Hibernate

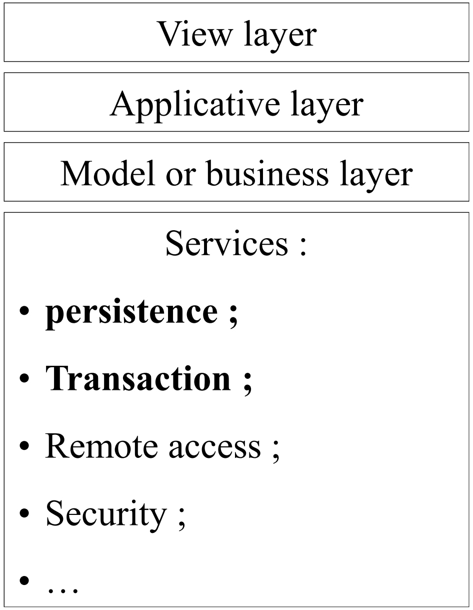
Reda Bendraou

Reda.Bendraou@lip6.fr

http://pagesperso-systeme.lip6.fr/Reda.Bendraou/

This course is inspired by the readings/sources listed in the last slide

Persistence service



View layer

Applicative layer

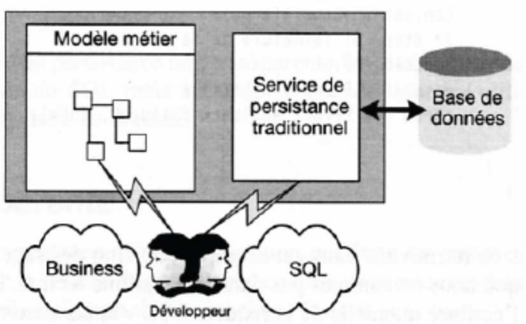
Model or business layer

Services :

- **persistence ;**
- **Transaction ;**
- Remote access ;
- Security ;
- …

Non-Transparent Persistency

- It is up to the developer to code the data access
- Object code mixed with SQL, Exception proper to DB connectivity (ex. SQL exceptions, etc.)
- Not natural object oriented programming
- Needs both expertise in the OO and in writing sound and optimal SQL requests



Non-Transparent Persistency: persistence with JDBC

```
Connection connexion = null;
try{
        //needs the driver to be loaded first using the class for name method + catch exceptions
        Connection connexion = DriverManager.getConnection( baseODBC, "", "");
        connexion.setAutoCommit( false );
        Float amount = new Float( 50 );
        String request = "UPDATE Bank SET balance=(balance -1000) WHERE holder=?";
        PreparedStatement statement = connexion.prepareStatement(request );
        statement.setString( 1, “name” );
        statement.executeUpdate();
        client2.check();      // throws an exception
        request = "UPDATE Bank SET balance =(balance +1000) WHERE holder =?";
        statement = connexion.prepareStatement(request );
        statement.setString( 1, “...” );
        statement.executeUpdate();
        connexion.commit() ;
} catch( Exception e ){
        connexion.rollback() ;
}
```

JDBC drawback

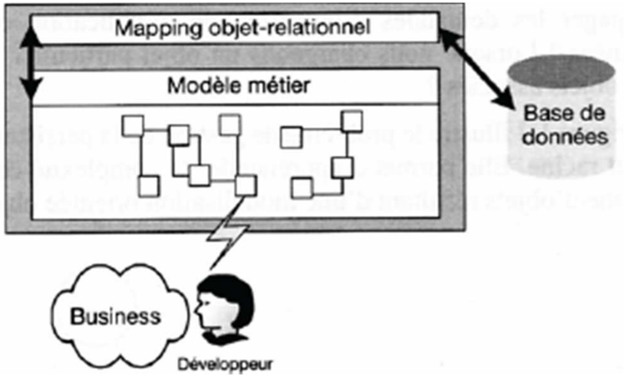
- Strong link between the object layer and the persistence layer:

- Any change in the object layer lead to write again SQL requests.

Transparent Persistency: ORM tools

- Natural OO programming. Developer does not have to deal with the persistency layer
- Less code related to data access, exceptions (up to 40% less)
- The SQL generated by the ORM tools has been proven to be more optimal than most developer's hand-written SQL requests

eclipse-link : framework
open source de
TOPLINK



Object-relational mapping with Hibernate 1/2

```
Session session = null;
    try{
SessionFactory sessionFactory =
new Configuration().configure().buildSessionFactory();

session = sessionFactory.openSession();

    //begin a transaction
    org.hibernate.Transaction tx = session.beginTransaction();
//create a contact and save it into the DB
    Contact contact = new Contact();
contact.setId(1);
    contact.setFirstName("Robbie");

    //save the contact into the DB
session.save(contact); // or session.persist(contact);
    //if you modify one of its properties, no need to save it again
contact.setFirstName("Robin");
    //mandatory to flush the data into the DB
    tx.commit();
}catch(Exception e){
      System.out.println(e.getMessage());
    }
```

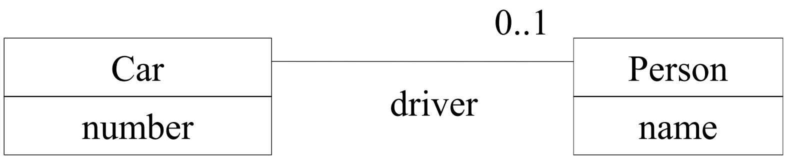
Contient tout ce qu'il faut savoir pour se conecter et communiquer avec le SGBD

Classes à connaitre : Session et SessionFactory

Cache de 1er niveau : Consomme beaucoup d'espace si mal géré.

Espace mémoire qui contient toutes informations → thread
liées au processus qui accède à la BD → portée session

Object-relational mapping with Hibernate 2/2



Serveur Application :
Possibilités de faire
des transactions réparties

Table Car	
NUMBER (pk)	PERSONNE_ID (fk)
234 GH 62	2
1526 ADF 77	1

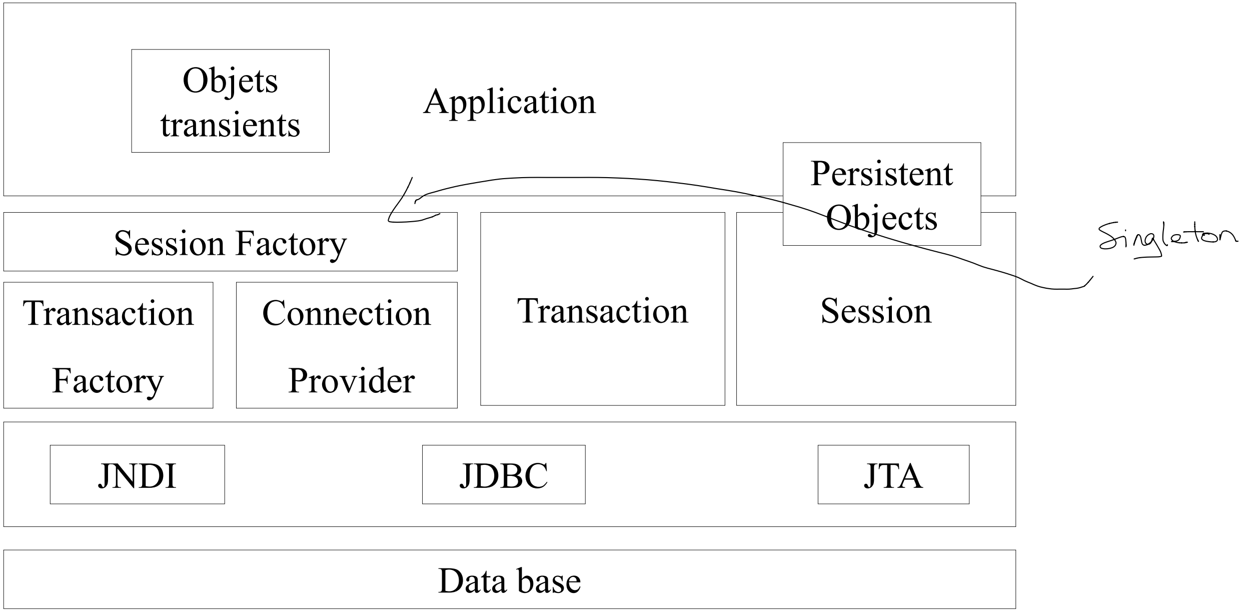
Table PERSON	
PERSONNE _ID(pk)	NAME
1	Dupond
2	Tintin

Hibernate : major points

- Pure object model above a database
- Association and inheritance are taken into account ;
- Mask different SQL implementations ;
- Two majors services :
  - Persistence
  - Transactions.
- Open source.

