Hibernate et la gestion de persistance

Claude Duvallet

Université du Havre
UFR Sciences et Techniques
25 rue Philippe Lebon - BP 540
76058 LE HAVRE CEDEX
Claude.Duvallet@gmail.com
http://litis.univ-lehavre.fr/~duvallet/

La gestion de la persistance avec Hibernate

Introduction

- Hibernate est un logiciel, écrit en java, qui permet de faire le mapping entre Objets Java et Objets stockés en base relationnelle.
- Il en assure aussi la persistance.
- Hibernate propose son propre langage d'interrogation HQL et a largement inspiré les concepteurs de l'API JPA.
- Hibernate 2.1.3 possède les fonctionnalités d'annuaire JNDI d'enregistrement et localisation des objets, la gestion des transactions.
- Hibernate a été écrit sous la responsabilité de Gavin King qui fait partie de l'équipe JBoss .
- Site de référence pour récupérer le produit et la documentation : `http://www.hibernate.org/`

La gestion de la persistance (1/2)

- La quasi-totalité des applications de gestion traitent des données dans des volumes plus ou moins important.
- Lorsque ce volume devient assez important, les données doivent être stockées dans une base de données.
- Il existe plusieurs types de base de données :
  - Hiérarchique :
    - Historiquement le type le plus ancien, ces bases de données étaient largement utilisées sur les gros systèmes de type mainframe.
    - Les données sont organisées de façon hiérarchique grâce à des pointeurs.
    - Exemple DL1, IMS, Adabas

La gestion de la persistance (2/2)

- Suite des types de bases de données :
  - Relationnelle (RDBMS / SGBDR) :
    - C'est le modèle le plus répandu actuellement.
    - Ce type de base de données repose sur les théories ensemblistes et l'algèbre relationnel.
    - Les données sont organisées en tables possédant des relations entre elles grâce à des clés primaires et étrangères.
    - Les opérations sur la base sont réalisées grâce à des requêtes SQL.
    - Exemple : MySQL, PosgreSQL, HSQLDB, Derby, etc.
  - Objet (ODBMS / SGBDO) : Exemple db4objects
  - XML (XDBMS) : Exemple : Xindice

$\Rightarrow$ La seconde catégorie (Relationnelle) est historiquement la plus répandue mais aussi une des moins compatible avec la programmation orienté objet.

Le modèle relationnel versus le modèle objet

- La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :
  - le modèle objet propose plus de fonctionnalités : héritage, polymorphisme, ...
  - les relations entre les entités des deux modèles sont différentes.
  - les objets ne possèdent pas d'identifiant en standard (hormis son adresse mémoire qui varie d'une exécution à l'autre).
  - Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique.
- La persistance des objets en Java possède de surcroît quelques inconvénients supplémentaires :
  - de multiples choix dans les solutions et les outils (standard, commerciaux, open source).
  - de multiple choix dans les API et leurs implémentations.
  - de nombreuses évolutions dans les API standards et les frameworks open source.

Les solutions de persistance avec Java (1/2)

- La première approche pour faire une correspondance entre ces deux modèles a été d'utiliser l'API JDBC fournie en standard avec le JDK.
- Cependant, cette approche possède plusieurs inconvénients majeurs :
  - nécessite l'écriture de nombreuses lignes de codes, souvent répétitives.
  - le mapping entre les tables et les objets est un travail de bas niveau.
  - ...
- Tous ces facteurs réduisent la productivité mais aussi les possibilités d'évolutions et de maintenance.
- De plus, une grande partie de ce travail peut être automatisé.

Les solutions de persistance avec Java (2/2)

- Face à ce constat, différentes solutions sont apparues :
  - des frameworks open source : le plus populaire est Hibernate qui utilise des POJOs.
  - des frameworks commerciaux dont Toplink était le leader avant que sa base de données devienne open source.
  - des API Standards : JDO, EJB entity, JPA.

Le mapping Objet/Relationnel (O/R) (1/2)

- Le mapping Objet/Relationnel (mapping O/R) consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objets de façon la plus facile possible.
- Un outil de mappping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :
  - Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités.
  - Proposer une interface qui permette de facilement mettre en œuvre des actions de type CRUD.
  - Éventuellement permettre l'héritage des mappings.
  - Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée.
  - Supporter différentes formes d'identifiants générés automatiquement par les bases de données (identity, sequence, ...).

Le mapping Objet/Relationnel (O/R) (2/2)

- Autres fonctionnalités proposées :
  - Support des transactions.
  - Gestion des accès concurrents (verrou, dead lock, ...).
  - Fonctionnalités pour améliorer les performances (cache, lazy loading, ...).
- Les solutions de mapping sont donc riches en fonctionnalités ce qui peut rendre leur mise en œuvre plus ou moins complexe.
- Les solutions de mapping O/R permettent de réduire la quantité de code à produire mais impliquent une partie configuration (généralement sous la forme d'un ou plusieurs fichiers XML ou d'annotations pour les solutions reposant sur Java 5).
- Depuis quelques années, les principales solutions mettent en œuvre des POJO (Plain Old Java Object).

L'architecture de gestion de la persistance

- Dans une architecture en couche, il est important de prévoir une couche dédiée aux accès aux données.
- Il est assez fréquent dans cette couche de parler de la notion de CRUD qui représentent un ensemble des 4 opérations de bases réalisable sur une données.
- Il est aussi de bon usage de mettre en œuvre le design pattern DAO (Data Access Object) proposé par Sun.

La couche de persistance (1/2)

- La partie du code responsable de l'accès aux données dans une application multi niveaux doit être encapsulée dans une couche dédiée aux interactions avec la base de données de l'architecture généralement appelée couche de persistance.
- Celle-ci permet notamment :
  - d'ajouter un niveau d'abstraction entre la base de données et l'utilisation qui en est faite.
  - de simplifier la couche métier qui utilise les traitements de cette couche
  - de masquer les traitements réalisés pour mapper les objets dans la base de données et vice et versa
  - de faciliter le remplacement de la base de données utilisée

La couche de persistance (2/2)

- La couche métier
  - utilise la couche de persistance et reste indépendante du code dédié à l'accès à la base de données.
  - La couche métier ne contient aucune requête SQL, ni code de connexion ou d'accès à la base de données.
  - La couche métier utilise les classes de la couche métier qui encapsulent ces traitements.
  - La couche métier manipule uniquement des objets pour les accès à la base de données.
- Le choix des API ou des outils dépends du contexte : certaines solutions ne sont utilisables qu'avec la plate-forme Enterprise Edition (exemple : les EJB) ou sont utilisables indifféremment avec les plates-formes Standard et Enterprise Edition.
- L'utilisation d'une API standard permet de garantir la pérennité et de choisir l'implémentation à mettre en œuvre.

Les opérations de type CRUD

- L'acronyme CRUD signifie : Create, Read, Update and Delete.
- Il désigne les quatre opérations réalisables sur des données (création, lecture, mise à jour et suppression).
- Exemple : une interface qui propose des opérations de type CRUD pour un objet de type Entite

```
public interface EntiteCrud {
  public Entite obtenir(Integer id);
  public void creer(Entite entite);
  public void modifier(Entite entite);
  public Collection obtenirTous();
  public void supprimer(Entite entite);
}
```


Le modèle de conception DAO (Data Access Object) (1/3)

- C'est un modèle de conception qui propose de découpler l'accès à une source de données.
- L'accès aux données dépend fortement de la source de données.
- Par exemple, l'utilisation d'une base de données est spécifique pour chaque fournisseur.
- Même si SQL et JDBC assurent une partie de l'indépendance vis-à-vis de la base de données utilisées, certaines contraintes imposent une mise à en œuvre spécifique de certaines fonctionnalités.
- Par exemple, la gestion des champs de type identifiants est proposée selon diverses formes par les bases de données : champ auto-incrémenté, identity, séquence, ...

Le modèle de conception DAO (Data Access Object) (2/3)

- Le motif de conception DAO proposé dans le blue print de Sun propose de séparer les traitements d'accès physique à une source de données de leur utilisation dans les objets métiers.
- Cette séparation permet de modifier une source de données sans avoir à modifier les traitements qui l'utilise.
- Le DAO peut aussi proposer un mécanisme pour rendre l'accès aux bases de données indépendant de la base de données utilisées et même rendre celle-ci paramétrable.
- Les classes métier utilisent le DAO via son interface et sont donc indépendantes de son implémentation.
- Si cette implémentation change (par exemple un changement de base de données), seul l'implémentation du DAO est modifié mais les classes qui l'utilisent via son interface ne sont pas impactées.
- Le DAO définit donc une interface qui va exposer les fonctionnalités utilisables.

Le modèle de conception DAO (Data Access Object) (3/3)

- Ces fonctionnalités doivent être indépendantes de l'implémentation sous jacente.
- Par exemple, aucune méthode ne doit avoir de requêtes SQL en paramètre.
- Pour les même raisons, le DAO doit proposer sa propre hiérarchie d'exceptions.
- Une implémentation concrète de cette interface doit être proposée.
- Cette implémentation peut être plus ou moins complexe en fonction de critères de simplicité ou de flexibilité.
- Fréquemment les DAO ne mettent pas en œuvre certaines fonctionnalités comme la mise en œuvre d'un cache ou la gestion des accès concurrents.

