statement all the time in order to improve performance.

ALL compares all columns, DIRTY compares only changed columns

```
import javax.persistence.Entity;
...
import org.hibernate.annotations.OptimisticLockType;



@Entity
@org.hibernate.annotations.Entity(optimisticLock=OptimisticLockType.ALL,
  dynamicUpdate=true)
public class Apple {
```

XML

```
  <class name="Apple" optimistic-lock="all" dynamic-update="true">
```

Limitations of this approach

Dynamic updates are slower, because Hibernate instead of reusing the same update query all the time, will recreate the update statement for every request.

Pessimistic locking

Pessimistic locking creates a lock in the database and enforces that no other user can edit the same data row. This is a suitable approach for Java application based on Swing or SWT. In a web application I discourage to use this approach. You don’t know, if the user is thinking, making a break or already on holiday. The lock on the object will stay until the HTTP session ends. If you forget to write a cleanup listener which rollback open transactions in your session, you might even have a transaction leak.

With modern Ajax technology you can work around this problem and send continuous ping from the client to the server. If the ping stops, you close down the lock on the database row.

How to lock a row with Hibernate?

You can lock an entity when you load it

```
      Apple appel = (Apple) session.get(Apple.class, id, LockOptions.UPGRADE);
```

or later

```
      session.buildLockRequest(LockOptions.UPGRADE).lock(apple);
```

It is even possible to lock entities using a query.

```
session.createQuery("from Apple a where a.name=?")
  .setParameter(0, theName)
  .setLockOptions(LockOptions.UPGRADE);
```

session.lock is deprecated

If you have some Hibernate experience or in old examples, you might be aware of code like *session.lock(customer)*. This method is now deprecated. Use the *buildLockRequest* method, which was demonstrated in this chapter.

Chapter 14. Integration with other technologies

In this chapter you will learn, how to integrate Hibernate into other technologies. The approaches shown, are proved in real world applications. I will name common pitfalls, you may come across, as well.

14.1. Hibernate and Spring

Spring is a beautiful framework, to implement the business logic of an application. It can integrate Hibernate in a very elegant way. You can download Spring from the page http://www.springframework.org/. This chapter requires that you have knowledge of Spring, as I will not explain any Spring basics.

The documentation of Spring presents three alternatives. I do not appreciate one of these approaches, therefore I will present two of them slightly adapted.

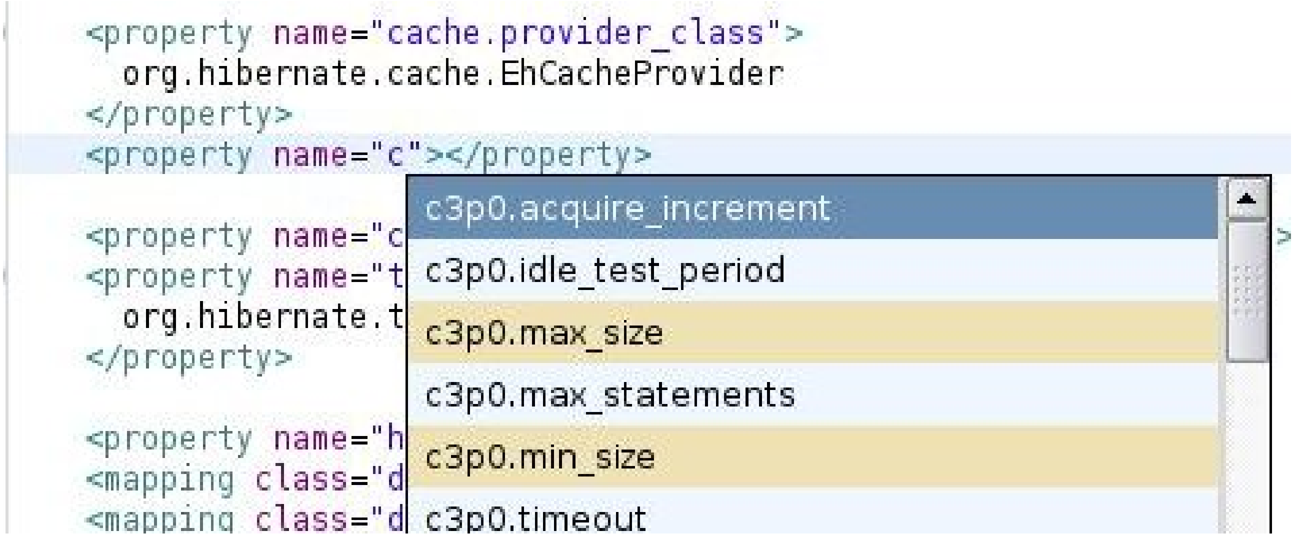
Both examples make use of Spring version 2 and annotations. Therefore you will need a JDK 5 or newer. If you still have to use Java 1.4, you have to configure transaction handling differently. I will give you some hints, when we speak about that topic.

14.1.1. Configuration

There are three approaches to configure Hibernate, when using Spring:

- Reference a Hibernate configuration from your Spring configuration
- Complete configuration of Hibernate inside your Spring configuration
- a mix of both approaches

I recommend the last approach, because some development environments provide auto completion for configuration settings in the Hibernate configuration file. Inside a Spring configuration file this is of course not supported. Below you can see a picture from the MyEclipse plugin for Eclipse showing auto-completion:



```
    <property name="cache.provider_class">
      org.hibernate.cache.EhCacheProvider
    </property>
    <property name="c"></property>

    <property name="c
    <property name="t
      org.hibernate.t
    </property>

    <property name="h
    <mapping class="d
    <mapping class="d
```

A configuration in Spring requires a datasource and a special SessionFactory provided by Spring. It is important that we use the SessionFactory provided by Spring, else the integration does not work and you may easily encounter connection leaks.

```
   <bean id="datasource"
      class="com.mchange.v2.c3p0.ComboPooledDataSource">
      <property name="driverClass" value="org.postgresql.Driver" />
      <property name="jdbcUrl"
         value="jdbc:postgresql://localhost:5432/learninghibernate" />
      <property name="user" value="postgres" />
      <property name="password" value="p" />
      <property name="minPoolSize" value="2" />
      <property name="maxPoolSize" value="4" />
   </bean>
   <bean id="hibernateSessionFactory"
      class=
"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
      <property name="dataSource" ref="datasource" />
      <property name="configLocation">
         <value>classpath:hibernate.cfg.xml</value>
      </property>
   </bean>
```

The Hibernate configuration file is quite short and does not include any settings for a database connection, Session handling and transaction.

```
<hibernate-configuration>
   <session-factory>
      <property name="dialect">
         org.hibernate.dialect.PostgreSQLDialect
      </property>
      <property name="cache.provider_class">
         org.hibernate.cache.EhCacheProvider
      </property>
      <property name="hbm2ddl.auto">none</property>
      <mapping class="de.laliluna.example.domain.Hedgehog" />
      <mapping class="de.laliluna.example.domain.WinterAddress" />
   </session-factory>
</hibernate-configuration>
```

Important tips

You should not configure your datasource in the Hibernate configuration file. If you still do it, you will encounter problems once your transactions are managed by Spring.

Alternatively, you can configure all settings of Hibernate in the Spring configuration.

```
   <bean id="datasource"
      class="com.mchange.v2.c3p0.ComboPooledDataSource">
      <property name="driverClass" value="org.postgresql.Driver" />
      <property name="jdbcUrl"
         value="jdbc:postgresql://localhost:5432/learninghibernate" />
      <property name="user" value="postgres" />
      <property name="password" value="p" />
      <property name="minPoolSize" value="2" />
      <property name="maxPoolSize" value="4" />
   </bean>

   <bean id="hibernateSessionFactory"
      class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
      <property name="dataSource" ref="datasource" />
         <property name="annotatedClasses">
         <list>
         <value>de.laliluna.example.domain.Hedgehog</value>
         <value>de.laliluna.example.domain.WinterAddress</value>
         </list>
         </property>
         <property name="hibernateProperties">
         <value>
         hibernate.hbm2ddl.auto=none
         hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
         hibernate.cache.provider_class=org.hibernate.cache.EhCacheProvider
         </value>
         </property>
   </bean>
```

My example uses a SessionFactory supporting annotations and XML.
If you want to use only XML mappings, you can change the class to *org.springframework.orm.hibernate3.LocalSessionFactoryBean*.

Why not using our simple Hibernate session factory?

We could consider to use our simple session factory, we created in the first example. In that case, we could select one of the patterns explained in chapter Session Handling Chapter 13, *Session and Transaction Handling*.

```
Session session = factory.getCurrentSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   tx.commit();

} catch (RuntimeException e) {

      try {
         if (tx != null)
            tx.rollback();
      } catch ( RuntimeException e1) {
         log.error("Transaction roleback not succesful", e1);
      }
      throw e;
}
```

Spring provides functionality to manage transactions in a comfortable way. Transaction management may even include Hibernate and separate SQL queries. If we use our own session factory, we cannot make use of this functionality.

If you use Spring in your project, I recommend to use the transaction handling of Spring as well. In that case, it is very important that you do not use the Hibernate transaction methods like session.beginTransaction.

Spring would not be able to release database connections or close open transactions.

