Persistance de données : JPA avec Hibernate

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Hibernate

Object-Relational Mapping (lien objet-relationnel)

- une couche d'abstraction à la base de données
- une classe qui permet à l'utilisateur d'utiliser les tables d'une base de données comme des objets
- consiste à associer :
  - une ou plusieurs classes à chaque table
  - un attribut de classe à chaque colonne de la table
- a comme objectif de ne plus écrire de requête SQL

Hibernate

Object-Relational Mapping (lien objet-relationnel)

- une couche d'abstraction à la base de données
- une classe qui permet à l'utilisateur d'utiliser les tables d'une base de données comme des objets
- consiste à associer :
  - une ou plusieurs classes à chaque table
  - un attribut de classe à chaque colonne de la table
- a comme objectif de ne plus écrire de requête SQL

Plusieurs ORM proposés pour chaque Langage de POO.

Hibernate

Pour Java

- **Hibernate**
- EclipseLink
- Java Data Objects (JDO)
- ...

Hibernate

Hibernate

- un ORM (le premier) Java
- open-source
- créé par JBoss (Entreprise productrice de serveurs d'application JEE JBoss)
- possédant une extension **NHibernate** pour la plateforme .NET (de Microsoft)
- pouvant être utilisé dans un projet Java ou JEE

Hibernate

Remarque

Pour définir une entité, **Hibernate** utilise

- Soit un fichier de Mapping appelé `entity.hbm.xml`
- Soit des annotations JPA comme `@Entity`, `@Id`...

Hibernate

JPA : Java Persistence API

- ensemble d'interfaces standardisé par Sun, qui permet l'organisation des données
- proposé par JSR (Java Specification Requests)
- s'appuyant sur l'utilisation des annotations pour définir le lien entre `Entity` (classe) et table (en base de données relationnelle) et sur le gestionnaire `EntityManager` pour gérer les données (insertion, modification...)

Hibernate

Ajouter le plugin Hibernate à Eclipse

- Dans le menu `Help`, choisir `Eclipse Marketplace`
- Saisir `Hibernate` dans la zone de saisie et cliquer sur `Go`
- Dans la liste, chercher `JBoss Tools` et lancer l'installation
- Attendre la fin d'installation et redémarrer Eclipse

Hibernate

Différentes étapes pour persister des données avec **Hibernate**

- Préparer une connexion
- Créer un projet (avec maven)
- Ajouter les dépendances pour *Hibernate* et *MySQL Connector* dans `pom.xml`
- Créer le fichier de configuration `hibernate.cfg.xml`
- Créer les entités
- Persister les données

Hibernate

Avant de créer une connexion

créer une base de données appelée `hibernate`

Hibernate

Avant de créer une connexion

créer une base de données appelée `hibernate`

Création d'une connexion à la base de données (`hibernate`)

- Aller dans menu `File` ensuite `New` et enfin choisir `Other`
- Chercher `Connection Profile`
- Sélectionner le SGBD (dans notre cas `MySQL`)
- Attribuer un nom à cette connexion dans `Name`
- Cliquer sur `New Driver Definition` devant la liste déroulante de `Drivers`
- Définir le driver en choisissant le dernier `MySQL JDBC DRIVER`, en précisant son emplacement dans la rubrique `JAR List`, en supprimant celui qui existait et en modifiant les données dans la rubrique `Properties`
- Vérifier ensuite l'`URL`, `User name`, `Password` et `DataBase Name` (`hibernate`) et Valider

Hibernate

Tester la connexion

- Aller dans l'onglet `Data Source Explorer`
- Faire un clic droit et ensuite choisir `Connect`

Hibernate

Créer un projet maven

- Aller dans menu `File` ensuite `New` et enfin choisir `Maven project`
- Cliquer sur `Next`
- Sélectionner `Internal` dans la liste `Catalog` puis choisir `maven-archetype-quickstart` puis cliquer sur `Next`
- Saisir `org.eclipse` dans `Group Id` et `Hibernate` dans `Artifact Id`
- Cliquer sur `Finish`

