KObject

Site de référence : www.kobject.net
KObject est un produit Open source sous licence GNU LGPL, disponible sur la forge logicielle sourceforge.net.

Principales fonctionnalités :

- Mapping relationnel/Objet
- Mise en place de MVC2 pour J2ee

Configuration logicielle

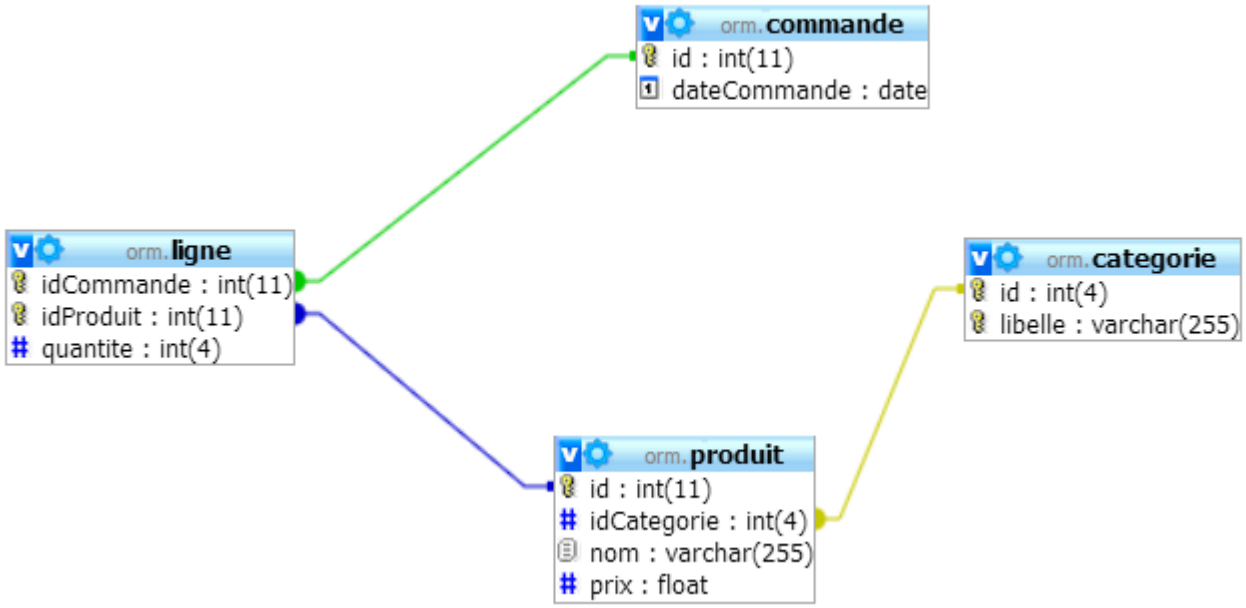
Vous disposez de :

- Eclipse Juno J2EE
- Kobject-library1.0.0.22f-beta1
- 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 le jar de Kobject et le driver JDBC pour mysql dans le dossier **WebContent/WEB-INF/lib**
- Copier le fichier de configuration de Kobject (**config.ko**) à la racine du projet.
- copier le fichier de configuration d'ehCache **ehCache.xml** dans le dossier **src**

Dans phpMyAdmin

- Créer la base de données **ormK** 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 de KObject dans la racine du projet :

Vérifiez les paramètres de connexion à Mysql.

Propriété	Valeur	Signification
package	metier	Package java dans lequel les classes métier seront définies
debug	SQL	Permet d'afficher les instructions SQL exécutées dans la console Eclipse

config.ko

```
base=ormK
classes=
controlClass=
cssFile=WebContent/css/css.properties
dbOptions=
dbType=mysql
debug=SQL
erFile=WebContent/conf/validation/er.properties
footerURL=WEB-INF/footer.jsp
headerURL=WEB-INF/header.jsp
host=127.0.0.1
koDateFormat=dd/MM/yyyy
mappingFile=WebContent/conf/mox.xml
messagesFile=WebContent/conf/validation/messages.properties
nullValue= 
package=net.kernel
password=
```

```
port=3306
sqlDateFormat=yyyy-MM-dd
useSetters=false
user=root
validationFile=WebContent/conf/kox.xml
webApp=false
useCache=true
cacheType=1
```