Les API standards

- JDBC : déjà vu.
- JDO : acronyme de Java Data Object.
  - Le but de cet API est de rendre transparent la persistance d'un objet.
  - Il repose sur l'enrichissement de byte-code à la compilation.
- Les Enterprise JavaBeans (EJB) 2.0 :
  - Les EJB (Enterprise Java Bean) proposent des beans de type Entités pour assurer la persistance des objets.
  - Les EJB de type Entité peuvent être de deux types :
    - CMP (Container Managed Persistance) : la persistance est assuré par le conteneur d'EJB en fonction du paramétrage fourni
    - BMP (Bean Managed Persistance) :
  - Les EJB bénéficient des services proposés par le conteneur cependant cela les rends dépendant de ce conteneur pour l'exécution : ils sont difficilement utilisables en dehors du conteneur (par exemple pour les tester).

Java Persistence API (JPA) et les EJB 3.0

- L'API repose sur
  - l'utilisation d'entités persistantes sous la forme de POJOs
  - un gestionnaire de persistance (EntityManager) qui assure la gestion des entités persistantes
  - l'utilisation d'annotations
  - la configuration via des fichiers xml
- JPA peut être utilisé avec Java EE dans un serveur d'application mais aussi avec Java SE (avec quelques fonctionnalités proposées par le conteneur en moins).
- JPA est une spécification : il est nécessaire d'utiliser une implémentation pour la mettre en œuvre.
- L'implémentation de référence est la partie open source d'Oracle Toplink : Toplink essential.
- La version 3.2 d'Hibernate implémente aussi JPA.
- JPA ne peut être utilisé qu'avec des bases de données relationnelles.

Java Data Object

- JDO (Java Data Object) est la spécification du JCP numéro 12 qui propose une technologie pour assurer la persistance d'objets Java dans un système de gestion de données.
- La spécification regroupe un ensemble d'interfaces et de règles qui doivent être implémentées par un fournisseur tiers.
- La version 1.0 de cette spécification a été validée au premier trimestre 2002.
- JDO propose de faciliter le mapping entre des données stockées dans un format particulier (bases de données ...) et un objet en fournissant un standard.

Présentation de JDO (1/2)

- Les principaux buts de JDO sont :
  - la facilité d'utilisation (gestion automatique du mapping des données).
  - la persistance universelle : persistance vers tout type de système de gestion de ressources (bases de données relationnelles, fichiers, ...).
  - la transparence vis à vis du système de gestion de ressources utilisé : ce n'est plus le développeur mais JDO qui dialogue avec le système de gestion de ressources.
  - la standardisation des accès aux données.
  - la prise en compte des transactions.

Présentation de JDO (2/2)

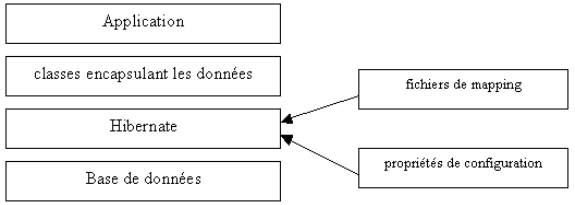
- Le développement avec JDO se déroule en plusieurs étapes :
  1. écriture des objets contenant les données (des beans qui encapsulent les données) : un tel objet est nommé instance JDO.
  2. écriture des objets qui utilisent les objets métiers pour répondre aux besoins fonctionnels. Ces objets utilisent l'API JDO.
  3. écriture du fichier metadata qui précise le mapping entre les objets et le système de gestion des ressources. Cette partie est très dépendante du système de gestion de ressources utilisé.
  4. enrichissement des objets métiers.
  5. configuration du système de gestion des ressources.
- JDO est une spécification qui définit un standard : pour pouvoir l'utiliser il faut utiliser une implémentation fournie par un fournisseur.
- Chaque implémentation est capable d'utiliser un ou plusieurs systèmes de stockage de données particulier (base de données relationnel, base de données objets, fichiers, ...).

Présentation d'Hibernate (1/2)

- Hibernate est un projet open source visant à proposer un outil de mapping entre les objets et des données stockées dans une base de données relationnelle.
- Ce projet ne repose sur aucun standard mais il est très populaire notamment à cause de ses bonnes performances et de son ouverture avec de nombreuses bases de données.
- Les bases de données supportées sont les principales du marché : DB2, Oracle, MySQL, PostgreSQL, Sybase, SQL Server, Sap DB, Interbase, ...
- Le site officiel http ://www.hibernate.org contient beaucoup d'informations sur l'outil et propose de le télécharger ainsi que sa documentation.

Présentation d'Hibernate (2/2)

- Hibernate a besoin de plusieurs éléments pour fonctionner :
  - une classe de type javabean qui encapsule les données d'une occurrence d'une table
  - un fichier de correspondance qui configure la correspondance entre la classe et la table
  - des propriétés de configuration notamment des informations concernant la connexion à la base de données
- Une fois ces éléments correctement définis, il est possible d'utiliser Hibernate dans le code des traitements à réaliser.
- L'architecture d'Hibernate est donc la suivante :



Création d'une classe encapsulant les données

- Cette classe doit respecter le standard des javabeans notamment encapsuler les propriétés dans ces champs private avec des getters et setters et avoir un constructeur par défaut.
- Les types utilisables pour les propriétés sont : les types primitifs, les classes String et Dates, les wrappers, et n'importe quelle classe qui encapsule une autre table ou une partie de la table.
- Dans notre exemple, il s'agira de la classe `Personnes`.
- On supposera qu'il existe une table `Personnes` dans une base de données de type MySQL.

La table `Personnes`

Elle contient les champs suivant :

- *idpersonne* de type *int*(11)
- *prenom* de type *varchar*(50)
- *nom* de type *varchar*(50)
- *datedenaissance* de type *datetime*

Création d'un fichier de correspondance (1/3)

- Pour assurer le mapping, Hibernate a besoin d'un fichier de correspondance (mapping file) au format XML qui va contenir des informations sur la correspondance entre la classe définie et la table de la base de données.
- Même si cela est possible, il n'est pas recommandé de définir un fichier de mapping pour plusieurs classes.
- Le plus simple est de définir un fichier de mapping par classe, nommé du nom de la classe suivi par ".hbm.xml".
- Ce fichier doit être situé dans le même répertoire que la classe correspondante ou dans la même archive pour les applications packagées.

Création d'un fichier de correspondance (2/3)

- Différents éléments sont précisés dans ce document XML :
  - la classe qui va encapsuler les données,
  - l'identifiant dans la base de données et son mode de génération,
  - le mapping entre les propriétés de classe et les champs de la base de données,
  - les relations,
  - etc.
- Le fichier débute par un prologue et une définition de la DTD utilisée par le fichier XML.
- Exemple :

```
<?xml version="1.0"?>
  <!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
```

Création d'un fichier de correspondance (3/3)

