
J.P.A. (Java Persistence Api) et Hibernate

Table des matières

I - Mapping Objet-Relationnel (ORM) et JPA.....	4
1. Problématique "O.R.M.".....	4
2. JPA (Java Persistence Api).....	5
II - Architecture d'Hibernate (Session,...).....	7
1. Hibernate: architecture & configurations.....	7
III - JPA : architecture & configuration.....	9
1. Présentation de JPA (Java Persistence Api).....	9
2. Unité de Persistance (META-INF/persistence.xml).....	9
3. Configuration du mapping JPA via annotations.....	12
IV - EntityManager et entités persistantes.....	14
1. Entity Manager et son contexte de persistance.....	14
2. Transaction JPA.....	16
3. Différents états - objet potentiellement persistant.....	17
4. Cycle de vie d'un objet JPA/Hibernate.....	17
5. Synchronisation automatique dans l'état persistant.....	18
6. objet persistant et architecture n-tiers.....	18
7. Principales méthodes JPA / EntityManager et Query.....	18

8. Contexte de persistance et proxy-ing.....	20
V - Langage de requêtes JPQL.....	22
VI - O.R.M. JPA (généralités).....	26
1. Vue d'ensemble sur les entités, valeurs et relations.....	26
2. Identité d'une entité (clef primaire).....	28
3. Propriétés d'une colonne.....	29
4. Relations (1-1, n-1, 1-n et n-n).....	29
VII - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...).....	31
2. Relations d'héritage & polymorphisme.....	41
VIII - JPA : aspects divers et avancés.....	45
1. Quelques spécificités de JPA.....	45
2. Cycle de persistance (pour @Entity) et annotations/callbacks associées pour "Listener".....	47
3. Map et apports de JPA 2.0.....	47
4. Api "Criteria" de JPA 2.....	52
IX - Outils pour JPA , 	56
1. Exemple de dépendances maven pour JPA.....	56
2. Outils et plugins pour JPA.....	56
3. Plugin maven "hibernate4-maven-plugin".....	57
4. Projet "JPA" d'eclipse.....	57
X - Configurations d'Hibernate (hbm,@...).....	59
1. Configuration Hibernate en xml (sans JPA).....	59
XI - Session Hibernate.....	62
1. Principales méthodes de l'objet Session:.....	62
2. Principales méthodes de l'objet Query:.....	62
XII - Intégration Spring , Hibernate Template.....	63
1. Grandes lignes sur l'intégration de Hibernate au sein de Spring.....	63
XIII - Criteria Hibernate.....	64
1. Criteria hibernate.....	64

XIV - Généralités ORM Hibernate.....	65
1. Vue d'ensemble sur les entités, valeurs et relations.....	65
2. Identité d'une entité (clef primaire et générateurs).....	68
3. Clef primaire composée.....	69
4. Structure générale d'une classe d'entité persistante.....	69
XV - ORM Hibernate (HBM: 1-1 , 1-n , n-n , ...).	71
1. Mapping (.hbm) des principales associations.....	71
2. Mapping Hibernate (.hbm.xml) pour l'héritage.....	73
3. 1-1 avec correspondance directe des "pk".....	74
4. Imbrications d'une collection de valeurs composites.....	74
XVI - Aspects divers et avancés (batch , xml, ...).....	76
1. Requêtes en grandes séries (par paquets / batch).....	76
2. Autres aspects divers de Hibernate.....	77
3. Mapping xml-relationnel (entité en xml).....	78
4. Outils et plugins pour Hibernate.....	80
5. Hbm2dll (SchemaExport).....	80

I - Mapping Objet-Relationnel (ORM) et JPA

1. Problématique "O.R.M."

1.1. Objectif & contraintes:

L'objectif principal d'une technologie de **mapping objet/Relationnel** est d'établir une **correspondance** relativement **transparente** et **efficace** entre :

un ensemble d'objets en mémoire

et

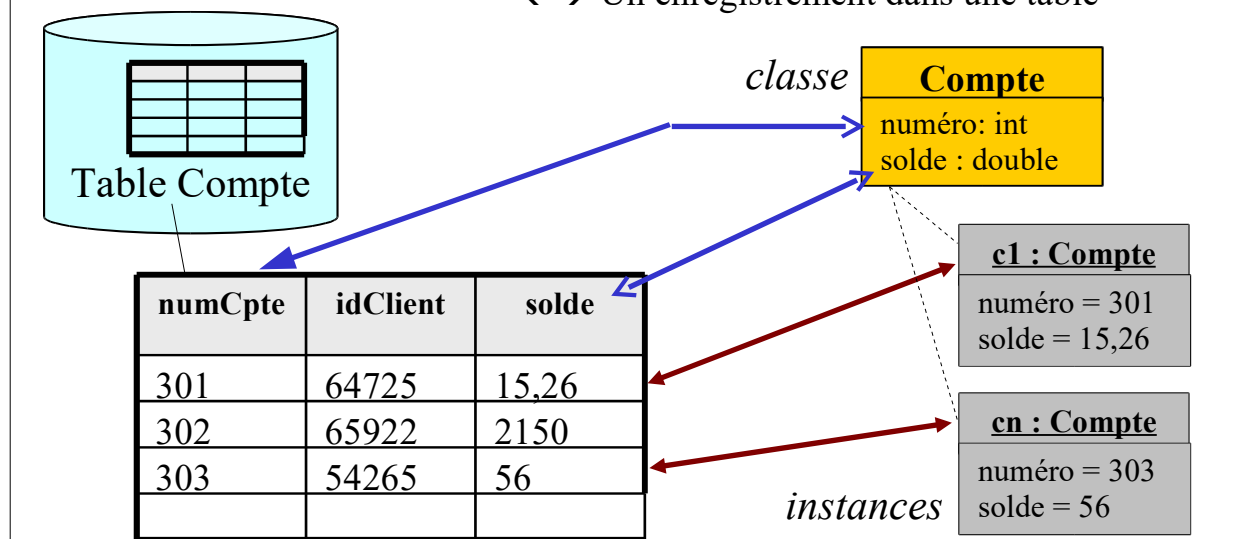
un ensemble d'enregistrements d'une base relationnelle.

O.R.M. (Mapping Objet-Relationnel)

Entité persistante = vue orientée objet d'un enregistrement

En gros (à peu près): Une instance (@Entity)

↔ Un enregistrement dans une table



Une telle technologie doit permettre à un programme orienté objet de ne voir que des objets dont certains sont des **objets persistants**. *Le code SQL est en très grande partie caché* car les objets persistants sont automatiquement pris en charge par la technologie "O.R.M."

Autrement dit, le code SQL n'est plus (ou très peu) dans le code "java" mais est généré automatiquement à partir d'une configuration de mapping (fichiers XML, annotations, ...).

Contraintes : pour être **exploitable** , une **technologie "O.R.M."** se doit d'être :

- **simple** (à configurer et à utiliser) et **intuitive**
- **portable** (possibilité de l'intégrer facilement dans différents serveurs d'application)
- **efficace** (bonnes performances, fiable , ...)
- capable de gérer les **transactions** ou bien de participer à une transaction déjà initiée
- ...

1.2. Éléments techniques devant être bien gérés

Au delà des simples associations élémentaires du type

"1 enregistrement d'une table <---> 1 instance d'une classe" ,

une **technologie "O.R.M." sophistiquée** doit savoir gérer certaines **correspondances évoluées** :

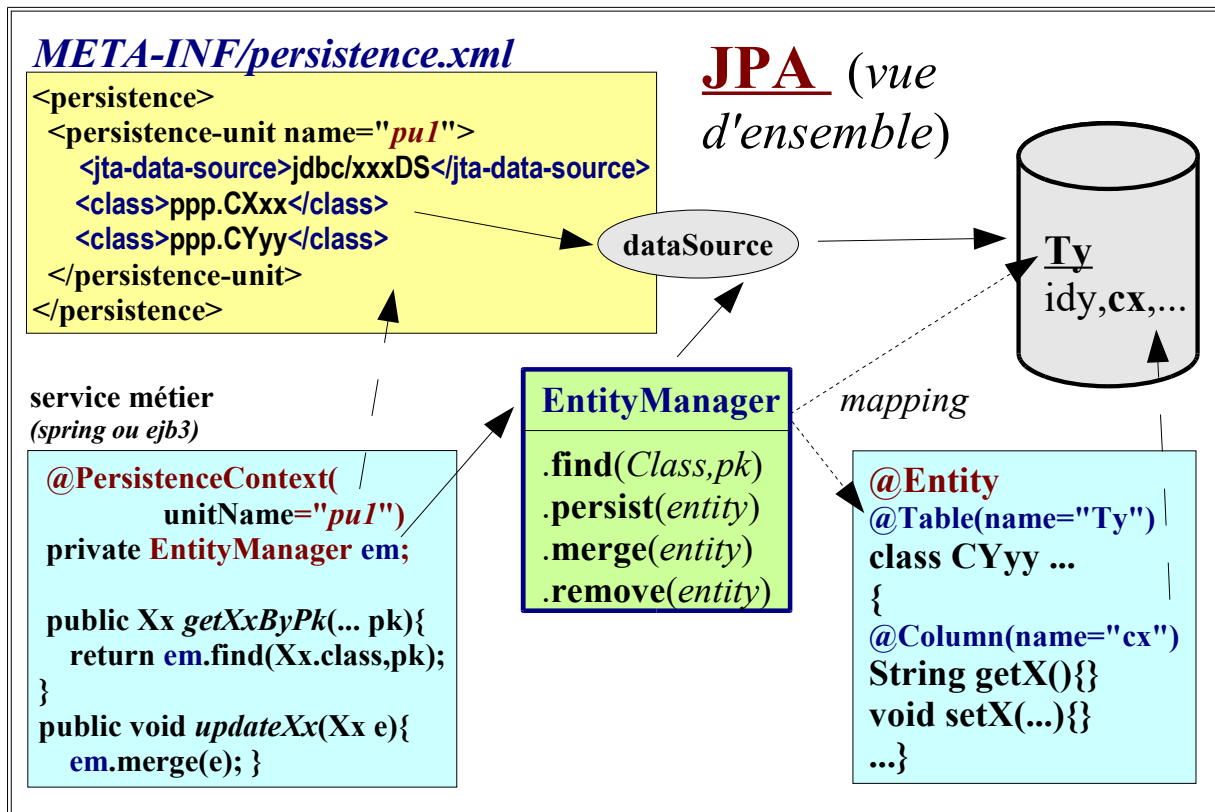
- **Associations "1-1" , "1-n" , "n-n"** (jointures entre tables <---> relations entre objets)
- **Objets "valeur"** (sous parties d'enregistrements , tables secondaires , ..)
- **Généralisation/Héritage/Polymorphisme** (<---> schéma relationnel ?)
- ...

Une technologie "O.R.M." doit sur ces différents points être **flexible et efficace** .

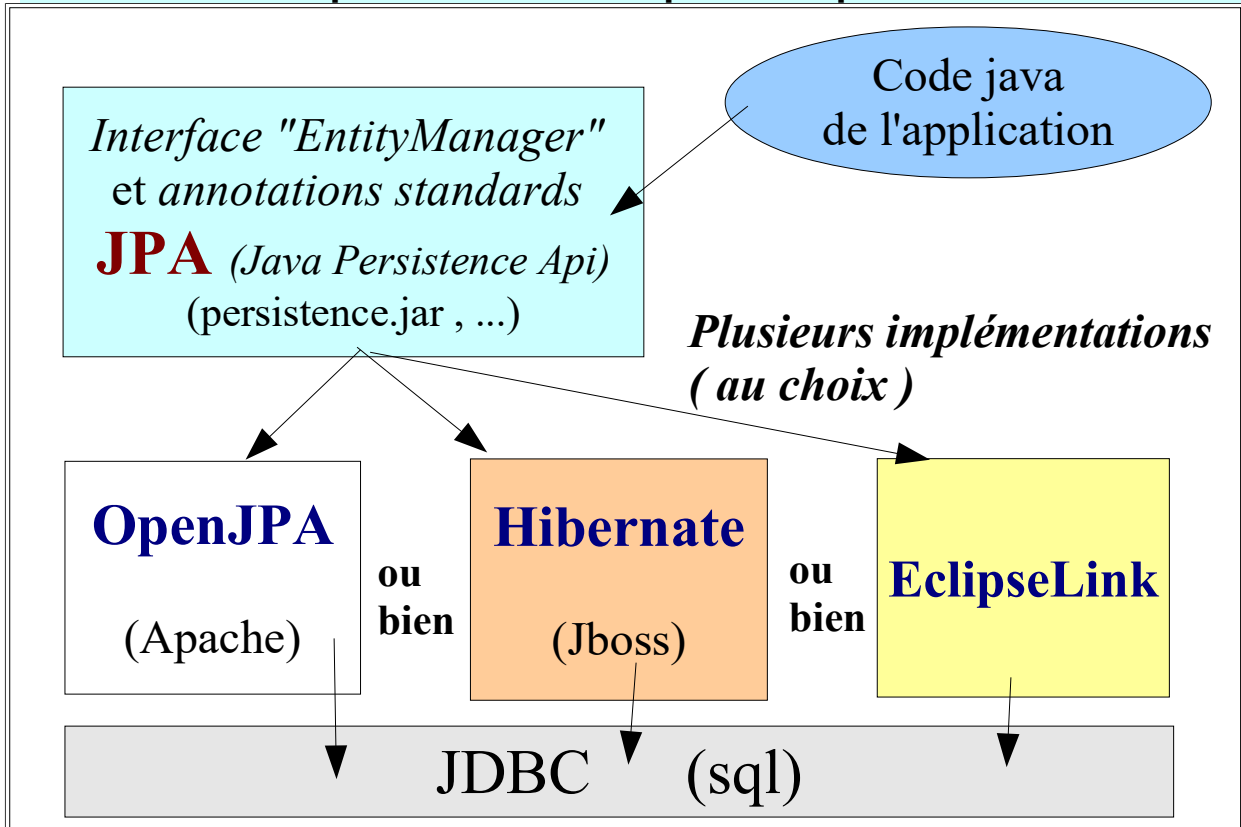
2. JPA (Java Persistence Api)

JPA (Java Persistence Api) et @Entity

- **API O.R.M. officielle de Java EE**
- **Configuration** basée sur des **annotations** (*avec configuration xml possible : orm.xml*).
- Utilisable dans les contextes suivants:
 - dans **module d'EJB3**
 - dans **module Spring**
 - de façon **autonome** (java sans serveur)
- Plusieurs **implémentations** internes disponibles (*OpenJPA , Hibernate* depuis version 3.2 ,)



2.1. Plusieurs implémentations disponibles pour JPA

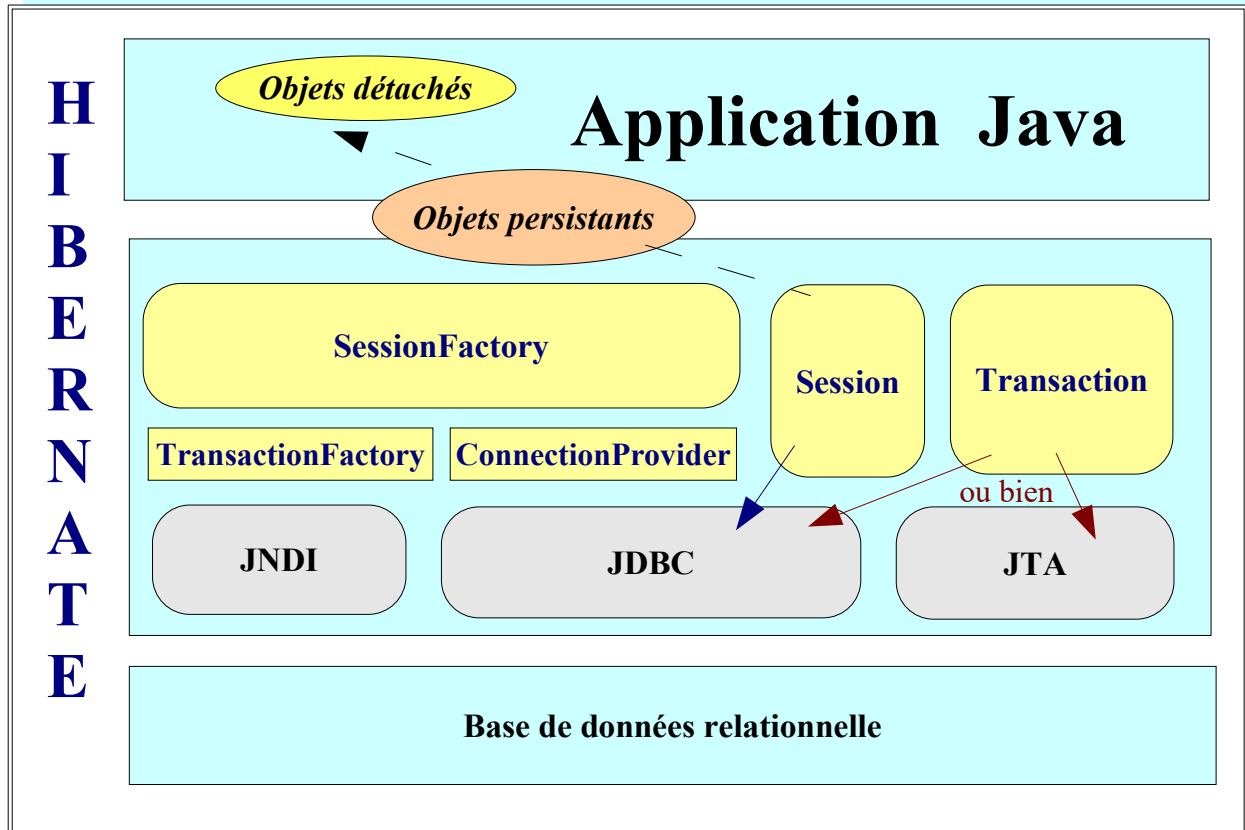


Selon le contexte, l'implémentation de JPA sera libre ou bien quasi imposée par le serveur d'applications (ex : JBoss Application Serveur utilise en interne Hibernate pour implémenter JPA).

II - Architecture d'Hibernate (Session,...)

1. Hibernate: architecture & configurations

1.1. Architecture d'hibernate



NB: Les transactions (commit/rollback) sont supervisées à partir de l'objet abstrait "Transaction" d'Hibernate. Celui-ci peut soit s'appuyer sur des transactions simples JDBC (sur une seule base de données) ou bien sur des transactions complexes "JTA" gérées par le serveur d'application hôte (ex: WebSphere, Weblogic, Jboss, Geronimo, ...).

D'autre part, l'utilisation exacte de l'objet "Transaction" d'hibernate est assez variable en fonction du style de programmation utilisé (explicite/implicite, déclaratif, ...).

L'objet "Session" d'hibernate est de loin le plus important : il correspond à une sorte de "D.A.O." générique permettant de déclencher les opérations habituelles de persistance (CRUD : Create, Retrieve, Update, Delete). L'objet "session" d'hibernate est soit "ouvert" ou "fermé" et ceci a une très grande incidence sur la connexion JDBC/SQL sous-jacente et sur l'état des objets (persistants ou détachés).

1.2. Principaux objets de l'API

SessionFactory	Un objet de ce type par base de données. cet objet représente un cache de tous les mappings "objet/relationnel" liés à une certaine base de données. Les traitements associés à cet objet sont ré-entrants (multi-threading possible) . Cet objet sert à fabriquer des objets "Session".
Session	Un objet de ce type représente une session utilisateur et est lié à un seul thread. Une session englobe un certain nombre d' objets persistants dont les valeurs sont mappées avec le contenu de certaines tables relationnelles. Chaque objet session incorpore une connexion JDBC et sert à récupérer ou fabriquer des objets de type "Transaction"
Transaction	Un objet de l'api hibernate représentant une transaction que l'on peut explicitement délimiter (begin , commit/rollback) . cet objet se base sur des transactions sous-jacentes de type JDBC, JTA ou
Objets persistants	Objets java (JavaBean / POJO) comportant des valeurs persistantes et des traitements "métier" . Tant qu'ils sont liés à une session, ces objets ont des valeurs synchronisées/synchronisables avec la base de données. Dès que la session est fermée, ces objets deviennent détachés et peuvent être directement réutilisés pour tout transfert ou affichage.
...	

1.3. Fichiers de configuration (selon variantes)

<i>Utilisation d'Hibernate</i>	<i>Configuration générale (base de données ,)</i>	<i>Configuration fine du mapping objet relationnel</i>
Hibernate seul	hibernate.cfg.xml	xxx.hbm.xml , yyy.hbm.xml
Hibernate intégré dans Spring	Partie de springConf.xml	xxx.hbm.xml, yyy.hbm.xml
Hibernate intégré dans JPA (lui même intégré dans EJB3 ou Spring ou ...)	META-INF/persistence.xml (+ ou)	annotations (@Entity ,@Table , @Column,) dans le code java des entités
...