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**

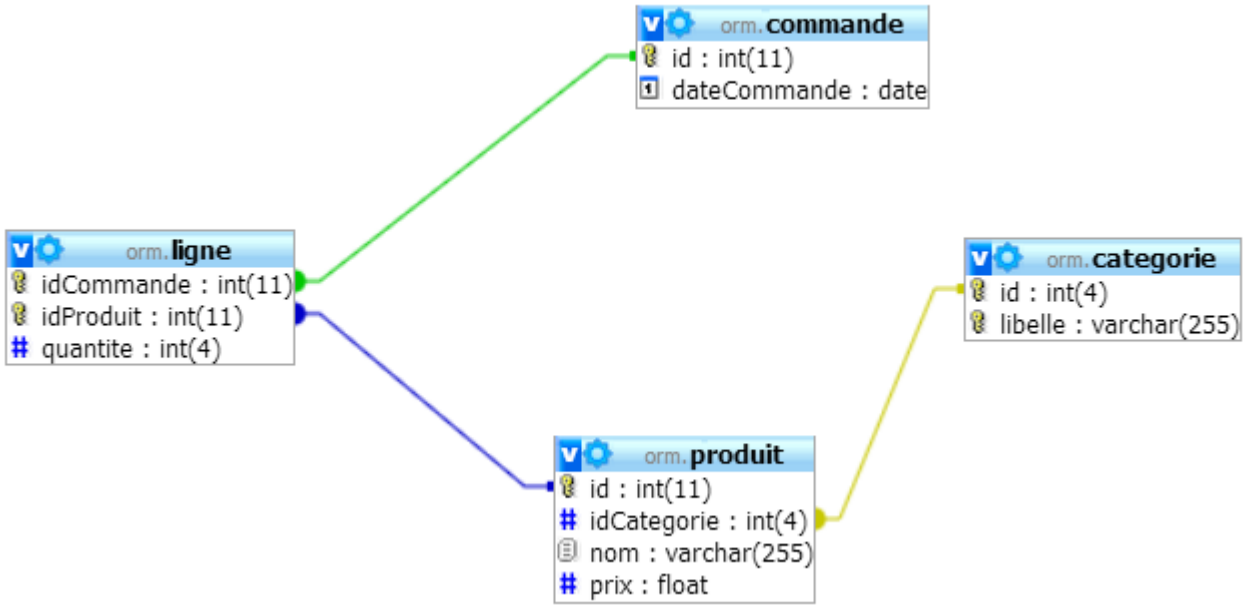
|h HibernateUtil.java

```
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(){
```

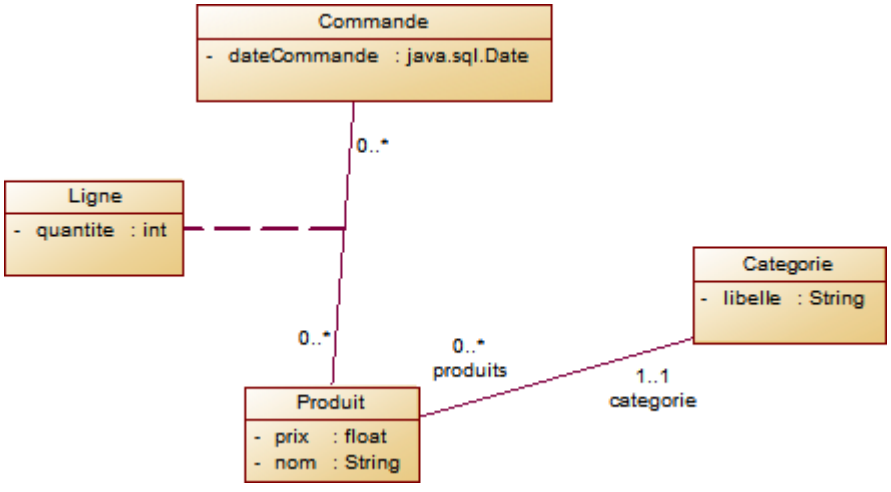
```
            return getSessionFactory().openSession();
        }
    }
```