- Le tag racine du document XML est le tag `<hibernate-mapping>`.
- Ce tag peut contient un ou plusieurs tag `<class>` : il est cependant préférable de n'utiliser qu'un seul tag `<class>` et de définir autant de fichiers de correspondance que de classes.
- Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class name="Personnes" table="personnes">
    <id name="idPersonne" type="int" column="idpersonne">
      <generator class="native"/>
    </id>
    <property name="nom" type="string" not-null="true" />
    <property name="prenom" type="string" not-null="true" />
    <property name="datedenaissance" type="date">
      <meta attribute="field-description">date de naissance</meta>
    </property>
  </class>
</hibernate-mapping>
```


Propriétés de configuration (1/5)

- Pour exécuter Hibernate, il faut lui fournir un certain nombre de propriétés concernant sa configuration pour qu'il puisse se connecter à la base de données.
- Ces propriétés peuvent être fournies sous plusieurs formes :
  - un fichier de configuration nommé hibernate.properties et stocké dans un répertoire inclus dans le classpath
  - un fichier de configuration au format XML nommé hibernate.cfg.xml
  - utiliser la méthode setProperties() de la classe Configuration
  - définir des propriétés dans la JVM en utilisant l'option -Dpropriété=valeur

Propriétés de configuration (2/5)

- Les principales propriétés pour configurer la connexion JDBC sont :

Nom de la propriété	Rôle
hibernate.connection.driver_class	nom pleinement qualifié de classe du pilote JDBC
hibernate.connection.url	URL JDBC désignant la base de données
hibernate.connection.username	nom de l'utilisateur pour la connexion
hibernate.connection.password	mot de passe de l'utilisateur
hibernate.connection.pool_size	nombre maximum de connexions dans le pool

- Les principales propriétés pour configurer une source de données (DataSource) à utiliser sont :

Nom de la propriété	Rôle
hibernate.connection.datasource	nom du DataSource enregistré dans JNDI
hibernate.jndi.url	URL du fournisseur JNDI
hibernate.jndi.class	classe pleinement qualifiée de type InitialContextFactory permettant l'accès à JNDI
hibernate.connection.username	nom de l'utilisateur de la base de données
hibernate.connection.password	mot de passe de l'utilisateur

Propriétés de configuration (3/5)

- Les principales autres propriétés sont :

Nom de la propriété	Rôle
hibernate.dialect	nom de la classe pleinement qualifiée qui assure le dialogue avec la base de données
hibernate.jdbc.use_scrollable_resultset	booléen qui permet le parcours dans les deux sens pour les connexions fournies à Hibernate utilisant les pilotes JDBC 2 supportant cette fonctionnalité
hibernate.show_sql	booléen qui précise si les requêtes SQL générées par Hibernate sont affichées dans la console (particulièrement utile lors du débogage)

- Hibernate propose des classes qui héritent de la classe Dialect pour chaque base de données supportée. C'est le nom de la classe correspondant à la base de données utilisées qui doit être obligatoirement fourni à la propriété hibernate.dialect.
- Pour définir les propriétés utiles, le plus simple est de définir un fichier de configuration qui en standard doit se nommer hibernate.properties.

Propriétés de configuration (4/5)

- Ce fichier contient des paires clé=valeur pour chaque propriété définie.
- Exemple de paramètres pour utiliser une base de données MySQL :

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

- Le pilote de la base de données utilisée doit être ajouté dans le classpath.
- Il est aussi possible de définir les propriétés dans un fichier au format XML nommé en standard `hibernate.cfg.xml`.
- Les propriétés sont alors définies par un tag `<property>`.

Propriétés de configuration (5/5)

- Le nom de la propriété est fourni grâce à l'attribut « name » et sa valeur est fourni dans le corps du tag.
- Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 2.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/MTP</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="dialect">net.sf.hibernate.dialect.MySQLDialect
    </property>
    <property name="show_sql">true</property>
    <mapping resource="Personnes.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```


Utilisation d'Hibernate (1/4)

- Pour utiliser Hibernate dans le code, il est nécessaire de réaliser plusieurs opérations :
  - création d'une instance de la classe
  - création d'une instance de la classe SessionFactory
  - création d'une instance de la classe Session qui va permettre d'utiliser les services d'Hibernate
- Si les propriétés sont définies dans le fichier hibernate.properties, il faut tout d'abord créer une instance de la classe Configuration.
- Pour lui associer la ou les classes encapsulant les données, la classe propose deux méthodes :
  - addFile() qui attend en paramètre le nom du fichier de mapping
  - addClass() qui attend en paramètre un objet de type Class encapsulant la classe.

  ⇒ Dans ce cas, la méthode va rechercher un fichier nommé `nom_de_la_classe.hbm.xml` dans le classpath (ce fichier doit se situer dans le même répertoire que le fichier .class de la classe correspondante).

Utilisation d'Hibernate (2/4)

- Une instance de la classe Session est obtenu à partir d'une fabrique de type SessionFactory.
- Cet objet est obtenu à partir de l'instance du type Configuration en utilisant la méthode buildSessionFactory().
- La méthode `openSession()` de la classe SessionFactory permet d'obtenir une instance de la classe Session.
- Par défaut, la méthode `openSession()` qui va ouvrir une connexion vers la base de données en utilisant les informations fournies par les propriétés de configuration.

Utilisation d'Hibernate (3/4)

- Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.Date;

public class TestHibernate1 {
  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();
    ...
```

- Il est aussi possible de fournir en paramètre de la méthode `openSession()` une instance de la classe `javax.sql.Connection` qui encapsule la connexion à la base de données.

Utilisation d'Hibernate (4/4)

- Pour une utilisation du fichier `hibernate.cfg.xml`, il faut :
  - créer une occurrence de la classe `Configuration`,
  - appeler sa méthode `configure()` qui va lire le fichier XML,
  - et appeler la méthode `buildSessionFactory()` de l'objet renvoyer par la méthode `configure()`.
- Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate1 {
  public static void main(String args[]) throws Exception {
    SessionFactory sessionFactory =
         new Configuration().configure().buildSessionFactory();
    ...
```

- Il est important de clôturer l'objet `Session`, une fois que celui ci est devenu inutile, en utilisant la méthode `close()`.

Persistance d'une nouvelle occurrence (1/2)

- Pour créer une nouvelle occurrence dans la source de données, il suffit de :
  - créer une nouvelle instance de la classe encapsulant les données,
  - de valoriser ces propriétés,
  - et d'appeler la méthode `save()` de la session en lui passant en paramètre l'objet encapsulant les données.
- La méthode `save()` n'a aucune action directe sur la base de données.
- Pour enregistrer les données dans la base, il faut réaliser un commit sur la connexion ou la transaction ou faire appel à la méthode `flush()` de la classe `Session`.

Persistance d'une nouvelle occurrence (2/2)

- Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.Date;

public class TestHibernate1 {
  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Personnes personne = new Personnes("nom3", "prenom3", new Date());
      session.save(personne);
      session.flush() ;
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```

Obtention d'une occurrence à partir de son identifiant (1/2)

- La méthode `load()` de la classe `Session` permet d'obtenir une instance de la classe des données encapsulant les données de l'occurrence de la base dont l'identifiant est fourni en paramètre.
- Il existe deux surcharges de la méthode :
  - la première attend en premier paramètre le type de la classe des données et renvoie une nouvelle instance de cette classe
  - la seconde attend en paramètre une instance de la classe des données et la met à jour avec les données retrouvées

Obtention d'une occurrence à partir de son identifiant (2/2)

- Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;

public class TestHibernate2 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();
    try {
      Personnes personne = (Personnes) session.load(Personnes.class, new Integer(3
      System.out.println("nom = " + personne.getNomPersonne());
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```


Le langage de requête HQL (1/2)

- Pour offrir un langage d'interrogation commun à toutes les bases de données, Hibernate propose son propre langage nommé HQL (Hibernate Query Language).
- Le langage HQL est proche de SQL avec une utilisation sous forme d'objets des noms de certaines entités.
- Il n'y a aucune référence aux tables ou aux champs car ceux-ci sont référencés respectivement par leur classe et leurs propriétés.
- C'est Hibernate qui se charge de générer la requête SQL à partir de la requête HQL en tenant compte du contexte.
- C'est-à-dire le type de base de données utilisée et défini dans le fichier de configuration et la configuration du mapping.

Le langage de requête HQL (2/2)

- La méthode `find()` de la classe Session permet d'effectuer une recherche d'occurrences grâce à la requête fournie en paramètre.
- La méthode `find()` possède deux surcharges pour permettre de fournir un seul ou plusieurs paramètres dans la requête.
- La première surcharge permet de fournir un seul paramètre : elle attend en paramètre la requête, la valeur du paramètre et le type du paramètre.

Rechercher toutes les lignes d'une table : exemple

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate3 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      List personnes = session.find("from Personnes");
      for (int i = 0; i < personnes.size(); i++) {
        Personnes personne = (Personnes) personnes.get(i);
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```


Rechercher une ligne d'une table : exemple

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate4 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      List personnes = session.find("from Personnes p where p.nomPersonne=?",
        "Duvallet", Hibernate.STRING);
      for (int i = 0; i < personnes.size(); i++) {
        Personnes personne = (Personnes) personnes.get(i);
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
```


Utilisation de la méthode `iterate ()` : exemple

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate5 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Iterator personnes = session.iterate("from Personnes ");
      while (personnes.hasNext()) {
        Personnes personne = (Personnes) personnes.next();
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```


Les possibilités du langage de requête HQL

- Il est aussi possible de trier les occurences d'une requête en utilisant la clause « order by » dans une requête HQL.

  Exemple :

```
List personnes = session.find("from Personnes p
                               order by p.nomPersonne desc");
```

- Il est possible d'utiliser des fonctions telles que « count() » pour compter le nombre d'occurrences.

  Exemple :

```
...
int compteur = ((Integer) session.iterate(
                 "select count(*) from Personnes").next() ).intValue();
System.out.println("compteur = " + compteur);
...
```

L'interface `Query` (1/2)

- Il est également possible de définir des requêtes utilisant des paramètres nommés grâce à un objet implémentant l'interface `Query`.
- Un objet de type `Query` est obtenu en invoquant la méthode `createQuery()` de la classe `Session` avec comme paramètre la requête HQL.
- Dans cette requête, les paramètres sont précisés avec un caractère « : » suivi d'un nom unique.
- L'interface `Query` propose de nombreuses méthodes `setXXX()` pour associer à chaque paramètre une valeur en fonction du type de la valeur (XXX représente le type).

L'interface `Query` (2/2)

- Chacune des méthodes possède deux surcharges permettant de préciser le paramètre (à partir de son nom ou de son index dans la requête) et sa valeur.
- Pour parcourir la collection des occurrences trouvées, l'interface `Query` propose la méthode `list()` qui renvoie une collection de type `List` ou la méthode `iterate()` qui renvoie un itérateur sur la collection.