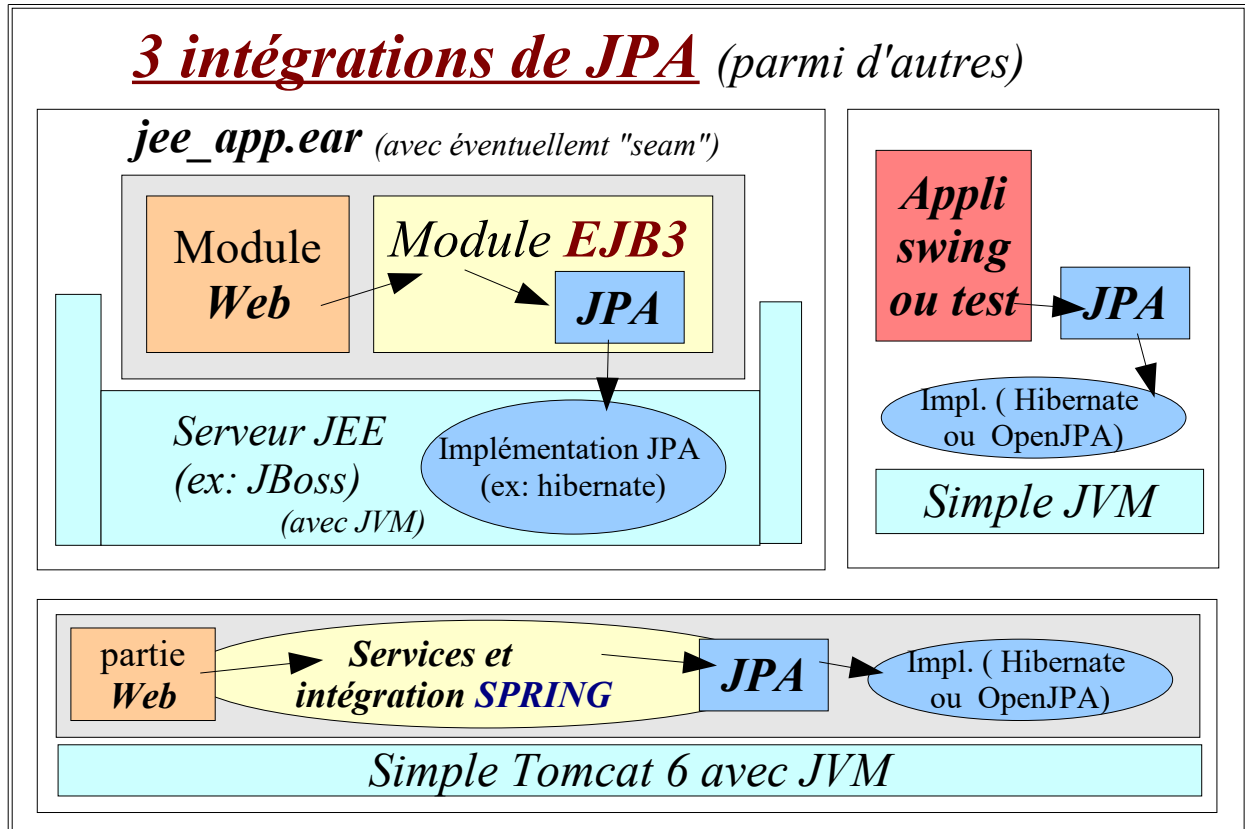
Structure ordinaire du code de l'application allant autour d'Hibernate:

DAO (ou Test) avec Hibernate seul	Ordres O.R.M. via " session " + gestion explicite des transactions + gestion explicite de la session
DAO ou Service avec " Spring+Hibernate "	Ordres O.R.M. via hibernateTemplate (+ try/catch)
Test ou DAO avec JPA (sans EJB3 ni Spring)	Ordres O.R.M. + gestion explicite des transactions + gestion explicite "entityManager"
Service avec JPA (intégré dans Spring ou EJB3)	Ordres O.R.M. via " entityManager " (+try/catch)

III - JPA : architecture & configuration

1. Présentation de JPA (Java Persistence Api)

L'api JPA peut être utilisée de façon très variable (dans EJB3 , dans Spring ou seul)



Une intégration "EJB3" s'appuie essentiellement sur ce qui est offert par le serveur d'applications (Jboss, WebSphere, ...). Par exemple JBoss 4.2.3 ou 5.1 comporte déjà en lui toutes les librairies (.jar) correspondant à l'API JPA (et son implémentation basée sur Hibernate).

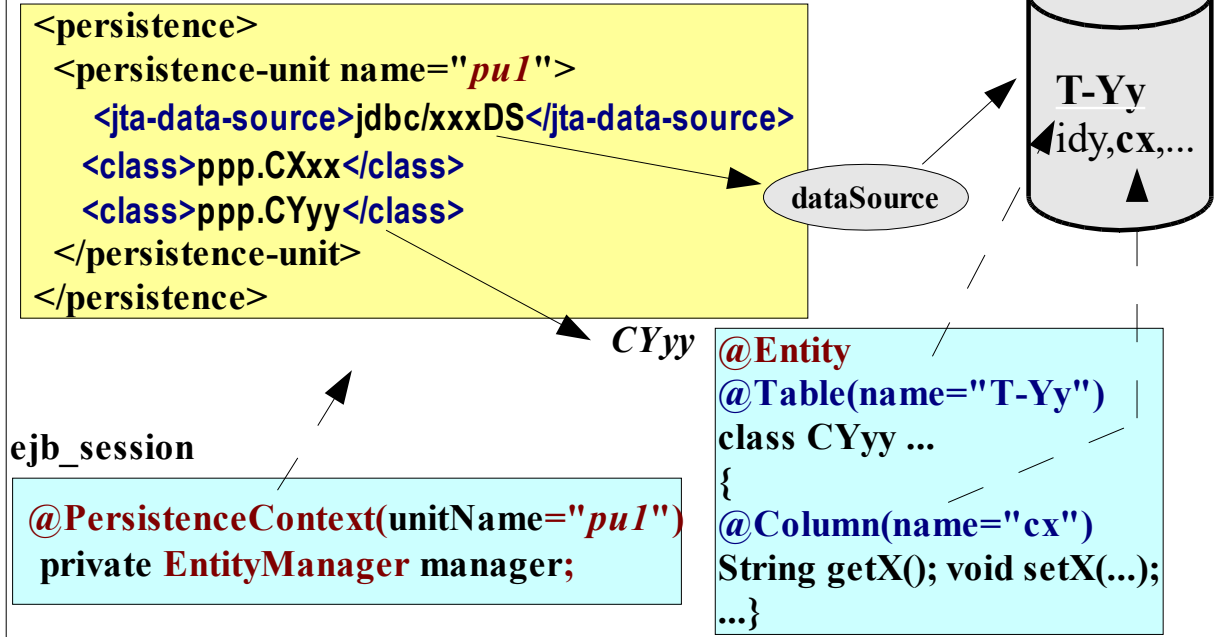
Une intégration "Spring" s'appuie par contre généralement sur des librairies choisies (Hibernate ou OpenJPA ou) et qui sont quelquefois livrées et déployées avec l'application (dans WEB-INF/lib par exemple) .

Une classe de Test "JUnit" exécutée avec une simple JVM peut très bien utiliser directement l'API JPA (avec ou sans Spring) en s'appuyant sur une des implémentations disponibles (Hibernate ou OpenJPA ou ...) .

2. Unité de Persistance (META-INF/persistence.xml)

Unité de persistance (*PersistenceUnit*)

META-INF/persistence.xml



META-INF/persistence.xml (exemple)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" >
  <persistence-unit name="OrderManagement">
    <description>....</description>
    <jta-data-source>java:/MyOrderDS</jta-data-source>
    <!-- <mapping-file>orm.xml</mapping-file> -->
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
    <properties>
      <!-- <property name="hibernate.hbm2ddl.auto" value="create-drop"/> -->
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

`<persistence-unit>` comporte un attribut optionnel `"transaction-type"` dont les valeurs possibles sont `"JTA"` ou `"RESOURCE_LOCAL"`.

La valeur par défaut est `"JTA"` dans un env. EE (ex : EJB3) et `"RESOURCE_LOCAL"` dans un env. SE.

La valeur de `<jta-data-source>` ou `<non-jta-data-source>` correspond au *nom JNDI global de la source de données SQL* (pool de connexions permettant d'atteindre une base de données précise).

[Rappel : le serveur Jboss A.S. ajoute le préfixe "java:/" sur les noms JNDI des "dataSources"]

L'ensemble des classes (avec annotations @Entity) devant être prises en compte et prises en charge par le contexte de persistance se définit via une (ou plusieurs et complémentaires) indication(s) suivante(s):

- Une liste explicite de classes (balises `<class>`)
- un ou plusieurs fichier(s) xml de mapping "objet/relationnel" (balises `<mapping-file>`)
- ...

NB: Dans le cas particulier d'un module EJB3 , les classes comportant des annotations @Entity seront automatiquement détectées et il n'est donc pas absolument nécessaire de les déclarées dans le fichier META-INF/persistence.xml . Par contre avec Spring , les `<class>p.EntityClass</class>` doivent être explicitées.

NB: un fichier xml de type **orm.xml** constitue soit une alternative vis à vis d'une configuration basée sur des annotations Java5 , soit un mécanisme de redéfinition.

Dans le cas où JPA est implémenté via la sous couche Hibernate , on peut éventuellement fixer la propriété **hibernate.hbm2ddl.auto** à l'une des valeurs suivantes :

- * **create-drop** [créer les tables au démarrage de l'application et les supprime à l'arrêt]
- * **update** [mise à jour des tables sans suppression]
- * **none** (ou le tout en commentaire)

====> Les "create table" sont générés à partir des informations @Table , @Column ,

====> Cette fonctionnalité peut être pratique en mode développement mais est très dangereuse en mode production !!!

Variante de META-INF/persistence.xml pour intégration dans Spring :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0" >
  <persistence-unit name="myPersistenceUnit"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>entity.persistence.jpa.Compte</class>
    <class>entity.persistence.jpa.XxxYyy</class>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect" />
      <!-- <property name="hibernate.hbm2ddl.auto" value="update" /> -->
    </properties>
  </persistence-unit>
</persistence>
```

NB : `<provider>.....HiernatePersistence</provider>` permet de préciser que la technologie Hibernate est choisie pour implémenter JPA .

3. Configuration du mapping JPA via annotations

Le mapping de JPA se configure généralement en insérant quelques annotations JAVA dans le code JAVA des entités.

3.1. Exemple d'entité persitante (@Entity)

```
@Entity
@Table(name = "T_CUSTOMER")
public class Customer implements Serializable {

@Id
private Long number;

@Column(name = "customer_name")
private String name;
private Address address;

@OneToMany(...)
private List<Order> orders = new ArrayList<Order>();

@ManyToMany(...)
private List<PhoneNumber> phones = new ArrayList<PhoneNumber>();

public Customer() {} // No-arg constructor

public Long getNumber() {return number;} // pk
public void setNumber(Long number) {this.number = number;}

public String getName() {return name;}
public void setName(String name) {this.name = name;}
public Address getAddress() {return address;}
public void setAddress(Address address) {this.address = address;}

public List<Order> getOrders() {return orders;}
public void setOrders(List<Order> orders) {
    this.orders = orders;}
public List<PhoneNumber> getPhones() {return phones;}
public void setPhones(List<PhoneNumber> phones) {
    this.phones = phones;}

// Business method to add a phone number to the customer
public void addPhone(PhoneNumber phone) {
    this.getPhones().add(phone);
// Update the phone entity instance to refer to this customer
    phone.addCustomer(this);}
}
```

Dans l'exemple précédent :

- l'annotation **@Entity** permet de marquer la classe Customer comme une classe d'objets

persistants

- l'annotation **@Table(name="T_CUSTOMER")** permet d'associer la classe java "Customer" à la table relationnelle "T_CUSTOMER".
- l'annotation **@Column(name="customer_name")** permet d'associer l'attribut "name" de la classe "Customer" à la colonne "customer_name" de la table relationnelle.
- l'annotation **@Id** permet de marquer le champ "number" comme **identifiant de l'entité (et comme clef primaire de la table)**.
- les autres annotations (**@OneToMany** , ...) seront approfondies dans un chapitre ultérieur.

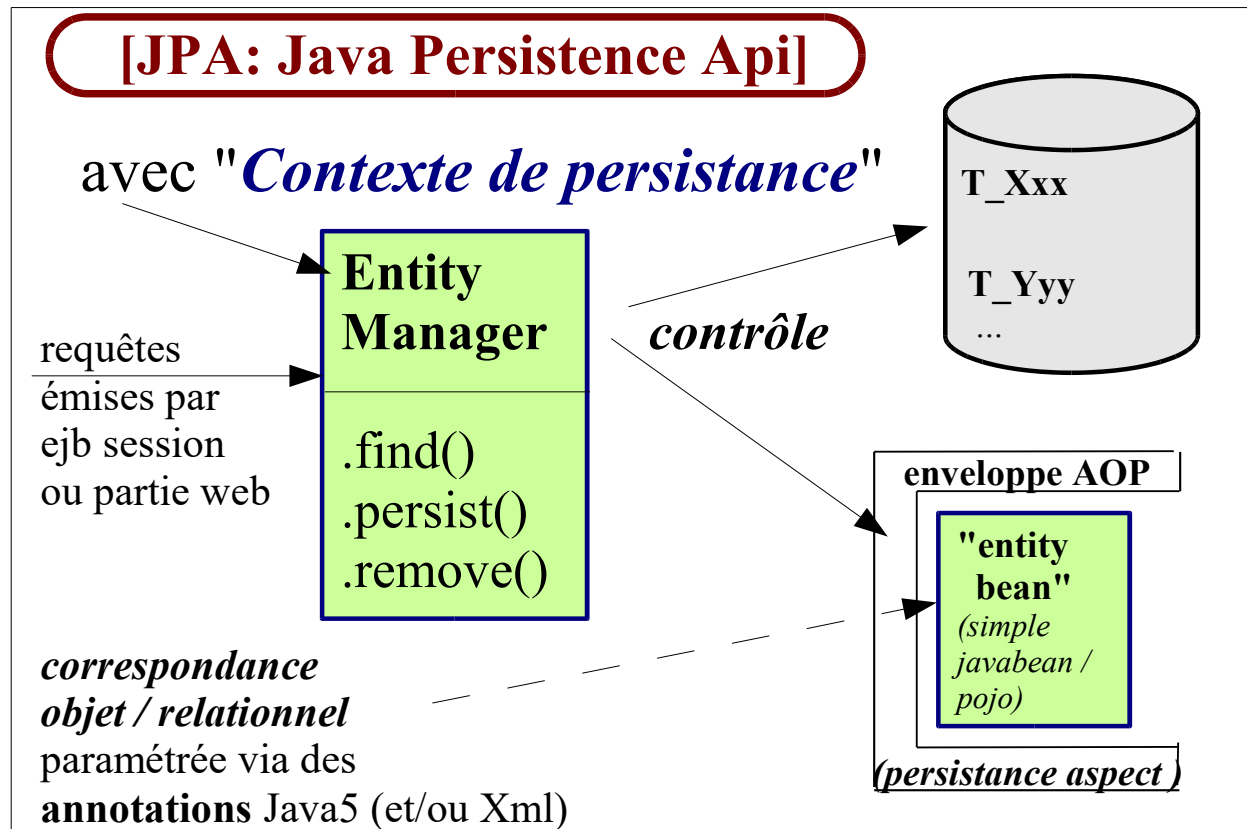
NB: Lorsqu'une annotation (@Column ou ...) n'est pas présente , les noms sont censés **coïncider** (sachant que les minuscules et majuscules ont quelquefois de l'importance du côté SQL/relationnel selon le système d'exploitation hôte (ex: linux)).

Remarques importantes:

- Les annotations paramétrant le mapping objet/relationnel (@Id , @Column, @OneToMany, ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "get".
[au dessus des "private" --> plus lisible car dans le haut de la classe]
[au dessus des "get" --> plus clair et/ou plus fonctionnel en cas d'héritage]
Attention: il faut absolument garder un style homogène (si quelques annotations au dessus des "privates" et quelques annotations au dessus des "getXxx()" --> ça ne fonctionne pas !!! sauf si annotation spéciale **@Access** de JPA 2 avec *AccessType.FIELD* ou *PROPERTY*)
- Les annotations paramétrant des injections de dépendances (@Resource , @PersistenceContext , ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "set".
- Même si une propriété (avec private +get/set) n'est marquée par aucune annotation , elle sera tout de même prise en compte lors du mapping objet-relationnel.
- Pour éventuellement (si nécessaire) désactiver le mapping d'une propriété d'une classe persistante, il faut utiliser l'annotation "**@Transient**" (au dessus du "private" ou du "getter" de la propriété).
- **@Enumerated(EnumType.STRING)** permet de stocker la valeur d'une énumération sous forme de chaîne de caractères dans une colonne d'une table relationnelle.
- **@Temporal(TemporalType.DATE** ou **TemporalType.TIME** ou **TemporalType.TIMESTAMP)** permet de préciser la valeur significative d'une date java (java.util.Date) .
- ...

IV - EntityManager et entités persistantes

1. Entity Manager et son contexte de persistance



Un objet persistant (@Entity avec toutes ces annotations) doit être pris en charge par un **gestionnaire d'objets persistants** (implicitement associé à un "*contexte de persistance*") de façon à ce que la liaison avec la base de données soit bien gérée.

Ci après , figure l'interface prédéfinie "**EntityManager**".

Celle-ci comporte tout un tas de **méthodes fondamentales** permettant de **déclencher/contrôler des opérations classiques de persistance** (*recherches, mises à jour , suppressions , ...*).

```
package javax.persistence;

/*Interface used to interact with the persistence context. */
public interface EntityManager {

    public void persist(Object entity); /* Make an instance managed and persistent.*/

    public <T> T merge(T entity); /* Merge the state of the given entity
                                   into the current persistence context. */

    public void remove(Object entity); /* Remove the entity instance. */

    public <T> T find(Class<T> entityClass, Object primaryKey); /* Find by primary key. */
    public <T> T getReference(Class<T> entityClass, Object primaryKey);
                                   /* Get an instance, whose state may be lazily fetched. */

    public void flush(); /* Synchronize the persistence context to the underlying database.*/
    public void setFlushMode(FlushModeType flushMode);
}
```

```

public FlushModeType getFlushMode();
public void lock(Object entity, LockModeType lockMode);
    /* Set the lock mode for an entity object contained in the persistence context. */

public void refresh(Object entity); /* Refresh the state of the instance from the database */

public void clear(); /* Clear the persistence context */
public boolean contains(Object entity);
    /* Check if the instance belongs to the current persistence context */
public Query createQuery(String qlString);
    /* Create an instance of Query for executing a JPQL statement */
public Query createNamedQuery(String name); /* in JPQL or native SQL */
public Query createNativeQuery(String sqlString);
public Query createNativeQuery(String sqlString, Class resultClass);
public Query createNativeQuery(String sqlString, String resultSetMapping);
public void joinTransaction(); /* join JTA active transaction */
public Object getDelegate(); /* implementation specific underlying provider */

public void close(); /* Close an application-managed EntityManager */
public boolean isOpen();
public EntityTransaction getTransaction();
}

```

Un **contexte de persistance** (associé à *EntityManager*) peut:

- soit être **créé et géré par le conteneur** (ex: EJB , éventuellement WEB)
puis injecté via l'annotation "**@PersistenceContext**" ou bien récupéré via un lookup JNDI .
- soit être explicitement créé via **createEntityManager()** de **EntityManagerFactory**().
Son cycle de vie est alors à gérer (==> **.close()** à explicitement appeler)

2. Transaction JPA

L'interface "EntityManager" d'un **contexte de persistance** comporte une méthode **getTransaction()** retournant de façon abstraite un objet permettant de contrôler (totalement ou partiellement) la transaction en cours:

```
public interface EntityTransaction {

    public void begin(); /* Start a resource transaction. */
    public void commit(); /* Commit the current transaction */
    public void rollback(); /* Roll back the current transaction.*/

    public void setRollbackOnly(); /* Mark the current transaction so that the only possible
    outcome of the transaction is for the transaction to be rolled back.*/
    public boolean getRollbackOnly();

    public boolean isActive();
}
```

Remarque importante:

Ces transactions JPA peuvent en interne s'appuyer sur:

- *des transactions simples et locales JDBC*
ou bien
- *des transactions complexes JTA (distribuées)*

Selon la configuration de **META-INF/persistence.xml** et la configuration du serveur d'application hôte (Jboss, WebSphere, Tomcat ,).

D'autre part, en fonction du contexte d'intégration de JPA, les transactions seront explicitement ou implicitement gérées d'une des façons suivantes:

- **gestion déclarative** (et automatique/implicite) des **transactions** au sein des **EJB3**
- **gestion déclarative** (et automatique/implicite) des **transactions** au sein de **Spring**
- **gestion explicitement programmée des transactions** (sans framework d'intégration)

Exemple (avec gestion explicite):