Modèle relationnel et modèle objet

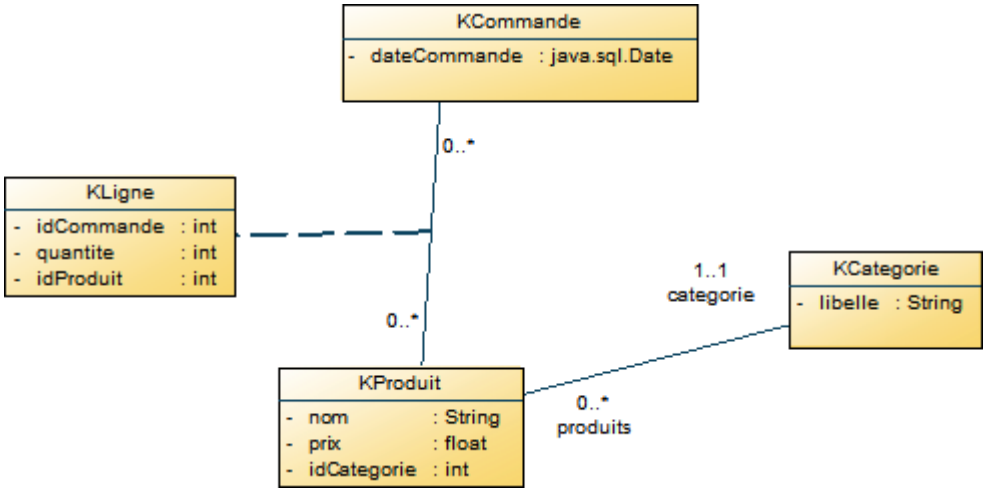
Modèle relationnel :



Modèle Objet (diagramme de classes) correspondant :



Nous allons modifier le modèle Objet pour le rendre compatible avec KObject et permettre le mapping, en ajoutant les membres qui serviront de clés étrangères (habituellement inutiles dans le monde Objet) :



Création des classes métier

La persistance des données est mise en place par Héritage et introspection. Les classes métier doivent hériter de la classe KObject.

Produits et catégories

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

KCategorie.java

```
1  package metier;
2
3  import net.ko.kobject.KListObject;
5  /**
6  * Classe KCategorie
7  */
8  @SuppressWarnings("serial")
9  public class KCategorie extends KObject {
10     private KListObject<KProduit> produits;
11     private String libelle;
12     public KCategorie() {
13        super();
14        keyFields="id";
15        tableName="categorie";
16        hasMany(KProduit.class);
17
18     }
```

*KProduit.java

```java
package metier;

import net.ko.kobject.KListObject;
/**
* Classe KProduit
*/
@SuppressWarnings("serial")
public class KProduit extends KObject {
    private KListObject<KLigne> lignes;
    private int idCategorie;
    private float prix;
    private KCategorie categorie;
    private String nom;

    public KProduit() {
        super();
        keyFields="id";
        tableName="produit";
        hasMany(KLigne.class);
        belongsTo(KCategorie.class);
    }
```

Les principes de construction des classes métiers à mapper avec **KObject** 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 KObject pour voir la correspondance entre les types)
- La classe doit être un **java Bean** pour permettre à KObject de travailler :
    - elle doit disposer d'un constructeur sans paramètre initialisant le membre keyFields définissant les champs présents dans la clé primaire
    - Chacun de ses membres doit disposer d'accesseurs

