# Hibernate

Site de référence : www.hibernate.org
Hibernate est un produit Open source sous licence GNU LGPL, développé par une équipe issue de la communauté JBOSS, aujourd'hui filiale de la société Red Hat.

**Principale fonctionnalité :**
Le rôle principal d'Hibernate est de remplacer l'accès aux bases de données par l'appel de méthodes objet de haut niveau.
Hibernate 3, version avec laquelle nous allons travailler, est capable de gérer la persistance avec des bases de données relationnelles, mais aussi avec des bases de données objet et des fichiers XML.

Il existe également une version d'Hibernate pour .net : NHibernate.

## Configuration logicielle

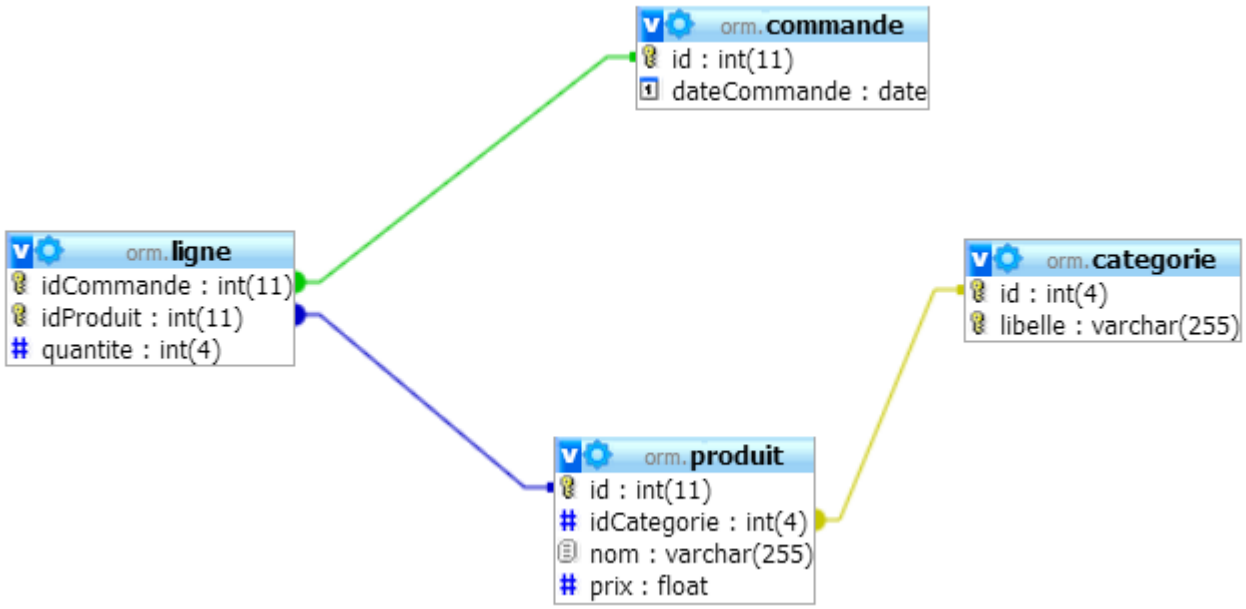
Vous disposez de :

- Eclipse Juno J2EE
- Hibernate 3
- Mysql Server
- Driver JDBC pour Mysql

## Contexte

Nous allons travailler à partir d'un cas simple, et assez couramment utilisé :

- Un SI composé de produits, classés en catégories (1 CIF).
- Des commandes de produits effectuées, dont le détail est stocké dans des lignes (1 CIM).



# Mise en place

Mise en place la configuration logicielle.

## Dans Eclipse

- Créer un nouveau **Dynamic Web Project** dans Eclipse
- Intégrer les jars d'Hibernate 3 et le driver JDBC pour mysql dans le dossier **WebContent/WEB-INF/lib**
- Copier le fichier xml de configuration d'Hibernate dans le dossier src du projet.

## Dans phpMyAdmin

- Créer la base de données **ORM** sur votre serveur Mysql en exécutant le script de création (la base est créée dans le script).