```
try {
    entityManager.getTransaction().begin();
    Adresse nouvelleAdresse = new Adresse("rue elle", "80000", "Amiens");
    Client nouveauClient = new Client();
    nouveauClient.setNom("nom du nouveau client");
    nouveauClient.setAdressePrincipale(nouvelleAdresse);
    this.entityManager.persist(nouveauClient);
    num_cli = nouveauClient.getIdClient();
    System.out.println("id du nouveau client: " + num_cli);
    entityManager.getTransaction().commit();
}
catch (Exception e) {
    this.entityManager.getTransaction().rollback();
    e.printStackTrace();
}
```


3. Différents états - objet potentiellement persistant

Etats objets	Caractéristiques
transient (proche état détaché)	Nouvel objet créé en mémoire, pris en charge par la JVM Java mais par encore contrôlé par une session Hibernate ou bien l'entityManager de JPA (le mapping objet/relationnel n'a jamais été activé). un tel objet n'a quelquefois par encore de clef primaire (elle sera souvent attribuée plus tard lors d'un appel à entityManager.persist() ou session.save())
persistant	objet actuellement sous le contrôle d'une session Hibernate ou de l'entityManager de JPA (<i>avant session.close() ou entityManager.close()</i>). Le mapping objet/relationnel est alors actif et une synchronisation (mémoire ==> base de données) est alors automatiquement déclenchée suite à une mise à jour (changement d'une valeur d'un attribut).
détaché	<p>objet qui n'est plus sous le contrôle d'une session Hibernate ou de l'entityManager de JPA (<i>après session.close() ou entityManager.close()</i>) . Les valeurs en mémoire sont conservées mais ne seront plus mises à jour (mapping objet/relationnel désactivé).</p> <p>Un objet détaché pourra éventuellement être ré-attaché à une session Hibernate (lors d'un update ou save_or_update() sur une session hibernate ou lors d'un appel à merge() sur l'entityManager de JPA) et ses éventuelles nouvelles valeurs pourront alors être synchronisées dans la base de données.</p>

4. Cycle de vie d'un objet JPA/Hibernate

transient ==> persistant (==> ...)

(via **new**)

(via **session.save()** ou
entityManager.persist())

insert into

ou bien (après hibernation prolongée):

persistant ==>

(via **query**)

select

détaché

(via session.close() ou
entityManager.close())

(==> **persistant**)

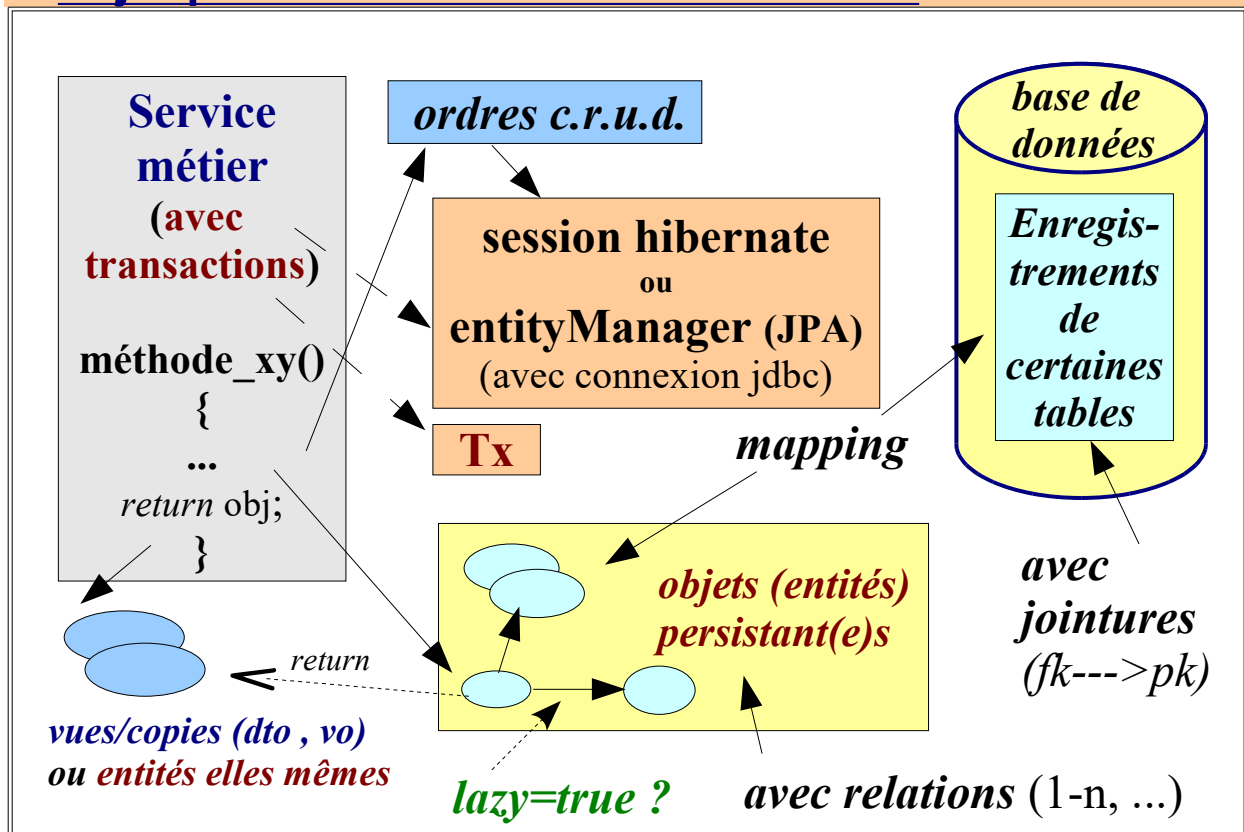
(via **session.update()** , ...
ou **entityManager.merge()**)

update set ...

5. Synchronisation automatique dans l'état persistant

Synchronisation automatique dans l'état persistant (.update() ou .persist() non obligatoire), mais .save() ou .merge() obligatoire pour un objet détaché.

6. objet persistant et architecture n-tiers



Remarques importantes:

- La technologie O.R.M. JPA ou hibernate sert essentiellement à remonter (et gérer) en mémoire un ensemble d'objets Java en tous points synchronisés avec les enregistrements d'une base de données relationnelle.
- Pour obtenir de bonnes performances (`lazy=true`) sans pour autant trop compliquer le code de l'application, il faut considérer les objets "entités persistantes" comme une structure d'ensemble orientée objet virtuellement liée à l'ensemble des données de la base (sachant que les mécanismes internes de JPA/Hibernate remontent les données en mémoire qu'en fonction des accès réellement effectués sur la structure objet [`lazy=true`]).
- La couche métier appelante (généralement développée avec Spring ou des EJB) doit souvent retourner des valeurs vers la couche présentation (IHM). Ces valeurs sont soit des références directes sur les entités persistantes elles mêmes ou bien des copies partielles sous formes de vues métiers (objets sérialisables / D.T.O. / V.O.) .
- Lorsque JPA/Hibernate est intégré dans Spring ou les EJB, les transactions sont automatiquement gérées par le conteneur et le code est alors significativement simplifié.

7. Principales méthodes JPA / EntityManager et Query

Principales méthodes de l'objet **EntityManager**:

méthodes	traitements
.find (class,pk)	Recherche en base et retourne un objet ayant les classe java et clef primaire indiquées
.createQuery (req_jpql)	Créer une requête JPQL et retourne cet objet
.refresh (obj)	Met à jour les valeurs d'un objet en mémoire en fonction de celles actuellement présentes dans la base de données (reload) .
.persist (obj_détaché) ou bien assez inutilement .persist (obj_déjà_persistant)	<u>Rend persistant un objet détaché :</u> Sauvegarde les valeurs d'un nouvel objet dans la base de données (insert into ...) et retourne une exception de type EntityExistsException si l'entité existe déjà . La clef primaire est quelquefois automatiquement calculée lors de cette opération
.merge (obj_détaché) ou bien assez inutilement .merge (obj_déjà_dans_contexte_persistence)	Sauvegarde ou bien met à jour (update ...) les valeurs d'un objet dans la base de données. Cette méthode s'appelle .merge() car l'entité passé en argument peut quelquefois être utilisée pour remplacer les valeurs d'une ancienne version (avec la même clef primaire) déjà présente dans le contexte de persistance.
.remove (obj)	Supprime l'objet (delete ... from where dans la base de données)
.flush () déclenché indirectement via .commit ()	Synchronise l'état de la base de données à partir des nouvelles valeurs des entités persistantes qui ont été modifiées en mémoire.

manager.**persist**() de **JPA** correspond à peu près au session.**save**() de **Hibernate**

manager.**find**() de **JPA** correspond à peu près au session.**get**() de **Hibernate**

manager.**remove**() de **JPA** correspond à peu près au session.**delete**() de **Hibernate**

manager.**merge**() de **JPA** correspond à peu près au session.**save_or_update**() de **Hibernate**

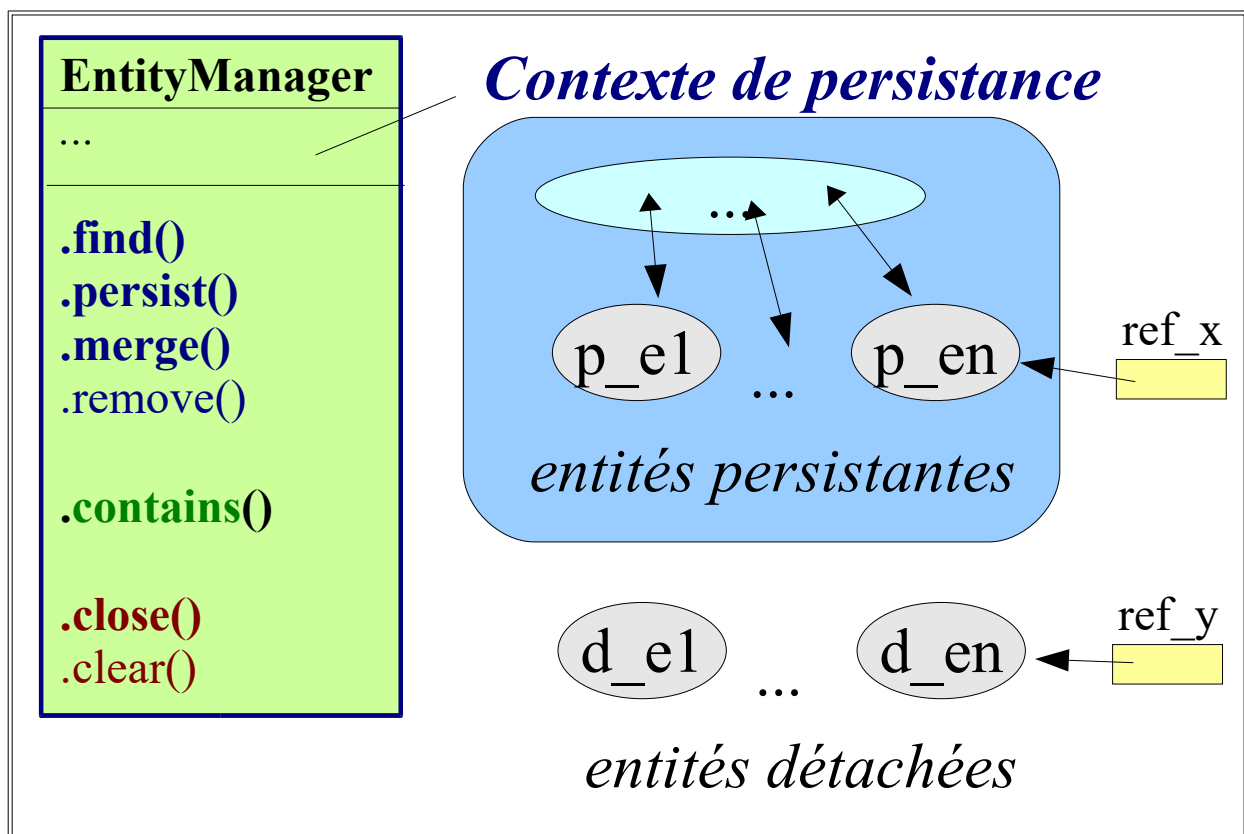
NB:

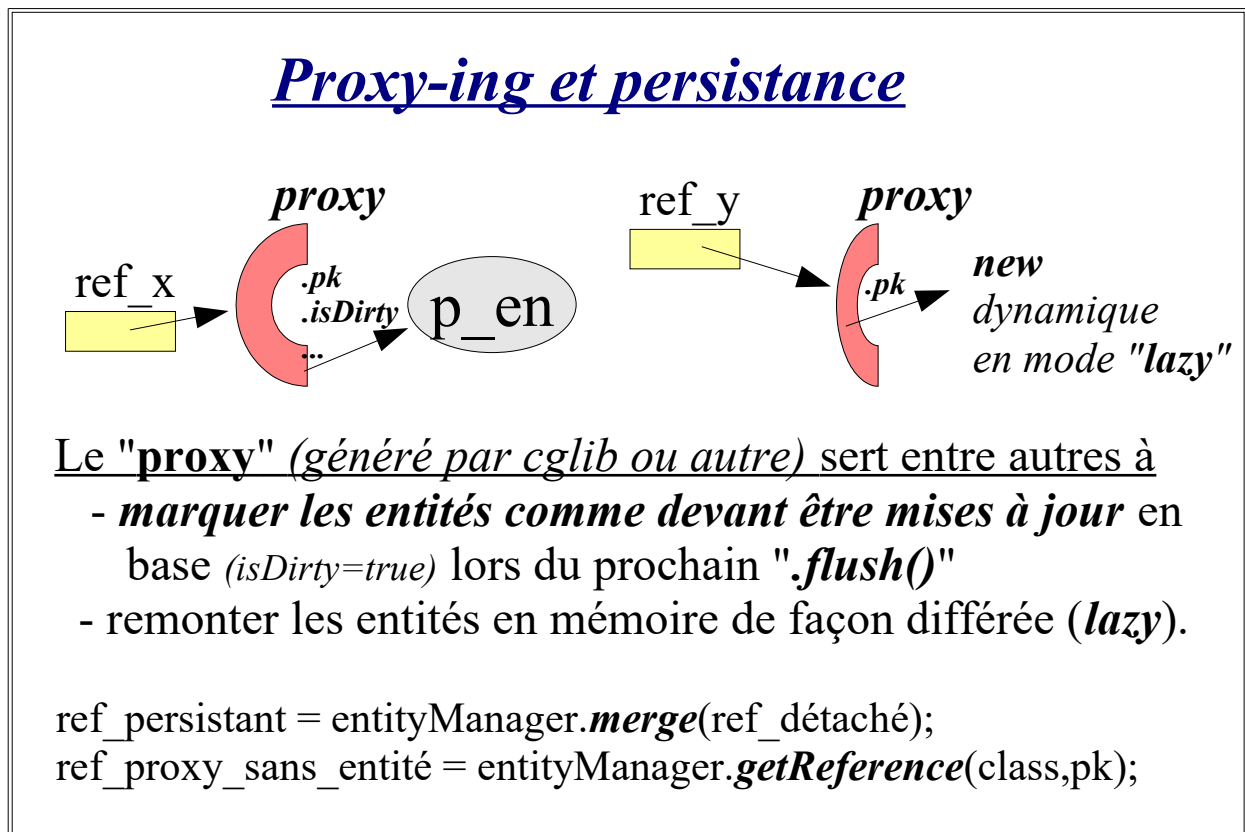
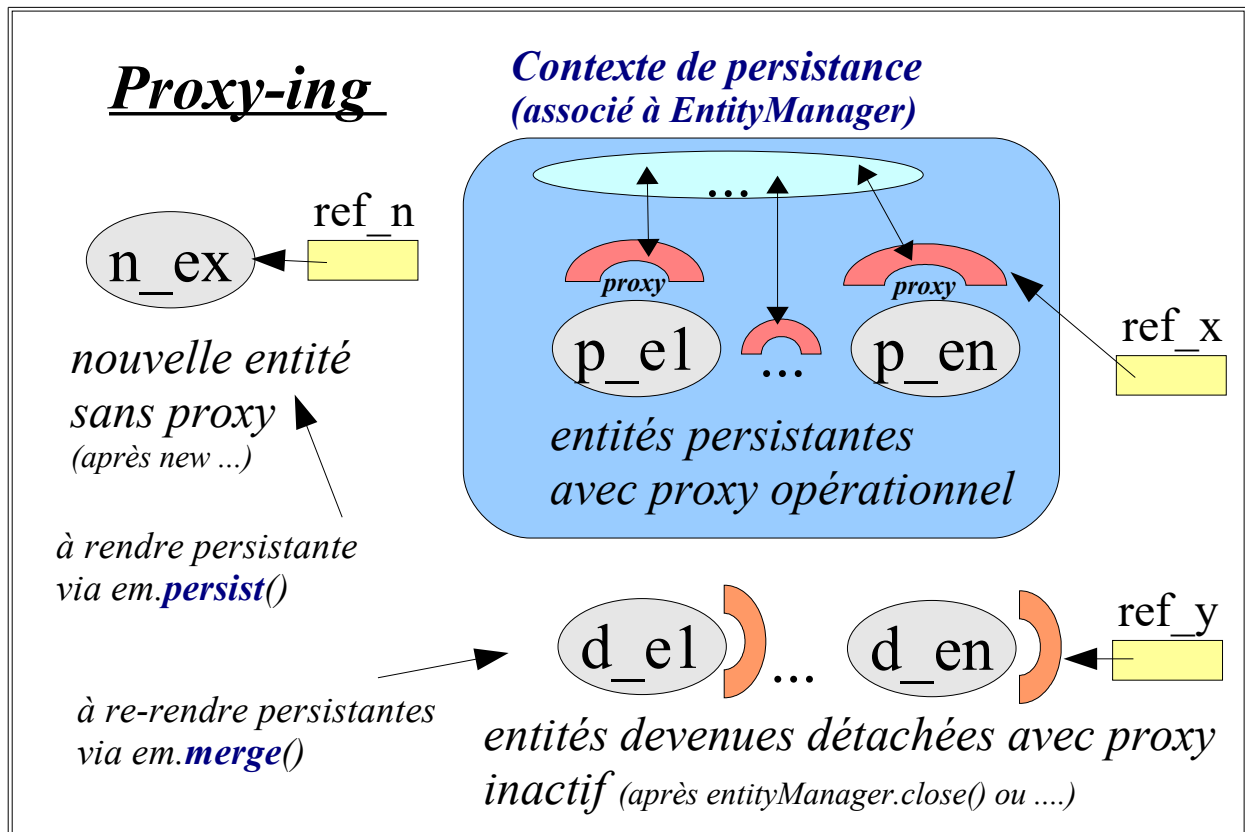
entityManager.getTransaction().commit() déclenche automatiquement **entityManager.flush()**

.clear() vide le contexte de persistance [tout passe à l'état détaché] sans effectuer de **.flush()**

Principales méthodes de l'objet **Query**:

méthodes	traitements
<code>.setParameter(paramName,paramValue)</code>	met à jour un paramètre de la requête jpql
<code>.getSingleResult()</code>	récupère l'unique objet d'une requête simple.
<code>.getResultList()</code>	retourne sous forme d'objet " <i>java.util.List</i> " la liste des objets retournés en résultat de la requête JPQL.
<code>.executeUpdate()</code>	exécute une requête autre qu'un select
<code>.setMaxResults()</code>	fixe le nombre maxi d'éléments à récupérer

8. Contexte de persistance et proxy-ing



V - Langage de requêtes JPQL

1.1. Présentation générale de HQL et JPQL

JPQL signifie *JPA Query Language*

HQL ou HbQL signifie *Hibernate Query Language*

Ces 2 langages correspondent à des *minis langages dérivés de SQL* et qui permettent d'exprimer des requêtes dans le cadre un mapping objet/relationnel géré par un contexte de persistance JPA ou par une session Hibernate.

1.2. Principaux éléments de syntaxe de JPQL et HQL

Le langage JPQL (ou HQL) n'est pas sensible à la casse pour sa partie SQL ("select" est équivalent à "SELECT") mais est sensible à la casse pour sa partie JAVA (la classe "Cat" n'est pas la même de "CAT" et la propriété c.weight n'est pas la même que c.WEIGHT).

Pour récupérer toutes les instances persistantes d'un certain type (c'est à dire tous les enregistrements de la table relationnelle associée) , une simple requête du genre "**from Cat**" (ou bien "**from ppp.Cat**" en précisant optionnellement le package java) est suffisante .

NB: La requête "**from Cat**" retourne non seulement des instances de la classe "**Cat**" mais également des instances des *éventuelles sous classes* (ex: "**DomesticCat**") . Le comportement de JPQL/HQL est bien en accord avec le "*polymorphisme*" du monde objet java.

De façon à pouvoir faire référence à une **instance** (du type précisé par from) dans une autre partie de la requête (ex: where) , on a généralement besoin d'un **alias** que l'on introduit par le mot clef optionnel "**as**" :

"from Cat **as** c" <==> "from Cat c"

"select c from Cat as c where c.weight > 8"

NB: la partie where est exprimée en utilisant la syntaxe **nom_alias_instance.nom_propriété** et non pas **nom_alias_table.nom_colonne** .

Autrement dit , la syntaxe de JPQL/HQL est centrée sur le monde objet (java) et pas sur le monde relationnel .

1.3. Quelques exemples de requêtes JPQL/HQL:

Rappel : syntaxe générale du **SELECT** du langage **SQL** :

```
SELECT fieldlist      FROM tablenames      [WHERE searchcondition]
      [GROUP BY fieldlist [HAVING searchconditions] ] [ ORDER BY fieldlist ]
```

Exemples JPQL/HQL:

```
SELECT cat.name FROM DomesticCat AS cat WHERE cat.name LIKE 'fri%'
```

```
SELECT avg(cat.weight), sum(cat.weight), max(cat.weight),
count(cat) FROM Cat cat ==> retourne une liste avec un seul
élément (ligne) de type Object[]
```

```
FROM Cat cat WHERE cat.mate.name is not null
```

```
SELECT c FROM Customer c JOIN c.orders o WHERE c.status = 1
(où c.orders est paramétré par @OneToMany ou @ManyToOne ...)
```

```
... where o.country IN ('UK', 'US', 'France')
```

```
... where town LIKE 'P%' où % signifie sous chaîne quelconque de 0 à n caractères
```

```
... where p.age BETWEEN 15 and 19
```

```
SELECT DISTINCT auth FROM Author auth
```

```
WHERE EXISTS
```

```
(SELECT spouseAuth FROM Author spouseAuth WHERE spouseAuth = auth.spouse)
```

```
SELECT auth FROM Author auth WHERE auth.salary >= ALL(SELECT a.salary FROM Author a
WHERE a.magazine = auth.magazine)
```

```
SELECT mag FROM Magazine mag WHERE (SELECT COUNT(art) FROM mag.articles art) > 10
```

```
SELECT pub FROM Publisher pub JOIN pub.magazines mag ORDER BY o.revenue, o.name
```

1.4. Lancement d'une requête JPQL (JPA)

Exemples:

```
import javax.persistence.TypedQuery;
...
public List<Client> getAllClient()
{
    TypedQuery<Client> query =
        entityManager.createQuery("Select c from Client as c" , Client.class);
    return query.getResultList();
} ...
```

```
public Categorie getCategorieprincipale() {
    Categorie cat=null;
    Query q = entityManager.createQuery("select c from Categorie as c "
                                     + "where c.parentId is null");
    cat = (Categorie) q.getSingleResult();
    return cat;
}

public List<Element> getElementByDesignation(String designation) {
    Query q = entityManager.createQuery("select e from Element as e "
                                     + "where e.designation=:design");
    q.setParameter("design", designation);      return q.getResultList();
}
```

Remarque importante:

Les API "Hibernate" et "JPA 1" sont nées à l'époque de la transition entre les jdk <=1.4 et les jdk >= 1.5 (lorsqu'il n'existait pas encore de "generic" : List au lieu de List<T>).

La méthode createQuery de l'interface EntityManager est surchargée.

Il existe une version qui renvoie un "TypedQuery" et qui accepte un second paramètre de type Class permettant de préciser le type de retour précis ce qui évite un warning .

D'autre part, les méthodes createQuery() et setParameter() ont été prévues pour s'enchaîner et l'on peut donc écrire :

```
return
entityManager.createQuery("select c from Client as c where c.age >=:ageMini", Client.class)
    .setParameter("ageMini",18)
    .getResultList();
```


1.5. Lancement d'une requête native avec JPA

```
Query query = entityManager.createNativeQuery(
    "SELECT NAME, SURNAME, AGE FROM PERSON");
List list = query.getResultList();
```

résultat ==>

une liste de tableaux (Object[]) avec NAME à l'index 0 , SURNAME à l'index 1 et AGE à l'index 2

Via une version surchargée de createNativeQuery ayant un second paramètre on peut quelquefois préciser le type de retour souhaité (si compatible) .

1.6. NamedQuery placée dans une classe d'entité persistante

```
@Entity
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
public class Country {
    ...
}
```

ou bien

```
@Entity
@NamedQueries({
    @NamedQuery(name="Country.findAll",
        query="SELECT c FROM Country c"),
    @NamedQuery(name="Country.findByName",
        query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
    ...
}
```

et appel via createNamedQuery("named_of_query" , ResultType.class) :

```
TypedQuery<Country> query =
    entityManager.createNamedQuery("Country.findAll", Country.class);

List<Country> results = query.getResultList();
```

Intérêt de "NamedQuery" par rapport aux requêtes ordinaires ?

---> c'est à la fois une **affaire de style** et une **affaire de performance**.

- Dans le cas où un service métier n'utilise aucun DAO en arrière plan et utilise l'entityManager en direct, les "NamedQuery" ont l'avantage de bien séparer détails de persistance avec logique métier.
- Etant placées au dessus des classes "@Entity" chargées en mémoire et analysées dès le démarrage de l'application, les requêtes de type @NamedQuery peuvent ainsi être **préparées** par le SGBDR et s'exécutent donc très vite.

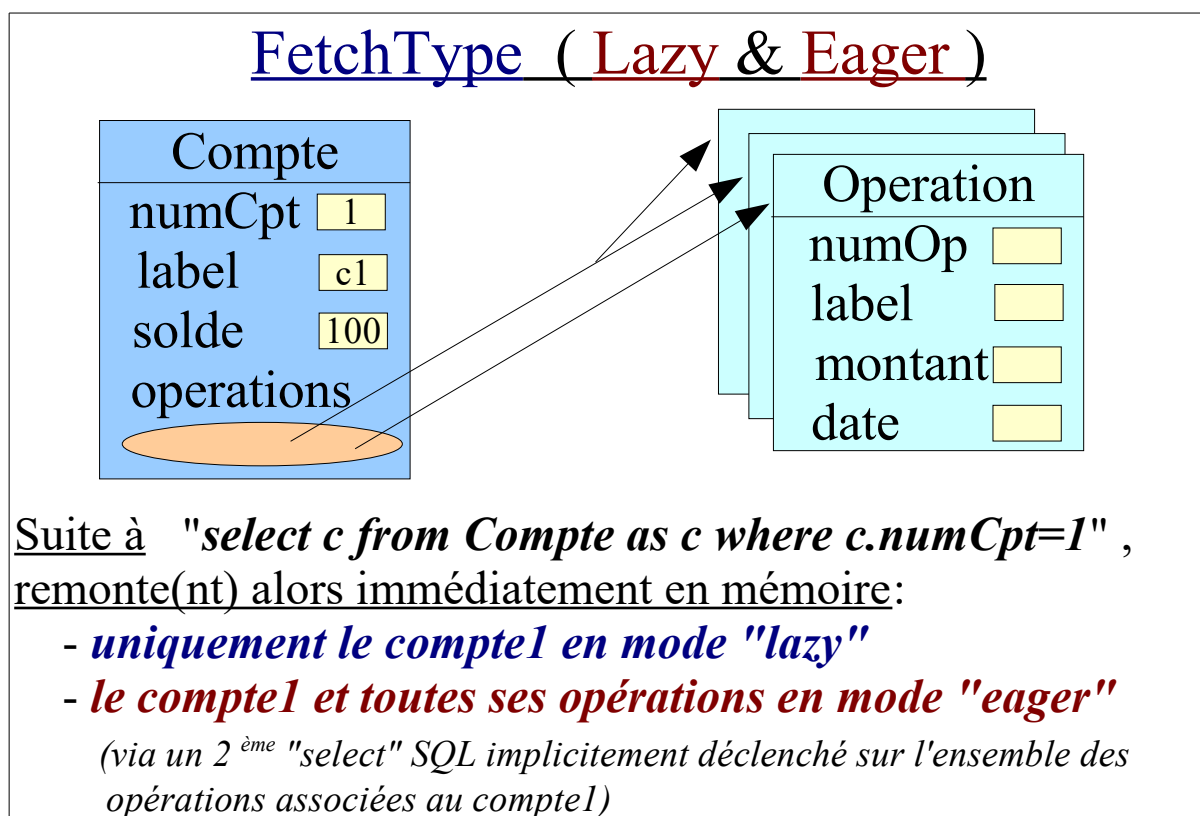
VI - O.R.M. JPA (généralités)

1. Vue d'ensemble sur les entités, valeurs et relations

1.1. Entités et objets valeurs (embedded)

- Une **entité** (@Entity) doit absolument comporter une **clef primaire** (@Id)
- Un sous objet imbriqué (@Embedded) n'a pas de clef primaire et doit absolument être référencé par une entité pour exister.

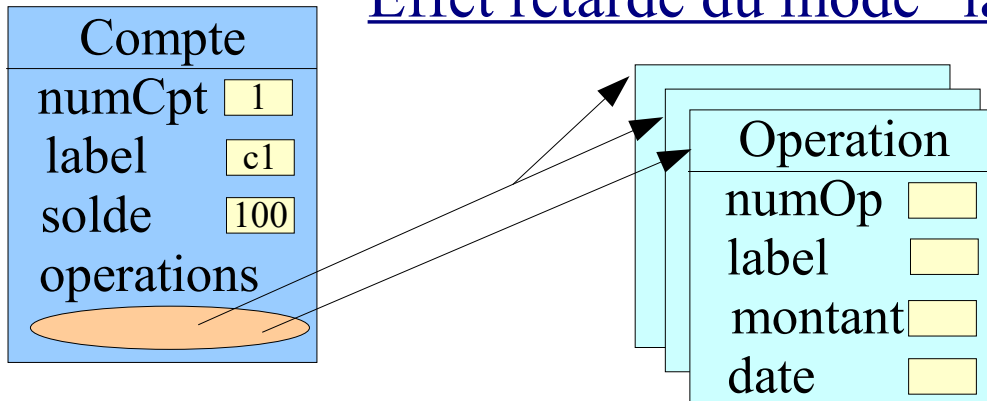
1.2. FetchType (lazy & eager)



L'attribut optionnel **lazy="true"** d'hibernate (ou bien **fetchType=Lazy** de JPA) permet de demander une *initialisation tardive* des membres d'une collection (les valeurs des éléments de la collection ne seront recherchées en base que lorsqu'elles seront consultées via par exemple un itérateur java). Ceci permet d'obtenir dans certains cas de meilleurs performances . Il faut cependant veiller à récupérer les valeurs de la collection avant le commit et la fermeture de la session (sinon ==> c'est trop tard : *LazyInitializationException*).

Il est fortement conseillé d'affecter explicitement la valeur "lazy" ou "eager" au paramètre **fetchType** (très important sur le plan des performances).

Effet retardé du mode "lazy"



- Si l'on accède qu'à "c1.getLabel()" et c1.getSolde() les opérations ne sont alors jamais remontées en mémoire.
- Par contre, *dès le parcours de la collection des opérations (via un itérateur ou une boucle for), Les entités "Opération" sont alors automatiquement remontées en mémoire par JPA via une série de petits "select" SQL (jusqu'à "1+N" en tout)".*

Exemple JPA:

---> fetch=FetchType.LAZY or fetch=FetchType.EAGER;

```
@OneToMany( fetch=FetchType.LAZY , mappedBy="compte" )
private List<Operation> dernieres_operations ;
```

NB : Si le mode "LAZY" est configuré dans une annotation JPA (ou dans un fichier hbm.xml), il est tout de même possible d'*expliquer ponctuellement une demande de chargement "tout d'un seul coup" via une requête JPQL* du type "select **distinct** c from Compte c **inner join fetch** c.operations inner join c.proprietaires cli where cli.numero = :numCli"

1.3. Operations en cascade

Lorsqu'une opération (delete , updade , save, ...) est effectuée sur une entité persistante, elle est alors automatiquement répercutée sur les sous objets de type "valeurs/embbded" .

Lorsque par contre on a affaire à une relation (1-1 ou 1-n) de type "entité-entité" , l'opération (save, update , delete) effectuée sur l'entité principale n'est pas automatiquement répercutée sur les entités liées. On peut cependant demander explicitement à JPA (ou hibernate) de répercuter une opération de mise à jour de l'entité principale vers une ou plusieurs entité(s) liée(s) via l'attribut **cascade** d'une relation .

Syntaxe JPA (dans les annotations):

```
@OneToMany(cascade=CascadeType.ALL , ...)
```

ou bien avec ensemble de cascades combinées:

```
cascade = {CascadeType.PERSIST, CascadeType.MERGE}, ...
```

NB: depuis JPA2 , la cascade "all" + "**delete-orphan**" permet en plus de supprimer automatiquement en base des entités "enfants" qui seraient détachées de leurs parents (suite un une

suppression de référence(s) dans une collection par exemple) .

NB: ceci fonctionne même si la mention "cascade" n'est pas précisée dans le schéma relationnel de la base de données.

2. Identité d'une entité (clef primaire)

2.1. Générateurs de clefs primaires

Une **clef primaire** (*ex*: numéro de facture) est assez souvent **générée automatiquement** via un **algorithme** basé sur la notion de **compteur**.

Exemple (JPA) :

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO) //pour auto incrémentation coté base
@Column(name="numero")
public long getNum() {
    return num;
}
```

2.2. Eventuelle clef primaire composée

Avec JPA:

Si une clef primaire est composée, il faut alors utiliser les annotations

@EmbeddedId (pour annoter la propriété liée au sous objet "clef primaire")

et

@Embeddable (pour annoter une *classe de clef primaire devant être sérialisable, publique, vue comme un "JavaBean" et devant coder equals() et hashCode()*).

Exemple de classe de clef primaire composée:

```
@Embeddable
public class CustomerPK implements java.io.Serializable
{
    private long id;
    private String name;

    public CustomerPK() { }

    public CustomerPK(long id, String name) {
        this.id = id;    this.name = name;
    }

    public long getId() { return id; }

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

```

public String getName() {    return name;  }

public void setName(String name) {    this.name = name;  }

public int hashCode() {
    return (int) id + name.hashCode();
}

public boolean equals(Object obj)
{
    if (obj == this) return true;
    if (!(obj instanceof CustomerPK)) return false;
    if (obj == null) return false;
    CustomerPK pk = (CustomerPK) obj;
    return pk.id == id && pk.name.equals(name);
}
}

```

```

@Entity
public class Customer implements java.io.Serializable
{
    CustomerPK pk;
    ...
    public Customer(){ }

    @EmbeddedId
    public CustomerPK getPk() {
        return pk;
    }

    public void setPk(CustomerPK pk){
        this.pk = pk;
    }
    ...
}

```

3. Propriétés d'une colonne

```
@Column(uptdatable = false, name = "flight_name", nullable = false, length = 50)
```

L'annotation @Column se transpose en sous balise <column> des fichiers de mapping hibernate (.hbm.xml).

4. Relations (1-1, n-1, 1-n et n-n)

<i>Annotations JPA</i>	<i>Significations</i>
@OneToOne	1-1 (fk [unique] ---> pk) [<i><many-to-one unique='true'> de hibernate</i>]
@OneToMany	1-n (collection avec clef [pk <----fk=clef])
@ManyToOne	n-1 (fk ----> pk)
@ManyToMany	n-n (fk1 ----> (k1,k2) ---> pk2)

NB: les annotations **@OneToOne** , **@OneToMany** et **@ManyToMany** peuvent éventuellement comporter un attribut "*mappedBy*" dont la valeur correspond à la *propriété qui sert à établir une correspondance inverse (et secondaire)* au sein d'une **relation bidirectionnelle** .

NB: les mises à jour des relations effectuées uniquement du côté secondaire d'une relation **bidirectionnelle** ne seront pas automatiquement sauvegardées (tant que le côté principal de la relation n'aura pas été explicitement réajusté).

L'annotation **@JoinColumn** permet d'indiquer (via *name*="...") le *nom de la colonne correspondant à la clef étrangère* dans la base de données.

exemple:

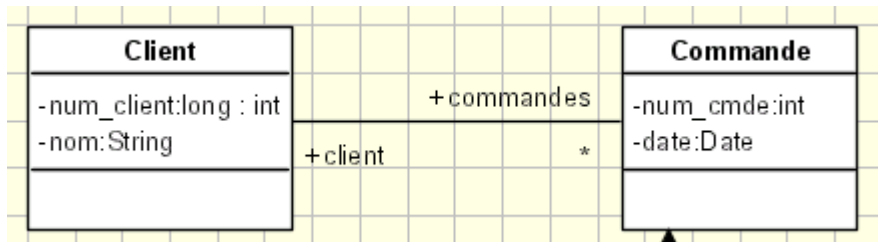
```
@OneToOne(cascade = {CascadeType.ALL})
@JoinColumn(name = "ADDRESS_ID")
public Address getAddress()
{
    return address;
}

public void setAddress(Address address)
{
    this.address = address;
}
```

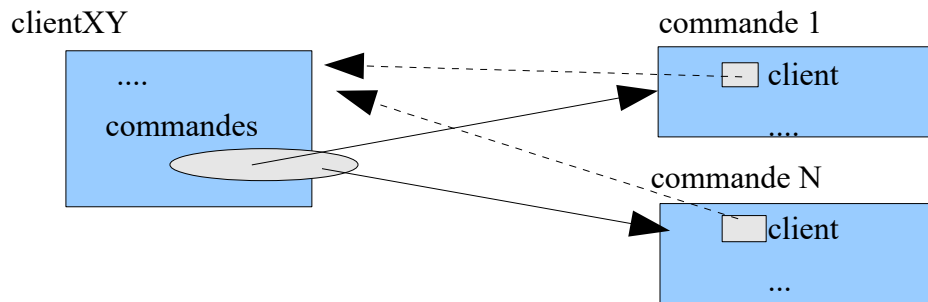
VII - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...)

1.1. Relations 1-n et n-1 (entité-entités)

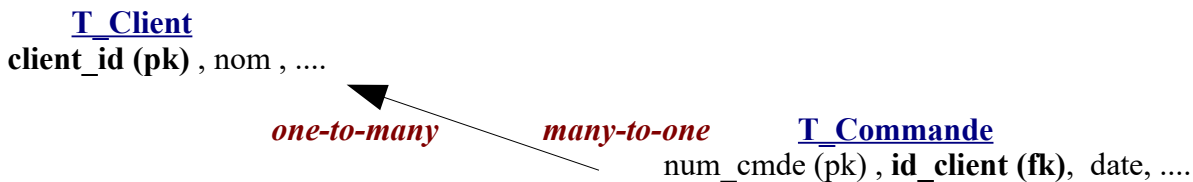
UML:



Java:



Base de données:



Annotations JPA:

```

@Entity
@Table(name = "T_Commande")
public class Commande
{
    @ManyToOne
    @JoinColumn(name = "id_client") //fk (nom de colonne dans la table)
    private Client client;

    ...

    public Client getClient() { return client; }
    public void setClient(Client client) { this.client = client; }
}
  
```

```

@Entity
@Table(name = "T_Client")
public class Client
{
    //NB : "client" (valeur de mappedBy) = nom (java) de la relation inverse
    @OneToMany(fetch=FetchType.LAZY, mappedBy="client")
    private List<Commande> commandes;

    ...

    public List<Commande> getCommandes() { return commandes; }
}
  
```

```

public void setCommandes(List<Commande> cmdes) { this.commandes = cmdes; }

public void addCommande(Commande c) {
    if (commandes == null) commandes = new ArrayList<Commande>();
    commandes.add(c); }
}

```

Rappels importants (pour JPA):

- La valeur de l'attribut "**mappedBy**" de l'annotation **@OneToMany** correspond à la *propriété (dans l'autre classe java) qui sert à établir une correspondance inverse* au sein d'une **relation bidirectionnelle**.
- L'annotation **@JoinColumn** permet d'indiquer (via name="...") le nom de la colonne correspondant à la clef étrangère dans la base de données.

1.2. Relations 1-n (avec JoinTable) évolutives vers du n-n

Bien qu'une relation 1-n (symétriquement n-1) soit souvent basée sur une jointure simple avec une clef étrangère en base, il est tout de même possible de mettre en place une relation 1-n avec une table de jointure en base. Déjà structurée en base comme une relation n-n, une telle relation 1-n a le mérite d'être facilement évolutive vers du n-n.

Exemple :

```

@Entity
@Table(name="Compte")
public class Compte {
    ...
    @ManyToOne //evolutif vers @ManyToMany
    @JoinTable(name="ClientCompte",
               joinColumns={@JoinColumn(name="num_compte")},
               inverseJoinColumns={@JoinColumn(name="num_client")})
    private Client client;
}

```

```

@Entity
@Table(name="Client")
public class Client extends Personne {
    ...
    @OneToMany(mappedBy="client")
    private List<Compte> comptes;
    ...
}

```

avec

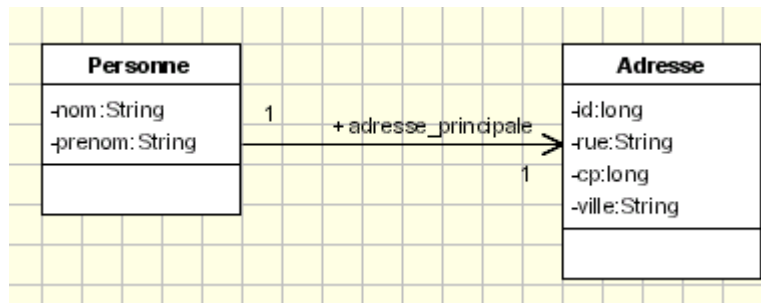
```

create table ClientCompte( num_client integer, num_compte integer,
                          primary key(num_client,num_compte));
ALTER TABLE ClientCompte ADD CONSTRAINT client_avec_compte_valide
    FOREIGN KEY (num_compte) REFERENCES Compte(numero);
ALTER TABLE ClientCompte ADD CONSTRAINT compte_avec_client_valide
    FOREIGN KEY (num_client) REFERENCES Client(num_client);