14.1.2. Use of the Spring template

In this chapter, we will have a closer look at the first approach integrating Spring and Hibernate. The full source code is provided in the project HibernateSpring. A Maven build file is so provided. If you use maven, just type

```
mvn compile
```

to download all libraries and source files. If you do not use Maven you can download the Spring framework at

http://www.springframework.org/

The example project makes use of the following libraries:

- spring.jar
- aspectjrt.jar
- spring-aspects.jar
- aspectjweaver.jar

Spring offers a template for different persistence technologies, which encapsulates the specific implementations and exceptions. Exceptions are wrapped by Spring exceptions. Templates exist for example for the following technologies:

- Hibernate,
- JDO,
- Ibatis,
- Toplink,
- JPA,
- JDBC.

If you use multiple of these technologies, the template is a good approach to achieve a common behaviour and transactional handling.

The first step is to create a Hibernate template. This template requires a Spring-Session-Factory as constructor argument.

HibernateTemplate template = new HibernateTemplate(factory);

The template encapsulates Hibernate technology. The following code example shows the usage. It is easier to read the code from the inside.

```
return (List<Hedgehog>) template.execute(new HibernateCallback() {

   public Object doInHibernate(Session session)
         throws HibernateException, SQLException {
      return session.createCriteria(Hedgehog.class).list();
   }

});
```

The *new HibernateCallback()* creates a anonymous implementation of the interface *HibernateCallback*. This interface has one method: doInHibernate. It is called by Spring and gets as parameter a Hibernate session. The code – doing our work – is in the single line return *session.createCriteria(Hedgehog.class).list()* .

This freshly created implementation is passed to the execute method of the template in order to run it.

This construction might look like a little bit unusual, but allows Spring to encapsulate the execution of the Hibernate code, to translate exceptions and to role back a transaction if an exception happens. So far, we had to do this on our own.

You might question, which transaction can be rolled back? We haven’t started one. Well, if we do not configure a transaction, Spring will set the transaction mode of the JDBC connection to auto-commit. One of the next chapter will explain how to do make use of transactions explicitly.

A hint: The Hibernate template provides convenient methods for many methods of the Hibernate session: save, saveOrUpdate, delete, update, etc. There is no problem in using them.

```
template.save(hedgehog);
```

Problematic methods of the Hibernate template

I recommend not to use the find methods of the Hibernate template. The template has global settings for maximum results, caching behaviour and many others and you can easily get unexpected results, if you forget to remove settings.

I recommend to implement queries with the approach, I have just explained with the HibernateCallBack and only to use template methods for simple methods like save, update etc.

14.1.3. Alternative approach (nicer)

There is probably nicer code than the template with a CallBack implementation. Therefore I want to show an approach not using this construct as well. The example project is called HibernateSpring2.

We will make use of the method *getCurrentSession* which exist in the Spring-Hibernate Session-Factory as well. Like our examples with a current session context we just call *getCurrentSession* to access the Hibernate session.

```
protected Session getCurrentSession() {
   return factory.getCurrentSession();
}
public void save(Hedgehog object) {
   getCurrentSession().saveOrUpdate(object);
}
```

This code is not yet complete. With this approach we must manage transactions explicitly and start one before we call getCurrentSession.

Exception handling

The Spring documentation asserts that with this approach, we have to handle Hibernate exceptions in our code instead of Spring exceptions.

This is not very precisely explained. The exception is translated a little bit later. In the first approach (template) the exception was translated to a Spring exception by the template. The second approach will translate the exception with the transaction handling after our method was processed.

As Hibernate exceptions are fatal RuntimeExceptions and should be handled in the frontend layer – for example a webapplication - , this is no disadvantage at all. We would not deal with a Hibernate exception inside of a business method.

In my opinion the second approach is a better choice because it is better readable and less code to write.

14.1.4. Transaction handling

It is quite simple to manage transaction with Spring. We only have to configure a transaction manager. In this example we will use a JDBC based transaction manager.

```
<tx:annotation-driven transaction-manager="txManager"/>
<bean id="txManager" class=
"org.springframework.orm.hibernate3.HibernateTransactionManager" >
   <property name="sessionFactory" ref="hibernateSessionFactory"/>
</bean>
```

The spring documentation explains how to configure JTA based transaction managers. JTA configuration is application server specific.

It is quite easy to integrate transactions in our code. We only have to add a *@Transactional* annotation in front of a method. Spring will start a transaction before the method and commit at the end.

```
@Transactional(propagation = Propagation.REQUIRED)
public void update(Hedgehog hedgehog) {
   hedgehogDao.update(hedgehog);
}
```

propagation defines which kind of exception we use.

Table 14.1. Transaction types

Type of transaction	Description
REQUIRED	transaction is required. If one exist it will be used else one will be created. This is the default.
SUPPORTS	transactions are supported but won’t be started, If a transaction exist the method runs in this transaction scope else the method runs without transaction scope.
MANDATORY	transaction is required but won’t be started, If no transaction exist an Exception will be thrown.
REQUIRES_NEW	will always start a new transaction, If one exists already it will be paused. This is not supported by all transaction managers.
NOT_SUPPORTED	method will not inside of a transaction, If one exists already it will be paused. This is not supported by all transaction managers.
NEVER	Transactions are not supported, If a transaction exist an exception will be thrown.
NESTED	A nested transaction will be started. You can use the transaction manager DataSourceTransactionManager to achieve this. Only a limited number of database and JTA transaction manager supports this configuration.

If a RuntimeException happens inside of our method, Spring will rollback the transaction. This will not be done if a checked exception happens. You can configure the rollback behaviour for checked exceptions explicitly. Have a look in the Spring documentation for further informations.

Transaction configuration per class

You can add the *@Transactional* annotation in front of the class as well. In this case Spring will add transaction handling to all public methods. If a method should use a special kind of transaction, you can overwrite the default with a annotation in front of the method.

Transactions without annotations

In my opinion, the annotation based transaction handling is a very beautiful solution. If you cannot use annotations because you don’t have the required JDK 5, you have to choose another approach.

In this case we would define transactions in the Spring configuration file, start and commit transactions explicitly in the source code or use an transaction template. All approaches are described in the Spring documentation.

14.2. Hibernate and Struts

You can find a simple example application in the project HibernateStruts. The application follows best practices and the approach can be reused in real projects. The application makes of of Ajax technology, displays a list of hedgehogs, provides paging and allows to create and edit hedgehog information. I used PostgreSQL but you may replace the database if you like.

I used the following frameworks:

- Struts 1.3
- Displaytag 1.1 (http://displaytag.sourceforge.net/11/)
- Hibernate 3.2 Core und Annotation
- Ajaxtags 1.2 (http://ajaxtags.sourceforge.net/)
- Prototype Ajax 1.5.0 (http://www.prototypejs.org/)
- Script.aculo.us 1.7.0 (http://script.aculo.us/)

The application demonstrates

- business and DAO layers
- a generic DAO implementation
- optimistic locking and dealing with resulting exception
- central Hibernate exception handling

14.2.1. Optimistic locking

Optimistic locking means that when you update an object, Hibernate checks if this object was not already changed by another user. This approach is described in chapter TODO: Referenz nicht gefunden. If the data was already changed, Hibernate throws a StaleObjectException. In this case we have to apply the normal exception handling – rollback transaction, close the session – and display the user an appropriate error message.

I propose to do this directly in the service class HedgehogServiceImp. Have a look in the method *saveOrUpdateHedgehog*. After the exception handling the exception is thrown again to allow handling in our Struts action method.

The Struts action class HedgehogAction adds an error message to the request and sends the user back to the input form. You can test the behaviour by opening two browser windows, editing the same hedgehog in both windows and than saving them.

In contrast to other RuntimeException which are normally fatal, we can offer the user a little bit more comfort, if we handle the *StaleObjectException* this way. The next chapter explains how to handle other Hibernate RuntimeExceptions.

14.2.2. Exception handling

If an exception happens it is important to rollback a open transaction and to close the Hibernate session. If we use the CurrentSessionContext (see chapter Session Handling and Transactions

Chapter 13, *Session and Transaction Handling*) the closing of the session is handled by Hibernate, if we call commit or rollback.

```
<property name="current_session_context_class">thread</property>
```

I created a Struts ExceptionHandler, which is called if a RuntimeException happens in Struts. This exception handler will rollback the transaction and display a general exception page.

First we have to configure the exception handler in the Struts configuration file struts-config.xml.

```
<global-exceptions>
   <exception type="java.lang.RuntimeException"
      handler="de.laliluna.example.struts.SevereExceptionHandler"
      key="errors.severe" path="/error.jsp">
   </exception>
</global-exceptions>
```

The class *SevereExceptionHandler* is my *ExceptionHandler* rolling back the transaction.

```
try {

   Transaction transaction = InitSessionFactory
      .getInstance().getCurrentSession().getTransaction();

   if (transaction.isActive())
      transaction.rollback();
   log.debug("Rolled back transaction after exception.");

} catch (RuntimeException e) {
   log.error("Error rolling back transaction");
}
```

If your database is not available or another exception happens, you will see an error page now. My example uses the C3P0 connection pool. This pools tries to reach the database a number of times before throwing an exception. If you test the exception handling by shutting down the database, you will have to wait a moment before getting the exception page.

Part IV. Configuration, Performance, Validation and Full Text Search

Chapter 15. Hibernate Full Text Search

Lucene is a popular full text search engine. You can index documents, websites or arbitrary other data. The index can be searched with a API. Hibernate Search integrates Lucene Search with Hibernate. Entities can be indexed easily and with a special session, you can perform a full text search for your entities. A lot of databases provides already their own mechanism for full text search. But those solutions are not portable across databases and Lucene is probably more powerful and flexible than proprietary solutions.

