# 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

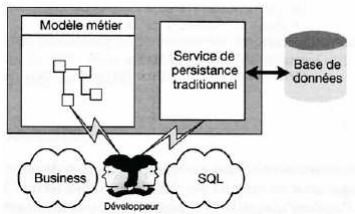
Services :

- **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() ;
}
```

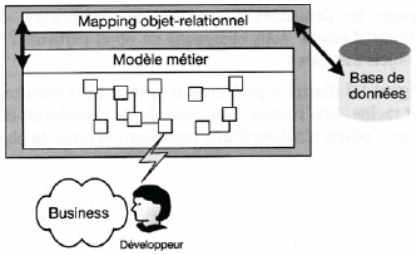
# JDBC drawback

- 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






# 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());
    }
```



# Object-relational mapping with Hibernate 2/2


| 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

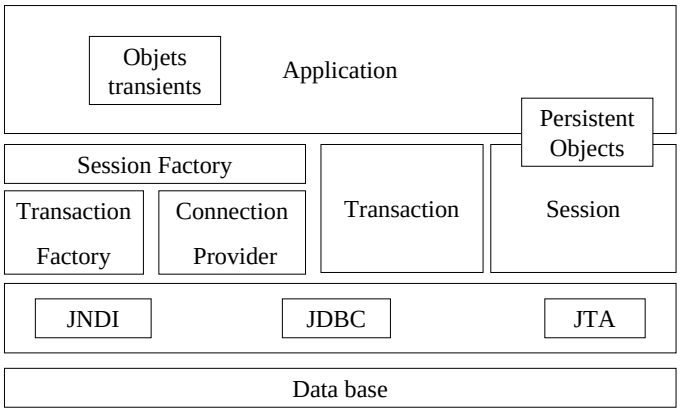
- Pure object model above a database
- Association and inheritance are taken into account ;
- Mask different SQL implementations ;
- Two majors services :
  - Persistence
  - Transactions.
- Open source.



# Hibernate architecture



# Hibernate architecture






# Hibernate Core

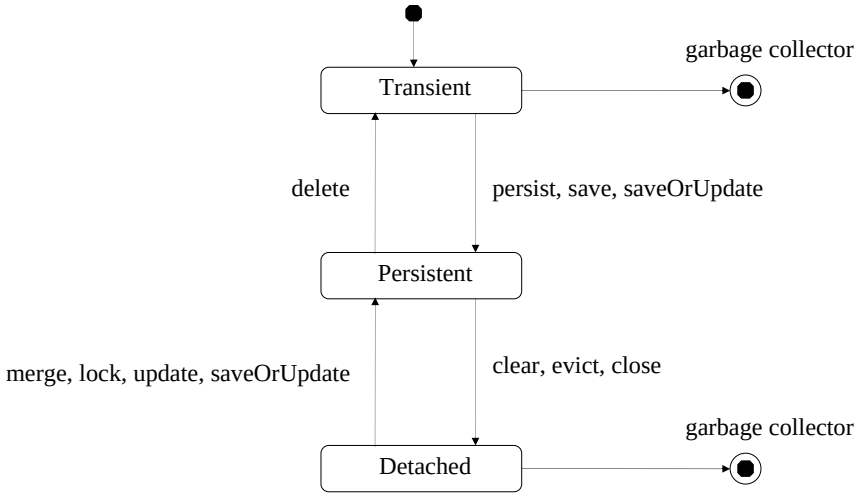
**Session Factory**

- 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






# 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/dfj_hibernate</property>
      <property name="hibernate.connection.username">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
```



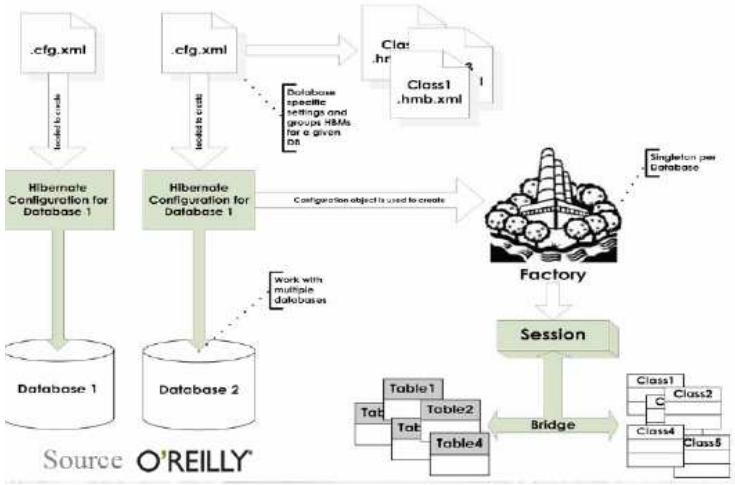
# Hibernate: Mapping File

- 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



## 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 ot public.
  - inheriting classes :
    - Must have a constructor without argument ;
    - Can have n 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

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

Name of the primary key column.

- 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**

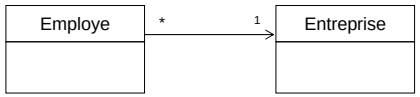
**Property type**

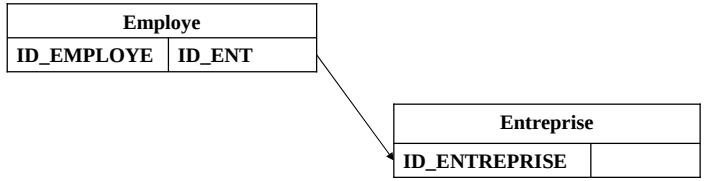
**The column name for this property in the generated table. By default, the property name in the class**



# Association many-to-one (N-1)

A table references another table via a foreign key



# Association one-to-many (1-N)

If the previous association should be bidirectional

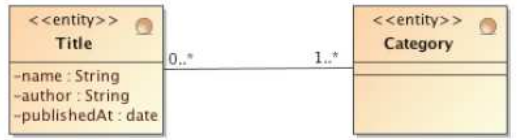
```
<class name="Entreprise" table="ENTREPRISE">
    . . .
    <set name="employes"  inverse=" true ">
          <key column="ID_ENT"/>
          <one-to-many class="Employe"/>
    </set>
    </class>
```

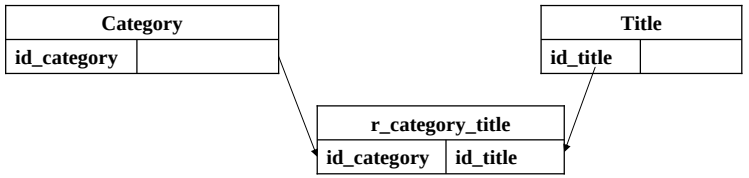
Column of table **EMPLOYE** (not **ENTREPRISE!!!!**) foreign key towards **ENTREPRISE**



# Association many-to-many

Need for an intermediary table.






# 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>
```