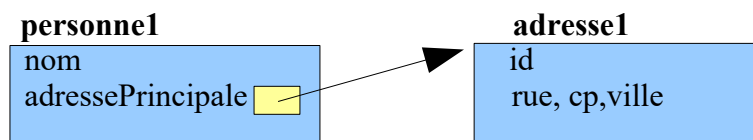
```


1.3. Relation 1-1 via clef étrangère (évolutif vers n-1)

UML:



Java:



====> exemple: `personne1.getAdressePrincipale().getVille();`

Base de données:

T_Personne

id (pk) , nom , prenom , , id_adr (fk)

@OneToOne (JPA)

T_Adresse

id (pk) , rue , cp , ville

1-1 ici évolutif vers n-1 (exactement structuré et paramétré comme n-1 et @ManyToOne) .
Lorsque c'est possible une contrainte **unique=true** sur la colonne id_adr (fk) peut indiquer une intention de 1-1 plutôt que n-1 .

JPA:

```

@Entity
public class Personne
{
    @OneToOne()
    @JoinColumn(name = "id_adr" , unique=true) //fk

    private Adresse adressePrincipale;

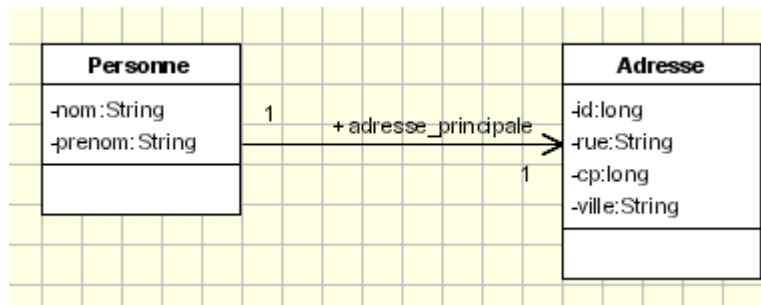
    ...

    public Adresse getAdressePrincipale() {
        return adressePrincipale;
    }

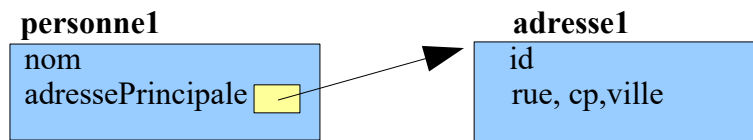
    public void setAdressePrincipale(Adresse adressePrincipale) {
        this.adressePrincipale = adressePrincipale;
    }
}
  
```

1.4. Relation 1-1 via Table secondaire et correspondance pk

UML:



Java:



====> exemple: `personne1.getAdressePrincipale().getVille();`

Base de données:

T_Personne

id (pk) , nom , prenom , ...

@OneToOne (JPA)

T_Adresse_De_Personne

idPers (pk) , rue , cp , ville

Les valeurs de idPers (clef primaire) de la table secondaire "T_Adresse_De_Personne" doivent absolument correspondre à celle de "id" (clef primaire) de la table principale "T_Personne". Cette correspondance rigide entre les clefs primaires des tables principale et secondaire n'est pas évolutif vers du n-1 . c'est du 1-1 pour toujours .

JPA:

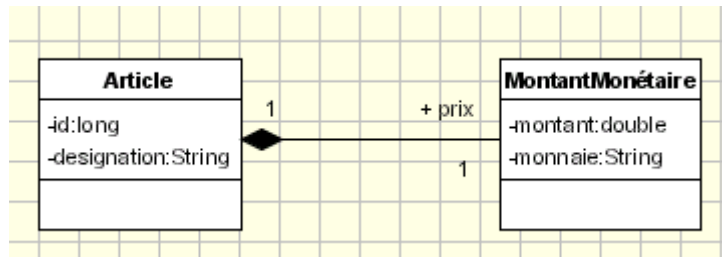
<pre> @Entity @Table(name="T_Personne") public class Personne { @Id @GeneratedValue(strategy=...IDENTITY) private Long id ; @OneToOne(optional=true , mappedBy="personne") private AdresseDePersonne adressePrincipale; ... } </pre>	<pre> @Entity @Table(name="T_Adresse_De_Personne") public class AdresseDePersonne { @Id private Long idPers; private String rue ; private String cp; private String ville; @OneToOne(optional=false) @PrimaryKeyJoinColumn private Personne personne; } </pre>
---	--

NB : Dans le cas rare où idPers de AdresseDePersonne ne serait pas explicitement définie comme clef primaire / id , l'annotation @PrimaryKeyJoinColumn comporte un attribut facultatif "referencedColumnName" .

1.5. Sous objets incorporable au sein d'une entité

Appelés objets "*Valeur composite*" ou "*Component*" dans la terminologie *Hibernate* , et appelés objets "*Embeddable*" dans la terminologie *JPA*, d'éventuels sous objets composés (non partageables et sans clef primaire) peuvent être utilisés pour structurer une entité.

UML:



En base de données:

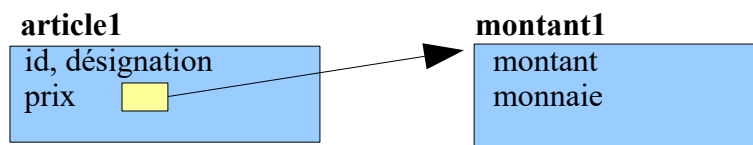
T Article

id (pk) , designation , *montant* , *monnaie* ,

==> mapping relationnel associé (par défaut):

- pas de table relationnelle séparée pour les sous objets "valeurs"
- liste des propriétés du composant directement incluse dans la table de l'objet contenant .

En Java:



==> `xxx.getPrix().getMontant()`; et `xxx.getPrix().getMonnaie()`;

NB: La classe java du composant imbriqué n'a pas d' ID (pas de clef primaire) . Elle respecte néanmoins les conventions "**JavaBean**" pour les éléments internes (ici montant et monnaie) .

JPA:

Une classe d'objet incorporable doit normalement être annotée via "**@Embeddable**" .

Une propriété d'une entité qui référence un objet imbriqué doit être annotée via **@Embedded** .

Les spécifications JPA se limitent à un seul niveau d'imbrication.

exemple:

@Embeddable

```

public class MontantMonetaire
{
    private double montant; // + get/set
    private String monnaie; // + get/set
    public MontantMonetaire() { }

    public MontantMonetaire(double montant, String monnaie) {
        this.montant = montant;    this.monnaie = monnaie;
    }
}
  
```

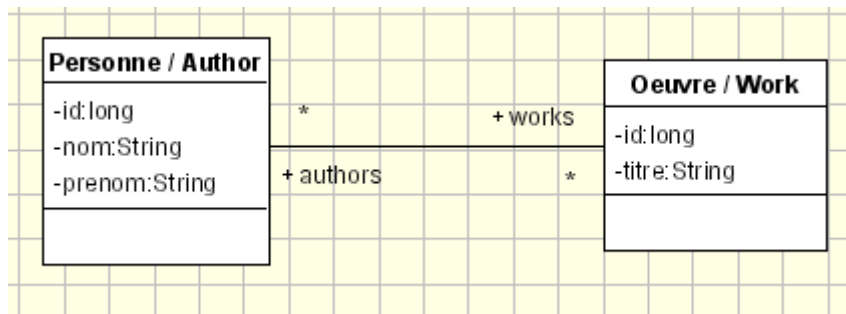
```
@Entity
@Table(name = "T_Article")
public class Article
    private long id; // avec get/set et @Id

    @Embedded
    /* @AttributeOverrides({
        @AttributeOverride(name = "montant", column = @Column(name = "montant")),
        @AttributeOverride(name = "monnaie", column = @Column(name = "monnaie"))
    }) */
    private MontantMonetaire prix;
    ...
    public MontantMonetaire getPrix() { return prix; }
    public void setPrix( MontantMonetaire prix) { this.prix = prix; }
}
```

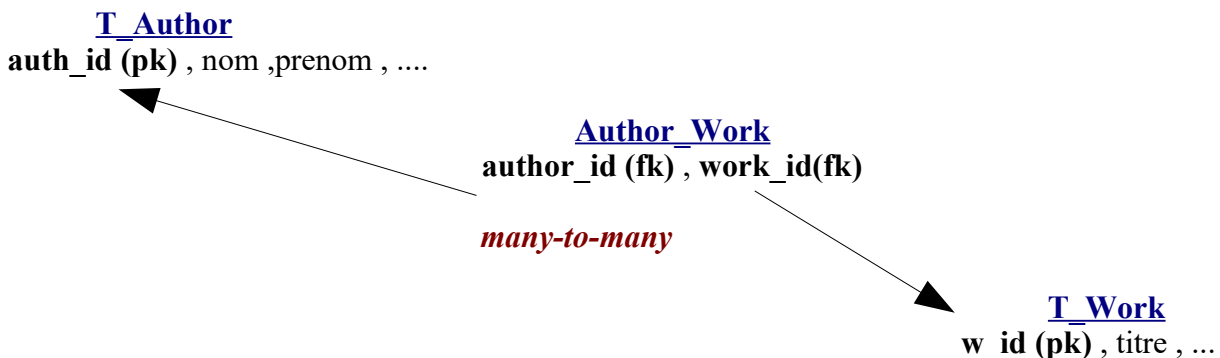
NB : `@AttributeOverride` n'est réellement utile que si les noms des colonnes sont différentes des noms des propriétés java de l'objet incorporé .

1.6. Relation n-n ordinaire

UML:



Mapping relationnel ==> *Table intermédiaire*



Java:

--> des collections dans les 2 sens
avec un sens principal et un sens secondaire (avec mappedBy).

JPA:

du côté "principal" de la relation (coté où les mises à jour seront "persistées"):

```
@Entity
public class Work
{
    @ManyToMany( fetch = FetchType.LAZY)
    @JoinTable(name = "Author_Work",
        joinColumns = {@JoinColumn(name = "work_id")},
        inverseJoinColumns = {@JoinColumn(name = "author_id")})
    private List<Author> authors;
    ...
    public List<Author> getAuthors() {
        return authors;
    }
    public void setAuthors(List<Author> authors) { this.authors = authors; }
    ..
}
```

NB: l'attribut **joinColumns** (de @JoinTable) correspond à la **clef étrangère pointant vers l'entité courante** et l'attribut **inverseJoinColumns** correspond à la **clef étrangère pointant vers les éléments de l'autre coté de la relation (ceux qui seront rangés dans la collection paramétrée)**.

et du côté inverse/secondaire:

```

@Entity
public class Author
{
    @ManyToMany(mappedBy="authors")
    private List<Work> works;
    ...
    public List<Work> getWorks() {
        return works;
    }
    public void setWorks(List<Work> works) { this.works = works; }
}

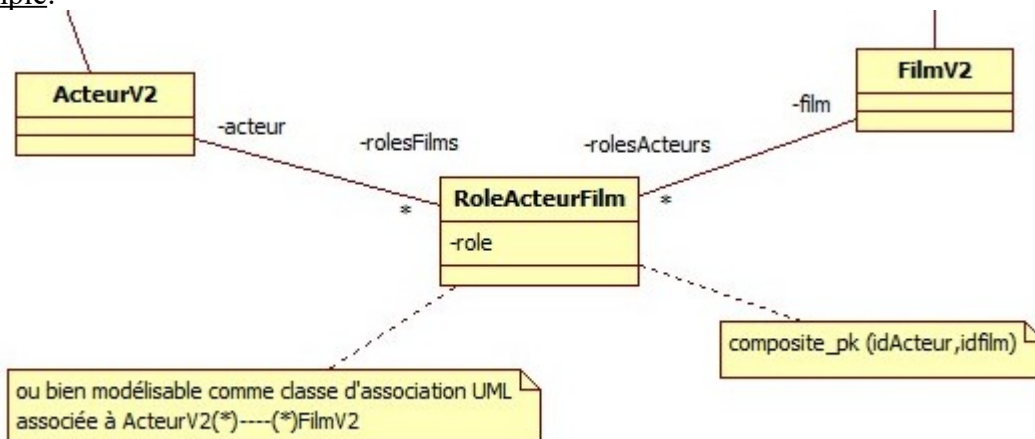
```

1.7. Relation n-n avec détails d'association

Remarque importante sur les relations (n-n):

Lorsque la table de jointure comporte des informations supplémentaires autres que les clefs étrangères, on est alors obligé de décomposer la relation n-n en deux relations (1-n + n-1) et les éléments de la table de jointure sont alors vus comme des entités ayant une clef primaire (souvent composée par les 2 clefs étrangères).

Exemple:



```

CREATE TABLE ActeurFilmV2
( idActeur integer,
  idFilm integer ,
  role VARCHAR(64),
  primary key(idActeur,idFilm));

ALTER TABLE ActeurFilmV2 ADD CONSTRAINT avec_acteur_valideV2
FOREIGN KEY (idActeur) REFERENCES Acteur(idActeur);
ALTER TABLE ActeurFilmV2 ADD CONSTRAINT avec_film_valideV2
FOREIGN KEY (idFilm) REFERENCES Film(idFilm);

```

@Entity

```

@Table(name="Acteur")
public class ActeurV2 {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long idActeur ;

    private String nom ;

    @OneToMany(mappedBy="acteur")
    private List<RoleActeurFilm> rolesFilms;

    ...
}

```

```

@Entity
@Table(name="ActeurFilmV2")
public class RoleActeurFilm {
    @EmbeddedId
    private RoleActeurFilmCompositePk pk ;

    private String role;

    @ManyToOne
    @JoinColumn(name="idActeur")
    @MapsId("idActeur") //pk.idActeur
    private ActeurV2 acteur;

    @ManyToOne
    @JoinColumn(name="idFilm")
    @MapsId("idFilm") //pk.idFilm
    private FilmV2 film;

    public RoleActeurFilm() {
        super(); pk = new RoleActeurFilmCompositePk();
    }

    public RoleActeurFilm(String role, ActeurV2 acteur, FilmV2 film) {
        super();
        this.role = role; this.acteur = acteur; this.film = film;
        pk=new RoleActeurFilmCompositePk(acteur.getIdActeur(),
                                           film.getIdFilm());
    }
    //+get/set , ...
}

```

NB: Avec **JPA 1.0** , @MapsId() n'existait pas et @JoinColumn(name="id_...", insertable=false, updatable=false) était indispensable pour que JPA puisse gérer un double mapping où les colonnes "idFilm" et "idActeur" sont déjà précisées au sein de la clef primaire (composite pk).

Depuis **JPA 2**, l'annotation **MapsId("id...")** permet de préciser la sous partie de la classe de clef primaire composite java qui sera associée à la clef étrangère (du @ManyToOne / @JoinColumn) .
 Un futur appel à `roleActeurFilm.setActeur(acteur)` déclenchera alors automatiquement une mise à jour de **pk.idActeur** .

```

@Embeddable
public class RoleActeurFilmCompositePk implements Serializable {
    private static final long serialVersionUID = 1L;

    private long idActeur;
    private long idFilm;

    public RoleActeurFilmCompositePk() {
        super();
    }

    public RoleActeurFilmCompositePk(long idActeur, long idFilm) {
        super();
        this.idActeur = idActeur;
        this.idFilm = idFilm;
    }

    //+get/set , equals et hashCode (generated by eclipse)
}

```

```

@Entity
@Table(name="Film")
public class FilmV2 {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long idFilm;
    private String producteur;
    private String titre;

    @Column(name="dateSortie")
    @Temporal (DATE) //partie significative = DATE et pas DateTime
    public Date date;

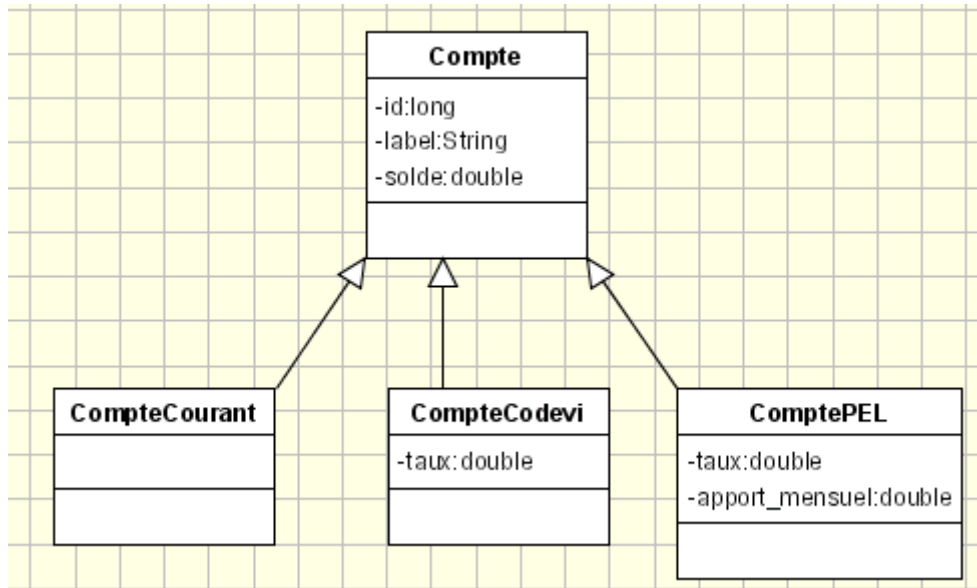
    @OneToMany (mappedBy="film")
    private List<RoleActeurFilm> rolesActeurs ;

    ...
}

```


2. Relations d'héritage & polymorphisme

UML:



2.1. stratégie "une seule table par hiérarchie de classes"

Stratégie "une table par hiérarchie de classes"
(avec colonne discriminante)

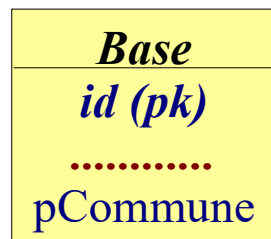


Table "Base_et_dérivés"

<i>id</i>	<i>typeDérivé</i> (discriminateur)	<i>pC.</i>	<i>p1a</i>	<i>p1b</i>	<i>p2</i>	<i>p3</i>
1	D1	xx	aa	bb	Null	Null
2	D2	yy	Null	Null	22	Null
3	D3	zz	Null	Null	Null	33

Colonne discriminante (*typeDérivé*)

Null (N/A)

Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.

Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

Cette stratégie (relativement simple) est assez pratique et adaptée dans le cas où il y a peu de différences structurelles entre les sous classes .

Contrainte : les colonnes liées aux attributs des sous classes ne peuvent pas avoir la contrainte "NOT NULL" . (*Pour une instance de la sous classe D1 , les colonnes inutilisées de la sous classe D2 doivent avoir la valeur NULL*) .

Pour la classe de base (JPA):

```
@Entity
@Table(name="Compte")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "typeCompte",
                    discriminatorType = DiscriminatorType.STRING)
/* @DiscriminatorValue("CompteCourant") //éventuellement en tant que valeur par défaut
// et si pas de sous classe CompteCourant mais utilisation directe du parent "Compte" */
public class Compte
{
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String label;
    private double solde;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getLabel() { return label; }
    public void setLabel(String label) { this.label = label; }

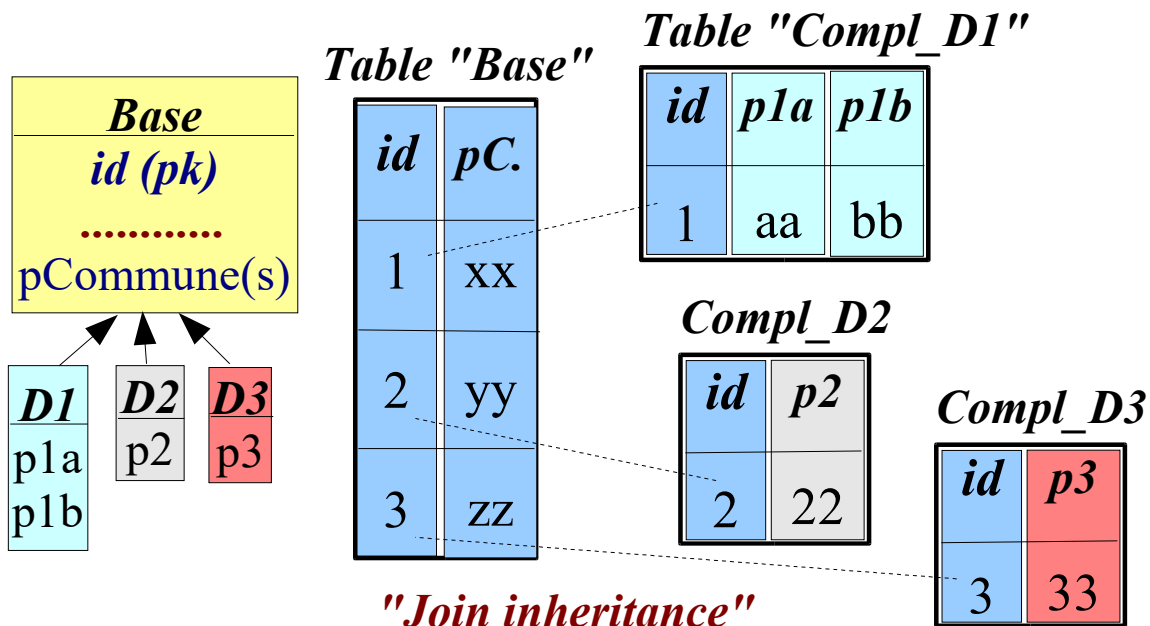
    public double getSolde() { return solde; }
    public void setSolde(double solde) { this.solde = solde; }
}
```

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("PEL")
public class ComptePEL extends Compte
{
    private double tauxInteret; //+get/set
    private double apportMensuel; //+get/set
    ...
}
```

2.2. Stratégie "Join_inheritance"

Stratégie "table principale

+ une table (de compléments) par classe fille"



```
@Entity
@Table(name="Compte")
@Inheritance(strategy=InheritanceType.JOINED )
public class Compte
{
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String label; // +get/set
    private double solde; // +get/set

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    ...
}
```

NB : Bien que "pas indispensable", une éventuelle colonne discriminante pourrait éventuellement être ajoutée si l'implémentation de JPA en tient compte du point de vue des performances.

```
@Entity
@Table(name="ComptePEL")
@Inheritance(strategy=InheritanceType.JOINED )
public class ComptePEL extends Compte
{
    private double tauxInteret; //+get/set
    private double apportMensuel; //+get/set

    ...
}
```

2.3. Stratégie (assez rare) "une table par classe"

Avec tout un tas de restrictions (voir documentation de référence si besoin)

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Compte
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    ...
    public int getId()    {    return id;    }

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

    ...
}
```

```
@Entity
public class ComptePEL extends Compte
{
    private double tauxInteret; // +get/set
    ...
}
```

2.4. Polymorphisme

Une requête JPQL/HQL exprimée avec un type correspondant à une classe parente retournera (par défaut / sans restriction explicite) toutes les instances de toute une hiérarchie de classes (classe mère + classe fille). Le type exact de chacune des instances retournées sera précis (bien que compatible avec le type de la sur-classe) ==> **polymorphisme complet** (java/mémoire + au niveau O.R.M.).

Autrement dit :

select c from Compte as c retournera tous les types de Compte (courant , codevi , pel).

"select c from ComptePEL as c" ne retournera que des comptes de types "PEL" .

VIII - JPA : aspects divers et avancés

1. Quelques spécificités de JPA

1.1. Une entité répartie dans 2 tables (principale, secondaire)

NB: avec JPA, il est également possible d'associer quelques propriétés d'une entité à une table secondaire :

```
@Entity
@Table(name = "CUSTOMER")
@SecondaryTable(name = "EMBEDDED_ADDRESS")
public class Customer implements java.io.Serializable
{
    @Column(name = "STREET", table = "EMBEDDED_ADDRESS")
    private String street ;

    @Column(name = "CITY", table = "EMBEDDED_ADDRESS")
    private String city ;

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

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

==> dans ce cas (clef primaire table_secondeire = clef_primaire_table_principale)
[cas du <one-to-one> des .hbm.xml de hibernate]

1.2. Éléments requis sur une classe d'entité (JPA):

- vraie classe (pas interface ni enum) non finale (pas de mot clef "final" , on doit pouvoir en hériter).
- un constructeur par défaut (sans argument) obligatoire et devant être public ou protégé (d'autres constructeurs sont possibles mais non indispensables).
- annotation "@Entity" ou bien <entity ...> dans un descripteur Xml.
- éventuelle implémentation de *java.io.Serializable* si besoin d'un transfert distant à l'état détaché.