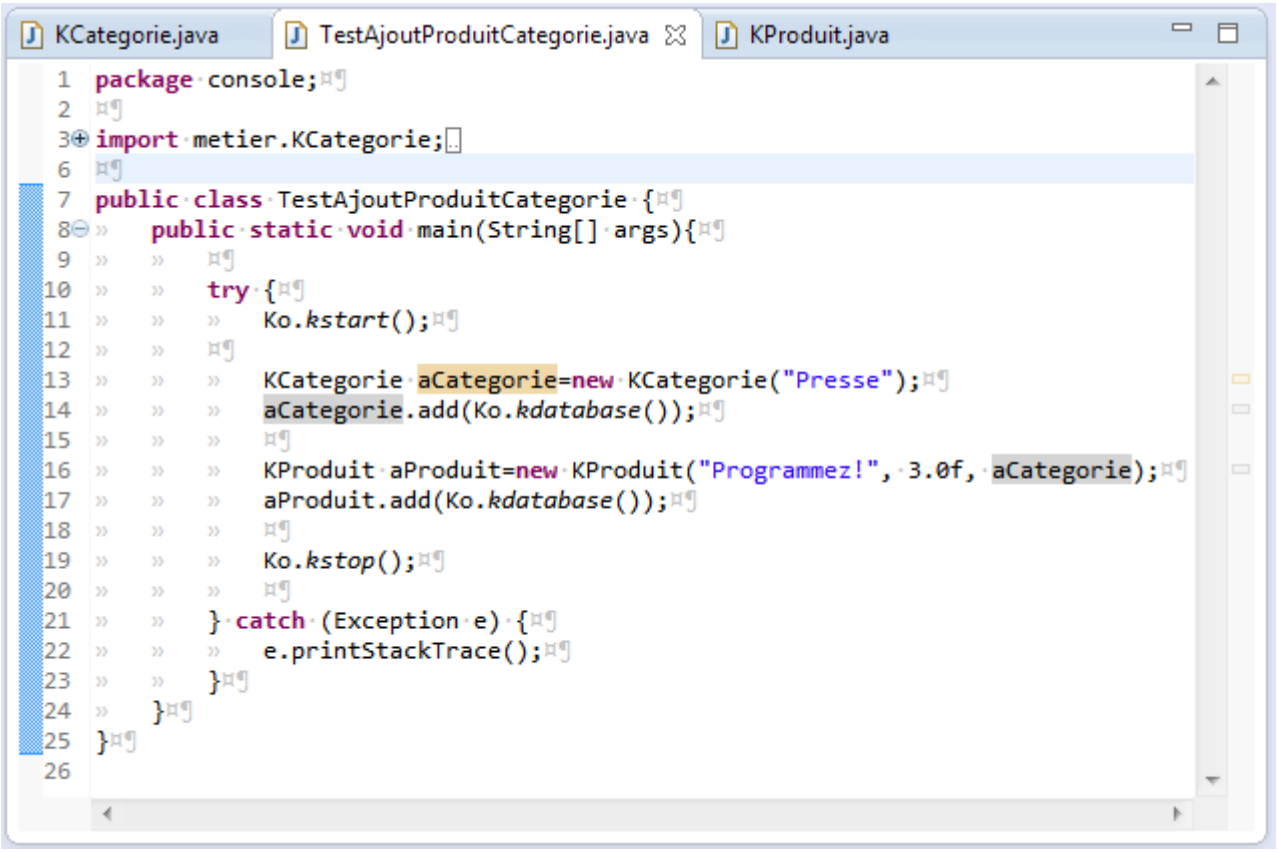
- Implémenter les classes **Categorie** et **Produit** dans le package **metier**
- Utiliser la complétion pour faire les imports
- Générer les accesseurs, le constructeur sans paramètre
- Ajouter un constructeur initialisant les membres de données et appelant le constructeur précédent
- Ajouter un toString 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. Réaliser un tableau montrant la correspondance de type (entier, chaine, etc.) entre les propriétés d'une classe et les champs d'une table.
5. 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 le constructeur)

Programme de test

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



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

1. À quoi correspond la méthode kstart() ?
2. Comment ont été traduits les liens objet entre le membre **categorie** et **produits** entre ces classes dans les tables de la base ?
3. Quelles requêtes SQL ont été créées par KObject pour réaliser la persistance ?
4. Que se passe t-il si l'insertion de la catégorie échoue ?

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

```
package console;
import metier.KProduit;
import net.ko.framework.Ko;
import net.ko.kobject.KObject;
public class TestLoadProduit {
    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            Ko.kstart();
            KProduit aProduit=(KProduit)KObject.kloadOne(KProduit.class, Ko.kdatabase(), 50);
            System.out.println(aProduit.getNom());
            System.out.println(aProduit.getCategorie().getLibelle());
        } catch (Exception e) {
            e.printStackTrace();
        }
        Ko.kstop();
    }
}
```

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

```
package console;
import metier.KCategorie;
import net.ko.framework.Ko;
import net.ko.kobject.KObject;
public class TestLoadCategorie {
    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            Ko.kstart();
            KCategorie aCategorie=(KCategorie)KObject.kloadOne(KCategorie.class, Ko.kdatabase(), 3);
            System.out.println(aCategorie.getLibelle());
            System.out.println(aCategorie.getProduits());
            System.out.println(aCategorie.getAttribute("produits"));
        } catch (Exception e) {
            e.printStackTrace();
        }
        Ko.kstop();
    }
}
```

A partir de ses 2 programmes et de leur exécution :

1. Précisez ce que charge exactement KObject lors du chargement d'un Objet
2. Précisez comment sont chargés les instances liées à un objet chargé pour les liens **belongsTo** et **hasMany**
3. En quoi consiste le chargement paresseux de KObject ?