Let’s have a first look at code sample before we talk about more details. You can find the full source code in the project LuceneSearch.

First, we need to configure the Lucene Search. If we use the AnnotationConfiguration to build the Hibernate session factory, there is only one property to be defined in the hibernate.cfg.xml. It is the location where the Lucene index should be stored.

```
<property name="hibernate.search.default.indexBase">
         /tmp
</property>
```

If you use a normal Configuration to build the session factory, it is required to configure a couple of event listener. Have a look in the reference documentation of Hibernate Search for more details. As it is possible to use XML with an AnnotationConfiguration as well, I propose to use this kind of configuration even with XML only mappings.

```
SessionFactory factory = new Configuration().configure().buildSessionFactory()
```

In the next step, the entities which should be searched have to be annotated.

```
@Entity
@Indexed
public class Article {
   @Id
   @DocumentId
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Integer id;

   @Field(index = Index.UN_TOKENIZED, store = Store.YES)
   private String title;

   @Field
   private String content;
// getter setter methods are missing
}
```

@Indexed marks an entity to be indexed, *@DocumentId* is required and defines how a document is identified in the Lucene index. @Field specifies that a field should be indexed. In the sample you can find two different settings for @Field.

@Field with default values

The content is indexed using the standard analyser. This analyser splits the text into words, transform them to lower case, removes characters like ;.' and removes a couple of very frequent English words like *a*, *is*, *in*

Only the indexed content will be stored in the Lucene index but not the content itself. If you use a tool like Luke to have a look at your Lucene index, you cannot see the original content.

The title is not tokenized or transformed. *@Field(index = Index.UN_TOKENIZED, store = Store.YES)*

As a consequence you cannot search individual words of the title, but we can search for the precise title or do a wild card search – find all titles starting with Foo.

In contrast to the field content, the title is stored in the index (store = Store.YES) and we can see it if we browse the index using Luke. I will tell you more about Luke at the end.

So, our entity is indexed and we can start to do full text searches in Hibernate. A Lucene search consists of three steps

- Creating a search session
- Creating a Lucene query
- executing the query.

A code sample.

```
Session session = SessionFactoryUtil.getFactory().getCurrentSession();
//      create a full text session
FullTextSession fSession = Search.getFullTextSession(session);
fSession.beginTransaction();
//      create a luceneQuery with a parser
QueryParser parser = new QueryParser("title", new StandardAnalyzer());
Query lucenceQuery = null;
try {
   lucenceQuery = parser.parse("content:hibernate");

} catch (ParseException e) {
   throw new RuntimeException("Cannot search with query string",e);
}
//      execute the query
List<Article> articles = fSession.createFullTextQuery(lucenceQuery, Article.class)
   .list();
for (Article article : articles) {
   System.out.println(article);
}
fSession.getTransaction().commit();
```

A search session is created from an open Hibernate session. Basically it is just a wrapper adding the search specific methods to the session. We use a StandardAnalyser to analyse the search string, which is the same \ used to index the content field. Finally we execute the full text query.

The field title was not tokenized. A search for title needs to use a different approach. You can use a precise search

```
List<Article> articles = fSession.createFullTextQuery(
     new TermQuery(new Term("title", "About Hibernate")), Article.class).list();
      for (Article article : articles) {
         System.out.println(article);
      }
```

or a wildcard search

```
List<Article> articles = fSession.createFullTextQuery(
     new WildcardQuery(new Term("title", "About*")), Article.class).list();
      for (Article article : articles) {
         System.out.println(article);
      }
```

You can adapt the indexing and the search string analysing to your needs. For example we could specify that the indexing of the field title goes through a toLowerCase filter. Emanuel Bernard has demonstrated a couple of new features on the Devoxx conference. You can use word stemming – run, runner, running \ {}- to find words with the same stem, phonetic searches with Soundex or Metaphone algorithm to find words with a close sound or approximate searches with ngram search. I will cover more of this approaches in the next updates.

Luke let you browse your index and perform searching on it. It is very useful to test your searches or debug a problem.

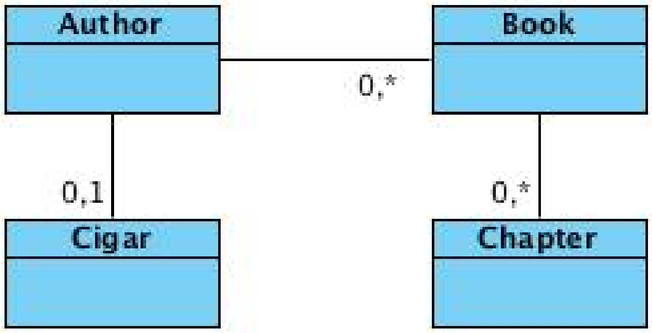
http://www.getopt.org/luke/

Chapter 16. Performance Tuning

There is a incredible choice of options to improve the performance of Hibernate based application. This chapter describes quite a number of them. Each part starts with a small use case describing the problem or requirement. After this you will find a detailed solution how to solve the problem.

Source code for the samples can be found in the project *mapping-examples-annotation*. Have a look in the package *de.laliluna.other.query*.

Some of the use cases make use of the following class structure:



16.1. Analysing performance problem

Scenario

If a dialog is too slow, there might happen a lot of unexpected queries. This can be caused by eager loading of relations or you might just reuse the queries of another dialog. Did you know for example that 1:1 and n:1 relations are loaded eagerly, if you use annotations whereas XML mappings are lazy by default.

Solution

The best approach is to analyse what is happening behind the scenes. Hibernate offers two configuration settings to print the generated SQL. The first one is a property in the hibernate.cfg.xml

```
<property name="show_sql">true</property>
```

If it is set to true, the SQL statements will be printed to the console. You will not see any timestamps, this is why I prefer the second approach, which uses the normal logging output.

```
# logs the SQL statements
log4j.logger.org.hibernate.SQL=debug

# Some more useful loggings
# Logs SQL statements for id generation
log4j.logger.org.hibernate.id=info
# Logs the JDBC-Parameter which are passed to a query (very verboose)
log4j.logger.org.hibernate.type=trace
# Logs cache related activities
log4j.logger.org.hibernate.cache=debug
```

There are more useful settings in the Hibernate configuration hibernate.cfg.xml

The property format_sql will nicely format the SQL instead of printing it on a single line.

```
<property name="format_sql">true</property>
```

The property use_sql_comments adds a comment to each SQL explaining why it was created. It let’s you identity if a HQL statement, lazy loading or a criteria query led to the statement.

```
<property name="use_sql_comments">true</property>
```

Another good source for information are the statistics of Hibernate.

You can enable the statistics in the Hibernate configuration or programmatically. The statistics class offers a number methods to analyse what has happened. Here a quick example:

```
Statistics statistics = sessionFactory.getStatistics();
statistics.setStatisticsEnabled(true);
statistics.logSummary();
```

Furthermore you can call getStatistics on a session as well to gather information about it.

16.2. Iterating through relations – batches

Scenario

The application retrieves all books for an author. It iterates through all chapters and counts the number of characters. An alternative scenario could go through orders of a customer and check if one of the order position can already be delivered.

The query for the books:

```
List<Book> books = session.createQuery(
   "from Book b where b.name like ?").setString(0, "Java%").list();
```

The following code printing the books will create one SQL query per book to initialize the chapters. We get 1+n queries in total. One for the books and n for the chapters, if we have n books.

```
for (Book book : books) {
   int totalLength = 0;
   for (Chapter chapter : book.getChapters()) {
      totalLength += (chapter.getContent() != null ?
      chapter.getContent().length() : 0);
   }
   log.info("Length of all chapters: " + totalLength);
}
```

Solution

One way to improve this is to define that Hibernate loads the chapters in batches. Here is the mapping:

Annotations.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(nullable = false)
@BatchSize(size = 4)
private Set<Chapter> chapters = new HashSet<Chapter>();
```

XML.

```
<set name="chapters" batch-size="4">
  <key column="book_id"></key>
  <one-to-many/>
</set>
```

When iterating through 10 books, Hibernate will load the chapters for the first four, the next four and the last two books together in a single query. This is possible because the *java.util.List* returned by Hibernate is bewitched. Take the sample code of this chapter and play around with the *batchsize* in the method efficientBatchSizeForRelation in the class PerformanceTest.

The best size of the batch size is the number of entries you print normally on the screen. If you print a book and print only the first 10 chapters, then this could be your batch size.

It is possible to set a default for all relations in the Hibernate configuration.

```
<property name="default_batch_fetch_size">4</property>
```

Use this property with care. If you print on most screens the first 5 entries from a collection, a batch size of 100 is pretty useless. The default should be very low. Keep in mind that a size of 2 reduces the queries already by 50 % and 4 by 75 %.

16.3. Iterating through relations – subqueries

Scenario

Same scenario again, this time we will load all chapters in one step.

Solution

The property subselect fetching defines that Hibernate should load all chapters of all books, when the first collection is being accessed.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(nullable = false)
@Fetch(FetchMode.SUBSELECT)
private Set<Chapter> chapters = new HashSet<Chapter>();
```

This is slightly different than the batch size, which loads only chapters for a couple of books. You might consider to use batch fetching if you want to load the chapters only for a couple of books, where as subselect fetching is useful if you always want to load the chapters for all books of the result list. There are less use cases for subselect fetching as it is to aggressive for most situations.

You can play around with subselect fetching in the method *efficientBatchSizeForRelation* in the class *PerformanceTest*. Just uncomment the *@Fetch* annotation in the book class.