NB:

- Des héritages sont possibles (entre "entité" et "non entité" et vice versa) et le polymorphisme peut s'appliquer sans problème.
- L'état d'une entité est liée à l'ensemble des valeurs de ses attributs (devant être privés ou protégés) .

- L'accès aux propriétés d'une entité doit s'effectuer via des "getter/setter" (conventions JavaBean).

La persistance (mapping ORM) s'applique sur toutes les propriétés non "transientes" d'une entité.

Sachant qu'une éventuelle propriété non persistante doit être marquée via "private transient" et par l'annotation "@Transient" .

1.3. Verrous (optimistes et pessimistes)

```
@Version
@Column(name = "OPTLOCK")
public Integer getVersion()
{
    return version;
}

public void setVersion(Integer i)
{
    version = i;
}
```

2. Cycle de persistance (pour @Entity) et annotations/callbacks associées pour "Listener"

<i>Annotations (callback)</i>	<i>déclenchement (étape du cycle de persistance)</i>
@PrePersist	juste avant passage à l'état persistant
@PostPersist	juste après passage à l'état persistant (clef primaire souvent affectée)
@PreRemove	juste avant suppression
@PostRemove	juste après suppression et destruction
@PreUpdate	juste avant mise à jour
@PostUpdate	juste après mise à jour
@PostLoad	juste après chargement en mémoire (suite à une recherche)

```

@Entity
@Table(name = "CUSTOMER")
@EntityListeners(CustomerCallbackListener.class)
public class Customer implements java.io.Serializable
{
    ...
}

```

```

public class CustomerCallbackListener
{
    @PrePersist
    public void doPrePersist(Customer customer) {
        System.out.println("doPrePersist: About to create Customer: " + ...);
    }

    @PostPersist public void doPostPersist(Customer customer) {
        System.out.println("doPostPersist: Created Customer: " + ...);
    }

    @PreRemove public void doPreRemove(Customer customer) {
        System.out.println("doPreRemove: About to delete Customer: " + ...);
    }

    @PostRemove public void doPostRemove(Customer customer) {
        System.out.println("doPostRemove: Deleted Customer: " + ...);
    }

    @PreUpdate public void doPreUpdate(Customer customer) {
        System.out.println("doPreUpdate: About to update Customer: " + ...);
    }

    @PostUpdate public void doPostUpdate(Customer customer) {
        System.out.println("doPostUpdate: Updated Customer: " + ...);
    }

    @PostLoad public void doPostLoad(Customer customer) {
        System.out.println("doPostLoad: Loaded Customer: " + ...);
    }
}

```

3. Map et apports de JPA 2.0

3.1. Détails sur persistance d'une énumération

@Enumerated(*EnumType.STRING*)

ou bien

@Enumerated(*EnumType.ORDINAL*) //by default

à placer au dessus d'une propriété dont le type est une énumération java.

3.2. Tables d'associations (map)

Des relations 1-n (et n-n) peuvent soit être matérialisées comme des collections , soit être codées comme des tables d'associations ("map") .

Une table d'associations ("map") nécessite absolument une clef (à préciser) .

La propriété d'un objet de la "Map" qui sera associée à cette clef peut être renseignée via l'annotation **@MapKey**(name="...") .

Si l'indication "@MapKey" n'est pas présente , la clef primaire (id) est alors utilisée comme clef par défaut.

JPA 1 ne supportait que des "map" de sous objets "Entity" .

JPA 2 est maintenant capable de gérer en plus des "map" d'éléments simples ("basic").

Etant donné qu'un élément simple n'a pas obligatoirement d'identifiant (pk) , la clef peut être renseignée par l'annotation @MapKeyColumn .

Exemple (JPA 1 ou 2):

```
...
import java.util.Map;
import javax.persistence.*;

@Entity
@Table(name="Qcm")
public class Qcm {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long idQcm;

    private String titre;

    @OneToMany(mappedBy="qcm")
    @MapKey(name="numQuestion")
    private Map<Long,Question> questions;

    //+get/set
}
```


3.3. Collection d'objets "embedded" (sans id):

Depuis la version 2 de JPA, il est possible de définir (via `@ElementCollection` et `@CollectionTable`) des collections de sous objets imbriqués (sans id). Cependant ceci ne fonctionne qu'à un seul niveau (pas de sous-sous objets (sans id) dans des sous-objets (sans id)).

En UML, ceci se modélise généralement comme une composition (agrégation forte / losange noir) avec une multiplicité "*" ou "1..*".

La sémantique associée induit des cascades automatiques !!!

Exemple:

Qcm avec Entités "Question" ayant les caractéristiques suivantes:

- clef primaire composé de (idQcm, numQuestion)
- collection de sous-objets "Réponse" imbriqués (sans id)

Les Questions sont stockés dans une table Question.

Les sous objets réponses seront stockés dans une table "Reponse" ayant une clef primaire composée (idQcm, numQuestion, numReponse). Cette composition est basée sur

- une sous partie (idQcm, numQuestion) correspondante à la clef primaire de l'entité parente
Cette sous partie "fk" de liaison avec le parent sera précisée via l'attribut *"joinColumns"* de l'annotation `@CollectionTable`
- une sous partie complémentaire quelconque (numReponse ou numLigne ou index)

@Embeddable

```
public class QuestionPK implements Serializable{
```

```
    @Column(name="idOfQcm")
    private long idQcm;
```

```
    @Column(name="numQuestion")
    private int num;
```

```
    ...
```

```
}
```

@Embeddable

```
public class Reponse {
```

```
    // NB: dans cette classe, on ne fait pas apparaître les propriétés (idQcm,numQuestion) pourtant
    // présentes dans la table "Reponse", les mécanismes JPA s'en chargeront automatiquement !!!
```

```
    private char numReponse;
    private String texte;
    private boolean ok;
    // + get/set
```

```
}
```

```

@Entity
public class Question /* de qcm */ {
    @Id
    private QuestionPK pk;

    @Column(insertable=false,updatable=false) // pour éviter conflit avec partie "num" ok pk
    private int numQuestion;

    @ManyToOne
    @JoinColumn(name="idOfQcm",insertable=false,updatable=false)
    private Qcm qcm;

    private String texte;

    @ElementCollection()
    @CollectionTable(name="Reponse", joinColumns={
        @JoinColumn(name="idOfQcm"),
        @JoinColumn(name="numOfQuestion")})
    private List<Reponse> reponses;
    // + get/set
}

```

Structure de la base de données:

```

CREATE TABLE Qcm(idQcm integer auto_increment PRIMARY KEY, titre VARCHAR(128));
INSERT INTO Qcm Values (1,'qcm1'); INSERT INTO Qcm Values (2,'qcm2');

CREATE TABLE Question ( idOfQcm integer, numQuestion integer, texte VARCHAR(64),
                        primary key (idOfQcm,numQuestion));
ALTER TABLE Question ADD CONSTRAINT question_avec_qcm_valide
                        FOREIGN KEY (idOfQcm) REFERENCES Qcm(idQcm);
INSERT INTO Question Values (1,2,"question2-qcm1"); INSERT INTO Question Values (1,1,"question1-qcm1");
INSERT INTO Question Values (2,1,"question1-qcm2"); INSERT INTO Question Values (2,2,"question2-qcm2");

CREATE TABLE Reponse ( idOfQcm integer, numOfQuestion integer, numReponse CHAR,
                        texte VARCHAR(64), ok boolean, primary key (idOfQcm,numOfQuestion,numReponse));
ALTER TABLE Reponse ADD CONSTRAINT reponse_avec_question_valide
                        FOREIGN KEY (idOfQcm,numOfQuestion) REFERENCES Question(idOfQcm,numQuestion);

INSERT INTO Reponse Values (1,1,'A',"repA_question1-qcm1",false);
INSERT INTO Reponse Values (1,1,'B',"repB_question1-qcm1",true);
INSERT INTO Reponse Values (1,2,'A',"repA_question2-qcm1",true);
...

```

3.4. Quelques autres apports de JPA 2

JPA 1 était simplement capable de gérer une relation 1-n (vers des entités) en la définissant (via `mappedBy="..."`) comme la relation inverse d'une relation n-1.

Autrement dit, la spécification de la colonne clef étrangère devait absolument être effectuée du côté "n-1" (via `@JoinColumn` à côté de `@ManyToOne`) et le lien devait donc être :

- soit unidirectionnel (que du côté n-1 / `@ManyToOne`)
- soit bidirectionnel

Depuis JPA 2, il est maintenant possible d'établir un lien 1-n (unidirectionnel sans lien inverse n-1) en spécifiant la clef étrangère du côté 1-n (`@JoinColumn` proche de `@OneToMany`).

Bien que possible, cette nouvelle fonctionnalité ne devrait idéalement être utilisée que pour établir un lien unidirectionnel 1-n entre une entité d'un package et des entités d'un autre package métier/fonctionnel.

Entre deux types d'entités d'un même package les liaisons bidirectionnelles ne posent aucun problème.

4. Api "Criteria" de JPA 2

4.1. Apports et inconvénients de l'API "Criteria"

L'api "Criteria" (qui est apparue avec la version 2 de JPA) correspond à une syntaxe alternative à JPQL pour exprimer et déclencher des requêtes retournant des entités persistantes "JPA".

Contrairement à JPQL, l'api des criteria ne s'appuie plus sur un dérivé de SQL mais s'appuie sur **une série d'expression de critères de recherche qui sont entièrement construits via des méthodes "java"**.

L'api "Criteria" est plus "orientée objet" et plus "fortement typée" que JPQL.

Le **principal avantage** réside dans la **détection de certaines erreurs dès la phase de compilation**.

A l'inverse, la plupart des erreurs dans l'expression d'une requête JPQL ne sont en général détectées que lors de l'exécution des tests unitaires.

Du côté négatif, l'API "Criteria" de JPA n'est syntaxiquement pas très simple (voir franchement complexe) et il faut un certain temps pour s'y habituer (le code est moins "lisible" que JPQL au premier abord).

4.2. Exemple basique (pour la syntaxe et le déclenchement)

```
public List<Devise> getAllDevise() {
    /*return this.entityManager.createQuery(
        "select d from Devise d", Devise.class).getResultList();*/

    //construction:
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

    //préparation (expression des critères de recherche)
    Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);
    criteriaQuery.select(deviseRoot);

    //déclenchement (assez semblable) à celui d'une requête JPQL :
    return this.entityManager.createQuery(criteriaQuery).getResultList();
}
```

4.3. Exemple avec prédicat pour partie ".where"

```
public Devise getDeviseByName(final String deviseName) {
    /* return this.entityManager.createQuery(
        "select d from Devise d where d.monnaie = :deviseName",
        Devise.class)
        .setParameter("deviseName", deviseName)
        .getSingleResult(); */
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

    Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);

    Predicate pEqMonnaie = cb.equal(deviseRoot.get("monnaie") , deviseName);

    criteriaQuery.select(deviseRoot);
    criteriaQuery.where(pEqMonnaie);

    return this.entityManager.createQuery(criteriaQuery).getSingleResult();
}
```

4.4. Exemple relativement simple avec jointure:

```
public List<Compte> findComptesByNumCli(long numCli) {
    /*
    return this.getEntityManager().createQuery("SELECT cpt FROM
    Client cli JOIN cli.comptes cpt WHERE cli.numero= :numCli",Compte.class)
    .setParameter("numCli", numCli)
    .getResultList();
    */
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Compte> criteriaQuery = cb.createQuery(Compte.class);

    Root<Client> clientRoot = criteriaQuery.from(Client.class);

    Predicate pEqNumCli = cb.equal(clientRoot.get("numero") , numCli);
    //Predicate pEqNumCli = cb.equal(clientRoot.get(Client_.numero) , numCli);

    Join<Client, Compte> joinComptesOfClient = clientRoot.join("comptes");
    //Join<Client, Compte> joinComptesOfClient = clientRoot.join(Client_.comptes);

    criteriaQuery.select(joinComptesOfClient);
    criteriaQuery.where(pEqNumCli);

    return this.entityManager.createQuery(criteriaQuery).getResultList();
}
```

NB: L'exemple précédent n'est pas "fortement typé" et certaines erreurs ("types incompatibles , "propriété n'existant pas ou mal orthographiée") peuvent passer inaperçues lors de la phase de compilation.

4.5. Classes "modèles" pour des expressions rigoureuses des critères de recherche :

```
...
//Predicate pEqNumCli = cb.equal(clientRoot.get("numero") , numCli);
Predicate pEqNumCli = cb.equal(clientRoot.get(Client_.numero) , numCli);

//Join<Client, Compte> joinComptesOfClient = clientRoot.join("comptes");
Join<Client, Compte> joinComptesOfClient = clientRoot.join(Client_.comptes);
....
```

Au sein de cette écriture beaucoup plus rigoureuse ,

`Client_.numero` sera interprété comme une propriété `".numero"` qui existe bien sur une entité de type `"Compte"` et qui de plus est de type `long` .

Ces informations sont issue d'une classe java "modèle" dont le nom se termine (par convention) par un caractère "underscore" et dont le début coïncide avec la classe de l'entité persistante décrite.

(La classe modèle `"Client_"` décrit la structure de la classe `"Client"` préfixée par `@Entity`).

Une classe modèle doit absolument être placée dans le même package java que la classe d'entité persistante à décrire. On pourra cependant paramétrer maven et/ou l'IDE eclipse de façon à placer les classes "modèles" dans un autre répertoire car celles-ci sont générées automatiquement.

Exemple de classe "modèle"

```
package tp.myapp.minibank.core.entity;

import javax.annotation.Generated;
import javax.persistence.metamodel.ListAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Client.class)
public abstract class Client_ {

    public static volatile SingularAttribute<Client, String> password;
    public static volatile SingularAttribute<Client, Long> numero;
    public static volatile ListAttribute<Client, Compte> comptes;
    public static volatile SingularAttribute<Client, Adresse> adresse;
    public static volatile SingularAttribute<Client, String> telephone;
    public static volatile SingularAttribute<Client, String> nom;
    public static volatile SingularAttribute<Client, String> prenom;
    public static volatile SingularAttribute<Client, String> email;
}
```

NB : La classe modèle `_Client` a été générée via un plugin maven `"hibernate-jpamodelgen"` (ou autre) qui peut se déclencher au moment de la compilation des classes ordinaires `"Client"` , ...

.../...

```

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-jpamodelgen</artifactId>
  <version>4.3.1.Final</version>
  <scope>provided</scope>
</dependency>

...

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.3</version>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
    <compilerArguments>
      <processor>
        org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor
      </processor>
    </compilerArguments>
  </configuration>
</plugin>

```

NB : il existe une autre façon de déclencher `org.hibernate.jpamodelgen` (plus paramétrable) .

Les classes modèles "XXX_" et les packages java sont par défaut générés dans le répertoire "**target/generated-sources/annotations**"

Il faut ensuite paramétrer le projet eclipse pour qu'il voit les classes générées :

- 1) activer le menu contextuel "**properties / compiler / annotation-processing**" depuis le projet eclipse "...-ejb"
- 2) **enable specific** / "**target/generated-sources/annotations**"
- 3) ajouter une utilisation de XXX_.yyy dans le code du DAO

IX - Outils pour JPA , ...

1. Exemple de dépendances maven pour JPA

```
<properties>
    <org.hibernate.version>4.3.6.Final</org.hibernate.version>
</properties>
...
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <!-- with indirect/transitive <artifactId>hibernate-core</artifactId> -->
    <version>${org.hibernate.version}</version>
</dependency>

<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.1.2.Final</version>
    <scope>runtime</scope>
</dependency>
```

à placer/ajuster dans **pom.xml**

2. Outils et plugins pour JPA

Parmi les différents outils et plugins facilitant l'usage d'hibernate/JPA, on peut citer:

- Plugin **maven** "hibernate3-maven-plugin" ou "**hibernate4-maven-plugin**"
- Assistants "**JPA**" d'eclipse
- Plugin "**Jboss Tools**" pour eclipse

3. Plugin maven "hibernate4-maven-plugin"

Ce plugin permet de générer (sur demande) un fichier schema.sql comportant (entre autres) des instructions "CREATE TABLE" en fonction de la structure des objets java (@Entity).

C'est une des manières de mettre en œuvre l'approche "Top-down" .

```
<!-- a lancer via mvn hibernate4:export -->
<plugin>
  <groupId>de.juplo</groupId>
  <artifactId>hibernate4-maven-plugin</artifactId>
  <version>1.1.0</version>
  <executions>
    <execution>
      <goals>
        <goal>export</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <target>none</target> <!-- genere schema.sql dans target -->
    <!-- <target>script</target> --> <!-- genere schema.sql dans target
        et affiche le contenu de schema.sql dans la console -->
  </configuration>
</plugin>
```

à placer/ajuster dans la partie build/plugins de pom.xml

4. Projet "JPA" d'eclipse

Mode opératoire à un peu adapter selon la version exacte d'eclipse (3.4 , 3.5 ou 3.6)

- 1) Paramétrer une connexion vers une base de données existante au niveau d'eclipse
(menu "Windows / **Show View** / Other ... / connectivity / **DataSourceExplorer** / **New**).
Paramétrer à ce niveau l'accès à la base de données (chemin du ".jar" du driver JDBC ,
url , ...).
Tester la connexion SQL via l'explorateur de base de données (menu "Windows / Show
View // **DataSourceExplorer** / / **connect**).
- 2) Créer un nouveau *projet JPA* (ou bien ajouter la *facette "JPA"* à un projet java existant)
- 3) depuis le projet JPA déclencher le menu contextuel "**JPA Tools ...**" / "**Generate Entities**"
choisir la connexion SQL , le schéma de la base de données , les tables à prendre en comptes
et quelques options (package java , ...)
==> **résultats = classes d'entités persistantes avec des annotations JPA (@Entity ,
@Table , @Id , @OneToMany ,).**

Remarque:

Ceci permet de générer une version "brute" des classes d'entités persistantes.

Il est souvent conseillé de recopier ces classes (par exemple dans un autre projet de type "maven") puis les adapter pour les améliorer un peu.

ANNEXES

X - Configurations d'Hibernate (hbm,@...)