Afficher le concepteur pour visualiser les tables, et les relations : Pour chaque table, notez les contraintes d'intégrité :

1. d'entité (clé primaire)
2. référentielle (relations)

**Exemple :**

**Produit :**

- **id** (primary key)
- **idCategorie** (foreign key references **categorie.id**)

## Hibernate

Ouvir le fichier de configuration d'Hibernate dans le dossier src :

Vérifiez les paramètres de connexion à Mysql.

| Propriété | Valeur | Signification |
|---|---|---|
| hbm2ddl.auto | validate | Permet de vérifier la correspondance entre le schéma de la base et les classes métiers |
| show_sql | true | Permet d'afficher les instructions SQL exécutées dans la console Eclipse |

Les lignes suivantes vont permettre d'assurer la persistance des classes que nous allons créer : **Categorie** et **Produit**.

```
    <mapping class="metier.Categorie" />
    <mapping class="metier.Produit" />
```

hibernate.cfg.xml

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

<hibernate-configuration>
```

```
    <session-factory >
        <property name="connection.url">jdbc:mysql://localhost/orm</property>
        <property name="connection.username">root</property>
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="connection.password"></property>
        <property name="connection.pool_size">10</property>
        <property name="current_session_context_class">thread</property>
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <property name="hbm2ddl.auto">validate</property>
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

    <mapping class="metier.Categorie" />
    <mapping class="metier.Produit" />

    </session-factory>
</hibernate-configuration>
```

## Création d'une classe de lancement de session Hibernate

Il est maintenant nécessaire de créer une classe Java permettant de piloter une session Hibernate. Cette classe n'ayant besoin d'être instanciée qu'une seule fois pour ensuite nous permettre d'effectuer le mapping entre classes et base de données, nous utiliserons une classe statique, ou un singleton. (c'est une pratique recommandée par la communauté JBoss).

Créer la classe **HibernateUtil** dans le package **hibernate**

|h HibernateUtil.java

```
package hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory =
buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new
AnnotationConfiguration().configure().buildSessionFactory();

        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." +
ex);
```

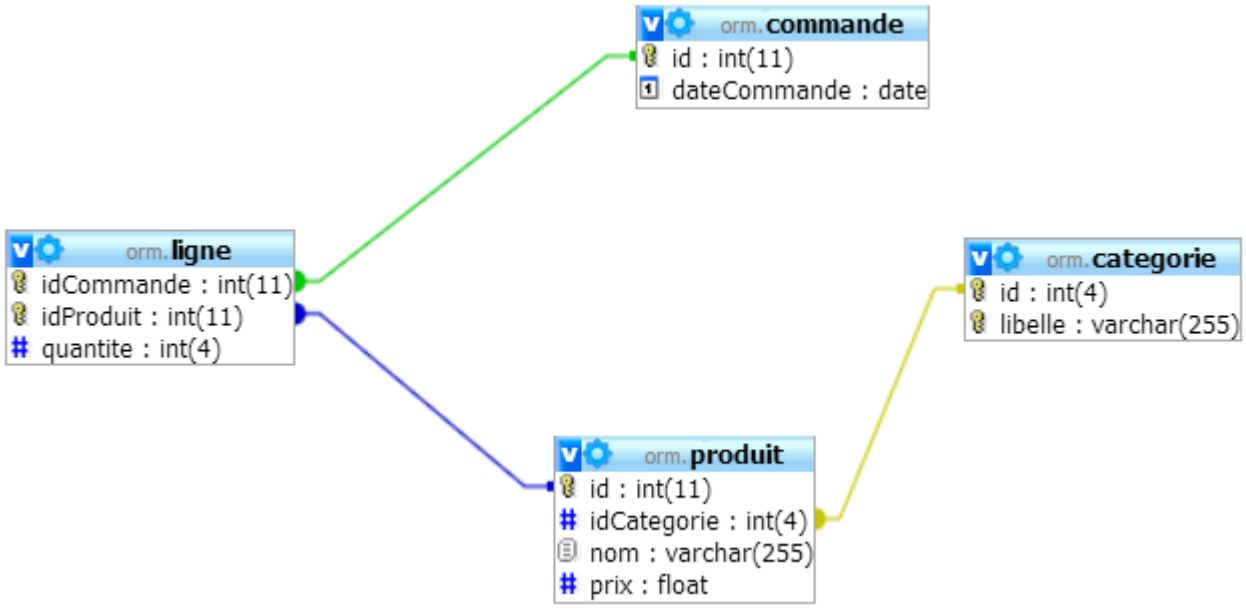
```
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