16.4. Iterating through relations – single query

Scenario

There is still the same problem but we solve it with a different query.

Solution

With join fetch we can tell Hibernate to load associations immediately. Hibernate will use a single SQL select which joins the chapters to the book table.

```
List<Book> books = session.createQuery(
   "select b from Book b left join fetch b.chapters where b.name like ?")
   .setString(0, "Java%")
   .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();
```

The same with a criteria query:

```
List<Book> books = session.createCriteria(Book.class)
   .setFetchMode("chapters", org.hibernate.FetchMode.JOIN)
   .add(Restrictions.like("name", "Java", MatchMode.START))
   .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();
```

The performance is very good but we must be aware that we will load all chapters into memory. Finally, don’t use join fetch with multiple collections, you will create a rapidly growing Cartesian product. A join will combine all possible combinations. Let’s do a SQL join on data where every book have two chapters and two comments.

select * from book b left join chapters c on b.id=c.book_id left join comment cm on b.id=cm.book_id

Book id	**Other book columns**	**Chapter id**	**Other chapter columns**	**Comment id**	**Other comment columns**
1	…	1	…	1	…
1	…	1	…	2	…
1	…	2	…	1	…
1	**…**	**2**	**…**	**2**	**…**

16.5. Reporting queries

Scenario

For a report you need to print the name of the author, the number of books he wrote and the total of chapters in his books. If your dataset consists of 10 authors with 10 books each and each book having 10 chapters, you will end up with 1000 objects in memory. The report requires only 10 java.lang.String and 20 java.lang.Integer.

Solution

The problem can easily be solved with a reporting query. The following query returns a list of Object arrays instead of entitys.

```
List<Object[]> authorReportObjects = session.createQuery("select a.name, " +
   "count(b) as totalBooks, count(c) as totalChapters " +
   "from Author a join a.books b join b.chapters c group by a.name")
   .list();
```

```
for (Object[] objects : authorReportObjects) {
   log.info(String.format("Report: Author %s, total books %d, total chapters %d",
      objects[0], objects[1], objects[2]));
}
```

An alternative is to fill a Java class dynamically. If you use HQL you might call the constructor with a corresponding arguments or with both HQL and criteria you can use an AliasToBeanResultTransformer. In that case the Java class needs to have the same properties as the column names of your query.

Constructor approach.

```
List<AuthorReport> authorReports = session.createQuery(
   "select new de.laliluna.other.query.AuthorReport(a.id, a.name, " +
   "count(b), count(c)) " +
   "from Author a join a.books b join b.chapters c group by a.id, a.name")
   .list();
for (AuthorReport authorReport : authorReports) {
   log.info(authorReport);
}
```

AliasToBeanResultTransformer approach.

```
List<AuthorReport> authorReports = session.createQuery(
   "select a.name as name, count(b) as totalBooks, count(c) as totalChapters " +
   "from Author a join a.books b join b.chapters c group by a.name")
   .setResultTransformer(new AliasToBeanResultTransformer(AuthorReport.class))
   .list();
for (AuthorReport authorReport : authorReports) {
 log.info(authorReport);
}
```

16.6. Iterating through large resultsets

Scenario

Our application should export all books and a summary of the chapters in XML format. We have a total of 100,000 books with 15 chapter each. It is just impossible to load all the entities into memory.

Solution

Instead of a normal query we are going to use a ScrollableResults. It will load the data step by step as we are iterating through the result. From time to time, we will call session.clear to remove the entities from the session. This allows the garbage collection to take them away and will limit our memory consumption.

```
ScrollableResults results = session.createQuery(
   "select b from Book b left join fetch b.chapters")
   .scroll(ScrollMode.FORWARD_ONLY);
while (results.next()) {
  Book book = (Book) results.get(0);

  /* display entities and collection entries in the session */
  log.debug(session.getStatistics().toString());
  // create XML for book
  for (Chapter chapter : book.getChapters()) {
    chapter.getContent();
    // create XML for chapter
  }
  session.clear();
}
results.close();
```

For debugging purpose I am printing the number of entities in the session using session.getStatistics().toString().

Pitfall warning

If you change entities, you must call session.flush before calling session.clear. Clear clears you session and all open SQL statements not yet sent to the database will be sent as well.

16.7. Caches

16.7.1. General

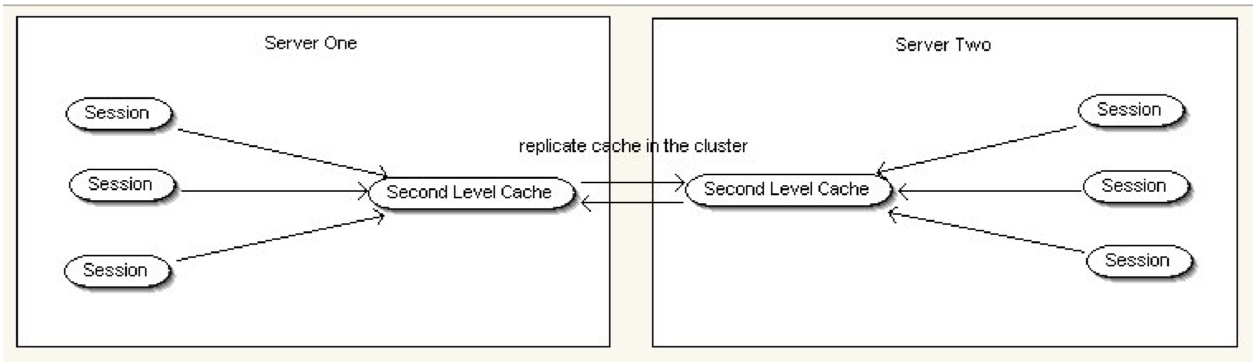
The purpose of a cache is to reduce access to the database.

There are two kinds of caches in a Hibernate application. The Hibernate session is the first one, the second one can be used optionally.

The Hibernate session is a transactional cache, i.e. it shows the changes a user has already flushed within an open transaction. As a session belongs to a unique user, theses changes are not propagated to other sessions. I remind you that a session is not thread-safe, do not share it across threads!

The second level cache is shared across a Java virtual machine instance or, if you use clustering, it can even be shared across multiple servers across a network. All your sessions share this cache. The second level cache normally contains only committed data. (except for transaction caches, where you can configure the isolation level)

A cache does not reference your objects, so changes to the objects will not destroy the cache. A cache resembles more a Map having a combination of id and class as key and an array as value, where the array holds all the field values.



Key	**Field 1**	**Field 2**	**Field 3**
de.laliluna.Developer#674	*Karl*	1234	77.55
de.laliluna.Developer#680	***Peter***	**123**	**12.22**

This is of course a simplification. A cache does also handle information about the creation time, last access time, change time and some very effective search algorithms.

Aspects to consider

A cache holds objects to prevent database queries. A cache is not aware of changes which bypass Hibernate for example queries issued by JDBC or from other non Java applications.

Situations where a cache is useful

- The same data is queried repeatedly
- Amount of data is relative small as compared to the available memory
- Queries are slow and the database is on the network
- All applications use Hibernate and the same cache

Choosing the cache mode

There are four cache modes available.

- read-write
- read-only
- nonstrict-read-write
- transactional

Readonly

By far the fasted cache is read-only but if you specify this in a class you cannot save or update any entries. You may consider using special mappings only to display data in combination with a readonly cache. This would push the performance to the limit.

Read-write

This is probably the most commonly used setting. As indicated by the Hibernate reference this cache cannot be used if you use the transaction isolation level serializable, which is probably used only in very rare cases.

Nonstrict-read-write

This cache mode does not guaranty strict transaction isolation. An update to the same dataset happens rarely by two concurrent threads at the same moment. This is true when your application uses optimistic locking with short transactions and is not heavily stressed by simultaneous transactions to the same data. When a transaction takes 50 milli-seconds you must have two threads accessing the same data at the same moment.

Transactional

Transactional cache modes are fully supported by the cache and you can configure the isolation level of the cache. Most caches do not support this behaviour.

Standard cache

The standard cache is filled when you load any data with methods like

- session.load
- session.get
- session.createQuery(…).list()
- …..

The cache is used when Hibernate has to create an instance from a given id. This is done when you use for example

- session.load
- session.get
- traversing a relation which is not yet initialised

You can configure the size of the cache and when items should expire. Expiring is useful when you update data with other applications from time to time but you can accept a delay during which the user sees stale data.

You do not have to set a timeout when your application is the only one accessing the database.

You can evict objects from the cache using the session factory (not the session itself). It provides methods like evict and others to deal with data in the cache.

Using the cache

Configure the cache in your *hibernate.cfg.xml* or in the *hibernate.properties*.

```
<property name="cache.provider_class">
   org.hibernate.cache.EhCacheProvider
</property>
```

Add a configuration file for your selected cache. You can find more detailed information about each cache in the following chapters. Here is a short example.

```
<cache name="myCache"
    maxElementsInMemory="6000"
    eternal="true"
    timeToIdleSeconds="0"
    timeToLiveSeconds="0"
    overflowToDisk="true"
/>
```

Add the cache tag to your class and to all relations you want to cache. You can specify a cache region to use a special configuration. For example I used the myCache configuration but you can also leave out the region definition in order to use the default configuration.

```
<class name="Developer" table="tdeveloper">
 <cache usage="read-only"  region="myCache" />
```

```
<list name="computers" cascade="all">
  <cache usage="read-only"  region="myCache" />
  <key column="developer_id" not-null="true"></key>
  <list-index column="listindex"></list-index>
  <one-to-many class="Computer" />
</list>
```

That’s it.