Hibernate architecture

Hibernate architecture



Hibernate Core

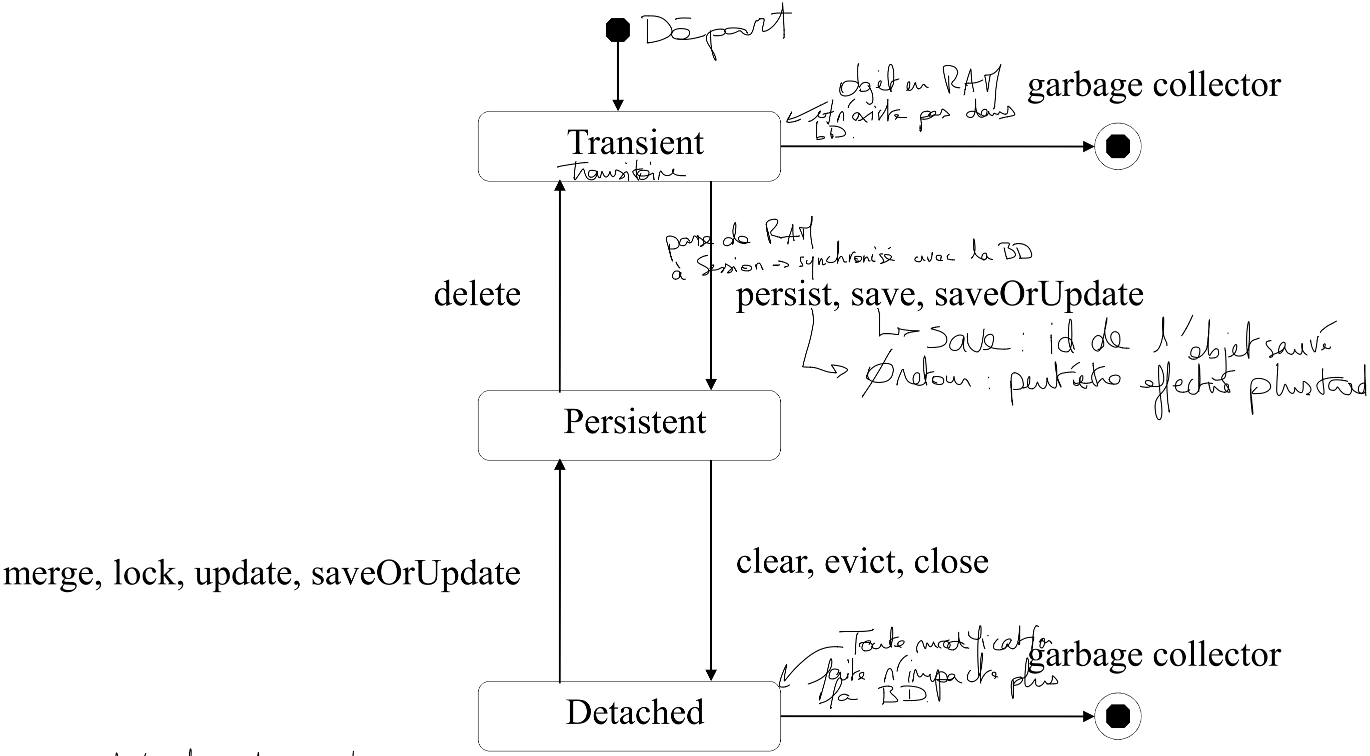
Session Factory

- Knows the mappings between classes and tables
- Provider of Sessions
- Heavy Object (apply singleton pattern)

Session

- Bridge between you application and the data base
- Allows CRUD operations on application’s objects
- Masks the JDBC Layer
- This is the first cache level
- Light Object

Objects life cycle within Hibernate



Départ

objet en RAM
n'existe pas dans BD.

garbage collector

Transient
Transitoire

delete

passe de RAM à Session -> synchronisé avec la BD

persist, save, saveOrUpdate

-> save : id de l'objet sauvé

-> ∅ retour : peut être effectué plus tard

Persistent

merge, lock, update, saveOrUpdate

clear, evict, close

Toute modification faite n'impacte plus la BD.

garbage collector

Detached

Un objet se détache très vite

-> Reflexe : être sur que l'objet est bien dans une session

objects life cycle management

- **object persistence :**

```
Person person = new Person();
person.setName( "Tintin" );
Long generatedID = (Long)session.save( person );
```

transient state

persistant state

```
session.save( person, new Long( 1234 );
```

- **Object loading :**

```
Person person = (Person)session.load( Person.class, generatedID );
```

```
Person person = new Person();
session.load( person, generatedID );
```

```
Person person = (Person)session.get( Person.class, id );
if( person == null ){
 person = new Person();
session.save( person, id );
}
```

```
session.refresh( person );
```

Object reloading.

Hibernate configuration

Hibernate configuration: the hibernate.cfg.xml file

```
<?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>
 <!– data base connection details-->
       <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
       <property name="hibernate.connection.url">jdbc:mysql://localhost/your_DB_name</property>
       <property name="hibernate.connection.username">your_db_password</property>
<property name="hibernate.connection.password">root</property>

       <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

 <!– here the value update means that the data base schema will not be created each time you run the
application but it will be just updated. To create it each time, put “create” instead-->

<property name="hbm2ddl.auto">update</property>

<!– link to mapping files -->

<mapping resource="org/lip6/hibernate/tuto/Contact.hbm.xml"/>
   </session-factory>

</hibernate-configuration>
```

Important: One file by project. In your root source package

- To get a SessionFactory :

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Hibernate configuration : Programmatic Way

- A program to define configuration properties :

```
Configuration cfg = new Configuration()
.addResource(“Content.hbm.xml")
.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
.setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test");
```

- A file (hibernate.properties) to define configuration properties :

```
hibernate.dialect org.hibernate.dialect.HSQLDialect
hibernate.connection.datasource java:comp/env/jdbc/test
```

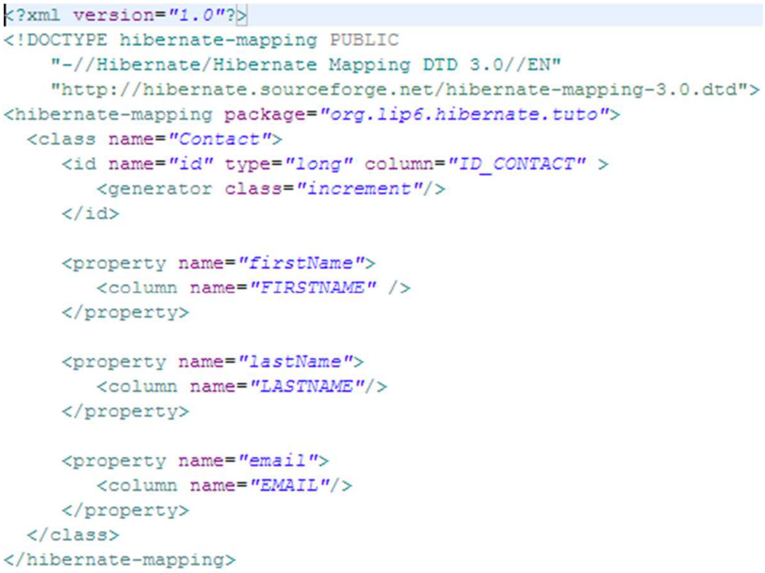
Hibernate configuration

- Many other properties:
  - JDBC configuration (autocommit, …) ;
  - Hibernate optimization (cache management, …) ;
  - …
  - To display SQL requests in the console:

```
hibernate.show_sql   true
```

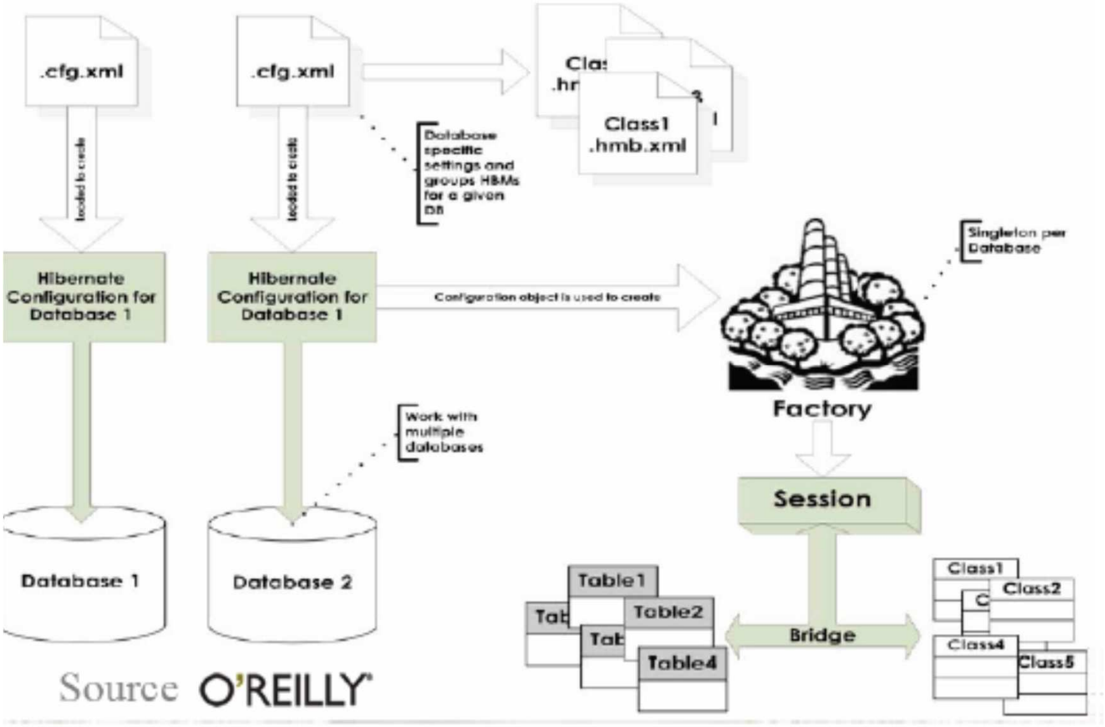
Hibernate: Mapping File

- Defines how a class will be mapped (made persistent) in the database
- File in XML format
- To put in the same package as the source class. Usually named ***MyClass*.hbm.xml** if the class is called ***MyClass***
- Introduced in more details further



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="org.lip6.hibernate.tuto">
  <class name="Contact">
     <id name="id" type="long" column="ID_CONTACT" >
        <generator class="increment"/>
     </id>

     <property name="firstName">
        <column name="FIRSTNAME" />
     </property>

     <property name="lastName">
        <column name="LASTNAME"/>
     </property>

     <property name="email">
        <column name="EMAIL"/>
     </property>
  </class>
</hibernate-mapping>
```

Hibernate cfg and hbm files: Principle


.cfg.xml
.cfg.xml
loaded to create
loaded to create
Class1
.hmb.xml
Database specific settings and groups HBMs for a given DB
Hibernate Configuration for Database 1
Hibernate Configuration for Database 1
Configuration object is used to create
Singleton per Database
Factory
Work with multiple databases
Session
Database 1
Database 2
Table1
Table2
Table4
Bridge
Class1
Class2
Class4
Class5

Source O'REILLY

Persistent Classes

Coding rules

- POJO allowed (Plain Old Java Object):
    - a constructor without argument ;
    - An ID property (used to define the primary key in the database table) :
        - int, float, … ;
        - Interger, Float, … ;
        - String or Date ;
        - a class which contains one of the above types.

Coding rules

- Other minor rules (recommended) :
  - no *final* class :
  - Accessors for persistent fields :
    - Persistent fields can be private, protected or public.
  - inheriting classes :
    - Must have a constructor without argument ;
    - Can have an identification property.

Object/Relational mapping basis

Java class

```
package familly;
public class Person{
        private Long id;
        private Date birthday;
        Personne mother;
        private void setId( Long id ){
                this.id=id;
        }
        public Long getId(){
                return id;
        }
        void setBirthday(Date date){
                birthday = date;
        }
        public Date getBirthday(){
                return birthday;
        }
        void setMother( Person mother){
                this. mother = mother;
        }
        public Person getMother() {
                return mother ;
        }
}
```

Mapping file

- mapping file example :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="familly">

<class name="Person" table="persons" >

   <id name="id"> <generator class="native"/> </id>

   <property name=" birthday " type="date" not-null="true" update="false"/>

   <many-to-one name="mother" column=" mother_id" update="false"/>

</class>

</hibernate-mapping>
```

Class element attributes

```
<class
        name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        proxy="ProxyInterface"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        where="arbitrary sql where condition«
        optimistic-lock="none|version|dirty|all"
        lazy="true|false"
        node="element-name"
        …
/>
```

Required

Optional, the name of the class if not provided

For inheritence

Interface for lazzy loading proxy.