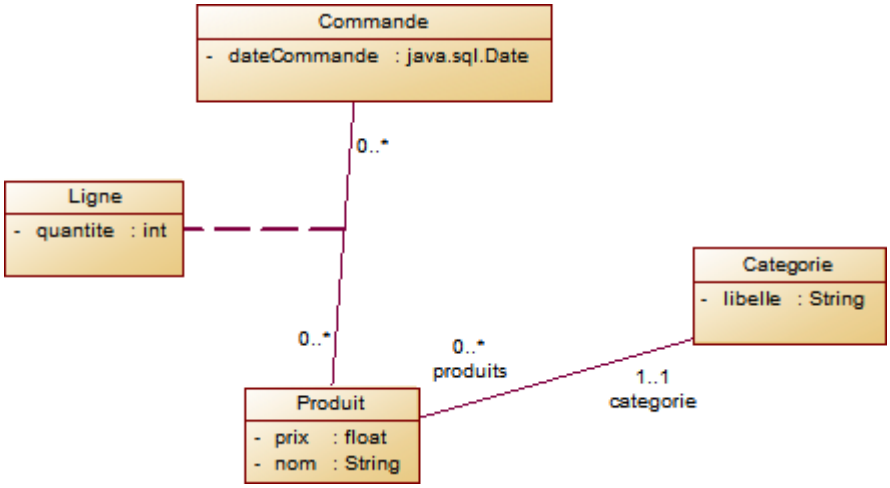
    public static void shutdown() {
        // Close caches and connection pools
        getSessionFactory().close();
    }
    public Session getSession(){
        return getSessionFactory().openSession();
    }
}
```

## Modèle relationnel et modèle objet

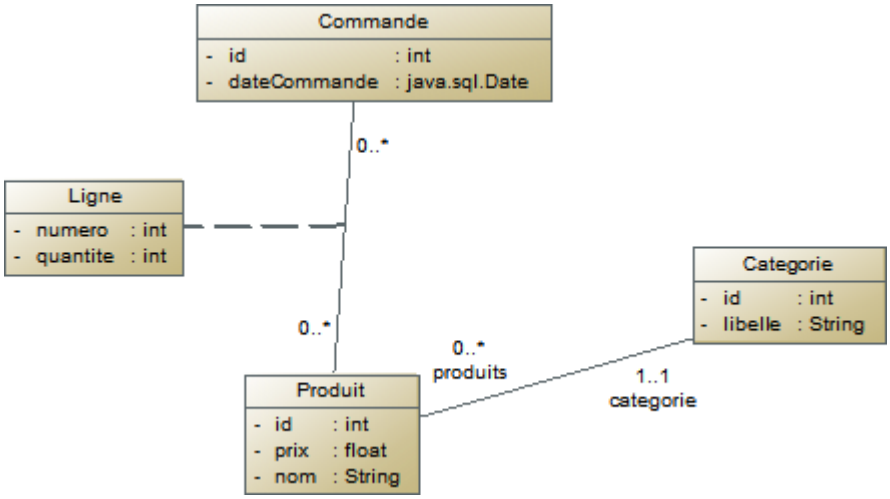
**Modèle relationnel :**




**Modèle Objet (diagramme de classes) correspondant :**




Nous allons modifier le modèle Objet pour le rendre compatible avec Hibernate et permettre le mapping, en ajoutant des membres qui serviront d'identifiants (habituellement inutiles dans le monde Objet) :




# Création des classes métier

La Java persistence API (JPA) sera utilisée pour définir le mapping relationnel/objet. Elle va permettre de mettre des annotations sur les classes métier de façon à définir leur persistance.

## Produits et catégories

Ci-dessous le début de l'implémentation des classes :

Categorie.java