Query cache

A query cache contains the object ids found by a query. The query and the parameters are used as key. As a consequence a query cache is only useful when you have queries with the same parameters issued regularly. When a table is updated, all cached queries touching the table are evicted from the cache as they are considered to be outdated.

To sum up, a query cache is useful when you have a lot of identical selects compared to the number of updates. Typical applications are content management systems, Internet forums with a lot of read access etc.

To use the query cache you must do at least the following:

In the *hibernate.properties* or *hibernate.cfg.xml* add the property

```
  <property name="hibernate.cache.use_query_cache">true</property>
```

In your query set CacheAble to true

```
List result = session.createQuery(
   "from Computer c where c.name like ? order by c.name desc, c.id asc")
   .setString(0,param)
   .setCacheable(true).list();
```

You can also specify a special cache region used for your query using setCacheRegion

```
List result = session.createQuery(
   "from Computer c where c.name like ? order by c.name desc, c.id asc")
   .setString(0,param)
   .setCacheable(true).setCacheRegion(“myQueryCache”).list();
```

That’s it.

As opposed to the table in the Hibernate reference all of the following cache providers support query caching.

Test all caches

We provided an example application named TestCache. It will output access times to the data.

In order to use the test, generate test data first. Adapt the *hibernate.cfg.xml* to suit your database and uncomment the *setupTest* method in the class TestMain to generate random data entries.

Have a look at the class *TestMain* to comment or uncomment the tests you want to run and to specify which mapping to load. We have mappings for all supported cache modes. A cache provider does not support every mode.

You can configure the number of threads for running two kinds of tests. The first uses session.get to randomly select entries. The other test queries data with random parameters. Both tests log how the access times are changing by the time when your cache is filled. Run a no cache test to compare the results.

In the hibernate.cfg.xml you can configure the cache provider. Configuration files for any of the following caches can be found in the project as well. Read in the following descriptions to learn more about needed libraries.

16.7.2. EH Cache

EH cache is the standard cache for Hibernate. It can be used as a distributed cache or as cache for JSP content. Detailed information can be found on the EH Cache website at

http://ehcache.sourceforge.net/documentation/

The configuration is done in the *ehcache.xml* file:

```
<ehcache>

    <!-- Sets the path to the directory where cache .data files are created.

         If the path is a Java System Property it is replaced by
         its value in the running VM.

         The following properties are translated:
         user.home - User's home directory
         user.dir - User's current working directory
         java.io.tmpdir - Default temp file path -->
    <diskStore path="java.io.tmpdir"/>

    <cache name="myCache"
        maxElementsInMemory="6000"
        eternal="true"
        timeToIdleSeconds="0"
        timeToLiveSeconds="0"
        overflowToDisk="false"
        />

    <cache name="net.sf.hibernate.cache.StandardQueryCache"
        maxElementsInMemory="20"
        eternal="false"
        timeToLiveSeconds="240"
        overflowToDisk="false"/>

  <cache name="net.sf.hibernate.cache.UpdateTimestampsCache"
        maxElementsInMemory="5000"
        eternal="true"
        overflowToDisk="false"/>

    <defaultCache
        maxElementsInMemory="7000"
        eternal="false"
        timeToIdleSeconds="180"
        timeToLiveSeconds="180"
        overflowToDisk="false"
        />
```

```
</ehcache>
```

Cache provider in the hibernate.cfg.xml:

```
<property name="cache.provider_class">
   org.hibernate.cache.EhCacheProvider
</property>
```

Needed libraries:

- eh-cache.jar

Supported cache modes:

- read-write
- read-only
- nonstrict-read-write

16.7.3. OS Cache

OS Cache can also be used as distributed cache or as cache for JSP content. Clustering using JMS and jgroups is supported. Detailed information can be found at http://www.opensymphony.com/oscache/.

Cache provider:

```
<property name="cache.provider_class">
   org.hibernate.cache.OSCacheProvider
</property>
```

Needed libraries:

- oscache.jar

Supported cache modes:

- read-write
- read-only
- nonstrict-read-write

16.7.4. Swarmcache

The swarm cache can be used as distributed cash as well. It uses jgroups for the clustering. Further information can be found at http://swarmcache.sourceforge.net/.

Cache provider:

```
<property name="cache.provider_class">
   org.hibernate.cache.SwarmCacheProvider
</property>
```

Needed libraries:

- swarmcache.jar
- jgroups-all library which you must download from the swarmcache website.

Supported cache modes:

- nonstrict-read-write

JBoss Treecache

JBoss is the only transactional cache in our list. It supports clustering as well. Further information can be found at http://www.jboss.org/products/jbosscache

You may consider using the treecache if you use Hibernate on JBoss.

Cache provider:

```
<property name="cache.provider_class">
   org.hibernate.cache.TreeCacheProvider
</property>
```

Needed libraries to run in a java application:

- jboss-cache.jar
- jboss-system.jar
- jboss-jmx.jar
- concurrent.jar
- jboss-common.jar
- jgroup library deliverd by Hibernate

For a web application deployed on a JBoss application server you will probably not need any of these libraries as they are already contained in a typical JBoss installation.

Supported cache modes:

- transactional

16.7.5. Bypass a cache

When you use *LockMode.Read* the cache is bypassed and the values are read directly from the database. You can specify the *LockMode* when you use *session.lock* or *session.load*.

Chapter 17. Configuration

17.1. Connection Pools

17.1.1. Built-in connection pool

If you do not configure a connection pool yourself, the built-in connection pool is used. Of course, it usually does work when you develop, but in some rare cases it does not work properly. So just don't use it. When you test your Hibernate layer without an application server use C3P0 or DBCP. Normally it is a good idea to use the connection pool provided by your application server if you only run tests within the application server. You may use C3P0 or DBCP also within an application server but there is normally no reason to deploy any further libraries when the functionality is already there.

C3P0

You can configure a minimal pool in Hibernate just by adding the following settings:

```
    <property name="c3p0.max_size">4</property>
    <property name="c3p0.min_size">2</property>
    <property name="c3p0.timeout">1800</property>
```

Warning

Do not use the following tag in combination with this connection pool.

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

The schema generation will close connections in a way the Pool cannot cope with.

Some optional settings are

```
    <property name="c3p0.acquire_increment">1</property>
    <property name="c3p0.idle_test_period">100</property><!-- seconds -->
    <property name="c3p0.max_statements">0</property>
    <property name="c3p0.AcquireRetryDelay">1001</property>
    <property name="c3p0.AcquireRetryAttempts">5</property>
    <property name="c3p0.MaxStatementsPerConnection">0</property>
```

Documentation can be found in the download package. The libraries are included in the Hibernate dialogue, but I recommend to download the most current version from http://sourceforge.net/projects/c3p0

The library needed is

- c3p0….some version.jar

DBCP

DBCP is not included in the Hibernate 3 distribution any more. If you want to use it have a look at http://wiki.apache.org/jakarta-commons/DBCP/Hibernate to get a custom connection provider. This connection provider must be specified in the hibernate.cfg.xml

```
<property name="connection.provider_class">
   org.hibernate.connection.DBCPConnectionProvider
</property>
```

Download and documentation can be found at http://jakarta.apache.org/commons/dbcp/

DBCP depends on two libraries, which are provided by jakarta as well:

- commons-dbcp

- commons-pool

To configure DBCP add the following properties to your *hibernate.cfg.xml*

```
<property name="connection.provider_class">
   org.hibernate.connection.DBCPConnectionProvider
</property>

<property name="dbcp.maxActive">17</property>
<property name="dbcp.maxWait">120000</property>
<property name="dbcp.maxIdle">3</property>
<!-- Action to take in case of an exhausted DBCP connection pool
   ( 0 = fail, 1 = block, 2= grow) -->
  <property name="dbcp.whenExhaustedAction">1</property>
  <property name="dbcp.ps.maxActive">150</property>
  <property name="dbcp.ps.whenExhaustedAction">2</property>
  <property name="dbcp.ps.maxWait">120000</property>
  <property name="dbcp.ps.maxIdle">10</property>
  <property name="dbcp.validationQuery">select 1</property>
  <property name="dbcp.testOnBorrow">true</property>
  <property name="dbcp.testOnReturn">false</property>
```

17.1.2. JNDI

JNDI is the Java Naming and Directory Interface. It allows to bind resources to names. You can bind Java classes like Data sources, session factories or other to JNDI and retrieve them using their name.

You have to setup the JNDI datasource in your application server or servlet engine.

Here is an extract of the standalone.xml of a JBoss application server.

```
<datasource jndi-name="java:/jboss/datasources/playDS" pool-name="playDS">
    <connection-url>jdbc:postgresql://localhost/play</connection-url>
    <driver>postgresql</driver>
    <security>
        <user-name>postgres</user-name>
        <password>p</password>
    </security>
</datasource>
```

In your Hibernate configuration, the JNDI is referenced.

```
<property name="connection.datasource">java:/jboss/datasources/playDS</property>
```

Appendix A. Appendix

A.1. Annotation Reference

The reference contains a description of many Hibernate annotations.

Default values of annotations

If you want to find out supported attributes you may have a look into the source code. In Eclipse you can press CTRL (alias STRG) and click on a class name to display the attached source code. The first time you make this, you have to define where the source code of the class can be found.

Below you can find the source code for the *javax.persistence.Table* annotation.