Utilisation de l'interface `Query` : exemple

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate8 {
  public static void main(String args[]) throws Exception {
    SessionFactory sessionFactory = new Configuration().configure()
                                                       .buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Query query = session.createQuery(
                      "from Personnes p where p.nomPersonne =:nom");
      query.setString("nom", "Duvallet");
      Iterator personnes = query.iterate();
      while (personnes.hasNext()) {
        Personnes personne = (Personnes) personnes.next();
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
```


Mise à jour d'une occurrence

- Pour mettre à jour une occurrence dans la source de données, il suffit d'appeler la méthode `update()` de la session en lui passant en paramètre l'objet encapsulant les données.
- Le mode de fonctionnement de cette méthode est similaire à celui de la méthode `save()`.
- La méthode `saveOrUpdate()` laisse Hibernate choisir entre l'utilisation de la méthode `save()` ou `update()` en fonction de la valeur de l'identifiant dans la classe encapsulant les données.

Suppression d'une ou plusieurs occurrences

- La méthode `delete()` de la classe `Session` permet de supprimer une ou plusieurs occurrences en fonction de la version surchargée de la méthode utilisée.
- Pour supprimer une occurrence encapsulée dans une classe, il suffit d'invoquer la classe en lui passant en paramètre l'instance de la classe.
- Pour supprimer plusieurs occurrences, voire toutes, il faut passer en paramètre de la méthode `delete()`, une chaîne de caractères contenant la requête HQL pour préciser les éléments concernés par la suppression.
- Exemple : suppression de toutes les occurrences de la table
```
session.delete("from Personnes");
```


Les relations

- Un des fondements du modèle de données relationnelles repose sur les relations qui peuvent intervenir entre une table et une ou plusieurs autres tables ou la table elle même.
- Hibernate propose de transcrire ces relations du modèle relationnel dans le modèle objet. Il supporte plusieurs types de relations :
  - relation de type 1 - 1 (one-to-one).
  - relation de type 1 - n (many-to-one).
  - relation de type n - n (many-to-many).
- Dans le fichier de mapping, il est nécessaire de définir les relations entre la table concernée et les tables avec lesquelles elle possède des relations.

Java Persistence API (JPA) (1/2)

- L'utilisation pour la persistance d'un mapping O/R permet de proposer un niveau d'abstraction plus élevé que la simple utilisation de JDBC.
- Ce mapping permet d'assurer la transformation d'objets vers la base de données et vice et versa que cela soit pour des lectures ou des mises à jour (création, modification ou suppression).
- Développée dans le cadre de la version 3.0 des EJB, cette API ne se limite pas aux EJB puisqu'elle aussi être mise en œuvre dans des applications Java SE.
- L'utilisation de l'API ne requiert aucune ligne de code mettant en œuvre l'API JDBC.

Java Persistence API (JPA) (2/2)

- L'API propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données.
- L'API Java Persistence repose sur des entités qui sont de simples POJOs annotés et sur un gestionnaire de ces entités (`EntityManager`) qui propose des fonctionnalités pour les manipuler (ajout, modification suppression, recherche).
- Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

Les entités (1/2)

- Les entités dans les spécifications de l'API Java Persistence permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables.
- Ce sont de simples POJO (Plain Old Java Object).
- Un POJO est une classe Java qui n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.
- Un objet Java de type POJO mappé vers une table de la base de données grâce à des méta data via l'API Java Persistence est nommé bean entité (Entity bean).
- Un bean entité doit obligatoirement avoir un constructeur sans argument et la classe du bean doit obligatoirement être marquée avec l'annotation `@javax.persistence.Entity`.
- Cette annotation possède un attribut optionnel nommé name qui permet de préciser le nom de l'entité dans les requêtes.
- Par défaut, ce nom est celui de la classe de l'entité.

Les entités (2/2)

- En tant que POJO, le bean entity n'a pas à implémenter d'interface particulière mais il doit en plus de respecter les règles de tous Javabeans :
  - Être déclaré avec l'annotation `@javax.persistence.Entity`
  - Posséder au moins une propriété déclarer comme clé primaire avec l'annotation `@Id`.
- Le bean entity est composé de propriétés qui seront mappés sur les champs de la table de la base de données sous jacente.
- Chaque propriété encapsule les données d'un champ d'une table.
- Ces propriétés sont utilisables au travers de simple accesseurs (getter/setter).
- Une propriété particulière est la clé primaire qui sert d'identifiant unique dans la base de données mais aussi dans le POJO.
  - Elle peut être de type primitif ou de type objet.
  - La déclaration de cette clé primaire est obligatoire.

Le mapping entre le bean entité et la table

- La description du mapping entre le bean entité et la table peut être fait de deux façons :
  - Utiliser des annotations
  - Utiliser un fichier XML de mapping
- L'API propose plusieurs annotations pour supporter un mapping O/R assez complet.

Annotation	Rôle
@javax.persistence.Table	Préciser le nom de la table concernée par le mapping.
@javax.persistence.Column	Associé à un getter, il permet d'associer un champ de la table à la propriété.
@javax.persistence.Id	Associé à un getter, il permet d'associer un champ de la table à la propriété en tant que clé primaire.
@javax.persistence.GeneratedValue	Demander la génération automatique de la clé primaire.
@javax.persistence.Basic	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut.
@javax.persistence.Transient	Demander de ne pas tenir compte du champ lors du mapping.

L'annotation `@Table`

- L'annotation `@javax.persistence.Table` permet de lier l'entité à une table de la base de données.
- Par défaut, l'entité est liée à la table de la base de données correspondant au nom de la classe de l'entité.
- Si ce nom est différent alors l'utilisation de l'annotation `@Table` est obligatoire.
- C'est notamment le cas si des conventions de nommage des entités de la base de données sont mises en place.
- L'annotation `@Table` possède plusieurs attributs :

Attribut	Rôle
name	Nom de la table
catalog	Catalogue de la table
schema	Schéma de la table
uniqueConstraints	Contraintes d'unicité sur une ou plusieurs colonnes

L'annotation `@Column` (1/2)

- L'annotation `@javax.persistence.Column` permet d'associer un membre de l'entité à une colonne de la table.
- Par défaut, les champs de l'entité sont liés aux champs de la table dont les noms correspondent.
- Si ces noms sont différents alors l'utilisation de l'annotation `@Column` est obligatoire.
- C'est notamment le cas si des conventions de nommage des entités de la base de données sont mises en place.

L'annotation `@Column` (2/2)

- L'annotation `@Column` possède plusieurs attributs :

Attribut	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi-table
unique	Indique si la colonne est unique
nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update
columnDefinition	Précise le DDL de définition de la colonne
length	Indique la taille d'une colonne de type chaîne de caractères
precision	Indique la taille d'une colonne de type numérique
scale	Indique la précision d'une colonne de type numérique

- Hormis les attributs name et table, tous les autres attributs ne sont utilisés que par un éventuel outil du fournisseur de l'implémentation de l'API pour générer automatiquement la table dans la base de données.

La définition de la clé primaire (1/2)

- Il faut obligatoirement définir une des propriétés de la classe avec l'annotation `@Id` pour la déclarer comme étant la clé primaire de la table.
- Cette annotation peut marquer soit le champ de la classe concernée soit le getter de la propriété.
- L'utilisation de l'un ou l'autre précise au gestionnaire s'il doit se baser sur les champs ou les getter pour déterminer les associations entre l'entité et les champs de la table.
- La clé primaire peut être constituée d'une seule propriété ou composées de plusieurs propriétés qui peuvent être de type primitif ou chaîne de caractères.
- La clé primaire composée d'un seul champ peut être une propriété d'un type primitif, ou une chaîne de caractères (`String`).

La définition de la clé primaire (2/2)

- La clé primaire peut être générée automatiquement en utilisant l'annotation `@javax.persistence.GeneratedValue`.
- Cette annotation possède plusieurs attributs :

Attributs	Rôle
strategy	Précise le type de générateur à utiliser : TABLE, SEQUENCE, IDENTITY ou AUTO. La valeur par défaut est AUTO
generator	Nom du générateur à utiliser

- Le type `AUTO` est le plus généralement utilisé : il laisse l'implémentation générer la valeur de la clé primaire.
- Le type `IDENTITY` utilise un type de colonne spécial de la base de données.
- Le type `TABLE` utilise une table dédiée qui stocke les clés des tables générées.
- L'utilisation de cette stratégie nécessite l'utilisation de l'annotation `@javax.persistence.TableGenerator`.

La définition de la clé primaire : Exemple

```
import java.io.Serializable;
import javax.persistence.*;

@Entity public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;
  private String prenom;
  private String nom;
  private static final long serialVersionUID = 1L;

  public Personne() { super(); }

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

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

  public String getPrenom() { return this.prenom; }

  public void setPrenom(String prenom) { this.prenom = prenom; }

  public String getNom() { return this.nom; }

  public void setNom(String nom) { this.nom = nom; }
}
```


L'annotation `@TableGenerator`

- L'annotation `@TableGenerator` possède plusieurs attributs :

Attributs	Rôle
name	Nom identifiant le `TableGenerator` : il devra être utilisé comme valeur dans l'attribut `generator` de l'annotation `@Id`
table	Nom de la table utilisé
catalog	Nom du catalogue utilisé
schema	Nom du schéma utilisé
pkColumnName	Nom de la colonne qui précise la clé primaire à générer
valueColumnName	Nom de la colonne qui contient la valeur de la clé primaire générée
pkColumnValue	
allocationSize	Valeur utilisée lors de l'incrémentation de la valeur de la clé primaire
uniqueConstraints	

- Le type SEQUENCE utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle.

L'annotation `@SequenceGenerator`

- L'utilisation de la stratégie « SEQUENCE » nécessite l'utilisation de l'annotation `@javax.persistence.SequenceGenerator`.
- L'annotation `@SequenceGenerator` possède plusieurs attributs :

Attributs	Rôle
name	Nom identifiant le SequenceGenerator : il devra être utilisé comme valeur dans l'attribut `generator` de l'annotation `@Id`
sequenceName	Nom de la séquence dans la base de données
initialValue	Valeur initiale de la séquence
allocationSize	Valeur utilisée lors l'incrémentation de la valeur de la séquence

- L'annotation `@SequenceGenerator` s'utilise sur la classe de l'entité.

Exemple d'utilisation de l'annotation `@SequenceGenerator`