```java
package metier;

import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="Categorie")
public class Categorie {
    @Id
    @Column(name="id")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    @Column(name="libelle")
    private String libelle;

    @OneToMany(mappedBy="categorie")
    private List<Produit> produits;

    public Categorie() {
    }
```

Produit.java

```java
package metier;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;


@Entity
@Table(name="Produit")
public class Produit {
    @Id
    @Column(name="id")
    private int id;

    @Column(name="nom")
    private String nom;

    @Column(name="prix")
    private float prix;

    @ManyToOne
    @JoinColumn(name="idCategorie")
    private Categorie categorie;

    public Produit() {
    }
```

Les principes de construction des classes métiers à mapper avec **hibernate** sont les suivants :

- A chaque champ de la base correspond un membre de données de la classe
- les types de données java et sql doivent être compatibles (consulter la documentation hibernate pour voir la correspondance entre les types)
- La classe doit être un **java Bean** pour permettre à Hibernate de travailler :
  - elle doit disposer d'un constructeur sans paramètre
  - Chacun de ses membres doit disposer d'accesseurs
  - Le mapping avec la base de données est défini au travers des annotations JPA posées sur les membres de la classe.

- Implémenter les classes **Categorie** et **Produit** dans le package **metier**
- Utiliser la complétion pour faire les imports
- Générer les accesseurs, le constructeur sans paramètre
- Ajouter un constructeur initialisant les membres de données
- Ajouter un toString affichant les champs proprement concaténés

A partir de l'observation de cette première implémentation et en utilisant à bon escient la documentation, répondez aux questions suivantes :

1. Comment est déclarée la table assurant la persistance d'un objet ?
2. Comment est déclaré le mapping entre un membre de la classe et un champ de la table relationnelle ?
3. Comment est déclarée la clé primaire de la table ?
4. Quelles sont les possibilités de déclaration des clés primaires ?

5. Réaliser un tableau montrant la correspondance de type (entier, chaine, etc.) entre les propriétés d'une classe et les champs d'une table.
6. Montrez à l'aide d'un schéma (par ex. deux classes liées au dessus de deux tables liées) comment se paramètre le lien bidirectionnel entre deux classes (en spécifiant les éléments à fournir dans les annotations)

#### Programme de test

- Implémenter le programme suivant dans un package nommé **console**, rendez le exécutable.
- Exécutez le code puis observez la base de données.

TestAjoutProduitCategorie.java

```
1  package console;
2
3  import metier.Categorie;
10
11 public class TestAjoutProduitCategorie {
12     public static void main(String[] args){
13         Session session= HibernateUtil.getSession();
14         Transaction trans= session.beginTransaction();
15
16         Categorie aCategorie=new Categorie("Presse");
17         session.persist(aCategorie);
18
19         Produit aProduit=new Produit("Programmez!", 3.0f, aCategorie);
20         session.persist(aProduit);
21
22         trans.commit();
23         session.close();
24     }
25 }
26
```

Analysez le code du programme et répondez aux questions en vous aidant au besoin de la documentation :

1. À quoi correspond la méthode persist() ?
2. À quoi correspond la méthode commit () ?
3. Comment ont été traduits les liens objet entre le membre **categorie** et **produits** entre ces classes dans les tables de la base ?
4. Quelles requêtes SQL ont été créées par Hibernate pour réaliser la persistance ?
5. Pourquoi comportent t-elles des points d'interrogation ?

## Chargement d'un objet

Observation du chargement d'un objet, par l'intermédiaire de sa clé primaire.

#### Programme de chargement d'un produit

- Implémenter le programme suivant dans le package nommé **console**, rendez le exécutable.

- Exécutez le code et regardez le résultat.
- Mettez le point d'arrêt indiqué et inspectez l'objet aProduit, et son membre catégorie .

TestLoadProduit.java