Only changed values are taken into account.

Check the column timestamp

All columns are checked

Main attributes for the id element

Name of the primary key column.

```
<id
        name="propertyName"
        type="typename"
        column="column_name"
        <generator class="native"/>
</id>
```

- Strategy to generate the key:
  - native : depends on the database
  - increment
  - UUID : choice of an algorithm
  - ...

Some Strategies for generating the id

increment
generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table.

identity
supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.

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

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

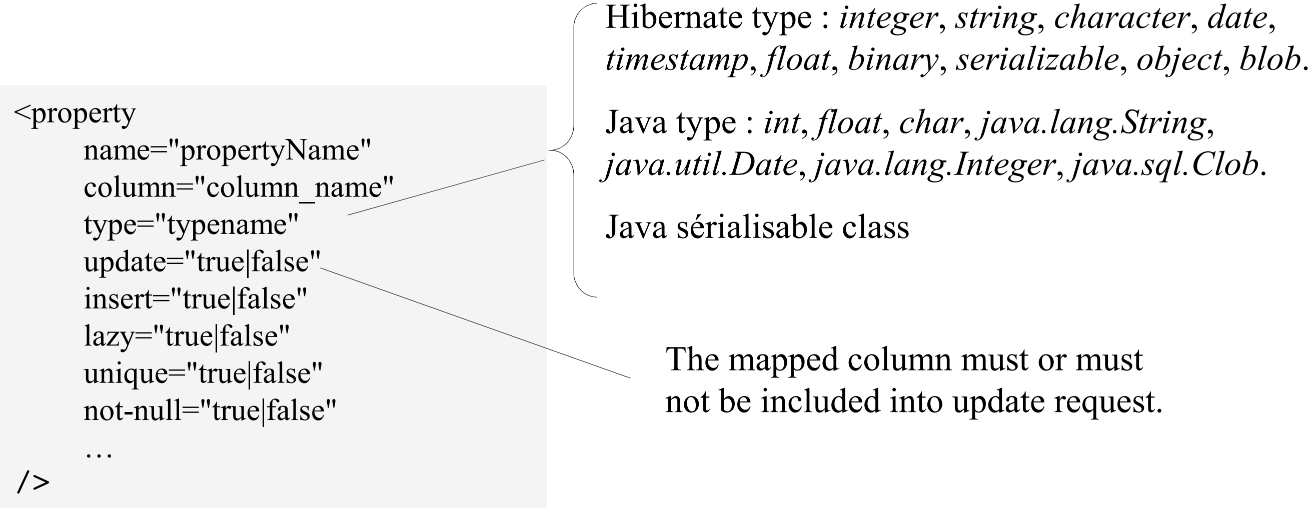
native
selects identity, sequence or hilo depending upon the capabilities of the underlying database.

assigned
lets the application assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified.

select
retrieves a primary key, assigned by a database trigger, by selecting the row by some unique key and retrieving the primary key value.

foreign
uses the identifier of another associated object. It is usually used in conjunction with a <one-to-one> primary key association.

Main attributes for the property element



```
<property
        name="propertyName"
        column="column_name"
        type="typename"
        update="true|false"
        insert="true|false"
        lazy="true|false"
        unique="true|false"
        not-null="true|false"
        …
/>
```

Hibernate type : *integer*, *string*, *character*, *date*, *timestamp*, *float*, *binary*, *serializable*, *object*, *blob*.

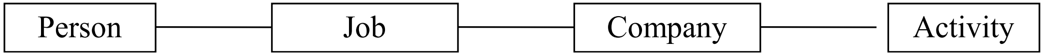
Java type : *int*, *float*, *char*, *java.lang.String*, *java.util.Date*, *java.lang.Integer*, *java.sql.Clob*.

Java sérialisable class

The mapped column must or must not be included into update request.

Lazy loading

- Example :



```
Person person  = (Person)session.get(Person.class, new Long(1));
```

The association with Job is set to null !

```
Activity a = person.getJob().getCompany().getActivity();
```

The objects graph is loaded

Association mappings

Associations

The more complex part when using Hibernate

Type: Uni or bidirectional

Multiplicities of associations

1-1, 1-N, N-1, M-N

Hibernate Tags for associations

For collections : **<set>**, **<list>**, **<map>**,**<bag>**, **<array>** et **<primitive-array>**

For multiplicities : **<one-to-one>**, **<one-to-many>**, **<many-to-one>**,**<many-to-many>**

Association many-to-one (N-1)

Case: **unidirectionnelle** Employé –> Entreprise :

```
<class name="Employe">
      <id name="id" column="ID_EMPLOYE">
      . . .
      </id>
      . . .
      <many-to-one name="entrp" column=" ID_ENT"
      class="Entreprise"/>
</class>
```

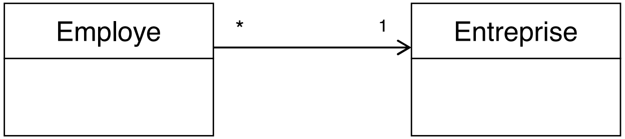
Property name in class Employe

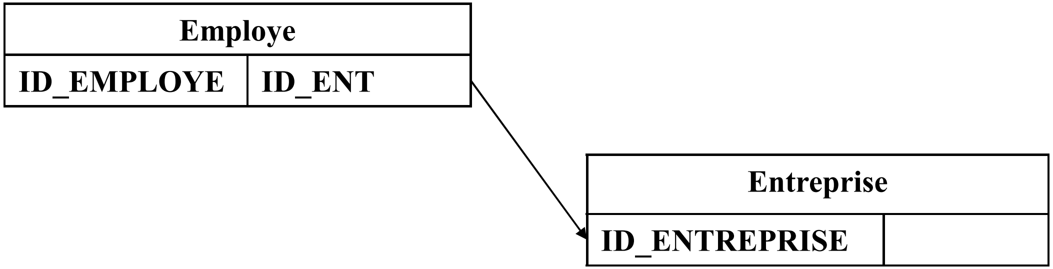
Property type

The column name for this property in the generated table. By default, the property name in the class

Association many-to-one (N-1)

A table references another table via a foreign key





Employe	
ID_EMPLOYE	ID_ENT

Entreprise	
ID_ENTREPRISE	

Association one-to-many (1-N)

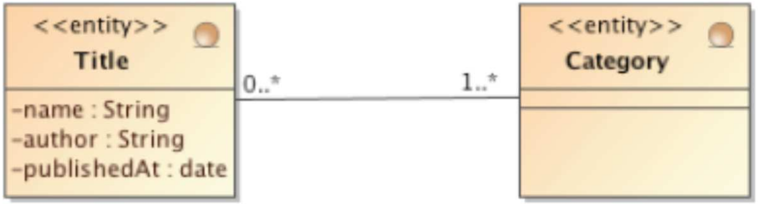
If the previous association should be bidirectional

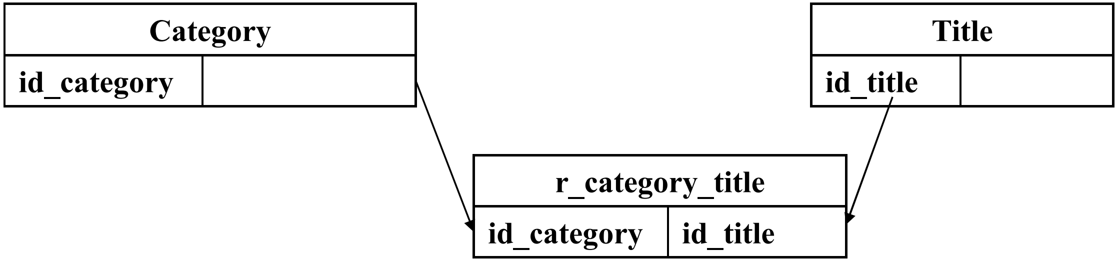
```
<class name="Entreprise" table="ENTREPRISE">
    . . .
    <set name="employes"  inverse=" true ">
        <key column="ID_ENT"/>
        <one-to-many class="Employe"/>
    </set>
    </class>
```

Column of table
EMPLOYE (not **ENTREPRISE!!!!**)
foreign key towards
ENTREPRISE

Association many-to-many

Need for an intermediary table.





Category

id_category

Title

id_title

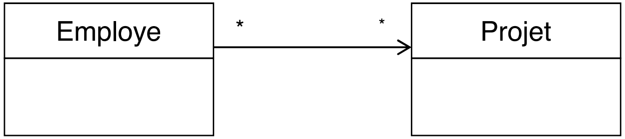
r_category_title

id_category | id_title

Association many-to-many (M-N)

case: Unidirectional

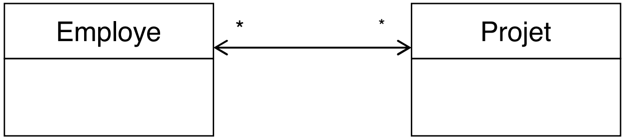
A new table is needed to link the two tables



```
<class name="Employe">
. . .
<set name="projets"   table="PARTICIPATION">
      <key column="ID_EMPLOYE"/>
      <many-to-many class="Projet" column="ID_PROJET"/>
</class>
```

Association many-to-many (M-N)

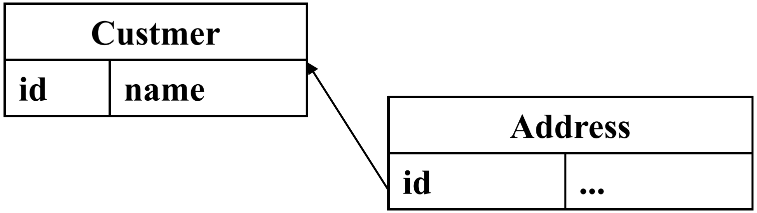
case: Bidirectional



```
<class name="Employe">
. . .
<set name="projets"   table="PARTICIPATION">
      <key column="ID_EMPLOYE"/>
      <many-to-many class="Projet" column="ID_PROJET"/>
</class>
<class name="Projet">
. . .
<set name="employes"   table="PARTICIPATION" inverse="true">
      <key column="ID_PROJET"/>
      <many-to-many class="Employe" column="ID_EMPLOYE"/>
</class>
```

Association one-to-one (1-[0..1])

Both tables share the same primary key (used for implementing a 1-[0..1]link).