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 :

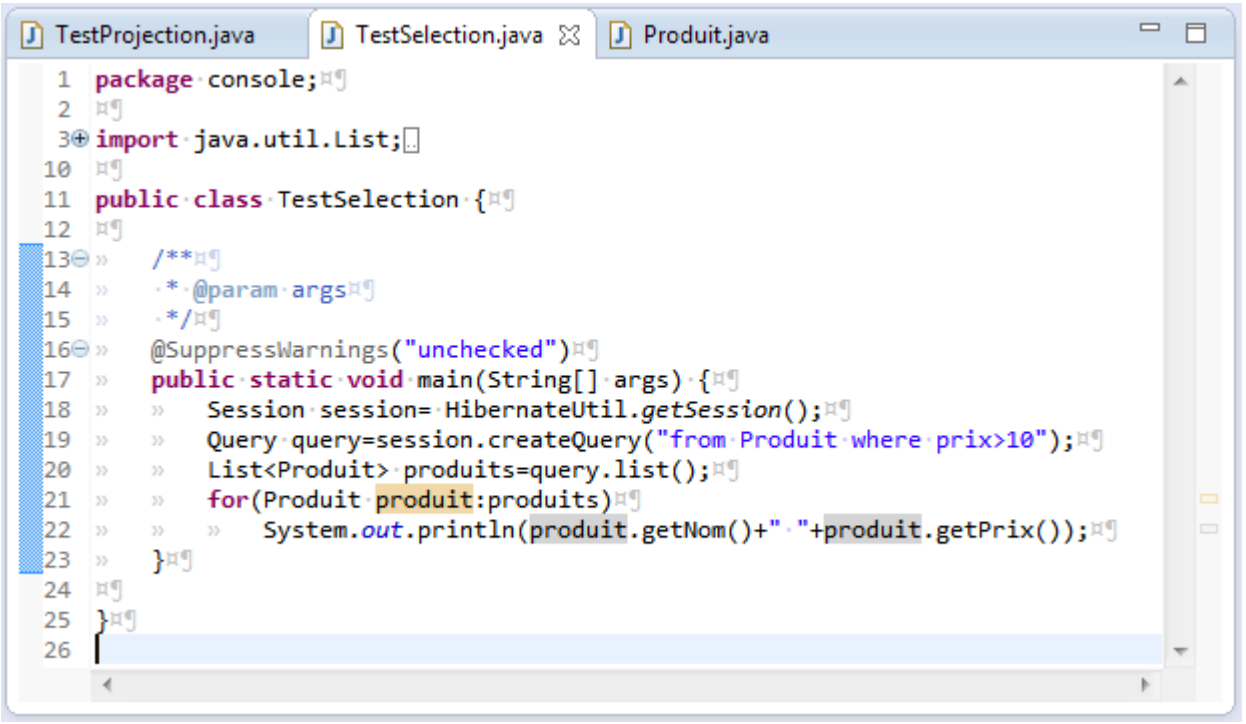
```
        for(Categorie categorie:categories)
            System.out.println(categorie);
```

A partir de l'exécution du programme modifié :