```
1  package console;
2
3  import hibernate.HibernateUtil;
7
8  public class TestLoadProduit {
9
10     public static void main(String[] args) {
11         Session session= HibernateUtil.getSession();
12
13         Produit aProduit=(Produit) session.get(Produit.class, 50);
14         System.out.println(aProduit.getNom());
15         System.out.println(aProduit.getCategorie().getLibelle());
16         session.close();
17     }
18
19 }
20
```

**Programme de chargement d'une catégorie**

- Implémenter le programme suivant dans le package nommé **console**, rendez le exécutable.
- Exécutez le code et regardez le résultat.
- Mettez le point d'arrêt indiqué et inspectez l'objet aCategorie, et son membre produits.
- Exécutez en pas à pas (step over) et inspectez toujours l'objet aCategorie, et son membre produits.

TestLoadCategorie.java

```
1  package console;
2
3  import hibernate.HibernateUtil;
7
8  public class TestLoadCategorie {
9
10     public static void main(String[] args) {
11         Session session= HibernateUtil.getSession();
12
13         Categorie aCategorie=(Categorie) session.get(Categorie.class, 3);
14         System.out.println(aCategorie.getLibelle());
15         System.out.println(aCategorie.getProduits().get(0).getNom());
16         System.out.println(aCategorie.getProduits().get(3).getNom());
17         session.close();
18     }
19
20 }
21
```

A partir de ses 2 programmes et de leur exécution :

1. Précisez ce que charge exactement Hibernate lors du chargement d'un Objet
2. Précisez comment sont chargés les instances liées à un objet chargé pour les liens **onToMany** et **manyToOne**

3. En quoi consiste le chargement paresseux d'Hibernate et la qualification **lazy** (rechercher dans l'aide)

# Chargement de listes d'objets

Interrogation de données avec Hibernate :

## Projection

Créer le programme suivant

TestProjection.java

```
1  package console;
2
3  import java.util.List;
4
5  import hibernate.HibernateUtil;
6
7  import metier.Categorie;
8
9  import org.hibernate.Query;
10 import org.hibernate.Session;
11
12 public class TestProjection {
13
14     /**
15      * @param args
16      */
17     @SuppressWarnings("unchecked")
18     public static void main(String[] args) {
19         Session session= HibernateUtil.getSession();
20         Query query=session.createQuery("from Categorie");
21         List<Categorie> categories=query.list();
22         for(Categorie categorie:categories)
23             System.out.println(categorie.getLibelle());
24     }
25
26 }
27
```

A partir de ce programme :

1. Interprétez la forme de la requête passée à la méthode **createQuery**, pourquoi n'est-elle pas complète ?

Modifiez la méthode toString de la classe Categorie :