You can download the source code for Hibernate annotations at http://www.hibernate.org or use the Maven repository of JBoss. https://repository.jboss.org/nexus/index.html

```
/*
 * The contents of this file are subject to the terms
 * of the Common Development and Distribution License
 * (the License).  You may not use this file except in
 * compliance with the License.
 *
 * You can obtain a copy of the license at
 * https://glassfish.dev.java.net/public/CDDLv1.0.html or
 * glassfish/bootstrap/legal/CDDLv1.0.txt.
 * See the License for the specific language governing
 * permissions and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL
 * Header Notice in each file and include the License file
 * at glassfish/bootstrap/legal/CDDLv1.0.txt.
 * If applicable, add the following below the CDDL Header,
 * with the fields enclosed by brackets [] replaced by
 * you own identifying information:
 * "Portions Copyrighted [year] [name of copyright owner]"
 *
 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
 */
package javax.persistence;

import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target(TYPE)
@Retention(RUNTIME)

public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
}
```

A.1.1. Entity and table annotation

@javax.persistence.Entity

Specifies that a class is used by Hibernate as entity

name="EntityName"
Optional, name which can be used in queries, default: class name

Sample.

```
@Entity
public class Tiger implements Serializable {
```

@org.hibernate.annotations.Entity

Hibernate extension, must be used in addition to @javax.persistence.Entity

mutable=true,
if false then the entity is treated as read only, default is true

dynamicInsert=false,
if true then only changed columns will be included in the insert statement. The default is false which is better in most cases as the same prepared statement can be reused all the time.

dynamicUpdate=false,
if true then only changed columns will be included in the update statement. The default is false which is better in most cases as the same prepared statement can be reused all the time.

selectBeforeUpdate=false,
Makes a select and compares the data and will only create an update if the data was changed. Default is false.

polymorphism=PolymorphismType.IMPLICIT,
Selects of parent classes in a inheritance hierarchy can return all entities as instance of the queried class (explicit polymorphism) or as instance of the class it should actually be (implicit polymorphism)

persister="",
Default is "". It allows to modify how the data is stored. It is rarely changed but allows for example to store data in JNDI.

optimisticLock= OptimisticLockType.VERSION
Defines how optimistic locking works. Default is a version column but you can compare all columns (OptimisticLockType.ALL) or just the changed columns (OptimisticLockType.DIRTY) to verify if a row has been updated by someone else.

@javax.persistence.Table

Optional, specifies database table a class is mapped.

name = "tableName",
Optional, name of the database table to which the class is mapped, default: class name

catalog = "catalogName",
Optional, name of the database catalogue (only supported by some databases), default: name, defined in the Hibernate configuration

schema = "dbSchemaName",
Optional, name of database schema (only supported by some databases), default: name, defined in the Hibernate configuration

uniqueConstraints = { @UniqueConstraint(columnNames = { "databaseColumn}", “anotherColumn” }) }
Optional, no influence on Hibernate but on the table generation. It will generate unique key constraints.

@org.hibernate.annotations.Table

Hibernate extension to EJB 3, generates indexes, when generating tables. Can be used with the annotations *@javax.persistence.Table* and *@javax.persistence.SecondaryTable*

appliesTo = "tableName",
database table name

indexes = { @Index(name = "forest_idx", columnNames = {"indexedColumn"}) }
one or more indexes

@org.hibernate.annotations.Index

Hibernate extension to EJB 3, specifies an index

name = "forestidx",
Name of the index

columnNames = {"colA","colB"}
Database column names which are included in the index

Sample.

```
@Entity()
@javax.persistence.Table(name = "tableName",
uniqueConstraints = { @UniqueConstraint(columnNames = { "unique_column" }) })
@org.hibernate.annotations.Table(appliesTo = "tableName",
indexes = { @Index(name = "forestidx", columnNames = { "indexedcolumn" }) })
public class Garden implements Serializable {
```

@javax.persistence.SecondaryTable

Map one bean to two tables. You specify for a column, that it belongs to the secondary tables (→ @Column(table=”secondaryTable”)

name = "tableName",
Required, specifies the name of the database table to which the class is mapped.

catalog = "catalogName",
Optional, specifies the catalog catalogue name (only support by some databases), default: name, defined in the Hibernate configuration..

schema = "dbSchemaName",
Optional, specifies the schema name (only support by some databases), default: name, defined in the Hibernate configuration.

pkJoinColumns = { @PrimaryKeyJoinColumn(name = "id", referencedColumnName = "id", columnDefinition = "int4") }
One or more PrimaryKeyJoinColumn to define how the tables are mapped.

@javax.persistence.UniqueConstraint

Specifies a unique key constraint. This information is used when Hibernate generates tables.

columnNames = { "databaseColumnName}", “anotherColumn” }
Required, specifies the table column names.

A.1.2. Primary key annotations

Source code for examples can be found in the project *mapping-examples-annotation* package de.laliluna.primarykey.

@javax.persistence.Id

Specifies that an attribute belongs to the primary key. Is applied to all primary key attributes.

@javax.persistence.GeneratedValue

Specifies that the value is generated by a sequence, increment, …

strategy=GenerationType.SEQUENCE,
Default: GenerationType.AUTO (selects generator depending on the database GenerationType.SEQUENCE uses a sequence, default name is hibernate_seq, (Oracle PostgreSql and other) GenerationType.TABLE uses a table to store the latest primary key value (all databases) GenerationType.IDENTITY special column table (MS SQL and other)

generator="generatorName"
Optional: References the generator which can be defined in front of the class or field.
Default: Name of the default provider, depends on the generation type (e.g. SEQUENCE uses hibernate_seq as sequence.

Sample.

```
@Entity
public class Cheetah implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;

@Entity
public class Tiger implements Serializable {
    @Id
    @TableGenerator(name = "puma_gen", table="primary_keys")
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "puma_gen")
    private Integer id;
```

@javax.persistence.IdClass

Specifies a composite primary key, i.e. a key consisting of multiple columns

value=SpottedTurtleId.class
: the class which is used for the primary key. It must have a column for each @Id column.

Sample.

```
@IdClass(SpottedTurtleId.class)
public class SpottedTurtle implements Serializable {
    @Id
    private String location;
    @Id
    private String favoriteSalad;
```

@javax.persistence.EmbeddedId

Specifies a composite primary key, i.e. a key consisting of multiple columns. As opposed to IdClass the fields are not included in the class, but only in the primary key.

Sample.

```
@Entity
public class BoxTurtle implements Serializable {

    @EmbeddedId
    private BoxTurtleId id;
```

@javax.persistence.SequenceGenerator

Specifies a generator which can be referenced from @GeneratedValue annotation. It is a HiLo algorithm using a database sequence.

name=”generatorName”,
: name, which can be referenced from @GeneratedValue

sequenceName=”dbSequenceName”,
: optional, database sequence name, default: *hibernate_id_seq*

initialValue=50,
: when the sequence is generated, defines the initial value, default: 1

allocationSize=1
: default: 50, The SequenceGenerator is in fact a HiLo sequence generator. A value is retrieved from a sequence, then the generator creates an allocationSize number of unique primary keys. With an allocation size of 5 you can create 5 entities and need only to retrieve one time the sequence value. Generators should be global but Hibernate has an implementation bug here. If you share a sequence across a number of generators, the generated id will frequently jump by the allocation size instead of incrementing. Either use one generator per class or set the allocationSize

to 1. If you insert lots of entities, for example during an import job, an allocationSize of 50, will save 49 sequence database queries per 50 inserts. This could improve performance by nearly 100 %.

Sample.

```
@Entity
@SequenceGenerator(name = "puma_seq", sequenceName = "puma_id_seq")
public class Puma implements Serializable {
```

@javax.persistence.TableGenerator

Specifies a generator which can be referenced from @GeneratedValue annotation. It is a HiLo algorithm using a database table.

name=”generatorName”,
name, which can be referenced from @GeneratedValue

table=”databaseTable”,
table, which stores the last primary key value

catalog=”databaseCatalog”,
database catalog, qualifier for table (only supported by some databases)

schema=”schemaName”,
database schema, qualifier for table (only supported by some databases)

pkColumnName=""
primary key column name identifying the generator of a class, default: sequence_name

valueColumnName=""
column holding the next hi value, default: sequence_name

pkColumnValue
primary key value, default: name of the class initialValue: default: 0 allocationSize: default: 50

uniqueConstraints={@UniqueConstraint(columnNames={"col_A, col_B"})}
Optional, no influence on Hibernate but on the table generation. It will generate unique key constraints.

Sample.

```
@Id
@TableGenerator(name = "puma_gen", table="primary_keys")
@GeneratedValue(strategy = GenerationType.TABLE, generator = "puma_gen")
private Integer id;
```

@org.hibernate.annotations.GenericGenerator

Hibernate specific generator. It references the id strategies of Hibernate Specifies a generator which can be referenced from @GeneratedValue annotation.

name=”generatorName”,
name, which can be referenced from @GeneratedValue

strategy = "seqhilo",
The strategies are explained in chapter . Can be any of the following identity, sequence, seqhilo, guid, native, select, hilo, assigned, foreign, uuid, increment

parameters = { @Parameter(name = "max_lo", value = "5") }
array of parameters. Each parameter has a name and a value.

Sample.

```
@Id
@GenericGenerator(name = "aName", strategy = "seqhilo",
   parameters = { @Parameter(name = "max_lo", value = "5") })
@GeneratedValue(generator = "aName")
private Integer id;
```

A.1.3. Column annotations

Tip: Often you do not need an annotation for your columns. By default all columns are mapped and the column name is the field name. Specify an annotation, if you have a special column (date, enum, …) or if the defaults do not suit your needs.