```
@Entity
@Table(name="PERSONNE")
@SequenceGenerator(name="PERSONNE_SEQUENCE",
                   sequenceName="PERSONNE_SEQ")
public class Personne implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE,
                  generator="PERSONNE_SEQUENCE")
  private int id;
```

Clé primaire composée de plusieurs colonnes

- Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes.
- L'API Java Persistence propose deux façons de gérer ce cas de figure :
  - L'annotation `@javax.persistence.IdClass`
  - L'annotation `@javax.persistence.EmbeddedId`
- L'annotation `@IdClass` s'utilise avec une classe qui va encapsuler les propriétés qui composent la clé primaire.
- Cette classe doit obligatoirement :
  - Être sérialisable.
  - Posséder un constructeur sans argument.
  - Fournir une implémentation dédiée des méthodes `equals()` et `hashCode()`.

Exemple de clé primaire sur plusieurs colonnes (1/6)

- La clé primaire est composée des champs nom et prenom
- Cet exemple présume que deux personnes ne peuvent avoir le même nom et prénom.

```
public class PersonnePK implements java.io.Serializable {

  private static final long serialVersionUID = 1L;
  private String nom;
  private String prenom;

  public PersonnePK() {}

  public PersonnePK(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() { return this.nom; }

  public void setNom(String nom) { this.nom = nom; }

  public String getPrenom() { return prenom; }

  public void setPrenom(String prenom) { this.nom = prenom; }
```


Exemple de clé primaire sur plusieurs colonnes (2/6)

```
  public boolean equals(Object obj) {
    boolean resultat = false;

    if (obj == this) {
      resultat = true;
    } else {
      if (!(obj instanceof PersonnePK)) {
        resultat = false;
      } else {
        PersonnePK autre = (PersonnePK) obj;
        if (!nom.equals(autre.nom)) {
          resultat = false;
        } else {
          if (prenom != autre.prenom) resultat = false;
          else resultat = true;
        }
      }
    }
    return resultat;
  }

  public int hashCode() { return (nom + prenom).hashCode(); }
}
```


Exemple de clé primaire sur plusieurs colonnes (3/6)

- Il est nécessaire de définir la classe de la clé primaire dans le fichier de configuration `persistence.xml`.
- Exemple :

```
<?xml version="1.0"  encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        version="1.0"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit  name="MaBaseDeTestPU">
         <provider>oracle.toplink.essentials.PersistenceProvider</provider>
         <class>Personne</class>
         <class>PersonnePK</class>
    </persistence-unit>
</persistence>
```

- L'annotation `@IdClass` possède un seul attribut `Class` qui représente la classe encapsulant la clé primaire composée.
- Il faut utiliser l'annotation `@IdClass` sur la classe de l'entité.

Exemple de clé primaire sur plusieurs colonnes (4/6)

- Il est nécessaire de marquer chacune des propriétés de l'entité qui compose la clé primaire avec l'annotation `@Id`.
- Ces propriétés doivent avoir le même nom dans l'entité et dans la classe qui encapsule la clé primaire.
- Exemple :

```
import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {
  private String prenom;
  private String nom;
  private int taille;

  private static final long serialVersionUID = 1L;
```


Exemple de clé primaire sur plusieurs colonnes (5/6)

```
  public Personne() {
    super();
  }

  @Id
  public String getPrenom() {  return this.prenom; }

  public void setPrenom(String prenom) {  this.prenom = prenom; }

  @Id
  public String getNom() {  return this.nom; }

  public void setNom(String nom) { this.nom = nom; }

  public int getTaille() {  return this.taille; }

  public void setTaille(int taille) { this.taille = taille;  }
}
```

- Remarque : il n'est pas possible de demander la génération automatique d'une clé primaire composée.

Exemple de clé primaire sur plusieurs colonnes (6/6)

- Les valeurs de chacune des propriétés de la clé doivent être fournies explicitement.
- La classe de la clé primaire est utilisée notamment lors des recherches.
- Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");
Personne personne = entityManager.find(Personne.class, clePersonne);
```


L'annotation @EmbeddedId (1/2)

- L'annotation `@EmbeddedId` s'utilise avec l'annotation `@javax.persistence.Embeddable`
- Exemple :

```
import javax.persistence.Embeddable;

@Embeddable
public class PersonnePK implements java.io.Serializable {
  private static final long serialVersionUID = 1L;
  private String nom;
  private String prenom;

  public PersonnePK() {  }

  public PersonnePK(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() { return this.nom; }

  public void setNom(String nom) { this.nom = nom; }
```

L'annotation @EmbeddedId (2/2)

```
  public String getPrenom() { return prenom; }

  public void setPrenom(String prenom) { this.nom = prenom; }

  public boolean equals(Object obj) {
    boolean resultat = false;
    if (obj == this) resultat = true;
    else {
      if (!(obj instanceof PersonnePK)) resultat = false;
      else {
        PersonnePK autre = (PersonnePK) obj;
        if (!nom.equals(autre.nom)) {
          resultat = false;
        } else {
          if (prenom != autre.prenom) resultat = false;
          else resultat = true;
        }
      }
    }
    return resultat;
  }

  public int hashCode() {  return (nom + prenom).hashCode(); }
}
```


Suite de l'exemple

```
import java.io.Serializable;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class Personne implements Serializable {
  @EmbeddedId
  private PersonnePK clePrimaire;
  private int taille;

  private static final long serialVersionUID = 1L;

  public Personne() { super(); }

  public PersonnePK getClePrimaire() { return this.clePrimaire; }

  public void setNom(PersonnePK clePrimaire) { this.clePrimaire = clePrimaire; }

  public int getTaille() { return this.taille; }

  public void setTaille(int taille) { this.taille = taille; }
}
```


Recherche d'éléments

- La classe qui encapsule la clé primaire est utilisée notamment dans les recherches
- Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");
Personne personne = entityManager.find(Personne.class, clePersonne);
```


L'annotation `@AttributeOverrides`

- L'annotation `@AttributeOverrides` est une collection d'attribut `@AttributeOverride`.
- Ces annotations permettent de ne pas avoir à utiliser l'annotation `@Column` dans la classe de la clé ou de modifier les attributs de cette annotation dans l'entité qui la met en œuvre.
- L'annotation `@AttributeOverride` possède plusieurs attributs :

Attributs	Rôle
name	Précise le nom de la propriété de la classe imbriquée
column	Précise la colonne de la table à associer à la propriété

L'annotation `@AttributeOverrides` : exemple

```
import java.io.Serializable;

import javax.persistence.AttributeOverrides;
import javax.persistence.AttributeOverride;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Column;

@Entity
public class Personne4 implements Serializable {
  @EmbeddedId
  @AttributeOverrides({
    @AttributeOverride(name="nom", column=@Column(name="NOM") ),
    @AttributeOverride(name="prenom", column=@Column(name="PRENOM") )
    })
  private PersonnePK clePrimaire;
  private int taille;
```


Les annotation de mapping des colonnes

- Par défaut, toutes les propriétés sont mappées sur la colonne correspondante dans la table.
- L'annotation `@javax.persistence.Transient` permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.
- L'annotation `@javax.persistence.Basic` représente la forme de mapping la plus simple.
- C'est aussi celle par défaut ce qui rend son utilisation optionnelle.
- Ce mapping concerne les types primitifs, les wrappers de type primitifs, les tableaux de ces types et les types `BigInteger`, `BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp`.

L'annotation `@Basic`

- L'annotation `@Basic` possède plusieurs attributs :

Attributs	Rôle
fetch	Permet de préciser comment la propriété est chargée selon deux modes : ■ LAZY : la valeur est chargée uniquement lors de son utilisation. ■ EAGER : la valeur est toujours chargée (valeur par défaut). Cette fonctionnalité permet de limiter la quantité de données obtenue par une requête
optionnal	Indique que la colonne est nullable

- Généralement, l'annotation `@Basic` peut être omise sauf dans le cas ou le chargement de la propriété doit être de type LAZY.

Exemple d'utilisation de l'annotation `@Basic`

```
import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch=FetchType.LAZY, optional=false)
  private String prenom;
  private String nom;
  ...
```

L'annotation `@Temporal`

- L'annotation `@javax.persistence.Temporal` permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (`@Date` et `@Calendar`) sont associées aux colonnes dans la table (date, time ou timestamp).
- La valeur par défaut est timestamp.

Exemple d'utilisation de l'annotation `@Temporal`