```
    @Override
    public String toString() {
        return "Categorie [id=" + id + ", libelle=" + libelle + ", produits="
                + produits + "]";
    }
```

Modifiez le programme pour qu'il utilise cette méthode :

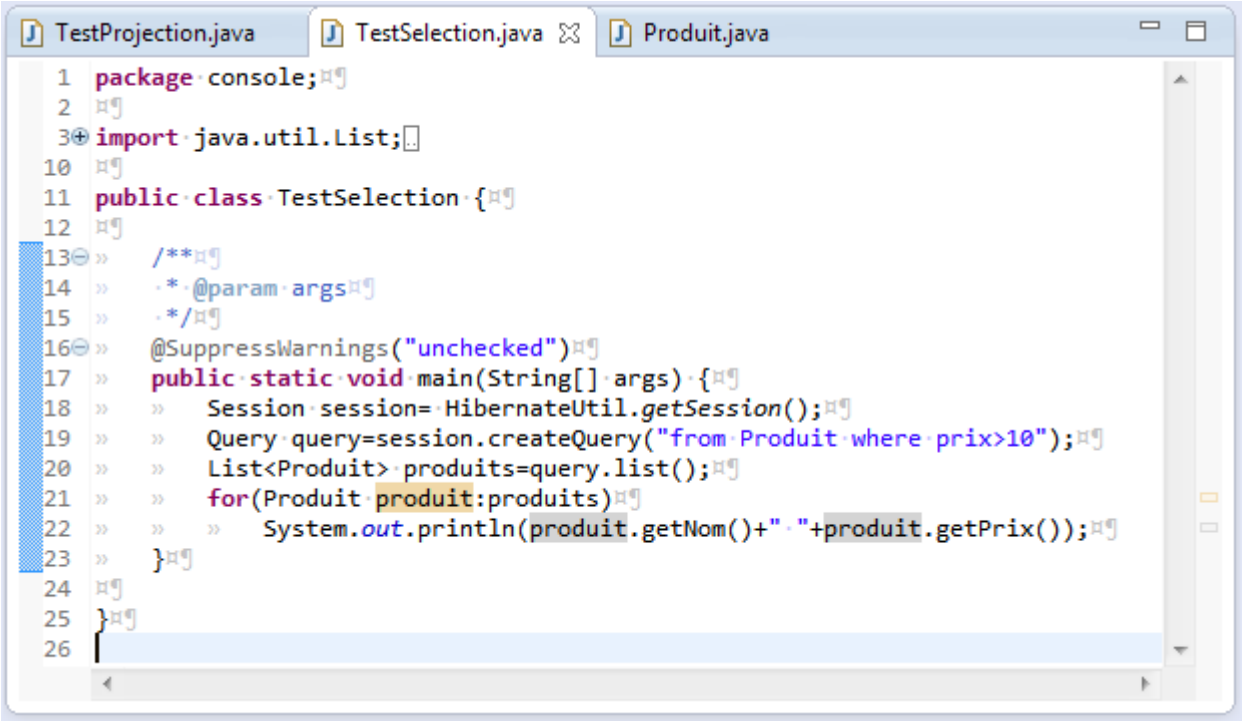
```
        for(Categorie categorie:categories)
            System.out.println(categorie);
```

A partir de l'exécution du programme modifié :

1. Interprétez et expliquez le résultat obtenu

## Sélection

Créer le programme suivant




A partir du programme :

1. Interprétez les requêtes SQL exécutées par Hibernate

Créer le programme suivant

TestProjection.java | TestSelection.java | Produit.java | Categorie.java

```
1  package console;
2
3  import java.util.List;
10
11 public class TestSelection {
12
13     /**
14      * @param args
15      */
16     @SuppressWarnings("unchecked")
17     public static void main(String[] args) {
18         Session session= HibernateUtil.getSession();
19         Query query=session.createQuery("from Produit prod where prod.categorie.libelle='beauté'");
20         List<Produit> produits=query.list();
21         for(Produit produit:produits)
22             System.out.println(produit.getNom()+" "+produit.getCategorie());
23     }
24
25 }
26
```

A partir du programme :

1. Interprétez les requêtes SQL exécutées par Hibernate

### Sélection avec distinct et projection

Créer le programme suivant

TestProjection.java | TestSelection.java | Produit.java | Categorie.java | TestDistinctSelection.java

```
1  package console;
2
3  import java.util.List;
10
11 public class TestDistinctSelection {
12
13     /**
14      * @param args
15      */
16     @SuppressWarnings("unchecked")
17     public static void main(String[] args) {
18         Session session= HibernateUtil.getSession();
19         Query query=session.createQuery("select distinct prod.categorie from Produit prod where prod.prix>10");
20         List<Categorie> categories=query.list();
21         for(Categorie categorie:categories)
22             System.out.println(categorie.getLibelle());
23     }
24
25 }
26
```

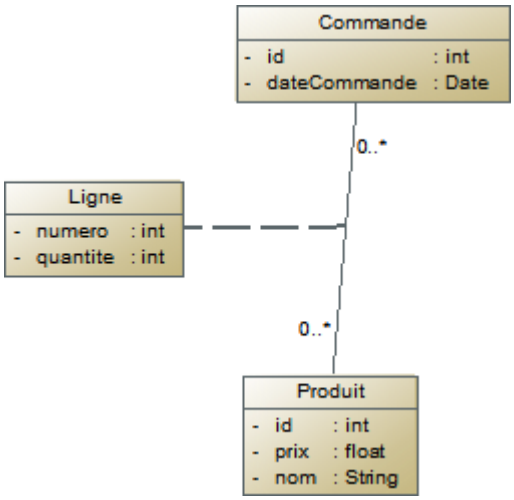
A partir du programme :

1. Expliquer ce que fait le programme
2. Interprétez la requêtes SQL exécutée par Hibernate : quel est le service rendu par l'ORM dans ce cas ?

## Gestion des commandes

Implémenter les classes métier Commande et Ligne, en utilisant le début de leur implémentation donné ci dessous, et le diagramme de classe :

- Ne pas oublier les règles citées précédemment (bean + réf dans le fichier de configuration + annotations JPA)
- Ajouter un constructeur avec paramètres permettant d'instancier correctement une ligne et une commande




## Commande

Commande.java