1. Interprétez et expliquez le résultat obtenu

Sélection

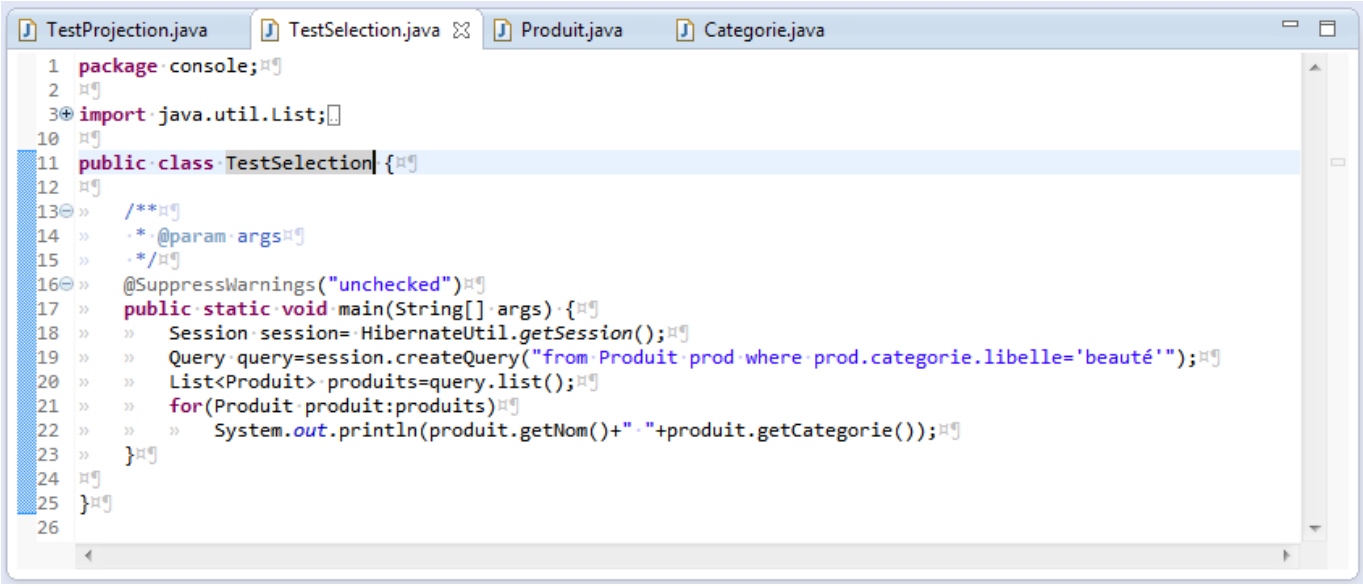
Créer le programme suivant



A partir du programme :

1. Interprétez les requêtes SQL exécutées par Hibernate

Créer le programme suivant

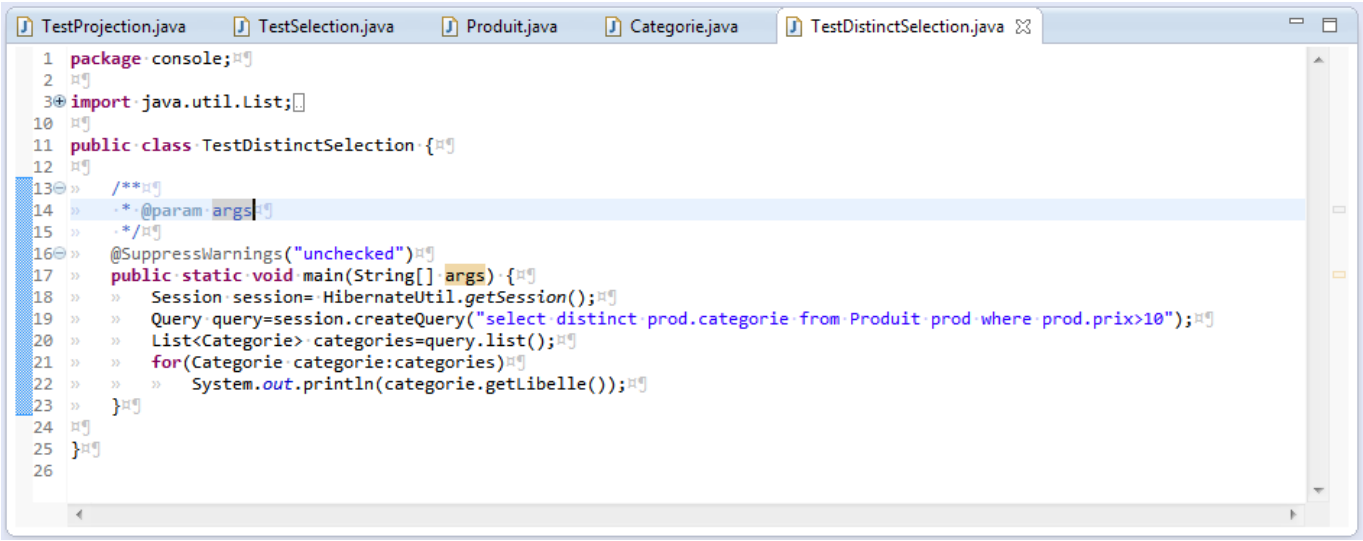


A partir du programme :

1. Interprétez les requêtes SQL exécutées par Hibernate

Sélection avec distinct et projection

Créer le programme suivant



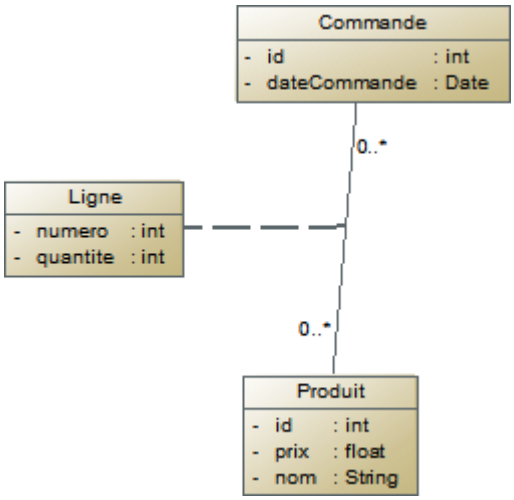
A partir du programme :

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.

.