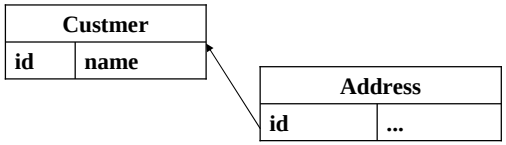
# Association many-to-many (M-N)

## 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).






# 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>
```



# 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>
```



# 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>
```



# 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!
  - Two persistent instances can not share the same collection => duplicate the collection.
  - A collection attribute can not be null.



# 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>
```

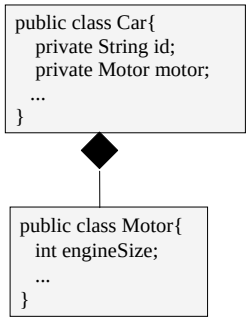
Exp. a Set containing Integer(s) ordered by « size »

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

# Compositions mapping



# Composition persistence = persistence by value

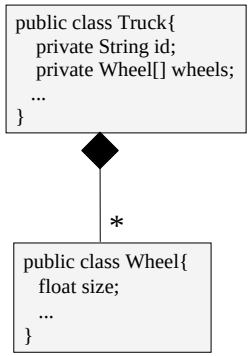



```
<class name=“Car" table=“car">

  <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



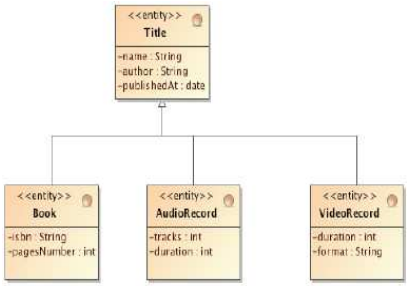

```
<class name=“Truck" table=“truck">

  <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.





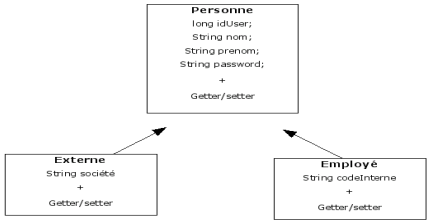

# One table per class hierarchy

| 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.



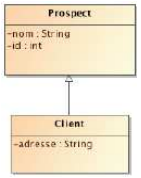
# One table per class hierarchy




```
<!-- 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.



# Persistence of an associated set of objects



# Transitive Persistence - Cascade

To add or to delete an object in cascade

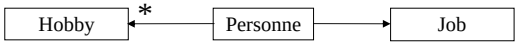
Option « cascade » in mapping files

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



# 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();
```

# 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 de propagate the persistence to the instances job and hobby.



# 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.



# 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 :


```
StringBuffer requeteS = new StringBuffer();
requeteS.append(“select person, job from Person person, Job job”);
Query requete = session.createQuery( chaineRequete.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, ... ;
- Jointures are allowed.

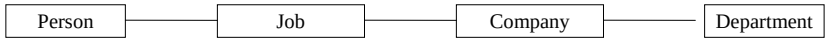


# The select clause

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

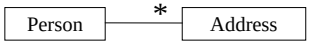
```
select person.name, person.age from Person person
```

  - associations:




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

  - collection elements:




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

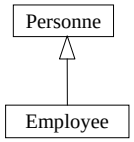
# From clause

- 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:


  - 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)
```



## 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();
```



## 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

# Readings/Sources

- The hibernate specification : *https://www.**hibernate**.org/*
- Book: JPA and Hibernate: Patricio Anthony, Eyrolles, 2008
- Book: Java Persistence with Hibernate, Christian Bauer et Gavin King, 2007, Manning publications
- B. Charroux Courses at EFREI (slides in French not provided online)
- H. Ouahidi Courses, UniConsulting (slides in French not provided online)