```
import java.io.Serializable;
import java.util.Date;
import javax.persistence.*;

@Entity public class Personne implements Serializable {
  @Id
  @GeneratedValue private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;
  private String nom;

  @Temporal(TemporalType.TIME) private Date heureNaissance;

  public Personne() { super(); }

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

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

  public Date getHeureNaissance() { return heureNaissance; }

  public void setTimeCreated(Date heureNaissance) {
    this.heureNaissance = heureNaissance;
  }
```


Beans entités (1/5)

- Ce sont eux aussi des POJOs mais leurs spécifications ont été définies à part car ils ont subi beaucoup de modifications dans la norme 3.0.
- L'annotation `@Entity (import javax.persistance.Entity;)` définit le bean comme étant de type entité.
- Le bean doit posséder au moins un constructeur par défaut et devra hériter de l'interface Serializable afin d'être utilisable à travers le réseau pour la gestion de la persistance.
- On peut spécifier deux méthodes différentes pour la gestion de la persistance au moyen de l'option *access* :
  - `@Entity(access=AccessType.FIELD)` permet d'accéder directement aux champs à rendre persistant.
  - `@Entity(access=AccessType.PROPERTY)` oblige le fournisseur à utiliser les accesseurs.

Beans entités (2/5)

- La clé primaire peut-être simple ou composée et doit être déclarée avec l'annotation `@Id`.
  - Par exemple, pour obtenir une clé qui s'incrémente automatiquement : `@Id(generate=GenerateType.AUTO)`.
- Pour les clés composées, il faut respecter certains principes :
  - La classe de la clé primaire doit être public et posséder un constructeur sans arguments.
  - Si l'accès est de type PROPERTY, la classe peut-être soit de type public soit de type protected.
  - La classe doit être sérialisable (implémenter `Serializable`).
  - Implémentation des méthodes `equals()` et `hashCode()`.
  - Si la clé primaire est mappée à de multiples champs ou propriétés, les noms des champs de cette clé doivent être les mêmes que ceux utilisés dans le bean entité.
- Les annotations permettent d'effectuer le mapping objet/relationnel et la gestion des relations entre les entités.

Beans entités (3/5)

- Lors de la création d'un bean entité, il faut effectuer le mapping de tous ses champs. Un mapping par défaut intervient lorsqu'aucune annotation précède le champ : `@Basic` spécifie ce comportement.
- `@Table` définit la table correspondant à la classe, elle prend en argumant le nom de la table.

```
@Entity(access=accessType.FIELD)
@Table(name="PAYS")
public class Pays implements Serializable {
  @Id(generate=GeneratorType.AUTO) private int id;
  @Basic private String nom;

  public Pays() {
  }

  public Pays(String nom) {
    this.nom = nom;
  }

  public int getId() {
    return id;
  }
}
```


Beans entités (4/5)

- On peut faire correspondre une valeur à un champ spécifique de la base de données en utilisant l'annotation `@Column` et des options comme `name` qui spécifie le nom de la colonne, ou des options pour définir si champ peut être nul, ...

```
@Column(name="DESC", nullable=false)
public String getDescription() {
  return description;
}
```

- Il existe les relations `OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany` (définies par les annotations correspondantes). Dans ces cas, il ne faut pas oublier de spécifier les colonnes faisant les jointures.

```
@ManyToOne(optional=false)
@JointColumn(name = "CLIENT_ID", nullable = false, updatable = false)
public Client getClient (){
  return client;
}
```


Beans entités (5/5)

- Les beans entités se manipulent par l'intermédiaire d'un `EntityManager`.
- Cet `EntityManager` peut être obtenu à partir d'un Bean Session par injection de dépendance.

```
@Stateless public class EmployeeManager{
  @Resource EntityManager em;

  public void updateEmployeeAddress (int employeeId, Address address) {
    //Recherche d'un bean
    Employee emp = (Employee)em.find ("Employee", employeId);
    emp.setAddress (address);
  }
}
```


Beans entités : exemple (1/12)

- Principe de base :
  - On crée un bean entité qui permettra de manipuler des objets persistants.
  - Dans le bean entité, on met en place un mapping entre les attributs du bean et une table de la base de données.
  - Le client n'accède pas directement aux beans entités mais passe par des beans sessions qui effectueront les manipulations nécessaires sur les beans entités.
- Les beans de l'exemple :
  - `ArticleBean` : il s'agit du bean entité.
  - `ArticleAccessBean` : il s'agit d'un bean session sans état. Il est composé de la classe `ArticleAccessBean.java` et de l'interface `ArticleAccess.java`.

Beans entités : exemple (2/12)

- L'interface `ArticleAccess.java` : définition des méthodes métiers.

```
package article;

import javax.ejb.Remote;
import java.util.*;

@Remote public interface ArticleAccess {
  public int addArticle (String libelle, double prixUnitaire, String categorie);
  public void delArticle (int idArticle);
  public InfosArticle rechercherArticle (int idArticle);
  public List rechercherLesArticlesParCategorie (String categorie);
  public List rechercherTousLesArticles ();
}
```


Beans entités : exemple (3/12)

- La classe `ArticleAccessBean.java` : la méthode `addArticle`

```
package article;

import javax.ejb.*;
import javax.persistence.*;
import java.util.*;

@Stateless public class ArticleAccessBean implements ArticleAccess {

  @PersistenceContext(unitName="Articles")
  EntityManager em;

  public int addArticle (String libelle, double prixUnitaire, String categorie){
    ArticleBean ab = new ArticleBean ();
    ab.setCategorie (categorie);
    ab.setLibelle (libelle);
    ab.setPrixUnitaire (prixUnitaire);
    em.persist(ab);
    em.flush();
    System.out.println ("AccessBean : Ajout de l'article "+ab.getIdArticle ());
    return ab.getIdArticle ();
  }
```


Beans entités : exemple (4/12)

- La classe `ArticleAccessBean.java` : la méthode `delArticle`

```
  public void delArticle (int idArticle){
        Query query = em.createQuery("DELETE FROM ArticleBean AS a WHERE a.idArticle="+idArticle);
  }

  public InfosArticle rechercherArticle (int idArticle){
    Query query = em.createQuery("SELECT a FROM ArticleBean AS a WHERE a.idArticle="+idArticle);

    List<ArticleBean> allArticles = query.getResultList();

    ArticleBean article = allArticles.get(0);
    return new InfosArticle (article.getIdArticle(), article.getLibelle(),
                             article.getPrixUnitaire(), article.getCategorie());
  }
```


Beans entités : exemple (5/12)

- La classe `ArticleAccessBean.java` : la méthode `rechercherTousLesArticles`

```
  public List rechercherTousLesArticles (){
        Query query = em.createQuery("SELECT a FROM ArticleBean AS a");
        List<ArticleBean> articlesbean = query.getResultList();
        Vector<InfosArticle> articles = new Vector();
        Iterator i = articlesbean.iterator();
        ArticleBean article;
        while (i.hasNext()) {
            article = (ArticleBean) i.next();
            InfosArticle infos = new InfosArticle (article.getIdArticle(), article.getLibelle(),
                                                   article.getPrixUnitaire(), article.getCategorie());
            articles.add(infos);
        }
        return articles;
   }
```

Beans entités : exemple (6/12)

- La classe `ArticleAccessBean.java` : la méthode `rechercherLesArticlesParCategorie`

```
  public List rechercherLesArticlesParCategorie (String categorie){
        Query query = em.createQuery("SELECT a FROM ArticleBean AS a WHERE categorie='"+categorie+"'");
        List<ArticleBean> articlesbean = query.getResultList();
        Vector<InfosArticle> articles = new Vector();
        Iterator i = articlesbean.iterator();
        ArticleBean article;
        while (i.hasNext()) {
            article = (ArticleBean) i.next();
            articles.add(new InfosArticle (article.getIdArticle(), article.getLibelle(),
                                           article.getPrixUnitaire(), article.getCategorie()));
        }
        return articles;
    }
}
```


Beans entités : exemple (7/12)

- Le fichier de gestion de la persistance : `persistence.xml`

```
<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_1_0.xsd"
   version="1.0">
   <persistence-unit name="Articles">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```


Beans entités : exemple (8/12)

- Le client `ArticleClient.java` : la méthode `creerArticle`

```
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;
import java.util.*;
import article.*;

public class ArticleClient {

  public void creerArticle(String libelle, double montantUnitaire, String categorie) {
        ArticleAccess ah;
        Properties props = System.getProperties();
        props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
        props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

        try {
            Context ctx = new InitialContext(props);
            ah = (ArticleAccess) ctx.lookup("ArticleAccessBean/remote");
            System.out.println ("Ajout d'un article : "+ah);
            int id = ah.addArticle(libelle, montantUnitaire, categorie);
            System.out.println ("Affichage de l'article "+id);
            afficherArticle(id);
        } catch (Throwable th) {
            System.out.println("Erreur dans creerArticle : " + th);
        }
    }
```


Beans entités : exemple (9/12)