```
3  import java.util.ArrayList;
15 
16 @Entity
17 @Table(name="Commande")
18 public class Commande {
19     @Id
20     @Column(name="id")
21     @GeneratedValue(strategy=GenerationType.IDENTITY)
22     private int id;
23 
24     @Column(name="dateCommande",insertable=false,updatable=false)
25     private Date dateCommande;
26 
27     @OneToMany(mappedBy="commande",cascade=CascadeType.PERSIST)
28     private List<Ligne> lignes;
29 
30     public Commande() {
31         lignes=new ArrayList<>();
32     }
```

## Ligne

Ligne.java

```java
package metier;
import javax.persistence.Column;
@Entity
@Table(name="Ligne")
public class Ligne {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="numero")
    private int numero;

    @Column(name="quantite")
    private int quantite;

    @ManyToOne
    @JoinColumn(name="idCommande")
    private Commande commande;

    @ManyToOne
    @JoinColumn(name="idProduit")
    private Produit produit;

```

1. Justifiez les annotations permettant de mettre en oeuvre la contrainte d'intégrité multiple
2. Interprétez l'annotation sur le membre dateCommande, en allant voir comment ce champ est défini dans la base de données

## Création de commandes

Implémenter le programme suivant.
Exécutez le.

CreateCommande.java

```java
package console;
import java.util.List;
public class CreateCommande {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Session session= HibernateUtil.getSession();
        Transaction trans= session.beginTransaction();

        Produit produit;
        Commande cmd=new Commande();
        Ligne ligne;
        Query query=session.createQuery("from Produit");
        List<Produit> produits=query.list();
        produit=produits.get(3);
        ligne=new Ligne(3, cmd, produit);
        cmd.getLignes().add(ligne);
        session.persist(ligne);

        produit=produits.get(6);
        ligne=new Ligne(10, cmd, produit);
        cmd.getLignes().add(ligne);
        session.persist(ligne);
        session.persist(cmd);

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

}
```

1. Analysez puis commentez chaque ligne (dans le code) de ce programme
2. Interprétez la réponse apportée par Hibernate à l'exécution

Corriger le programme en conséquence, vérifiez les résultats obtenus dans la base de données.

# Test Web

- Dans un package **technics**, Écrire une classe **Gateway** disposant de méthodes statiques permettant d'obtenir les résultats suivants :
    1. La liste des catégories ;
    2. la liste des produits d'une catégorie donnée ;
    3. enregistrant une ligne ;
    4. enregistrant une commande.
- Créer une classe **Display** disposant d'une méthode statique
    1. Retournant une liste au format HTML à partir d'une ArrayList passée en paramètre
- Construire les **pages JSP** répondant au service suivant :
    1. l'utilisateur lance l'application dans son navigateur ;
    2. il voit une liste déroulante de catégories ;

3. il sélectionne une catégorie dans la liste et voit une liste déroulante des produits de cette catégorie ;
4. il sélectionne un produit et saisit une quantité voulue ;
5. un bouton lui permet de continuer le remplissage de son panier (retour à 2)
6. un bouton lui permet de valider son panier : la commande est alors enregistrée.

Ajouter toutes les classes (servlet) et méthodes nécessaires pour éviter d'avoir à effectuer un quelconque traitement dans les JSP.

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
**http://slamwiki2.kobject.net/slam4/orm/hibernate?rev=1353778356**

Last update: **2019/08/31 14:39**