Custmer

id	name

Address

id	...

Association one-to-one

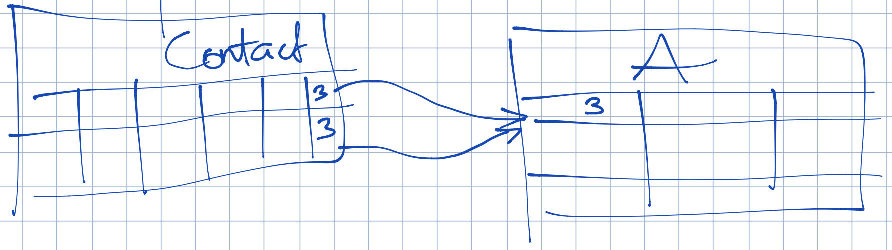
case: Unidirectional

Two ways. The 1rst one by using many-to-one

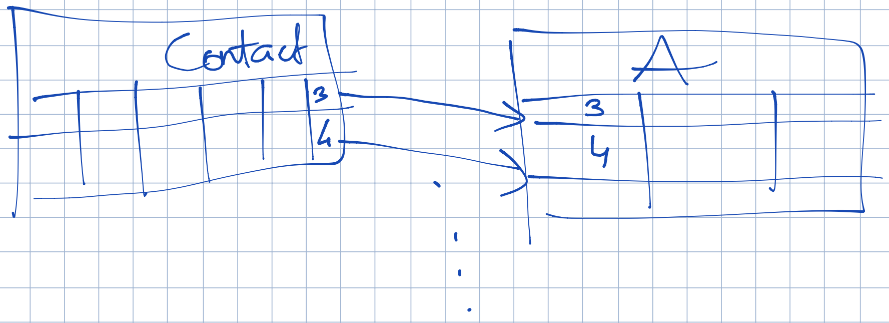
Exp. a person has a unique address

```
<class name="Person">
    <id name="id" column="personId">
            <generator class=“increment"/>
    </id>
    <many-to-one name="address" column="addressId" unique="true" not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
            <generator class=“increment"/>
    </id>
</class>
```

many to one



one to one



Association one-to-one

case: Unidirectional

The second way by using one-to-one

Exp. a person has a unique address

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="increment"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>     </generator>
    </id>
<one-to-one name="person" constrained="true"/>
</class>
```

ici on choisit de générer l'id @ == idContact

↳ ne peut pas être null

Association one-to-one

case: Bidirectional

Two ways. The 1rst one by using many-to-one

Exp. a person has a unique address

```
<class name="Person">
    <id name="id" column="personId">
        <generator class=« increment"/>
    </id>
    <many-to-one name="address" column="addressId" unique="true" not-
    null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class=« increment"/>
    </id>
    <one-to-one name="person" property-ref="address"/>
</class>
```

Association one-to-one

case: Bidirectional

The second way by using one-to-one

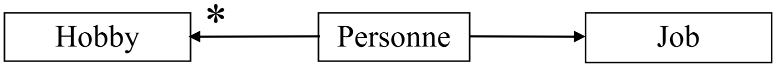
Exp. a person has a unique address

```
<class name="Person">
     <id name="id" column="personId">
          <generator class=« increment"/>
     </id>
     <one-to-one name="address"/>
</class>

<class name="Address">
     <id name="id" column="addressId">
          <generator class="foreign">
                    <param name="property">person</param>          </generator>
     </id>
     <one-to-one name="person" constrained="true"/>
</class>
```

Transitive Persistence of objects

The persistence of associated objects

- Starting from the class diagram:



- How to persist associated objects ?
  - First solution : explicitly persist all the instances.

```
Person person = new Person();
Hobby hobby = new Hobby();
Job job = new Job();
person.addHobby( hobby );
person.setJob( job );

Session session = HibernateUtil.getSession();
Transaction tx = session.beginTransaction();
session.persist( person );
session.persist( hobby );
session.persist( job );
tx.commit();
tx.close();
```

← ⚠ Sauvegarde dans le bon ordre

Phone numbers: all-delete-orphan.

The persistence with Cascade

- Second solution: to use *cascade*.

```
<class name="Person">
   <id name="id" column="personId">
      <generator class="native"/>
   </id>
  <many-to-one name=“job" class= " Job " column= " jobId"
cascade=“persist"/>
   <set name="hobbies" fetch="select" cascade="persist">
      <key column="personId"/>
      <one-to-many class="Hobby"/>
   </set>
</class>


<class name=“Hobby" lazzy="true">
   …
</class>

<class name=“Job" lazzy="true">
   …
</class>
```

The persistence with Cascade

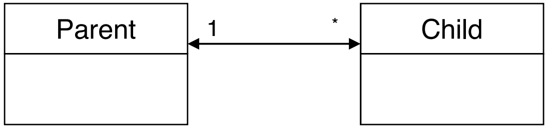
- The following program:

```
Person person = new Person();
Hobby hobby = new Hobby();
Job job = new Job();
person.addHobby( hobby );
person.setJob( job );

Session session = HibernateUtil.getSession();
Transaction tx = session.beginTransaction();
session.persist( person );
tx.commit();
tx.close();
```

- Allows to propagate the persistence to the instances job and hobby.

Transitive Persistence - Cascade

all-delete-orphan



```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

- **Will not remove c from the database.**
- **Only removes the link to p and causes a NOT NULL constraint violation.**

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

- Child cannot exist without its parent.
- So if we remove a Child from the collection, we do want it to be deleted. To do this, we must use cascade="all-delete-orphan".

```
<set name="children" inverse="true" cascade="all-delete-orphan">
```

Collections mapping

Collections persistence

- Hibernate supports the following collections:
  - *java.util.Set*, *java.util.Collection*, *java.util.List*, *java.util.Map*, *java.util.SortedSet*, *java.util.SortedMap*,
  - a collection defined by a user *org.hibernate.usertype.UserCollectionType*.

- Attention!
  - Must always use interfaces
  - Two persistent instances can not share the same collection => duplicate the collection.
  - A collection attribute can not be null.
  - Collection should be instanciated right away
    - Not delegated to constructor method or setters

      List MyList=new ArrayList();

Simple type Collections

Exp. a Set containing String(s)

```
<set name="names" table="person_names">
        <key column="person_id"/>
        <element column="person_name" type="string"/>
</set>
```

Exp. a Set containing Integer(s) ordered by « size »

```
<bag name="sizes" table="item_sizes" order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

Tips for Bidirectional Maintenance

Bidirectional Maintenance

Developers must maintain bidirectional associations in order to keep in-memory objects up-to-date

```
aParent.getChildren().add(aChild);
aChild.setParent(aParent);
```

Hibernate recommends a strategy to ensure this process=>

- ✓ **Maintain associations on a single side of the relationship**

May not be effective in all situations

- ✓ 1:1, M:M,
- ✓ redefinition of HashCode(), equals(), contains, etc?
- ✓ **Proposition of another strategy**

Bidirectional Strategy M:M

Between Account and Ebiller classes

EBiller Add Method

```
public void addAccount(Account account){
        this.accounts.add(account);

        if (!account.getEbillers().contains(this)) {
            account.addEbiller(this);
        }
}
```

EBiller Remove Method

```
        public void removeAccount(Account account) {
            this.accounts.remove(account);
        if (account.getEbillers().contains(this)) {
            account.removeEbiller(this);
        }
}
```

Bidirectional Strategy M:M

Between Account and Ebiller classes

Account Add Method

```
public void addEbiller(EBiller ebiller) {
        this.ebillers.add(ebiller);
        if (!ebiller.getAccounts().contains(this)) {
        ebiller.addAccount(this);
        }
}
```

Account Remove Method

```
public void removeEbiller(EBiller ebiller) {
        this.ebillers.remove(ebiller);
        if (ebiller.getAccounts().contains(this)) {
        ebiller.removeAccount(this);
        }
}
```

Bidirectional Strategy 1:1

Between Ebill and Transaction classes

Need to handle potential null object on each side

- **Objects might not be initialized**
- **Example: EBill has one Transaction / Transaction has one EBill**

EBill Set Method

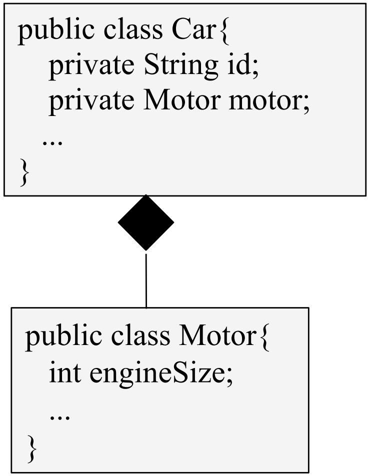
```
public void setTransaction(Transaction transaction) {
this.transaction = transaction;
if (transaction !=null && (transaction.getEbill() == null||!transaction.getEbill().equals(this)))
{
    transaction.setEbill(this);
}
}
```

Transaction Set Method

```
public void setEbill(EBill ebill) {
this.ebill = ebill;
if (ebill != null && (ebill.getTransaction() == null || !ebill.getTransaction().equals(this))) {
ebill.setTransaction(this);
}
}
```

Compositions mapping

Composition persistence = persistence by value



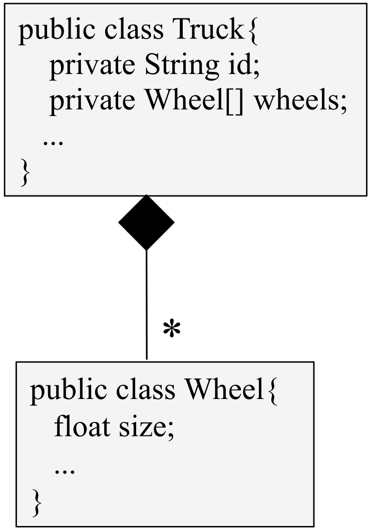
```
public class Car{
   private String id;
   private Motor motor;
  ...
}
```