1. Configuration Hibernate en xml (sans JPA)

1.1. Fichiers de configuration

```
Configuration cfg = new Configuration();
```

```
    cfg.configure(); // prend en compte hibernate.cfg.xml
/* ou bien */ cfg.configure("hibernate.cfg2.xml");
```

```
SessionFactory sf = cfg.buildSessionFactory();
```

Cet objet de configuration *cfg* est automatiquement initialisé avec un jeu de propriétés (*hibernate.properties*) ou bien via un fichier de configuration xml (*hibernate.cfg.xml*) à placer dans le CLASSPATH .

Exemple de fichier *hibernate.cfg.xml* (avec *datasource*) :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.datasource">java:comp/env/jdbc/ex_hib</property>
    <property name="show_sql">false</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!-- Mapping files -->
    <mapping resource="Client.hbm.xml"/> <!-- mapping classe java - table -->
    <mapping resource="Commande.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

NB:

La propriété *connection.datasource* doit avoir une valeur du type "*java:comp/env/jdbc/xxxDS*" avec *Tomcat* et une valeur du type "*java:xxxDs*" avec *JBoss* .

Dans le cas où l'on préfère laisser à hibernate le soin de gérer le pool de connexions JDBC lui même , les propriétés à renseigner (à la place de *hibernate.connection.datasource*) sont les suivantes:

```
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
```

```
<property name="connection.url">jdbc:mysql://localhost/mydatabase</property>
<property name="connection.username">mydbuser</property>
<property name="connection.password">mypwd</property>
... ( dans hibernate.cfg.xml sans datasource)
```

Les **principaux dialectes SQL** sont *DB2Dialect*, *PostgreSQLDialect*, *MySQLDialect*, *OracleDialect*, *SybaseDialect*, *SQLServerDialect*, *InformixDialect*, *HSQLDialect*, ... à préfixer par *org.hibernate.dialect* (V3) ou *net.sf.hibernate.dialect* (V2)

A chaque **classe d'objet persistant** doit correspondre un **fichier xml de mapping**.

Un fichier de mapping hibernate a classiquement l'extension **".hbm.xml"** ; il doit être placé dans le CLASSPATH (comme un fichier de ressource) et doit être référencé depuis le fichier de configuration principal **"hibernate.cfg.xml"** (via une balise <mapping>).

Remarque importante:

La technologie **Hibernate** est assez souvent utilisée conjointement avec **Spring**.

Lorsqu' hibernate est utilisé avec Spring, le fichier de configuration **"hibernate.cfg.xml"** n'est plus nécessaire. il est remplacé par une **configuration "Spring" équivalente**:

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost/minibankdb" />
  <property name="username" value="mydbuser" />
  <property name="password" value="mypwd" />
</bean>
...
<bean id="myHibernateSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="myDataSource" />
  <property name="mappingResources">
    <list>
      <value>Compte.hbm.xml</value>
      <value>Client.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props><prop key="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect</prop>
    </props>
  </property>
</bean>
...
<bean id="xxxxDAO" class="dao.HbXxxDAO">
  <property name="sessionFactory" ref="myHibernateSessionFactory" />
</bean>
...
```

Exemple de fichier de mapping (xxx.hbm.xml):

```
<?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>
<class name="tp.Chat" table="CAT">
  <id name="num" unsaved-value="0" > <!-- ou unsaved-value="null" si pk type String -->
    <column name="CAT_ID" not-null="true"/>
    <generator class="native"/> <!-- native si auto_increment mysql ou equivalent -->
  </id>
  <property name="name">
    <column name="NAME" length="16" not-null="true"/>
  </property>
  <property name="sex"/>
  <property name="weight"/>
</class>
</hibernate-mapping>

```

NB:

- Les noms des propriétés sont celles de la classe java ["num" ==> `getNum()` / `setNum(...)`].
- La balise `id` permet d'indiquer la clef primaire , les autres propriétés sont renseignées via `<property>` .
- Lorsque le type d'une propriété n'est pas précisé , hibernate s'appuie sur le résultat d'une introspection java (*java.lang.reflect*).
- Lorsque le nom d'une colonne n'est pas renseigné , le même nom que celui de la propriété est utilisé (idem pour le type avec adaptation éventuelle liée au dialecte SQL).
- La classe "tp.Chat" est un simple JavaBean avec un constructeur par défaut (sans argument).

1.2. Exemple de code java utilisant Hibernate seul (sans spring):

```

public void creerNouveauChat(.....){
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
//SessionFactory sessionFactory = MyHibernateSessionFactorySingleton.getInstance();
Session session = sessionFactory.openSession();
Transaction tx = null;
tx = session.beginTransaction();
    Cat nouveauChat = new Cat();      nouveauChat.setName(chNomChat); ....
    session.save(nouveauChat);
tx.commit();
session.close(); ....
}

```

NB: `HibernateException` hérite de *RuntimeException* (c'est une *unchecked* exception) ==> les *try/catch* sont donc facultatifs .

XI - Session Hibernate

1. Principales méthodes de l'objet Session:

méthodes	traitements
.get(xxx.class,pk)	Recherche en base et retourne un objet ayant les classe java et clef primaire indiquées
.delete(obj)	Supprime l'objet (delete ... from where dans la base de données)
.update(obj)	Met à jour d'un objet dans la base de données (update sql)
.save(obj) ou .persist(obj)	Sauvegarde les valeurs d'un nouvel objet dans la base de données (insert into ...) . La clef primaire est quelquefois automatiquement calculée lors de cette opération
.save_or_update(obj)	Sauvegarde ou bien met à jour les valeurs d'un objet dans la base de données.
.createQuery(req_hbql)	Créer une requête HQL (de type Query) et retourne cet objet
.refresh()	pour que les versions en mémoire tiennent compte des liens bi-directionnels ou pour rafraîchir explicitement les valeurs .

2. Principales méthodes de l'objet Query:

méthodes	traitements
.setXxx(paramName,paramValue) ou Xxx = String , Int , ...	met à jour un paramètre de la requête hbql
.iterate()	retourne un itérateur (java.util.Iterator) pour parcourir l'ensemble d'une collection (résultat d'un select)
.list()	retourne sous forme d'objet " java.util.List " la liste des objets retournés en résultat de la requête HQL.

XII - Intégration Spring , Hibernate Template

1. Grandes lignes sur l'intégration de Hibernate au sein de Spring

Outre la configuration *sans hibernate.cfg.xml* mentionnée au sein du chapitre "configuration", l'intégration d'hibernate au sein de Spring repose essentiellement sur l'objet technique "*hibernateTemplate*" fourni par Spring:

```
import org.springframework.orm.hibernate3.HibernateTemplate;
```

```
public class CompteDaoHbTp implements CompteDAO {

    private HibernateTemplate hibernateTemplate=null;

    public void setSessionFactory(SessionFactory sf)
    {
        hibernateTemplate = new HibernateTemplate(sf);
    }

    public Compte getCompteByNum(long num_cpt) throws DataAccessException {
        Compte cpt = null ;
        cpt=(Compte)hibernateTemplate.get(Compte.class,
                                         Long.valueOf(num_cpt));
        return cpt;
    }

    public Collection getCompteOfClient(int num_client)
        throws DataAccessException {
        return hibernateTemplate.find("select c from Compte as c
                                     where c.num_cli = " +num_client);
    }

    public void updateCompte(Compte cpt) throws DataAccessException {
        hibernateTemplate.update(cpt);
        System.out.println("(via HibernateTemplate)");
    }
}
```

L'objet **hibernateTemplate** de Spring reprend presque toutes les fonctionnalités de l'objet "session" d'hibernate (méthodes très ressemblantes). Sa principale valeur ajoutée réside dans le fait qu'il gère automatiquement :

- les ouvertures et les fermetures de session hibernate.
- les `session.beginTransaction()` , `tx.commit()` ,...

en fonction de la configuration Spring et des méthodes appelantes au niveau des services métiers.

==> approfondir si nécessaire ce point via la documentation Spring .

XIII - Criteria Hibernate

1. Criteria hibernate

- Requêtes formulées à base de critères (syntaxe java). Mixage possible "HQL + criteria" .
- Une expression des critères de recherche en java peut constituer une alternative à une requête HQL.

La méthode `createCriteria()` de l'objet `session` permet de créer une instance d'une classe implémentant l'interface `org.hibernate.Criteria`.

Cette instance (Criteria) correspond à une composition de critères (`org.hibernate.criterion`) qui permettent d'exprimer des **restrictions** sur le résultat à obtenir (*un peu comme une clause Where*).

1.1. Exemples de Criteria:

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

```
List cats = session.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Iz%") )
    .add( Restrictions.gt( "weight", new Float(minWeight) ) )
    .addOrder( Order.asc("age") )
    .list();
```

Quelques exemples de `org.hibernate.criterion.Restrictions` (à imbriquer dans `.add(..)`):

```
Restrictions.or(
    Restrictions.eq( "age", new Integer(0) ),
    Restrictions.isNull("age"))
```

```
Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } )
```

```
Restrictions.between("weight", minWeight, maxWeight)
```

Trier un résultat:

```
.addOrder( Order.asc("name") )
.addOrder( Order.desc("age") )
```

1.2. Pour un déclenchement des "Criteria Hibernate" depuis JPA:

Lorsqu'on est certain que Hibernate est utilisé en tant qu'implémentation interne de JPA , on peut établir un pont entre le "entityManager" de JPA et la "session" d'Hibernate via la méthode `".getDelegate()"` .

```
((Session) entityManager.getDelegate() )....
```


XIV - Généralités ORM Hibernate

1. Vue d'ensemble sur les entités, valeurs et relations

1.1. Problématique

Une technologie de **mapping objet – relationnel** telle qu'**hibernate** ou **JPA** doit pouvoir s'appuyer sur un **modèle objet** qui soit :

- **naturel d'un point de vue "langage orienté objet"** tel que **java** (==> utilisation de **JavaBean** et de collections classiques)
- **compatible avec la structure générale d'une base de données relationnelle** (jointures entre *entités* : clefs étrangères référençant des *clefs primaires*).

1.2. Entités

Entités

Client
num_cli (pk)
 nom , adresse, ...

Une entité (avec id/pk) existe même si elle n'est pas référencée .

Commande
num_cmde (pk)
 date , ...

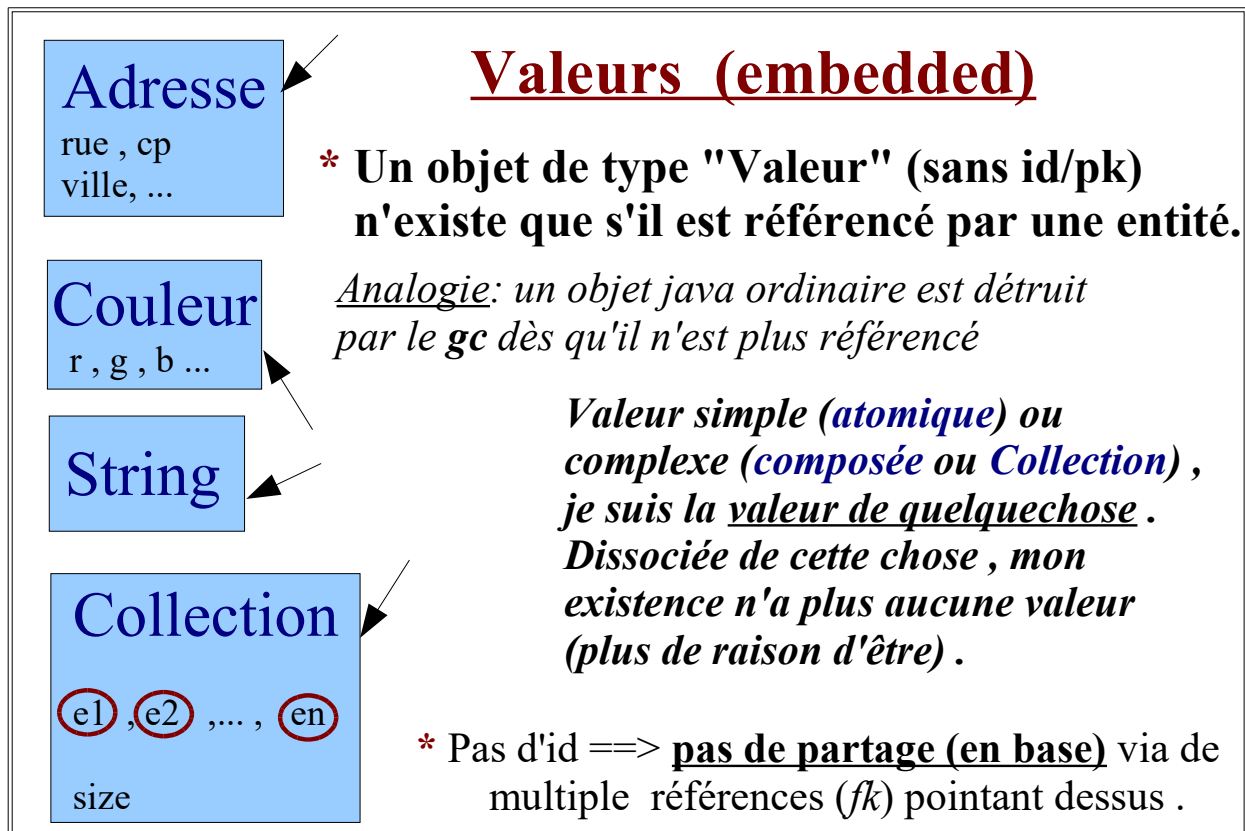
***Je ne pense pas
 mais j'ai une identité
 (clef primaire)
 donc je suis .***

To be or not to be : that is the question .

Pour un objet informatique, l'essence précède l'existence .

Sous hibernate et JPA, un objet java de type "entité" sera un simple **JavaBean** comportant obligatoirement une **propriété "id"** (ou "numéro" ou "...") faisant office d'identifiant et correspondra généralement en base à un enregistrement identifié par une clef primaire .

1.3. Objet "Valeurs" ou bien "imbriqué/incorporé (embedded)"



Concrètement, un objet hibernate/JPA de type "Valeur"/"Embedded" peut correspondre en base :

- à un ensemble de champs (colonnes) d'une certaine table [Valeur composée/component]
- pour Hibernate (mais pas pour JPA 1.0) : à des enregistrements secondaires (non partagés et référencés par une seule clef étrangère) [Collection de "component"] . Bien que pouvant être techniquement associés à des clefs primaires cachées, ces enregistrements secondaires n'ont pas de réelles identités propres : ce ne sont que des détails d'une et une seule entité principale [cascade-delete] .

Au niveau des mécanismes internes et automatiques de hibernate/JPA, on peut aussi considérer comme étant implicitement un sous objet de type "Valeur/Embedded" un objet "Collection de références" "purement Java" qui n'est pas directement associé à un enregistrements en base mais qui est associé à un ensemble de clefs étrangères qui référencent d'autres entités (relation 1-n ou un des sens d'une relation n-n).

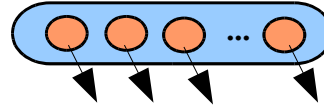
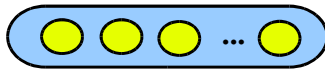
1.4. Collections et constituants

Collection et constituants

* Une collection est **globalement** un élément de type **valeur** (n'existant que si référencé) → 

* Les éléments d'une collection sont :

- soit des **valeurs** (**element** , **composite-element**)
- soit des **références vers d'autres entités**
(**one-to-many** , **many-to-many**, **many-to-any**).



* Une **clef de collection** (liée à une **clef étrangère** en base) permet de déterminer l'appartenance des éléments à une collection 

Types de collections "hibernate" --> **map**, **set** , **list** , **bag** ,**array**

Les **list** , **array** et **map** ont des **index**

Les **set** et **bag** n'ont pas d'**index**

NB: Une collection de type **java.util.Set** pose quelquefois de gros problèmes d'affichage (ex: impossible de générer un tableau (h:dataTable) dans JSF) . La notion de "**Bag**" correspond en fait à une **simple Collection d'éléments** (implémentée en interne à partir de **java.util.List**) mais n'ayant pas nécessairement toute la sémantique des listes (pas d'ordre/pas d'index) .

Dans le cadre d'une **association 1-n ordinaire** Hibernate et JPA choisiront généralement (par défaut, sans indication particulière) des "**Bag**" (compatibles avec **java.util.List**) comme implémentations de **java.util.Collection** .

1.5. Lazy (Hibernate)

L'attribut optionnel **lazy="true"** d'hibernate (ou bien **fetchType=Lazy** de JPA) permet de demander une *initialisation tardive* des membres d'une collection (les valeurs des éléments de la collection ne seront recherchées en base que lorsqu'elles seront consultées via par exemple un itérateur java). Ceci permet d'obtenir dans certains cas de meilleures performances . Il faut cependant veiller à récupérer les valeurs de la collection avant le commit et la fermeture de la session (sinon ==> c'est trop tard : *LazyInitialisationException*).

Il est fortement conseillé d'affecter explicitement la valeur "false" ou "true" au paramètre lazy (très important sur le plan des performances).

NB: Avec Hibernate (et sa configuration hbm.xml) , le paramètre lazy est non seulement présent au niveau d'une collection (ex <set ... lazy="...">) mais il est également possible de lui affecter une valeur au niveau d'une classe ordinaire (<class ... lazy="...">) ce qui aura pour effet de préciser si les valeurs d'une sous objet de la classe en question doivent être par défaut remontées dès de chargement d'un objet principal (ex: 1-1 ,) .

Exemple hibernate (.hbm.xml):

---> lazy="true" or lazy="false" .

```
<bag name="dernieres_operations" cascade="all" inverse="true" lazy="true">
    <key column="num_compte"/>
    <one-to-many class="domain.entity.Operation" />
</bag>
```

1.6. Operations en cascade (syntaxe hbm)

Syntaxe hibernate (dans les fichiers ".hbm.xml"):

<... cascade="all" ou cascade="none" ou cascade="delete" ou cascade="save-update" />

NB: cascade="all,delete-orphan" permet en plus de supprimer automatiquement en base des entités "enfants" qui seraient détachées de leurs parents (suite à une suppression de référence(s) dans une collection par exemple) .

2. Identité d'une entité (clef primaire et générateurs)

Une **clef primaire** (ex: numéro de facture) est assez souvent **générée automatiquement** via un **algorithme** basé sur la notion de **compteur**.

Un fichier de mapping hibernate (avec pour extension .hbm.xml) peut comporter l'indication d'un algorithme à utiliser pour générer une nouvelle clef primaire . Ceci s'effectue en imbriquant une sous balise <generator> ou sein de la balise <id> .

Exemples (hibernate):

```
<id name="num" unsaved-value="0">
    <column name="numero" not-null="true" />
    <generator class="native" />
</id>
```

```
</id>
```

```
<id name="id" type="long" column="cat_id">
  <generator class="hilo">
    <param name="table">uid_table</param>
    <param name="column">next_hi_value_column</param>
  </generator>
</id>
```

Principaux algorithmes/générateurs:

<i>Hibernate</i>	<i>JPA</i>	<i>Algorithme</i>
increment		Simple compteur (long, int , short) qui s'incrémente (nouvelle pk = max(pk)+1) ==> à ne pas utiliser si d'autres programmes (ou d'autres threads) sont susceptibles d'effectuer des insertions au même moments (ex: autre membre d'un cluster).
hilo		Même principe que "increment" cependant la dernière valeur du compteur est stockée dans une colonne d'une table spéciale (ce qui permet de bien gérer les verrous en cas d'insertions simultanées).
uuid.hex		universal unique id (128 bits) au format hexadécimal (style base des registres de Win32) ==> identifiant dont le type est compatible avec le format String de longueur >=32 caractères
identity ou sequence		identifiant géré par des mécanismes spécifiques à une certaine base de données (ex: sequence Oracle , auto_increment MySql ,)
native	AUTO	choisit identity , sequence ou hilo selon le type de base de données
....		Voir documentation de référence

3. Clef primaire composée

Avec Hibernate:

```
<class name="..." table="..." >
  <composite-id name="propertyName" class="pk_class">
    <key-property name="propertyName" type="typename" column="column_name"/>
    <key-property name="propertyName" type="typename" column="column_name"/>
    ou <key-many-to-one name="propertyName" class="ClassName" column="column_name"/>
    ....
  </composite-id>
</class>
}
```

4. Structure générale d'une classe d'entité persistante

Le **code java** associé à une **classe d'objet persistant hibernate** correspond à un simple **JavaBean** (classe Java respectant les conventions de nom **getXxx()** et **setXxx(...)** au niveau des propriétés).

On parle quelquefois en terme de **POJO** (**Plain Old Java Object** / **Plain Ordinary Java Object**) pour désigner le fait qu'une API n'impose quasiment rien sur la structure des objets java .

Un **constructeur sans argument** est **indispensable**.

NB: Sous *eclipse*, les méthodes publiques "*getXxx()*" et "*setXxx(...)*" peuvent être générées automatiquement à partir des attributs privés et le menu contextuel "*source/generate getter/setter*"

De façon à ce qu'un objet persistant hibernate puisse être comparé à un autre (de façon efficace), il est quelquefois conseillé de reprogrammer convenablement les méthodes **equals** et **hashCode** héritées de la classe **Object**.

Cependant, cette **optimisation** (*facultative*) est **délicate** dans le sens où le code de **equals** et **hashCode** doit prendre en charge uniquement les valeurs élémentaires de l'objet lui même (entiers, réels, chaînes) mais **ne doit surtout pas tenir compte des références (éventuellement nulles) sur les objets reliés par des associations** et d'autre part ces méthodes peuvent éventuellement ne pas tenir compte de la clef primaire car elle n'a pas toujours de valeur bien affectée (ex: objet nouvellement créé) [--> en pratique : bien décocher les propriétés à ne pas prendre en compte lors de la génération de code via l'assistant eclipse "generate equals & hashCode"].

Exemple:

```
package entites;
public class Operation {
    private String id; // PK
    private String label;    private double mouvement;
    private Compte compte; // référence vers objet associé (extrémité inverse "n-1" d'une relation "1-n"

    public String getId() { return id;}          public void setId(String id) {    this.id = id;}
    public String getLabel() {return label;}
    public void setLabel(String label) {this.label = label;}
    public double getMouvement() {return mouvement;}
    public void setMouvement(double mouvement) {this.mouvement = mouvement; }
    public Compte getCompte() {return compte;}
    public void setCompte(Compte compte) {this.compte = compte; }

    public boolean equals(Object other) {
        if (this == other) return true;    if ( !(other instanceof Operation) ) return false;
        final Operation op = (Operation) other;
        if ( !op.getLabel().equals( getLabel() ) ) return false;
        if ( op.getMouvement() != getMouvement() ) return false;
        return true;
    }
    public int hashCode() {
        int res= getLabel().hashCode();
        res=29*res + ((int) getMouvement());
        return res;
    }
}
```

+ habituelles annotations si JPA .

XV - ORM Hibernate (HBM: 1-1 , 1-n , n-n , ...)

1. Mapping (.hbm) des principales associations

1.1. Relations 1-n (et n-1)

Configuration Hibernate (.hbm.xml):

```
.... au sein de Client.hbm.xml ....
<bag name="commandes" inverse="true" lazy="true">
  <key column="id_client"/> <!-- fk -->
  <one-to-many class="entity.Commande"/>
</bag>
```

<bag> est souvent mieux que <set> ==> pas de problème affichage JSF , ...

et si bi-directionnel:

```
.... au sein de Commande.hbm.xml ....
<many-to-one name="client" column="id_client"/> <!-- fk -->
```

(NB: inverse="true" du côté secondaire d'une relation bi-directionnelle)

Dans l'exemple ci-dessus, inverse="true" a été positionné dans Client.hbm.xml pointant vers une collection de commandes [Client-->Commande(s) est donc ici le sens secondaire]

NB: Les changements faits uniquement sur l'extrémité inverse de l'association ne sont pas persistés !!!!

Exemple:

```
client.getCommandes().add(commande); // Le client est maintenant "au courant" de la relation
commande.setClient(client); // La commande est maintenant "au courant" de la relation
```

```
session.persist(client); // La relation ne sera pas sauvegardée (via persist ou update)!
                        // si pas ligne suivante
```

```
session.persist(commande); // La relation sera sauvegardée (même si pas ligne précédente)
```

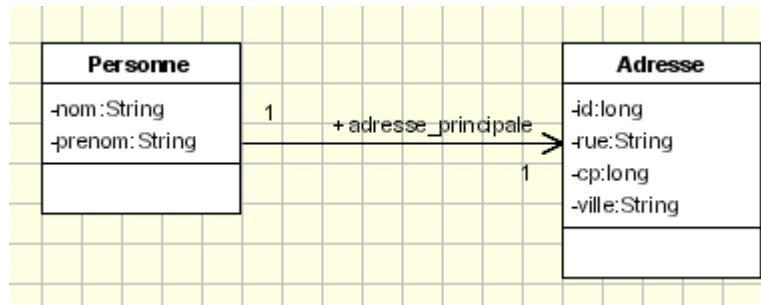
Sur une relation n-n (many-to-many) inverse=true est d'un côté quelconque

Sur une relation 1-n , inverse="true" est généralement du côté 1 pointant vers une collection de n .

NB: il est possible d'établir une **relation bidirectionnelle 1-n avec une table de jointure** (<one-to-many .../> est alors remplacé par <many-to-many unique='true' ... /> .

1.2. Relation 1-1 (entité - entité)

Mapping 1-1 hibernate (.hbm.xml):



Si relation classique (à base de clef étrangère) alors [*many-to-one unique="true"*]

T_Personne

id (pk) , nom , prenom , , id_adr (fk)

many-to-one unique="true"

T_Adresse

id (pk) , rue , cp , ville

.... dans Personne.hbm.xml ...

```

<many-to-one name="adressePrincipale"
  column="ID_ADR"
  unique="true"
  not-null="true"/>
    
```

Si pas de clef étrangère mais correspondance direct des clefs primaires:

alors one-to-one (dans .hbm.xml)

1.3. Incorporation d'objet valeur (1-1 par composition)

```

<component name="variable_instance_référençant_composant" class="classe_composant">
  <property name="propriété1_du_composant" />
  <property name="propriété1_du_composant" />
</component>
    
```

exemple:

```

<class name="entity.Article">
....
  <component name="prix" class="xxx.MontantMonetaire">
    <property name="montant"/>
    <property name="monnaie"/>
  </component>
....
</class>
    
```

1.4. Relations n-n

Mapping Hibernate n-n (dans .hbm.xml) :

Dans Personne/Auteur (.hbm.xml)

...


```
<bag name="works" table="author_work" inverse="true" lazy="true">
  <key column="author_id"/>
  <many-to-many class="xxx.Work" column="work_id"/>
</bag>
....
```

et

Dans Oeuvre/Travaux (.hbm.xml)

```
...
<bag name="authors" table="author_work" lazy="true">
  <key>
    <column name="work_id" not-null="true"/>
  </key>
  <many-to-many class="xxx.Author">
    <column name="author_id" not-null="true"/>
  </many-to-many>
</bag>
....
```

2. Mapping Hibernate (.hbm.xml) pour l'héritage

2.1. Stratégie "Une table par hiérarchie" / colonne discriminante

Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.

Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

```
<class name="xxx.compte" table="T_Compte">
  <id name="id" type="long" column="id_compte"> <generator class="native"/> </id>
  <discriminator column="typeCompte" type="string"/>
  <property name="label" />
  <property name="solde" />
  ...
  <subclass name="xxx.ComptePEL" discriminator-value="PEL">
    <property name="tauxInteret" />
    <property name="apportMensuel" />
  </subclass>
  <subclass name="xxx.CompteCodevi" discriminator-value="Codevi">
    ...
  </subclass>
</class>
```

NB: Il faut absolument respecter l'ordre "id , discriminator , property ," .

2.2. Stratégie "Une table par classe fille" – joined-subclass

1 table avec points communs (avec pk) + 1 table pour chaque classe fille avec suppléments et fk .

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
  <generator class="native"/>
```

```

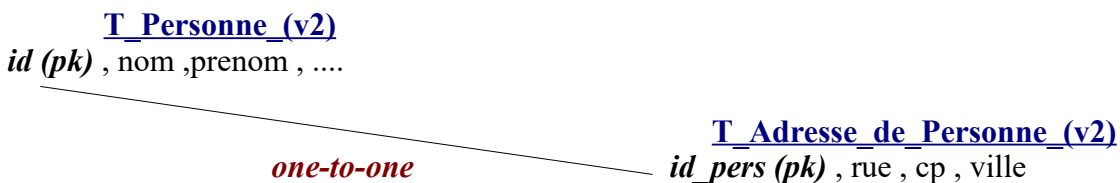
</id>
<property name="amount" column="AMOUNT"/>
...
<joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
  <key column="PAYMENT_ID"/>
  <property name="creditCardType" column="CCTYPE"/>
  ...
</joined-subclass>
<joined-subclass name="CashPayment" table="CASH_PAYMENT">
  <key column="PAYMENT_ID"/>
  ...
</joined-subclass> ...
</class>

```

3. 1-1 avec correspondance directe des "pk"

OK avec Hibernate (bien qu'assez rare).

Si correspondance direct des clefs primaires: alors one-to-one (dans .hbm.xml)



```

.... dans Personne.hbm.xml ...
<one-to-one name='adressePrincipale' />

```

NB: en base la correspondance attendue est :

"valeur clef primaire de Adresse (_ de _ Personne) = valeur de clef primaire de Personne"

4. Imbrications d'une collection de valeurs composites

OK avec Hibernate (pas d'équivalent avec JPA 1.0 , mais équivalent en JPA 2.0)

Exemple:

```

<list name="lignesCommande" table="T_LigneCommande" lazy="true">
  <key column="commande_id"/>
  <index column="num_ligne"/>
  <composite-element class="LigneCommande">

```

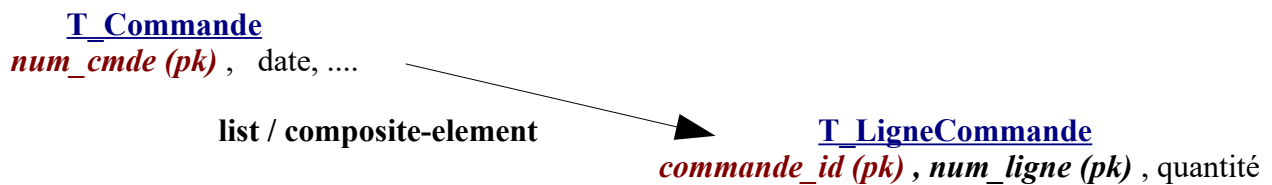
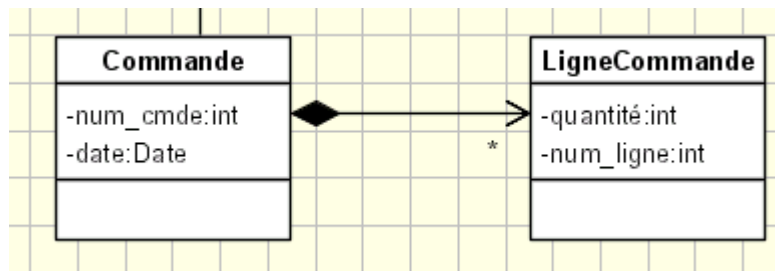
```

    <property name="quantité"/>
    <many-to-one name="produit" column="id_produit"/>
  </composite-element>
</list>

```

Attention , les index (numéros de lignes) sont prévus commencer à 0.

Si au sein de la base de données , le premier index trouvé est 1 , Alors dans la collection de lignes , une référence "null" sera associé à l'élément manquant en index=0 .



XVI - Aspects divers et avancés (batch , xml, ...)

1. Requêtes en grandes séries (par paquets / batch)

Pour obtenir de bonnes performances sur des gros volumes de données (nombreux objets/enregistrements) , on peut :

- activer l'utilisation des paquets au sein de JDBC:
`hibernate.jdbc.batch_size 20` (taille raisonnable : 10-50)
- éventuellement désactiver le cache de second niveau pour éviter de saturer la mémoire.
`hibernate.cache.use_second_level_cache false`
- vider régulièrement les buffers et le cache de premier niveau (avec des boucles qui coïncident avec la taille des paquets JDBC):

Exemple1 (insertion par paquets):

```
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, même taille que la taille du paquet JDBC
        //flush un paquet d'insertions et libère la mémoire :
        session.flush();
        session.clear();
    }
}
tx.commit();
session.
```

Exemple2 (mise à jour par paquets):

```
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 un paquet de mises à jour et libère la mémoire :
        session.flush();
        session.clear();
    }
}
tx.commit();
session.close();
```

NB: bien qu'un peu optimisé ce code est encore beaucoup moins performant qu'un UPDATE SQL

(ou HQL) direct sur la base de données.

Traitements par lots idéalement exprimés par des requêtes "HQL/DML" :

à encadrer par `tx.beginTransaction()` , `tx.commit()`; et `openSession()`; `session.close()`:

en mise à jour (update):

```
String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
    .setString( "newName", newName )
    .setString( "oldName", oldName )
    .executeUpdate();
```

en suppression (delete):

```
String hqlDelete = "delete Customer c where c.name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
    .setString( "oldName", oldName )
    .executeUpdate();
```

en récupération/insertion (insert into ... select ...):

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
    .executeUpdate();
```

2. Autres aspects divers de Hibernate

2.1. Mode de verrouillage (optimiste par défaut):

Etant donné qu'Hibernate est très souvent utilisé en mode 2-tiers ou 3-tiers , les sessions et transactions hibernates sont généralement assez courtes et le mode de verrouillage "optimiste" (utilisé par défaut par Hibernate) convient très bien dans la plupart des cas.

Bien que possible au niveau d'Hibernate, le mode de verrouillage pessimiste est rarement utile et d'autre part il dépend des fonctionnalités de la base de données.

2.2. Eventuelles (et rares) sessions longues :

```
// foo est une instance chargée antérieurement par une autre session
session.reconnect();// Obtention d'une nouvelle connexion JDBC
Transaction t = session.beginTransaction();
foo.setProperty("bar");
t.commit();//Terminer la transaction, propager les changements et vérifier les versions.
session.disconnect();// Retourner la connexion JDBC
```

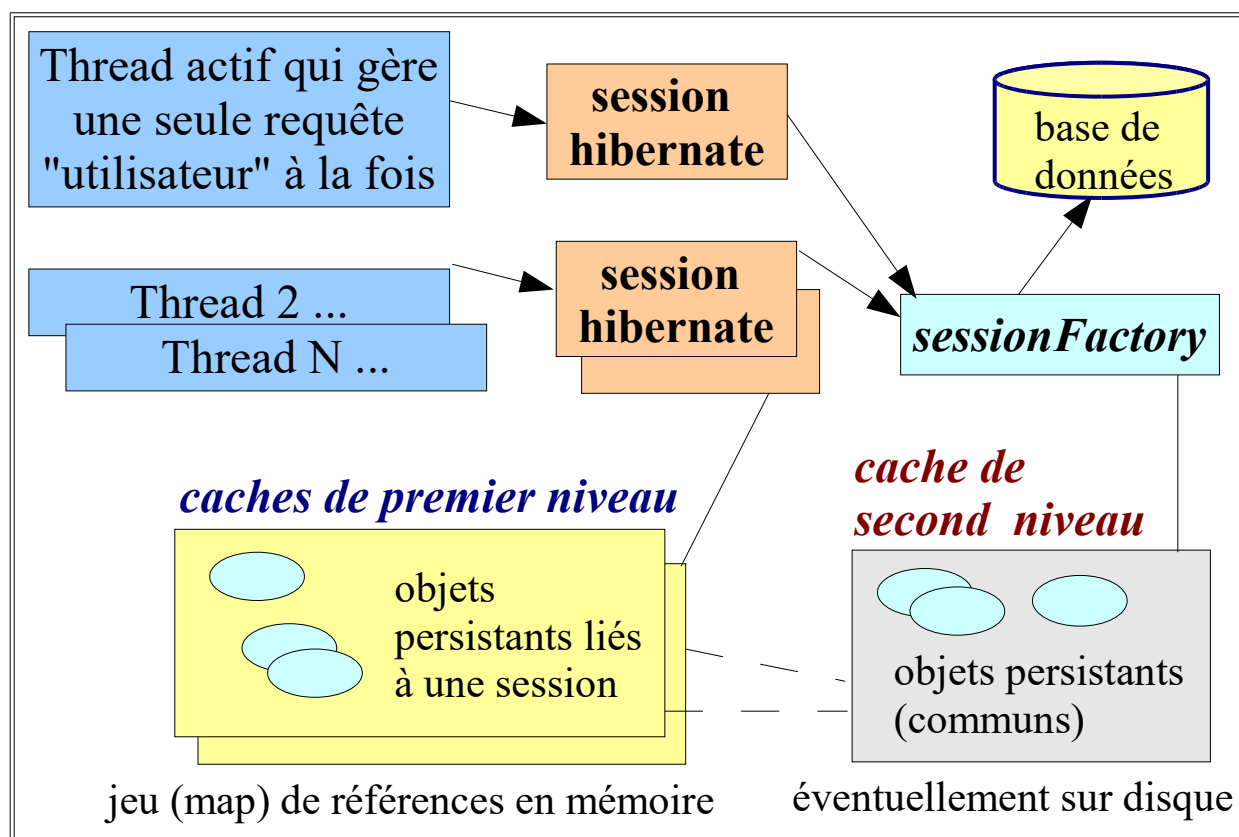
==> gestion automatique des versions : si pb de version ==> `HibernateException` .

2.3. Stratégies de chargement

Trop ou pas assez de chargement tardifs ("lazy") peuvent poser des problèmes au niveau des performances ou bien sur le plan des exceptions.

La sous section "Stratégies de chargement" du chapitre "Améliorer les performances" de la documentation de référence d'Hibernate explique comment paramétrer finement ce point.

2.4. Cache de second niveau



Le cache de premier niveau est associé à une session hibernate . Souvent associé à des transactions courtes , il est géré automatiquement sans problème notable.

Le **cache de second niveau** est associé à toute une JVM (voir un cluster complet). il est exactement lié à un objet "SessionFactory" . Le chapitre "*Améliorer les performances*" de la *documentation de référence* d'Hibernate explique comment paramétrer ce cache de second niveau de façon à optimiser les performances.

2.5. Autres optimisations diverses

Finalement, le chapitre "meilleures pratiques" de la documentation de référence et les éventuelles métriques JMX peuvent être utiles pour optimiser les performances.

3. Mapping xml-relationnel (entité en xml)

Hibernate permet (si on le souhaite) de gérer en mémoire des entités au format XML (au lieu d'utiliser des instances d'objet Java).

Les principaux nouveaux éléments de syntaxe sont les suivants:

- "element-name" - mappe vers l'élément XML nommé
- "@attribute-name" - mappe vers l'attribut XML nommé
- "." - mappe vers le parent de l'élément
- "element-name/@attribute-name" - mappe vers l'élément nommé de l'attribut nommé

On peut paramétrer un *double mapping* **Java<-->Relationnel** et **Xml<-->Relationnel**

(`<class` avec `name="javaClassName"` et `node=".."` >)
 ou bien un *simple mapping* **Xml<-->Relationnel**
 (`<class` avec `entity-name="..."` `node="..."` mais sans `name="javaClassName"` >).

Exemple:

```
<class entity-name="Account"
table="ACCOUNTS"
node="account">
    <id name="id"
        column="ACCOUNT_ID"
        node="@id"
        type="string"/>
    <many-to-one name="customerId"
        column="CUSTOMER_ID"
        node="customer/@id"
        embed-xml="false"
        entity-name="Customer"/>
    <property name="balance"
        column="BALANCE"
        node="balance"
        type="big_decimal"/>
    ...
</class>
```

structure Xml associée:

```
<account id="987632567" short-desc="Savings">
    <customer id="123456789"/>
    <balance>100.29</balance>
</account>
```

`embed-xml="false"` ==> liaison/référence sur correspondance entre id ou ...
`embed-xml="true"` ==> imbrication (sans id)

Construction directe d'un arbre Xml à partir d'une requête Hibernate :

```
Document doc = ....;
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();
List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}
```

```

}
tx.commit();
session.close();

```

NB: il est également possible de mettre la base de données à jour en déclenchant `dom4jSession.saveOrUpdate(...)` depuis un arbre Xml .

Ceci est très pratique pour effectuer des **"import/export XML"** .

Autre élément utile: `dom4jSession.replicate(...)` .

4. Outils et plugins pour Hibernate

Parmi les différents outils et plugins facilitant l'usage d'hibernate, on peut citer:

- **HibernateSynchronizer** (très ancien plugin pour eclipse permettant de générer le mapping "ORM" dans des cas simples. Attention: "D.A.O." bizarre à ne pas générer)
- **HibernateTools** (de JBoss)
- Plugin **maven**

5. Hbm2ddl (SchemaExport)

Le fichier **Hibernate3.jar** contient un package "org.hibernate.tool.hbm2ddl" qui comporte des classes java (ex: *SchemaExport*) permettant de générer des instructions SQL "create table" à partir d'une configuration Hibernate .

Certaines classes peuvent être lancées via du code java , des lignes de commandes ou bien des tâches ANT .

Exemple:

lancer_hbm2ddl_SchemaExport.bat

```

set LIB_DIR=D:\tp\JAVA_EE\back\orm_hibernate\wksp_hibernate_jpa_2010\hibernate_hbm_spring\lib
set HB_CP=%LIB_DIR%\hibernate3.jar;%LIB_DIR%\commons-logging-1.0.4.jar;%LIB_DIR%\log4j-1.2.11.jar;
%LIB_DIR%\dom4j-1.6.1.jar;%LIB_DIR%\cglib-2.1.3.jar;%LIB_DIR%\commons-collections-2.1.1.jar
set APP_CP=D:\tp\JAVA_EE\back\orm_hibernate\wksp_hibernate_jpa_2010\hibernate_hbm_spring\bin
set GLOBAL_CP=%HB_CP%;%APP_CP%

```

```

java -cp "%GLOBAL_CP%" org.hibernate.tool.hbm2ddl.SchemaExport
--config=hibernate.cfg.xml --output=my_schema.ddl --text --format

```

NB: Si la configuration Hibernate est sous forme de classes annotées (et pas sous forme de fichiers ".hbm.xml") , il faut écrire quelques lignes de code java plutôt qu'un déclenchement via une ligne de commande.