- Le client `ArticleClient.java` : la méthode `afficherArticle`

```
    public void afficherArticle(int numeroArticle) {
        ArticleAccess ah;
        Properties props = new Properties();
        props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
        props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
        props.put("java.naming.provider.url", "localhost:1099");
        try {
            Context ic = new InitialContext(props);
            ah = (ArticleAccess) ic.lookup("ArticleAccessBean/remote");
            InfosArticle infos = ah.rechercherArticle(numeroArticle);
            System.out.println("voici les infos sur l'article : " + infos.idArticle);
            System.out.println("   id : " + infos.idArticle);
            System.out.println("   libelle : " + infos.libelle);
            System.out.println("   prix unitaire : " + infos.prixUnitaire);
            System.out.println("   categorie : " + infos.categorie);
        } catch (Throwable th) {
            System.out.println("GereCommande.creerArticle : " + th);
        }
    }
```

Beans entités : exemple (10/12)

- Le client `ArticleClient.java` : la méthode `afficherArticlesParCategorie`

```
public void afficherArticlesParCategorie(String categorie) {

     ArticleAccess ah;
     Properties props = new Properties();
     props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
     props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
     props.put("java.naming.provider.url", "localhost:1099");
     try {
         Context ic = new InitialContext(props);
         ah  = (ArticleAccess) ic.lookup("ArticleAccessBean/remote");
         List<InfosArticle> articles = ah.rechercherLesArticlesParCategorie(categorie);
         Iterator i = articles.iterator();
         InfosArticle article;
         while (i.hasNext()) {
             article = (InfosArticle) i.next();
             afficherArticle(article.idArticle);
         }
     } catch (Throwable th) {
         System.out.println("Erreur dans rechercherArticlesParCategorie : " + th);
     }
 }
```


Beans entités : exemple (11/12)

- Le client `ArticleClient.java` : la méthode `afficherTousLesArticles`

```
public void afficherTousLesArticles() {

     ArticleAccess ah;
     Properties props = new Properties();
     props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
     props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
     props.put("java.naming.provider.url", "localhost:1099");
     try {
         Context ic = new InitialContext(props);
         ah  = (ArticleAccess) ic.lookup("ArticleAccessBean/remote");
         List<InfosArticle> articles = ah.rechercherTousLesArticles();
         Iterator i = articles.iterator();
         InfosArticle article;
         while (i.hasNext()) {
             article = (InfosArticle) i.next();
             afficherArticle(article.idArticle);
         }
     } catch (Throwable th) {
         System.out.println("Erreur dans rechercherArticlesParCategorie : " + th);
     }
 }
```


Beans entités : exemple (12/12)

- Le client `ArticleClient.java` : la méthode `main`

```
   public static void main(java.lang.String[] args) {

       ArticleClient a = new ArticleClient ();

       a.creerArticle("Les miserables", 21, "LIVRE");
       a.creerArticle("Celine Dion au stade de France", 120, "CD");
       a.creerArticle("Je l'aime a mourir", 28, "LIVRE");
       a.creerArticle("La mer", 38, "LIVRE");

       // Recherche de l'article 3
       System.out.println ("================================");
       a.afficherArticle(3);
       System.out.println ("================================");
       a.afficherTousLesArticles();
       System.out.println ("================================");

       // Recherche de la categorie
       a.afficherArticlesParCategorie("CD");
       System.out.println ("================================");

   }
}
```


Connexion avec une base de données MySQL (1/2)

- Créer une base de données (Exemple : JBossDB)
- Créer un fichier `mysql-ds.xml` (Le nom importe peu !)
- Copier ce fichier dans le répertoire de déploiement `$JBOSS_HOME$/server/default/deploy`.
- Télécharger le connecteur JDBC pour MySQL (fichier JAR) et copier dans le répertoire `$JBOSS_HOME$/server/default/lib`.

Connexion avec une base de données MySQL (2/2)

- Le fichier `mysql-ds.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
    <local-tx-datasource>
        <jndi-name>MySQLDS</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/JBossDB</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>duvallet</user-name>
        <password>duvallet</password>
        <exception-sorter-class-name>
          org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
        </exception-sorter-class-name>
        <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional) -->
        <metadata>
            <type-mapping>mySQL</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```


Les relations Un à Un (One To One)

- Une relation « One To One » est utilisée pour lier deux entités uniques indissociables.
- Ce type de relation peux-être *mappé* de trois manières dans la base de données :
  - en utilisant les mêmes valeurs pour les clés primaires des deux entités via l'annotation `@PrimaryKeyJoinColumn`.
  - en utilisant une clé étrangère d'un coté de la relation via l'annotation `@JoinColumn (name="identifiant", referencedColumnName="clé primaire dans l'autre table")`.
  - en utilisant une table d'association de liens entre les deux entités via l'annotation `@JoinTable(name = "NomDeLaTableDeJointure", joinColumns = @JoinColumn (name="1ère clé étrangère", unique=true), inverseJoinColumns = @JoinColumns(name="2ième clé étrangère", unique=true)`

Les relations Un à Plusieurs et Plusieurs à Un

Les relations Un à Plusieurs (One To Many) et Plusieurs à Un (Many To One)

- Ces deux relations sont utilisées pour lier à une unique instance d'une entité A, un groupe d'instances d'une entité B.
- L'association « Many To One » est définie par l'annotation `@ManyToOne`
- Dans le cas d'une relation bidirectionnelle, l'autre coté doit utiliser l'annotation `@OneToMany`.
- Les attributs de ces deux annotations correspondent à ceux de l'annotation `@OneToOne`.

Les relations Plusieurs à Plusieurs (Many To Many)

- Ces relations sont utilisées pour lier des instances de deux entités entre elles.
- Exemple : un article appartient à éventuellement plusieurs catégories et une catégorie contient plusieurs articles.
- Il est nécessaire d'utiliser une table d'association pour mettre en œuvre cette association.

Le langage EJB-QL

- EJB-QL (Entreprise JavaBean Query Language) est une spécification de langage de requêtes.
- C'est un langage qui est complètement portable et il peut donc être utilisé à l'identique quelque soit les implémentations du langage SQL présentes dans les SGBD.
- Il est traduit en lanage SQL lors de son exécution.
- Il permet d'utiliser directement les objets de type Entity Bean dans les requêtes.
- Ce langage a évolué entre la version 2 et la version 3 des EJB.

EJB-QL : un premier exemple

```
String requete = "SELECT utilisateur FROM Utilisateur AS utilisateur";
Query query = entityManager.createQuery (requete);
List<Utilisateur> tousLesUtilisateurs = query.resultList();

For (Utilisateur u : tousLesUtilisateurs) {
  // Traitements sur l'utilisateur
}
```

- On a un bean Entité nommé Utilisateur.
- Dans cet exemple, nous utilisons le langage EJB-QL, l'EntityManager et l'API Query pour :
  - récupérer la liste de tous les utilisateurs stockées dans la base de données,
  - stocker la liste dans une collection,
  - parcourir la liste et examiner chaque élément récupérer.

Les requêtes EJB-QL 3

- Des similarités avec les requêtes SQL.
- Elles permettent d'effectuer des selection (SELECT), des modifications (UPDATE) ou encore des suppressions (DELETE).
- Une requête EJB-QL contient les clauses suivantes :
  - SELECT : liste les Entity Beans et les propriétés retournées par la requête.
  - FROM : définit les Entity Beans utilisés. Ceux-ci doivent être déclarés via l'expression AS.
  - WHERE : permet d'appliquer des critères de recherche en utilisant les types JAVA ayant des équivalents en base de données ou encore des Entity Beans.
- L'opérateur « . » sert à naviguer entre les propriétés et les relations des Entity Beans.
- Exemple :

```
SELECT utilisateur FROM Utilisateur AS utilisateur WHERE utilisateur.id=5
```


Les requêtes SELECT en EJB-QL 3

- Il s'agit des requêtes les plus utilisées.
- Elles sont très similaires aux requêtes SQL bien qu'elles soient orientés objet.
- Elles permettent de manipuler des Beans Entités.
- On peut récupérer des objets de type "Entity Bean" ou encore uniquement quelques propriétés.
  - Exemple :

```
SELECT utilisateur.nom, utilisateur.prenom
FROM Utilisateur AS utilisateur
```

Les requêtes DELETE et UPDATE en EJB-QL 3

- Il s'agit des requêtes fournissant un ensemble d'opérations permetant de modifier ou supprimer un Bean Entité.
- Elles ne peuvent avoir qu'une seule entité dans la clause FROM.
- Pour modifier ou supprimer des enregistrements de plusieurs Beans Entités, il faut effectuer plusieurs requêtes.
- Un premier exemple avec DELETE :

```
DELETE FROM Utilisateur AS utilisateur
WHERE utilisateur.id=3
```

- Un second exemple avec UPDATE :

```
UPDATE FROM Utilisateur AS utilisateur
SET utilisateur.prenom="Claude"
WHERE utilisateur.id=1
```


Les suppressions en cascade

- La clause DELETE d'EJB-QL ne supporte pas la suppression en cascade même si la relation a été configurée avec `CascadeType.REMOVE` ou `CascadeType.ALL`.
- Il faut écrire manuellement les retraits en cascade de la base de données.
- Lorsqu'il existe des relations, le risque de soulèvement d'exception existe.
- Il existe plusieurs solutions :
  - Utiliser le gestionnaire de persistance qui applique la suppression en cascade.
  - Écrire explicitement toutes les requêtes EJB-QL de suppression dans le bon ordre.
  - Utiliser les possibilités de la base de données et gérer la suppression en cascade au niveau de celle-ci.