```
public class Motor{
  int engineSize;
  ...
}
```

```
<class name=“Car" table=“car">

   <id name="id" column=“carId"
type="string">
      <generator class=“increment"/>
   </id>

   <component name="Motor" class="Motor">
      <property name=" engineSize"/>
   </component>

</class>
```

- The car table has two columns : carID and engineSize.

Multiple composition persistence



```
public class Truck{
   private String id;
   private Wheel[] wheels;
  ...
}
```

```
public class Wheel{
   float size;
   ...
}
```

```
<class name=“Truck" table=“truck">

   <id name="id" column=“carId" type="string">
      <generator class="uuid"/>
   </id>

   <set name=" wheels" table="wheels"
lazy="true">
      <key column="id"/>
      <composite-element class="Wheel">
        <property name="size"/>
      </composite-element>
   </set>

</class>
```

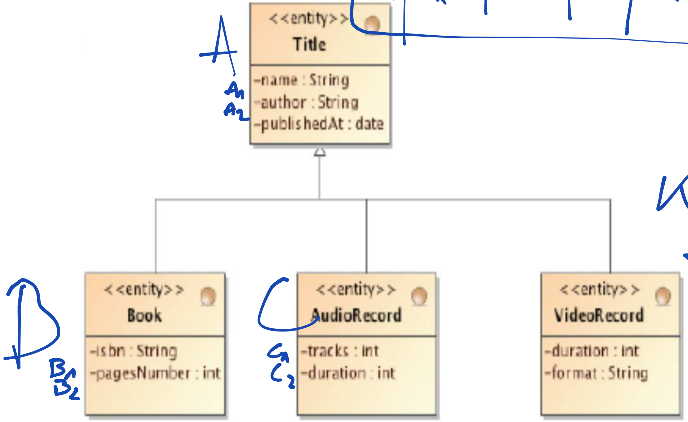
Inheritance mapping

Inheritance

- Hibernate supports 3 strategies for inheritance:
  - a table per class hierarchy;
  - a table per subclass ;
  - a table per concrete class.

A

A1	A2	B1	B2	C1	C2	Kind
X	X					A
X	X	X	X			B
X	X			X	X	C



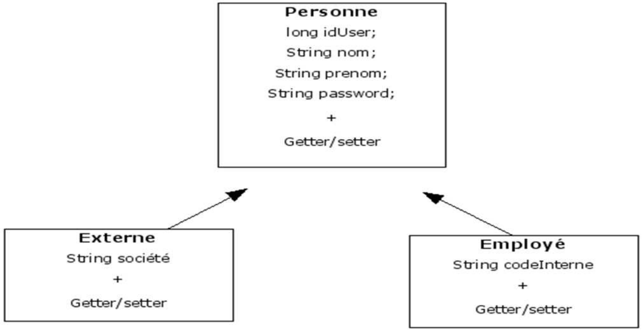
Kind : spécifie le type de l'objet (discriminant) pour pas que l'on puisse confondre.

One table per class hierarchy

name	author	isbn	discriminator
XML pour les nuls	Pascal Gerard	12121	B
Thriller	Mickeal Jackson		A

- Only one table is used ;
- A discriminator column is added to the table
- Constraint : subclass columns can not have not-null constraint.

One table per class hierarchy



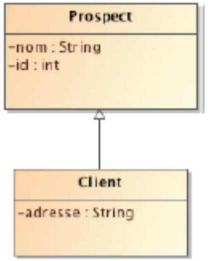
Personne
long idUser;
String nom;
String prenom;
String password;
+
Getter/setter

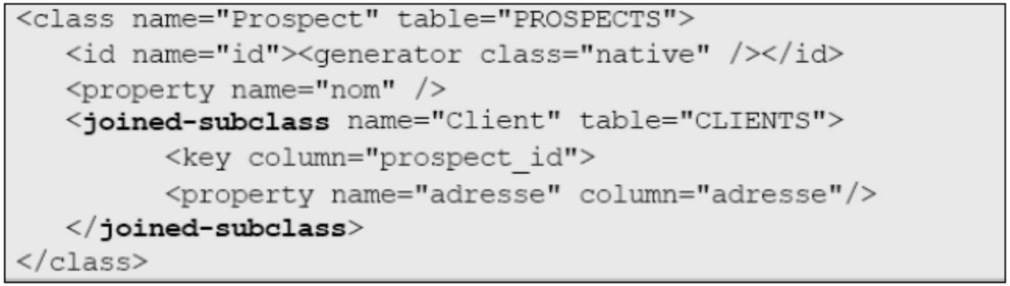
Externe
String société
+
Getter/setter

Employé
String codeInterne
+
Getter/setter

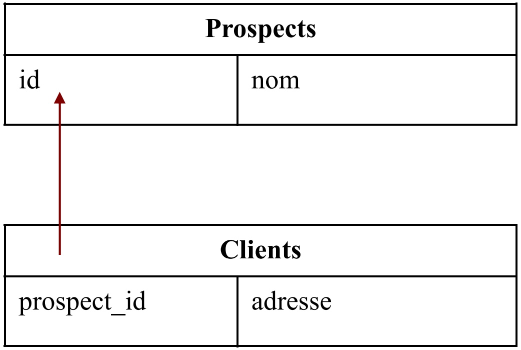
```
<!-- Strategie table-per-class hierarchy mapping -->
<hibernate-mapping>
<class name="Personne" table="PERSONNES" discriminator-value="P">
   <id name="idPers" column="idPers" type="long">
     <generator class="sequence"/>
   </id>
   <discriminator column="sousclasse" type="character"/>
   <property name="nom" column="nom" type="string"/>
   <property name="prenom" column="prenom" type="string"/>

   <subclass name="Employe" discriminator-value="E">
    <property name="codeInterne" column="codeInterne" type="string"/>
   </subclass>
   <subclass name="Externe" discriminator-value="X">
    <property name="societe" column="societe" type="string"/>
   </subclass>
</class>
</hibernate-mapping>
```

One table per class with a join





```
<class name="Prospect" table="PROSPECTS">
   <id name="id"><generator class="native" /></id>
   <property name="nom" />
   <joined-subclass name="Client" table="CLIENTS">
         <key column="prospect_id">
         <property name="adresse" column="adresse"/>
   </joined-subclass>
</class>
```



Prospects	
id	nom

Clients	
prospect_id	adresse

- A client will be in both tables
- A Prospects only in the prospects table
- For clients, you need a Join to extract the object

One table per concrete class

```
<class name="Vehicule">

   <id name="id" type="long" column="VEHICULE_ID">
      <generator class="sequence"/>
   </id>

   <union-subclass name=“Truck" table="VEHICULE_ROULANT">
      <property name=“wheels" column=“WHEELS"/>
   </union-subclass>

   <union-subclass name=“Plane" table=“FLYING_VEHICULE">
   </union-subclass>

</class>
```

- 2 tables are required
- drawback: all the classes need columns for inheritance properties.

HQL

(Hibernate Query Language)

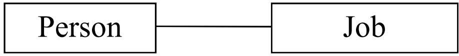
Hibernate Query Language

- HQL is an object version of the SQL;
- HQL is used to get objects from the database.
- Example :

retourne un proxy ← session.load

vs

retourne le vrai objet. ← session.get



Person

Job

```
StringBuffer requeteS = new StringBuffer();
requeteS.append("select person, job from Person person, Job job");
Query requete = session.createQuery( requeteS.toString() );
List resultats = requete.list();
(Object[]) firstResult = (Object[]) resultats.get( 0 );
Person person = (Person )firstResult[ 0 ];
Job job = (Job) firstResult[ 1 ];
```

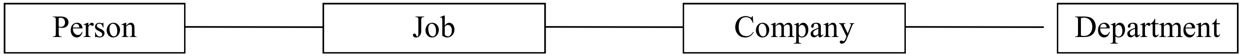
- HQL allows select, from, where, some operators =, >, ..., between, ... ;
- Joins are allowed.

The select clause

- Select is applied to :
    - An object or an object property:

```
select person.name, person.age from Person person
```

    - associations:



Person

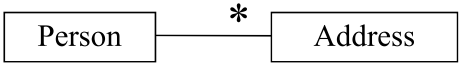
Job

Company

Department

```
select person.job.company.department from Person person
```

    - collection elements:



Person

*

Address

```
select elements(person.addresses) from Person person
```

From clause

- In select ... from ..., select is not required:

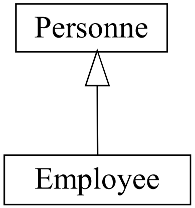
```
select person from Person person
```

⟺

```
from Person person
```

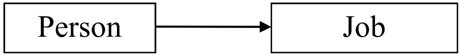
- from and the polymorphism:
  - The following program allows to get instances in a inheritance heriachy

```
StringBuffer requeteS = new StringBuffer();
requeteS.append( “from Person person” );
Query requete = session.createQuery(requeteS .toString() );
List results = requete.list();
(Object[]) firstResult = (Object[]) results.get( 0 );
if(firstResult instanceof Person ){
 Person  person = (Person )firstResult[0];
} else if(firstResult instanceof Employee ){
 Employee employee = (Employee )firstResult[0];
}
```



Associations loading

- With the following class diagram:



Person

Job

  - the request:

```
List results = session.createQuery( “select person from Person person” ).list();
```

Returns Person instances with the association non initialized (lazzy loading).

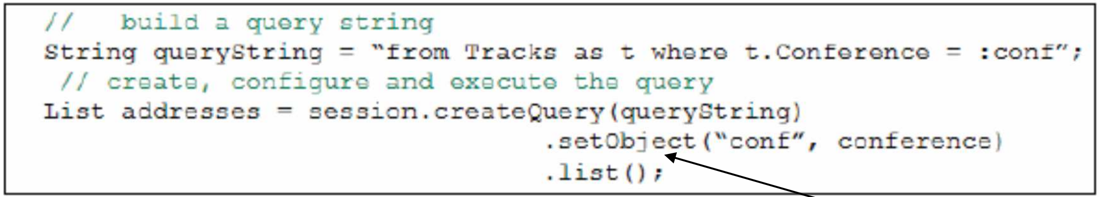
- The followong request:

```
StringBuffer requeteS = new StringBuffer();
requeteS.append( “select person from Person person” )
    .append( “left join fetch person.job job” );
List results = session.createQuery(requeteS .toString() ).list();
```

Returns Person instances with the association initialized.

HQL Parameterized Request

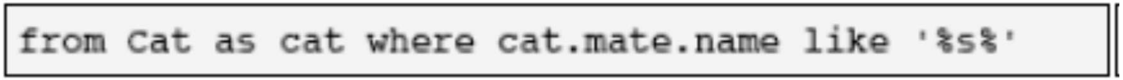
A Request with a « where » clause



```
//   build a query string
String queryString = “from Tracks as t where t.Conference = :conf”;
 // create, configure and execute the query
List addresses = session.createQuery(queryString)
                                .setObject(“conf”, conference)
                                .list();
```

setEntity() and not setObject in ver3

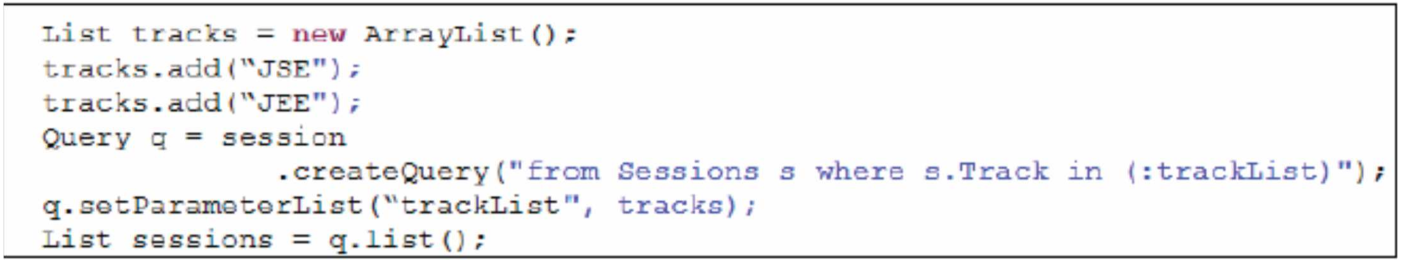
ATTENTION: SetInteger() and SetString() if the parameter is of Integer type or String type in the class



```
from Cat as cat where cat.mate.name like '%s%'
```

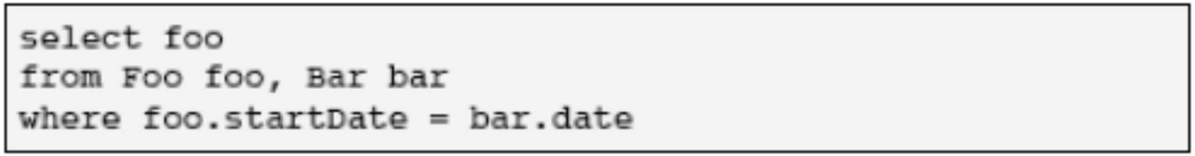
HQL Parameterized Request

Examples (with a list)



```
List tracks = new ArrayList();
tracks.add(“JSE");
tracks.add(“JEE");
Query q = session
             .createQuery("from Sessions s where s.Track in (:trackList)");
q.setParameterList(“trackList", tracks);
List sessions = q.list();
```

Join

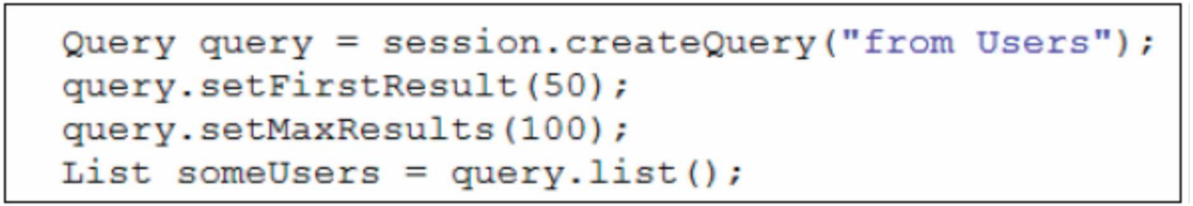


```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

HQL Parameterized Request

You can control result values

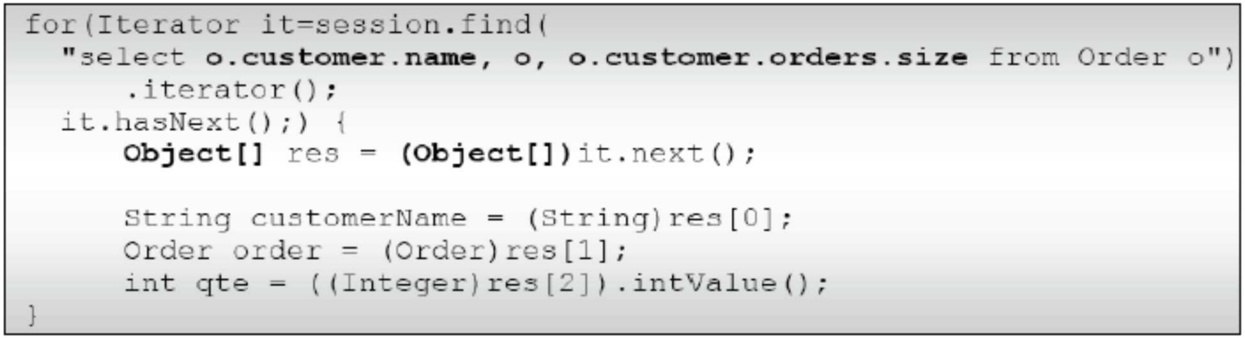
- ✓ Specification of the first requested record
- ✓ Specification of the max records requested



```
Query query = session.createQuery("from Users");
query.setFirstResult(50);
query.setMaxResults(100);
List someUsers = query.list();
```

HQL: Examples

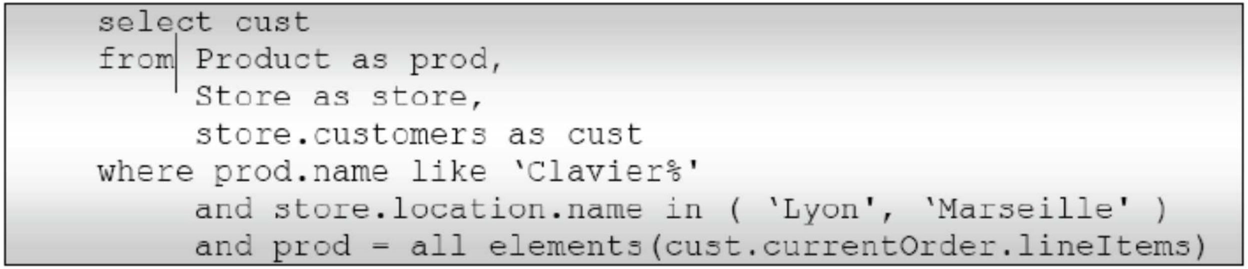
If a request has several elements in its « select » clause, the result is a table of objects



```
for(Iterator it=session.find(
  "select o.customer.name, o, o.customer.orders.size from Order o")
      .iterator();
  it.hasNext();) {
      Object[] res = (Object[])it.next();

      String customerName = (String)res[0];
      Order order = (Order)res[1];
      int qte = ((Integer)res[2]).intValue();
}
```

HQL: Examples

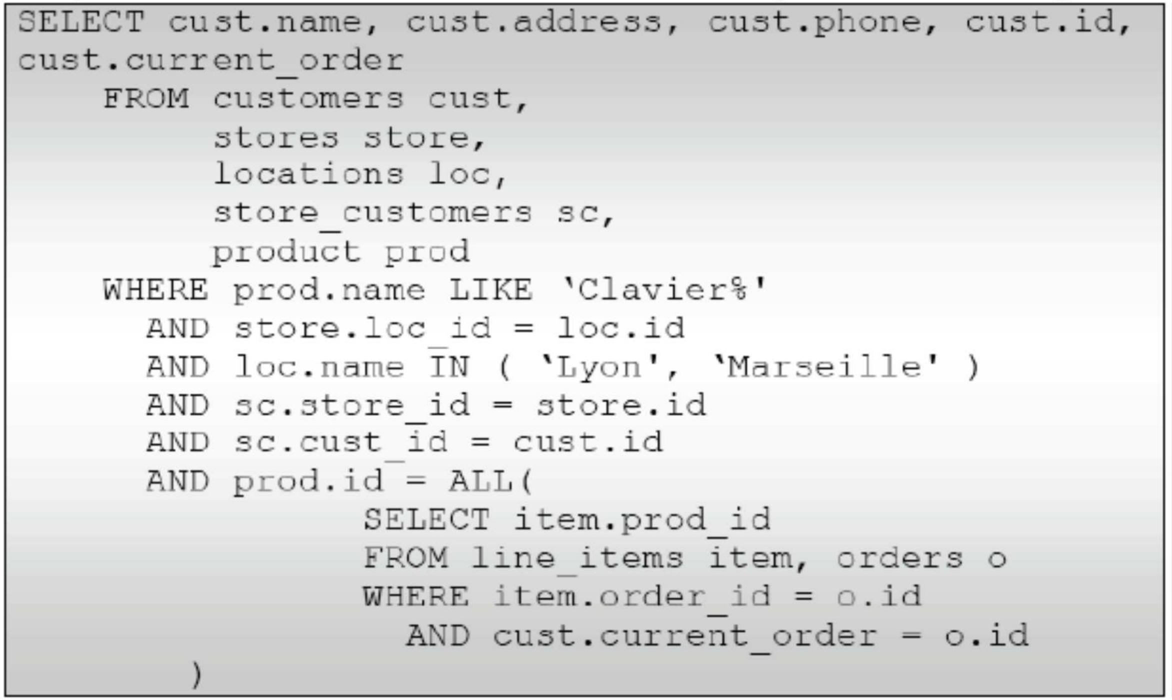
Select customers of stores located in Lyon or Marseille having a current order that contains only “Clavier”.



```
select cust
from Product as prod,
     Store as store,
     store.customers as cust
where prod.name like ‘Clavier%'
     and store.location.name in ( ‘Lyon', ‘Marseille' )
     and prod = all elements(cust.currentOrder.lineItems)
```