@javax.persistence.Column

Can be used to specify details for a column.

name="full_described_field",
database column name

unique=false,
If set to true, generates a unique key constraint, if the table is generated.

nullable=true,
If set to false, generates a not null constraint.

insertable=true,
Specifies if Hibernate can insert an object into the database.

updatable=true,
Specifies if Hibernate can update an object in the database.

columnDefinition="varchar(255)",
Can be used to explicitly define an SQL type for the column. Default: value as defined by the dialect in the Hibernate configuration.

table="tableName2",
Table name of a secondary table. If not specified the column belongs to the primary table.

length=255,
Length of the column (applies to String values only), Default: 255

precision=0,
Decimal precision, if column is a decimal value. Default: value as defined by the dialect in the Hibernate configuration.

scale=0

Decimal scale, if column is a decimal value. Default: value as defined by the dialect in the Hibernate configuration.)

@javax.persistence.Transient

Specifies that a column is not mapped to a database column.

Sample.

```
@Transient
private String transientField;
```

@javax.persistence.Basic

Optional, specifies that a field is mapped. By default all columns are mapped.

fetch=FetchType.EAGER,

Specifies if a field is loaded when the object is loaded (EAGER) or when the field is accessed (LAZY). Field based Lazy loading does only work with Bytecode instrumentation. If not enabled than the LAZY will not be recognized. Default: FetchType.EAGER

optional=true

A hint, if this field can be null.)

@javax.persistence.Lob

Specifies that a column is a large object type. Depending on the Java type a CLOB (character large object) or a BLOB (binary large object) is used. Depending on the database version and driver this might not work out of the box. See chapter about Lob Mapping Chapter 10, *Lob with Oracle and PostgreSQL*

@javax.persistence.Temporal

Precisely defines a date, time or timestamp column

value=TemporalType.DATE

Can be any of the following types: DATE, TIME, TIMESTAMP, NONE

Sample.

```
@Temporal(value=TemporalType.DATE)
private Date dateField;
```

@javax.persistence.Enumerated

Specifies an enum field. value = EnumType.STRING:: Can be one of the following values: EnumType.STRING or EnumType.ORDINAL STRING let Hibernate create a char column, holding the enum type as character, for example JUNGLE for ForestType.JUNGLE. ORDINAL let Hibernate create an integer column, holding an integer value for each enum type. The first enum type is 0, the second 1, The ORDINAL approach might be a problem, if later you want to insert enum types.

Sample.

```
public enum ForestType {JUNGLE, FOREST, NORDIC}
```

```
@Enumerated(EnumType.STRING)
private ForestType forestType;
```

@javax.persistence.Version

Specifies a version column to implement an optimistic locking strategy (see Fehler: Referenz nicht gefunden) Can be applied to the following types int, Integer, short, Short, long, Long, Timestamp. Use only one version column per class. Version column should be in the primary table (→ secondaryTable) Do not update the version column yourself. In some databases a timestamp might be not precisely enough, if your system is fast. Therefore I prefer to use int or long columns.

Sample.

```
@Version
private Long version;
```

@javax.persistence.AttributeOverride

Overwrites the column definition from a mapped super class or an embedded object.

name = "color",
    field of the class

column = @Column(name = "pullover_column")
    a column definition → see @Column

Sample.

```
 de.laliluna.component.simple.Sheep
@Embedded
@AttributeOverride(name = "color", column = @Column(name = "pullover_column"))
private Pullover pullover;
```

@javax.persistence.AttributeOverrides Overwrites multiple column definition from a mapped super class or an embedded object. value={@AttributeOverride(…), @AttributeOverride(…)}:: Array of @AttributeOverride

Sample.

```
@AttributeOverrides({@AttributeOverride(name = "color",
column = @Column(name = "pullover_column"))})
```

A.1.4. Special

@org.hibernate.annotations.AccessType

Specifies how Hibernate sets values. Hibernate can use a set method (property) or reflection to set the private field directly (field). By default, Hibernate verifies where the id is annotated. If the @Id annotation is in front of the field than field based access is assumed. If the annotation is in front of the get method then property based access is used. This annotation can be used in front of a class, a field or a property.

value=”field”
    Can be field or property

Sample.

```
@AccessType(value="field")
public class Turtle implements Serializable {
```

@org.hibernate.annotations.Formula

Can be used to get a formula calculated by the database. The column is of course read only.

value="10 * table_column + 5"
A formula being calculated in the DB.

Sample.

```
@Formula("10 * id + 5")
   public Integer formula;
```

@org.hibernate.annotations.Type Can be used to specify the type being used, instead of Hibernate let select it, based on the Java type or other annotation. You need it in rare cases. Type can be used to specify a Custom type. See TypeDefs for a sample. type=”nameOfTheType”,:: a name of type defined in a @TypeDef or a class parameters = { @Parameter(name = "paramName", value = "theValue") }:: array of parameters. Each parameter has a name and a value.

Sample.

```
@Type(type = "org.hibernate.type.BinaryType")
   private byte imageAsBytea[];
```

@org.hibernate.annotations.TypeDefs

Can be used to define a global name for a type. This is useful, if you work with custom types. value = { @TypeDef(name = "keyType", typeClass = KeyType.class) }:: array of @TypeDef

Sample.

```
import org.hibernate.annotations.Type;
import org.hibernate.annotations.TypeDef;
import org.hibernate.annotations.TypeDefs;
...... snip ......
@TypeDefs(value = { @TypeDef(name = "keyType", typeClass = KeyType.class) })
@Entity
public class YogaClub implements Serializable {

   @Type(type = "keyType")
   private String name;
```

@org.hibernate.annotations.TypeDef

name=”keyType”,
Global name of the type, to be used when properties are defined

typeClass = KeyType.class
Class of the user type

parameters = { @Parameter(name = "x", value = "y") }
Optional parameters (A custom type can be parametrized. Imagine a type providing lower case, upper case Strings, depending on a parameters.)

Sample see @TypeDefs

@org.hibernate.annotations.Parameter

Is used together with other annotations, to specify parameters. @TypeDef uses it to pass parameters to custom types. name = "x",:: name of the parameter value = "y":: value of the parameter

A.1.5. Relation annotations

@javax.persistence.OneToOne

Defines a one to one relation. Sample: de.laliluna.relation.one2one

targetEntity = Invoice1.class,
Normally guessed from the Class. Specifies the other side of the relation.

cascade = {CascadeType.ALL},
Array of cascadeTypes -see chapter explaining Cascasde

fetch=FetchType.EAGER,
Specifies if the other class is loaded, if this class is loaded. For one2one the default is EAGER. You can set it to LAZY as well.

optional=true,
Defines a not null contraint, if set to false.

mappedBy="otherSideProperty"
In a bi-directional relation, specifies that the other side manages the relation.

Sample.

```
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
...... snip .........
@OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "invoice_id")
   private Invoice1 invoice;
```

@javax.persistence.OneToMany

Specifies a one to many relation. Sample: de.laliluna.relation.one2many

targetEntity = Invoice1.class,
Can be guessed, if you use generics. Specifies the other side of the relation.

cascade = {CascadeType.ALL},
Array of cascadeTypes -see chapter explaining cascasding

fetch=FetchType.LAZY,
Specifies if the other class is loaded, if this class is loaded. For one2many the default is LAZY. You can set it to EAGER as well.

mappedBy="otherSideProperty"
In a bi-directional relation, specifies that the other side manages the relation.

Sample.

```
   @OneToMany
   @JoinColumn(name="club_id", nullable=false)
   private Set<JavaClubMember1> members = new HashSet<JavaClubMember1>();
```

@javax.persistence.ManyToOne

Specifies a one to many relation. Used with the class being the many side of the relation. Sample: de.laliluna.relation.one2many

targetEntity = Invoice1.class,
   Normally guessed from the Class. Specifies the other side of the relation.

cascade = {CascadeType.ALL},
   Array of cascadeTypes -see chapter explaining Cascasde

fetch=FetchType.EAGER,
   Specifies if the other class is loaded, if this class is loaded. For one2one the default is EAGER. You can set it to LAZY as well.

optional=true,
   Defines a not null contraint, if set to false. Default is true

Sample.

```
@ManyToOne
@JoinColumn(name = "club_id", nullable = false)
private JavaClub3 club;
```

@javax.persistence.ManyToMany

Specifies a many to many relation. Sample: de.laliluna.relation.many2many

targetEntity = Invoice1.class,
   Can be guessed, if you use generics. Specifies the other side of the relation.

cascade = {CascadeType.ALL},
   Array of cascadeTypes -see chapter explaining Cascasde

fetch=FetchType.EAGER,
   Specifies if the related objects are loaded, if this class is loaded. For many2many the default is LAZY. You can set it to EAGER as well.

mappedBy="otherSideProperty"
   In a bi-directional relation, specifies that the other side manages the relation.

Sample.

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(name = "concert_visitor_2",
   joinColumns = { @JoinColumn(name = "concert_id") },
   inverseJoinColumns = { @JoinColumn(name = "visitor_id") })
private List<Visitor2> visitors = new ArrayList<Visitor2>();
```

A.1.6. Join column annotations

@javax.persistence.JoinColumn

Used for relation mapping, specifies how tables are joined. Most values are guessed by Hibernate. Sample: de.laliluna.relation…

name = "club_id",
Column name storing the foreign key

referencedColumnName="id",
The name of the column which is referenced by the foreign key. By default the primary key column is referenced but you can specify another column as well.

unique=false,
Used, when Hibernate generates tables. This will generate a unique key constraint.

nullable = true,
Used, when Hibernate generates tables. This will generate a not null constraint.

insertable=true,
Specify if this column can be inserted, if an object is saved.

updatable=true,
Specify if this column can be updated, if an object is saved.

columnDefinition="int4",
The SQL type of the foreign key column.

table=""
If you use a secondary table, you can specify in which table your foreign key is placed. → see secondary table.

Sample.