La clause WHERE en EJB-QL 3 (1/2)

- Les opérateurs incluses dans la clause WHERE existent pour la plupart dans une forme similaire à celle utilisées en SQL.
- Voici quelques uns de ces opérateurs :
  - BETWEEN : opérateur conditionnel permettant de restreindre les résultats suivant un intervalle.

```
SELECT utilisateur FROM Utilisateur as utilisateur
WHERE utilisateur.id BETWEEN 10 AND 15
```

  - LIKE : permet de comparer la valeur d'un champ suivant un motif spécifié.

```
SELECT utilisateur FROM Utilisateur AS utilisateur
WHERE utilisateur.prenom LIKE 'Claude%'
```

  - IN : teste une appartenance à une liste de chaînes de caractères.

```
SELECT utilisateur FROM Utilisateur AS utilisateur
WHERE utilisateur.adresse.pays IN ('France', 'Espagne', 'Belgique')
```


La clause WHERE en EJB-QL 3 (2/2)

- Suite des opérateurs :
  - IS NULL : teste si une valeur est nulle.

```
SELECT utilisateur FROM Utilisateur AS utilisateur
WHERE utilisateur.login IS NULL
```

  - MEMBER OF : teste l'appartenance d'une instance à une collection.

```
SELECT utilisateur FROM Utilisateur AS utilisateur
WHERE ?1 MEMBER OF utilisateur.entrepreneurs
```

  - IS EMPTY : teste si une collection est vide.

```
SELECT utilisateur FROM Utilisateur AS utilisateur
WHERE utilisateur.entrepreneurs IS EMPTY
```

  - NOT : inverse le résultat de la condition. Il est possible de l'utiliser avec les précédents opérateurs (NOT BETWEEN, IS NOT NULL, ...).

```
SELECT utilisateur FROM Utilisateur AS utilisateur
WHERE utilisateur.entrepreneurs IS NOT EMPTY
```

La clause ORDER BY en EJB-QL 3

- Cette clause permet de trier les résultats à partir d'un ou plusieurs champs en utilisant l'ordre alphanumérique.
- Un premier exemple :

```
SELECT utilisateur FROM Utilisateur AS utilisateur
ORDER BY utilisateur.nom
```

- Il est possible de trier de façon ascendante (mot clef ASC) ou descendante (mot clef DESC).
- On peut combiner plusieurs critères :

```
SELECT utilisateur FROM Utilisateur AS utilisateur
ORDER BY utilisateur.nom, utilisateur.prenom DESC
```


Les fonctions d'agrégation

- Elles permettent d'effectuer des opérations sur des ensembles.
- On les retrouve en EJB-QL via les instructions suivantes :
  - `avg()` : retourne une moyenne par groupe.
  - `count()` : retourne le nombre d'enregistrement.
  - `max()` : retourne la valeur la plus élevée.
  - `min()` : retourne la valeur la plus basse.
  - `sum()` : retourne la somme des valeurs.
- Un petit exemple avec l'instruction `count()` :

```
SELECT COUNT(utilisateur)
FROM Utilisateur AS utilisateur
```


Les jointures

- Elles permettent de manipuler les relations existantes entre plusieurs entité.
- Un premier exemple permettant de récupérer des informations sur un utilisateur, ces informations étant stockées dans une autre table :

```
SELECT utilisateur.nom, compte.login
FROM Utilisateur AS utilisateur, CompteUtilisateur compte
WHERE utilisateur.id = compte.id
```

- La même chose en utilisant les propriétés relationnelles :

```
SELECT utilisateur.nom, utilisateur.compteUtilisageur.login
FROM Utilisateur AS utilisateur
```

- Et en utilisant l'instruction `JOIN` :

```
SELECT utilisateur.nom, compte.login
FROM Utilisateur AS utilisateur
JOIN utilisateur.compteUtilisateur as compte
```


Les transactions dans Java EE

- Une transaction est définie par un début et une fin.
- Une transaction est démarquée, généralement, par l'appel de la méthode `begin()` et par l'appel des méthodes `commit()` ou `rollback()` pour mettre fin à celle-ci.
- Pour les transactions locales, le gestionnaire de ressources se charge de gérer implicitement les transactions.
- Pour les transactions globales, il est nécessaire d'avoir recours aux API JTA/JTS :
  - JTS permet permet la démarcation, la connexion aux ressources, la synchronisation, et la propagation du contexte transactionnel.
  - JTA permet de se connecter à différents composants d'accès aux données au sein d'une même transaction.
- Le conteneur EJB propose un service sur lequel toute la gestion des transaction peut-être définie au sein des EJB.
- Il n'est donc pas obligatoire de paramétrer la gestion des transactions.

Les transactions gérées par le conteneur (1/2)

- Avec les EJB 3, la configuration des transactions est vraiment facilitée.
- Pour définir une méthode transactionnelle, on utilise l'annotation `@TransactionAttribute`.
- Cette annotation prends en paramètre le type de la transaction :
  - MANDATORY : l'exécution est effectué dans un contexte existant.
  - NEVER : la transaction est exécuté uniquement si aucun contexte transaction n'existe.
  - NOT_SUPPORTED : la transaction peut s'exécuter en dehors d'un contexte transactionnel.
  - REQUIRED : l'exécution se faite soit dans un contexte transactionnel existant soit il est créée.
  - REQUIRES_NEW : un nouveau contexte transactionnel est crée pour l'exécution de la transaction.
  - SUPPORTS : la transaction est exécuté dans le contexte existant si et seulement si l'appelant est associé à une transaction.

Les transactions gérées par le conteneur (2/2)

- Exemple :

```
@Stateless public class MonBeanSession implements MonInterface {
  ...
  @TransactionAttribute(TransactionAttributeType.SUPPORTS)
  public void maMethodeTansactionnelle(...){
    java.sql.Connection con1 , con2;
    java.sql.Statement stmt1, stmt2;
    //Récupération des connexions
    con1 = ...;  con2 = ...;
    stmt1 = con1.createStatement();
    stmt2 = con2.createStatement();
    // Execution des requêtes
    stmt1.executeQuery(...);
    stmt1.executeUpdate(...);
    stmt2.executeQuery(...);
    stmt2.executeUpdate(...);
    stmt1.executeUpdate(...);
    stmt2.executeUpdate(...);
    //Fermeture des connexions
    con1.close(); con2.close();
  }
}
```

- Pas besoin d'indiquer les démarcations car elles seront appliquées par le conteneur à EJB à cause des annotations.

Les transactions gérées par le Bean (1/4)

- Cette solution ne doit être utilisée que si le mode de gestion par le conteneur n'est pas possible.
- Ce mode permet une démarcation beaucoup plus fine au sein de la méthode.
- La transaction peut-être initialisée et gérée directement le Bean (Session ou MessageDriven uniquement).
- Le principe consiste à récupérer un objet de type `javax.transaction.UserTransaction` puis à appeler la méthode `begin()` pour initailiser la transaction.
- Pour terminer la transaction, il faut utiliser soit la méthode `commit()` ou soit la méthode `rollback()`.

Les transactions gérées par le Bean (2/4)

- Les principales méthodes de l'interface `UserTransaction` sont :
  - `void begin()` : crée une nouvelle transaction et l'assigne au processus courant.
  - `void commit()` : valide la transaction assignée au processus courant.
  - `int getStatus()` : retourne le statut de la transaction courante.
  - `void rollback()` : annule la transaction courante.
  - `void setRollbackOnly()` : modifie la transaction courante afin d'en signifier l'annulation. La transaction ne pourra plus être validée.
  - `void setTransactionTimeout(int secondes)` : modifie la valeur du temps maximum d'exécution de la transaction courante depuis le `begin()`.

Les transactions gérées par le Bean (3/4)

- En EJB 3, l'accès aux ressources est simplifié.
- Pour récupérer un objet `UserTransaction`, il suffit d'utiliser l'annotation `@Resource` sur la propriété du Bean (variable d'instance).
- Il est nécessaire de préciser que la gestion des transactions et de type « Bean Managed ».
- Cela se fait aussi grâce à une annotation (`@TransactionManagement`) placé au niveau de la classe du Bean.

Les transactions gérées par le Bean (4/4) : Exemple

```
@Statefull @TransactionManagement(BEAN)
public class MonBeanSession implements MonInterface {
  @Resource javax.Transaction.UserTransaction ut
  @Resource javax.sql.DataSource bd1;
  public void maMethode1(...){
    //Début de la transaction
    ut.begin();
  }
  public void maMethode2(...){
    java.sql.Connection con;
    java.sql.Statement stmt;
    //Ouverture de la connexion
    con = db.getConnection;
    stmt = con.createStatement();
    // Execution des requêtes
    stmt.executeQuery(...);
    stmt.executeUpdate(...);
    //Fermeture des connexions
    stmt.close();
    con.close();
  }
  public void maMethode3(...){
    //Validation de la transaction
    ut.commit();
  }
}
```