HQL: Equivalent in SQL

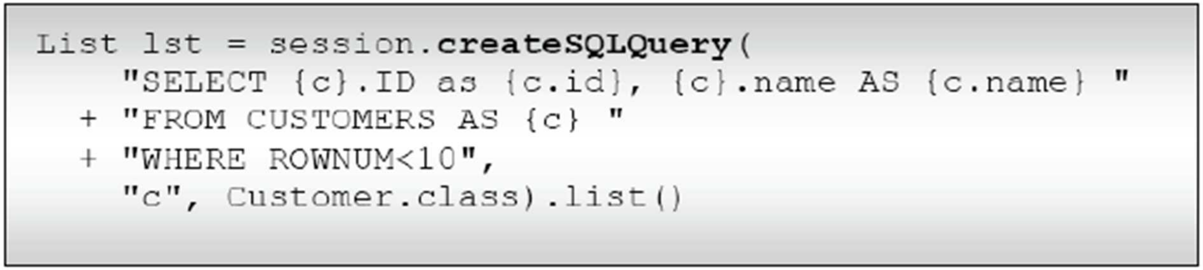
The same request using SQL



```
SELECT cust.name, cust.address, cust.phone, cust.id,
cust.current_order
    FROM customers cust,
         stores store,
         locations loc,
         store_customers sc,
         product prod
    WHERE prod.name LIKE ‘Clavier%'
      AND store.loc_id = loc.id
      AND loc.name IN ( ‘Lyon', ‘Marseille' )
      AND sc.store_id = store.id
      AND sc.cust_id = cust.id
      AND prod.id = ALL(
                SELECT item.prod_id
                FROM line_items item, orders o
                WHERE item.order_id = o.id
                  AND cust.current_order = o.id
        )
```

SQL Queries

createSQLQuery



```
List lst = session.createSQLQuery(
    "SELECT {c}.ID as {c.id}, {c}.name AS {c.name} "
  + "FROM CUSTOMERS AS {c} "
  + "WHERE ROWNUM<10",
    "c", Customer.class).list()
```

HQL: Criteria

You can use the **Restrictions** class to add more criteria on the request

```
List cats = sess.createCriteria(Contact.class)
.add(Restrictions.like("name", "Dupo%") )
.add(Restrictions.between("weight", minWeight, maxWeight) ) .list();
```

In case of a unique result

```
List cats = sess.createCriteria(Contact.class)
.add(Restrictions.like("name", "Dupo%") )
.uniqueResult();
```

HQL: Criteria - Examples

You can sort you results using **org.hibernate.criterion.Order**.

```
List cats = sess.createCriteria(Contact.class)
.add( Restrictions.like("name", "F%")
.addOrder(Order.asc("name") )
.addOrder(Order.desc("age") )
.setMaxResults(50)
.list();
```

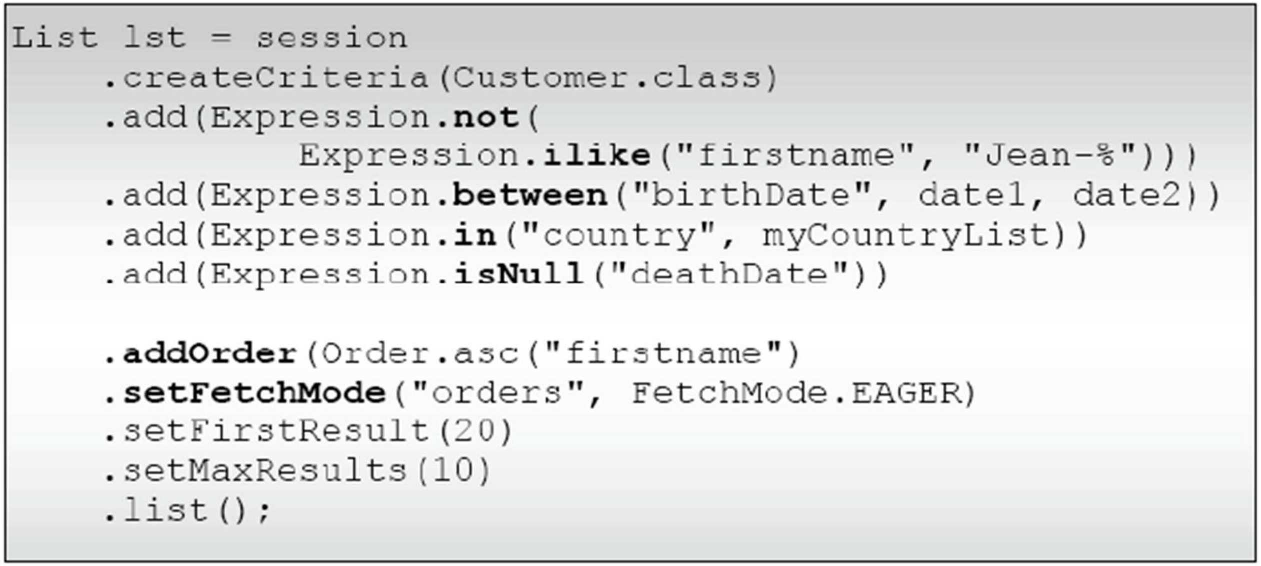
HQL: Criteria - Examples

You can specify constraints on classes and their **associations** using **createCriteria().**

```
List cats = sess.createCriteria(Contact.class)
.add(Restrictions.like("name", "F%") ) .createCriteria("profiles")
.add(Restrictions.like("name", "F%") )
.list();
```

HQL: Criteria - Examples

Multi-criteria search



```
List lst = session
    .createCriteria(Customer.class)
    .add(Expression.not(
            Expression.ilike("firstname", "Jean-%")))
    .add(Expression.between("birthDate", date1, date2))
    .add(Expression.in("country", myCountryList))
    .add(Expression.isNull("deathDate"))

    .addOrder(Order.asc("firstname")
    .setFetchMode("orders", FetchMode.EAGER)
    .setFirstResult(20)
    .setMaxResults(10)
    .list();
```

HQL: Request by the Example

The class **org.hibernate.criterion.Example** allows you to define a criterion given an instance of a class (object)

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
                      .add( Example.create(cat) )
                      .list();
```

HQL: Request by the Example

Advanced request

```
Example example = Example.create(cat)
    .excludeZeroes() //exclude zero valued properties
    .excludeProperty("color") //exclude the property named "color"
    .ignoreCase() //perform case insensitive string comparisons
    .enableLike(); //use like for string comparisons

List results = session.createCriteria(Cat.class) .add(example) .list();
```

Advanced Features

Automatic Dirty Checking

In case of modifying an object, No need to “Save” it again if it is already attached to the session

- ✓ A very efficient mechanism

Automatic update during the next flush of the session

- ✓ The session checks all the objects and generates an Update for every object modified since the last flush.
- ✓ Flush frequency needs to be properly set to not decrease system’s performance

flush

Configurable: flushMode

- ✓ Never (flushMode.MANUAL – flushMode.NEVER)
  - No synchronization between the session and the DB. Need to call explicitly “flush”
- ✓ **At Commit time (flushMode.COMMIT)**
  - Synchronization at commit time
- ✓ **Automatic (flushMode.AUTO)**
  - Default Mode
  - Flush is performed at commit time but before the execution of some requests in order to preserve the coherence of DB
- ✓ Always (flushMode.ALWAYS)
  - Synchronization before every request exeuction
  - Be carful for the performances. Deprecated sans bonnes raisons.

Transitive Persistence - Cascade

To add or to delete an object in cascade

Option « cascade » in mapping files

- « none », « persist », « save-update »
- « delete », « all »
- « all-delete-orphan »

Lazy loading

Can be set at different levels

Lazy Property

```
<property name="email" lazy="false“/>
```

Lazy Collections (primitive types or associations)

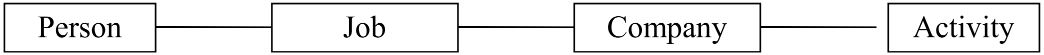
```
 <bag name="names" table="contact_names" lazy="true">
      <key column="ID_CONTACT"/>
      <element column="person_names" type="string"/>
</bag>

 <set name="profiles" inverse="true" lazy="true">
       <key column="id_contact"/>
       <one-to-many class="Profile"/>
</set>
```

Single association

```
<many-to-one name="address" column="addressId"  lazy="true"/>
```

Lazy loading

- Example :



```
Person person  = (Person)session.get(Person.class, new Long(1));
```

The association with Job is set to null !

```
Activity a = person.getJob().getCompany().getActivity();
```

The objects graph is loaded

Composite Id

- A table with a composite key can be mapped with multiple properties of the class as identifier properties.
- Element => **<composite-id>**
- 2 ways
  - 1) *Embedded* composite identifier
  - 2) *Mapped* composite identifier

Composite Id

- 1) Embedded composite identifier

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

- The persistent class *must* override equals() and hashCode() to implement composite identifier equality. It must also implement Serializable.

- Disadvantage: You must instantiate an instance of the persistent class itself and populate its identifier properties before you can **load()** the persistent state associated with a composite key (can’t use simply load(A.class, int x, int y))

- According to Hibernate: “*we discourage it for serious applications* ”

Composite Id

ContactKey {
FirstName;
Last name;
}

- 2) *Mapped* composite identifier

- The identifier properties named inside the <composite-id> element are duplicated on both the persistent class and a separate identifier class (that you have to create)

ContactKey C = new ContactKey
(" ", " ")

Session.get(Contact
, C);

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

- The identifier class must override equals() and hashCode() and implement Serializable

- Disadvantage: code duplication

Automatic Versioning

- System allows multiple users to work with and modify the same database record simultaneously
- Who ever updates last, wins!
  - Previous save is unknowingly lost
- Almost always overlooked by developers
  - Difficult to reproduce or validate a customer complaint
  - No record of anything failing
  - Needs to be pro-actively thought of

Automatic Versioning

- Pessimistic
  - Don’t allow other users to even read the record while you have it “checked out”
  - We’re pessimistic, believing someone will change it on us

- Optimistic
  - Everyone can access it, and we check to see if it’s been updated when we commit changes
  - If it’s been changed while a user was working on it, alert then so we can act accordingly
  - We’re optimistic, believing that nobody else will change it, but if it happens, at least we should know about it

Hibernate’s Optimistic Locking

- Hibernate will automatically check and see if it’s been modified while in your possession.
  - If it did, will throw a **StaleObjectStateException**

- Requires a column to keep track of versions
  - Dedicated ‘**Version**’ Column (usually a number)
  - Can use a **Timestamp**
    - Caution! – two transactions could occur at the exact same time

- When performing the update, version column is automatically included as part of the ‘where’ clause
  - Update employee set salary=(salary*1.10), version=version+1 where id=1 and version=1

Hibernate’s Optimistic Locking

1. **Decide on a versioning strategy**
   - Version column or date?

2. Add an attribute to every domain object Java class to represent the version

- private int version;
- No setter/getter required

3. Add a column to every domain object database table to represent the version

4. Update the object mapping files to make Hibernate aware of the version attribute

- <version name="version" access="field" column="version"/>
- MUST be placed immediately after the identifier property

Hibernate’s Optimistic Locking