```
   @ManyToOne
   @JoinColumn(name = "club_id", nullable = false)
   private JavaClub3 club;
```

@javax.persistence.JoinColumns

Is used to join classes with composite primary keys. Sample: *de.laliluna.component.joincolumns*

value = {@JoinColumn}
array of @JoinColumn

Sample.

```
@OneToOne
   @JoinColumns({
      @JoinColumn(name="articleGroup", referencedColumnName="articleGroup"),
      @JoinColumn(name="articleNumber", referencedColumnName="articleNumber") })
   private ElbowRest elbowRest;
```

@javax.persistence.JoinTable

Used to join a relation with a separate table. Sample: *de.laliluna.component.one2many* and *many2many*

name = "club_member",
database table name used for joining

catalog="",
database catalog (only supported by some databases)

schema="",
database schema (only supported by some databases)

joinColumns = { @JoinColumn(name = "member_id") },
Array of joinColumn. Defines the JoinColumns on this side of the relation. You need multiple columns for composite ids.

inverseJoinColumns = { @JoinColumn(name = "club_id") },
Array of joinColumn. Defines the JoinColumns on the other side of the relation. You need multiple columns for composite ids.

uniqueConstraints={@UniqueConstraint(columnNames={"club_id"})}
If you let Hibernate create tables, this information is used to create unique key constraints.

Sample.

```
@ManyToOne
   @JoinTable(name = "club_member",
      joinColumns = { @JoinColumn(name = "member_id") },
      inverseJoinColumns = { @JoinColumn(name = "club_id") })
   private JavaClub4 club;
```

@javax.persistence.PrimaryKeyJoinColumn

Defines how a table is joined to another table. Sample: *de.laliluna.relation.one2one*

name = "id",
Name of the column used in the secondary table. (Foreign key) Default is the name of the primary key column.

referencedColumnName = "id",
Name of the column used in the primary table. (Primary key). Default values are taken from the primary key definition.

columnDefinition = "int4"
SQL type of the foreign key column. Default values are taken from the primary key column.

Sample.

```
   @OneToOne(cascade = CascadeType.ALL,optional=false)
   @PrimaryKeyJoinColumn
   private Order3 order;
```

@javax.persistence.PrimaryKeyJoinColumns

Specifies how a table is joined. Can be used, if you have a composite id.

value={@PrimaryKeyJoinColumn}
: array of @PrimaryKeyJoinColumn

See the PrimaryKeyJoinColumn sample in *de.laliluna.relation.one2one* and the composite sample *de.laliluna.component.joincolumns* to get an impression, when this is used.

A.1.7. Components

@javax.persistence.Embedded

Specifies that another class is embedded in this class. The fields of the embedded class can be mapped to the same table.

Hint: In a newer EJB implementation the Embedded annotation has an array of @AttributeOverride. Hibernate might adopt this as well. example:

Sample (de.laliluna.component.simple.Sheep).

```
@Embedded
@AttributeOverrides({@AttributeOverride(name = "color",
   column = @Column(name = "pullover_column"))})
private Pullover pullover;
```

@javax.persistence.ElementCollection()

Maps a collection of components.

targetClass
: Target class, optional, normally taken from generic declaration

fetch=FetchType.EAGER
: Specifies how the data is fetched (EAGER | LAZY), default: LAZY

Sample (de.laliluna.component.collection1, de.laliluna.component.collection2).

```
@ElementCollection()
@JoinTable(name="pizza_ingredients", joinColumns =
@JoinColumn(name="pizza_id"))
private Set<Ingredient> ingredients = new HashSet<Ingredient>();
```

@org.hibernate.annotations.CollectionOfElements

Is now deprecated. Use *@javax.persistence.ElementCollection()* instead.

@javax.persistence.Embeddable Specifies that this class can be embedded into other classes.

@org.hibernate.annotations.Parent

Can be used in an embedded class, if you need a reference to the parent.

Sample.

```
import org.hibernate.annotations.Parent;
```

```
@Embeddable
public class DeliveryAddress implements Serializable {
   @Parent
   private PizzaClient client;
```

A.1.8. Inheritance

@javax.persistence.Inheritance

Specifies an inheritance mapping, ie. an inheritance structure of classes is mapped to 1 or more tables. strategy = InheritanceType.SINGLE_TABLE:: Specifies how the inheritance structure is mapped SINGLE_TABLE = All columns from parent and derived classes in the same table. TABLE_PER_CLASS = One table per parent or derived class. The derived tables have the columns from the parent class as well. JOINED = One table per parent or derived class. The parent table holds the data of common columns.

Sample (de.laliluna.inheritance.*).

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "plant_type",
   discriminatorType = DiscriminatorType.STRING)
public class Plant implements Serializable {

@Entity
public class Flower extends Plant {
```

@javax.persistence.DiscriminatorColumn

Is used with the inheritance type InheritanceType.SINGLE_TABLE. As all classes are stored in one table we need a column which defines what kind of class a row belongs to.

name = "plant_type",
: Database column name

discriminatorType = DiscriminatorType.STRING,
: Specifies the approach how the discriminator is stored. STRING gives an easily readable representation. Can be any of STRING, CHAR, INTEGER

columnDefinition="varchar(31)",
: optional, database column type, normally taken from the discriminatorType

length=20
: length of database column, default: 31

Sample (de.laliluna.inheritance.singletable.*).

```
@DiscriminatorColumn(name = "plant_type",
   discriminatorType = DiscriminatorType.STRING,
   columnDefinition="varchar(31)", length=31)
```

@javax.persistence.DiscriminatorValue

Optional, can be used with the inheritance type InheritanceType.SINGLE_TABLE, default: class name

value=”Flower”
: unique name specifying a derived class

Sample.

```
@Entity
@DiscriminatorValue("Flower")
public class Flower extends Plant {
```

@javax.persistence.MappedSuperclass

Specifies an inheritance mapping, where the parent class is not mapped itself. example: *de.laliluna.inheritance.mappedsuperclass.**

Sample.

```
@MappedSuperclass
public class MusicBand implements Serializable { .......

@Entity
@SequenceGenerator(name = "musicband_seq", sequenceName = "musicband_id_seq")
public class SoftrockGroup extends MusicBand{ ......
```

A.1.9. Queries

@javax.persistence.NamedQueries

Defines multiple named queries. See example below.

@javax.persistence.NamedQuery

Defines a query which can be reused at various places. Provides a performance advantage to building queries every time. example: *de.laliluna.other.namedquery.**

Sample.

```
@NamedQueries({
   @NamedQuery(name = "bookQuery", query =
   "from ComputerBook b where b.id > :minId and b.name = :name",
    hints = {
        @QueryHint(name = "org.hibernate.readOnly", value = "false"),
        @QueryHint(name = "org.hibernate.timeout", value = "5000")})
    })
```

Usage.

```
List<ComputerBook> list = session.getNamedQuery("bookQuery")
   .setString("name", "Hibernate")
   .setInteger("minId",10)
    .list();
```

@org.hibernate.annotations.NamedQueries

Defines multiple named queries. See example below. The related chapter explains the difference to *@javax.persistence.NamedQueries*.

@org.hibernate.annotations.NamedQuery

Defines a query which can be reused at various places. Provides a performance advantage to building queries every time. example: *de.laliluna.other.namedquery.**

Sample.

```
@org.hibernate.annotations.NamedQueries({
   @org.hibernate.annotations.NamedQuery(name = "bookQuery", query =
   "from ComputerBook b where b.id > :minId and b.name = :name",
   flushMode = FlushModeType.AUTO,
    cacheable = true, cacheRegion = "", fetchSize = 20, timeout = 5000,
    comment = "A comment", cacheMode = CacheModeType.NORMAL,
    readOnly = true)})
```

Usage.

```
List<ComputerBook> list = session.getNamedQuery("bookQuery")
   .setString("name", "Hibernate")
   .setInteger("minId",10)
    .list();
```

@javax.persistence.SqlResultSetMappings

Defines multiple result set mappings.

@javax.persistence.SqlResultSetMapping

If you use SQL in a query instead of HQL, you can still create entities as result. A result set mapping tells Hibernate how to transform the SQL result set into entities.

example: *de.laliluna.other.namedquery.**

Sample.

```
@SqlResultSetMappings({
    @SqlResultSetMapping(name = "bookReport", entities = {@EntityResult
        (entityClass = ComputerBook.class,
            fields = {@FieldResult(name = "id", column = "id"),
                @FieldResult(name = "name", column = "book_name")})})
})
```

Usage.

```
List<ComputerBook> books = session.createSQLQuery
    ("select id, book_name from computerbook")
    .setResultSetMapping("bookReport").list();
```

A.1.10. Not yet described

@javax.persistence.MapKey @javax.persistence.OrderBy @org.hibernate.NotFound @OnDelete @javax.persistence.OrderBy cache DiscriminatorFormula LazyToOne, LazyCollection, Fetch Batchsize, check, where indexcolumn, mapkey

Filter

Filter, FilterDef

Index