```
<class name="org.lip6.hibernate.Contact" table="Contact">
    <id name="contactId" column="CONTACT_ID">
            <generator class="increment"/>
    </id>
    <version name="version" access="field" column=“VERSION"/>

    <property name="firstName" column="FIRST_NAME"
    type="double"/>
</class>
```

Hibernate’s Optimistic Locking

- Hibernate allows you to disable automatic increment based on changes to a particular property or collection
  - Used if you don’t want to increment the version of an object based on the addition or removal of an associated object

- In the object’s mapping file, add the following to the property or collection mapping:

```
<many-to-one name="address" … optimistic-lock="false"/>
```

Hibernate's Pessimistic Locking

1. No version or timestamp attributes/columns required

2. Will NOT work for conversations

- Not valid for detached objects or sessions
- Only within the same session

3. Does not scale as well

- Can become a bottleneck while other users wait

4. Once object obtained, call the lock() method

- LockMode.UPGRADE
  - Waits for the lock if unavailable
- LockMode.UPGRADE_NOWAIT
  - Throws an exception if lock unavailable
- NOT AVAILABLE IN ALL DATABASE VENDORS
  - If not available, just returns the latest version

Batch Processing

Batch Processing

- When executing operations across large data sets, it is more optimal to run directly in the database (not in memory)
  - ✓ Avoids loading potentially thousands of records into memory to perform the exact same action

- In SQL, can be performed with 'Update' and 'Delete' commands
  - ✓ UPDATE ACCOUNT SET BALANCE=BALANCE*1.01;
  - ✓ DELETE FROM ACCOUNT;

Problem with Hibernate: it needs to cache and load objects into the session before performing requests

Batch Processing

2 ways to perform Batches

1) Play with the flush, the cache and JDBC batch size

2) Use The StatelessSession Interface

Example: Inserting 100000 records

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer); }
tx.commit();
session.close();
```

Batch Processing

1) Playing with the flush, the cache and JDBC batch size

- you will need to enable the use of JDBC batching.
  - Absolutely essential if you want to achieve optimal performance.
  - Set the JDBC batch size to a reasonable number (10-50, for example):

    hibernate.jdbc.batch_size 20

- you may need to disable second-level cache

  hibernate.cache.use_second_level_cache false

Batch Processing

When making new objects persistent flush() and then clear() the session regularly in order to control the size of the first-level cache

- **Example: Batch Update**

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
Customer customer = new Customer(.....);
session.save(customer);
if ( i % 20 == 0 ) {
//20, same as the JDBC batch size
//flush a batch of inserts and release memory:
session.flush();
session.clear(); }
}
tx.commit();
session.close();
```

Batch Processing

Example: Batch Insert

you need to use scroll() to take advantage of server-side cursors for queries that return many rows of data.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
ScrollableResults customers =session.getNamedQuery("GetCustomers")
      .setCacheMode(CacheMode.IGNORE)
      .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
Customer customer = (Customer) customers.get(0);
customer.updateStuff(...);
if ( ++count % 20 == 0 ) {
//flush a batch of updates and release memory:
session.flush(); session.clear(); } }
tx.commit();
session.close();
```

Batch Processing

2) Using The StatelessSession Interface

- Used for streaming data to and from the database in the form of detached objects
- A StatelessSession has no persistence context associated with it
- does not implement a first-level cache nor interact with any second-level or query cache.
- does not implement transactional write-behind or automatic dirty checking

Batch Processing

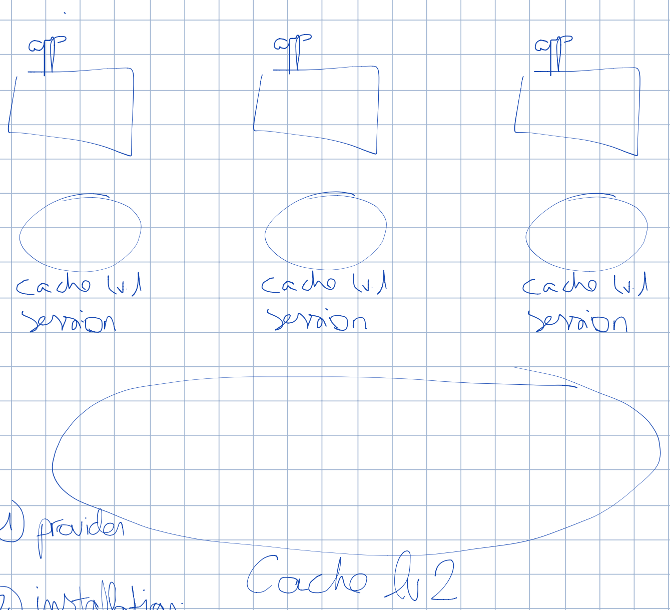
Example

```
StatelessSession session = sessionFactory.openStatelessSession(); Transaction tx
    = session.beginTransaction();
    ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0); customer.updateStuff(...);
    session.update(customer);
}
tx.commit();
session.close();
```

- In this code example, the Customer instances returned by the query are immediately detached. They are never associated with any persistence context

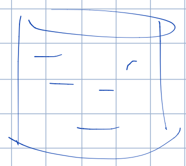
- Results in the immediate execution of a SQL UPDATE, no persistent context is created

2nd Level Cache



1. provider

2. installation:
   - JAR --> /lib
   - fichier XML de config.



Second Level Cache

- Performance increase for objects with a much greater READ to WRITE ratio

- Great for reference or immutable objects Not advised for

– Frequently changing data

– Tables accessed from other applications

- Requires a cache strategy and pluggable cache provider

Second Level Cache

- **Strategies: each level increases performance and risk of stale data**
  - Transactional
    - Slowest of caching, but most risk free
  - Read-write
  - Read-only
    - Fastest performance, but most risky.
    - Use if the data never changes

- Four cache providers are built into Hibernate
  - EHCache: Simple process scope cache in a single JVM
  - OSCache: Richer set of expiration policies and support
  - SwarmCache: Cluster cache, but doesn't suppor 'Query Cache'
  - JBoss Cache: Fully transactional, replicated clustering

Second Level Cache

How to fix provider and strategy properties

```
<session-factory>
...
<property name="cache.provider_class">
org.hibernate.cache.EhCacheProvider
</property>
<property name="cache.use_second_level_cache">
true
</property>
...
</session-factory>
```

Second Level Cache: ehcache.xml

```
<ehcache>
<diskStore path="java.io.tmpdir" />
<defaultCache maxElementsInMemory="10000" eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true" />

<!-- setup special rules for Account objects -->
<cache name="courses.hibernate.vo.Account" maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false" />
</encache>
```

Second Level Cache : Example

Example: Account mapping file

```
<class name="courses.hibernate.vo.Account" table="ACCOUNT">
    <cache usage="read-write" />
    <id name="accountId" column="ACCOUNT_ID">
        <generator class="native" />
    </id>
    <discriminator column="ACCOUNT_TYPE" type="string" />
    <version name="version" column="VERSION" type="long" access="field" />
    <property name="creationDate" column="CREATION_DATE"
    type="timestamp" update="false" />
...
</class>
```

Long Conversations with Hibernate

Long Conversations with Hibernate

- Unit of work that spans multiple Hibernate Sessions
  - think multiple requests
- Each request starts and ends its own Hibernate Session
- Two ways of solving this
  - Use detached objects
    - Session-per-request with detached objects
  - Use an extended Persistence Context
    - Session-per-conversation

Session-per-request with Detached Objects

- **Page 1 : Request 1**
  - Create Contact object
    - Contact is ‘transient’
- **Page 2 : Request 2**
  - Collect contact information, save contact to database
    - Contact transitions to ‘persistent’, but we still want that contact object available when we get to page 3, so we detach it from the Hibernate session
- **Page 3 : Request 3 – N**
  - Collect information for a single Profile, build a profile object, and associate it to contact, save to database
    - Profile starts off ‘transient’, transitions to ‘persistent’ for storing it to the database, and then ‘detached’ to continue using for later pages
    - Also reattaching the Contact object to save the relationship, and then detaching that again as well.
- **Page 4 : Request N+1**
  - Display saved information summary using ‘detached’ objects

Session-per-request with Detached Objects: Example

```
public void testUpdateContact() {
// Get a handle to the current Session
Session session =HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();

// Modify objects
Contact contact = new Contact();
session.saveOrUpdateContact(contact);

// Close the Session without modifying the persistent object
// Changes the state of the objects to detached
session.getTransaction().commit();
HibernateUtil.getSessionFactory().close();

// Modify detached object outside of session
contact.setEmail("belami@maupassant.fr");
…..
```

You can also use Merge() or Lock (i.e. without update) to reattach objects

Session-per-request with Detached Objects: Example

```
...
// Get handle to a subsequent Session
Session session2 =HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();

// Reattach the object to be persisted.
// An update statement will be scheduled regardless
// of whether or not object was actually updated
session2.update(contact);


// Commits/closes the session
// Saves the changes made in the detached state
session2.getTransaction().commit();
```

Session-per-Conversation

- Extending the persistence context
  - Keep the same persistence context across server requests
  - Reconnected upon subsequent requests
- No more ‘detached’ objects
  - All objects are either transient or persistent
  - No need for reattaching/merging
- Synchronized with database at end of the conversation
- Considered to be complicated by many developers

Session-per-Conversation

- Session is disconnected from the underlying JDBC connection upon commit
  - Persistence context is held on in some persistent state
    - **HTTPSession**
    - Stateful Session Bean
- Persistence context is reconnected when the session begins the next transaction
  - Objects loaded during the conversation are in the ‘persistent’ state; never detached
  - All changes are synchronized with the db when flush() is called
- **Need to disable automatic flushing of the session**
  - Set FlushMode.MANUAL when the conversation begins and the Session is opened
- Manually trigger the synchronization with the database
  - **session.flush() at the end of the conversation**

Conclusion

A very powerful and efficient Framework

Used in many projects in the industry

Has inspired the JPA Standard

- Provided as the implementation of JPA in many JEE servers

May disappear with the increasing success of JPA

Some advanced functionalities provided by Hibernate and still not provided by JPA