Hibernate

Ajouter une première dépendance pour Hibernate dans `pom.xml`

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core
   -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.7.Final</version>
</dependency>
```

Hibernate

Ajouter une première dépendance pour Hibernate dans `pom.xml`

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core
   -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.7.Final</version>
</dependency>
```

Ajouter une deuxième pour le driver de MySQL

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
</dependency>
```

Hibernate

Ajouter une première dépendance pour Hibernate dans `pom.xml`

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core
   -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.7.Final</version>
</dependency>
```

Ajouter une deuxième pour le driver de MySQL

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
</dependency>
```

Pour mettre à jour ces dépendances dans le projet, il faut faire `Maven > Update Project`

Hibernate

Créer le fichier de configuration `hibernate.cfg.xml`

- Faire un clic droit sur `src/main/java` et aller dans `New > Other`
- Chercher `Hibernate` puis sélectionner `Hibernate Configuration File (cfg.xml)` et cliquer sur `Next`

Deux solutions possibles

- Soit en cliquant sur `Get values from Connection`
- Soit en remplissant le formulaire champ par champ

Quelle que soit la solution, il faut que la valeur choisie pour `Hibernate version` correspond à celle de la dépendance ajoutée dans `pom.xml`

Hibernate

Si on veut remplir le formulaire champ par champ

- Choisir MySQL de la liste `Database dialect`
- Sélectionner le driver MySQL, `com.mysql.jdbc.Driver`, de la liste `Driver Class`,
- Choisir l'URL de connexion `jdbc:mysql://<host><:port>/<database>` de `Connection URL` et la remplacer par `jdbc:mysql://localhost:3306/hibernate`
- Saisir le nom d'utilisateur dans `Username` et le mot de passe dans `Password`
- Valider et aller vérifier les données dans le fichier XML généré

Hibernate

Contenu du fichier `hibernate.cfg.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd
     ">
<hibernate-configuration>
 <session-factory name="">
  <property name="hibernate.connection.driver_class">com.mysql.
     jdbc.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://
     localhost:3306/hibernate</property>
  <property name="hibernate.connection.username">root</property
     >
  <property name="hibernate.connection.password">root</property
     >
  <property name="hibernate.dialect">org.hibernate.dialect.
     MySQLDialect</property>
 </session-factory>
</hibernate-configuration>
```

Hibernate

On peut aussi ajouter la propriété `hbm2ddl.auto` qui peut prendre comme valeur

- `create` : crée les tables, les données précédemment présentes dans les tables seront perdues.
- `update` : met à jour les tables avec les valeurs données. si les tables ne sont pas présentes dans la base de données, elles seront créées.
- `validate` : si les tables ne sont pas présentes dans la base de données, elles ne seront pas créées et une exception sera levée.
- `create-drop` : créer les tables en détruisant les données précédemment présentes. Les tables de la base de données seront aussi supprimées lorsque la SessionFactory est fermée (à utiliser pour tester).

Hibernate

Pour afficher les requêtes SQL exécutées par Hibernate dans la console

- On peut ajouter la propriété `show_sql` qui prend soit `true` soit `false`

Hibernate

Nouveau contenu du fichier `hibernate.cfg.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory name="">
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.
    Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306
    /hibernate</property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect
    </property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
 </session-factory>
</hibernate-configuration>
```

Créons une classe `Personne` **dans** `org.eclipse.model`

```
public class Personne {
  private int num;
  private String nom;
  private String prenom;

  public int getNum() {
    return num;
  }
  public void setNum(int num) {
    this.num = num;
  }
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

Hibernate

Créer le fichier de configuration `Personne.hbm.xml`

- Faire un clic droit sur `src/main/java` et aller dans `New > Other`
- Chercher `Hibernate`
- Sélectionner `Hibernate XML Mapping File (hbm.xml)` puis cliquer sur `Next`
- Choisir le package contenant les entités `org.eclipse.model` et supprimer les autres puis cliquer sur `Next`
- Valider et aller vérifier les données dans le fichier XML généré

Hibernate

Exemple de contenu du fichier `Personne.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
   3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 2 ao?t 2018 06:34:34 by Hibernate Tools 3.5.0.Final -->
<hibernate-mapping>
    <class name="org.eclipse.model.Personne" table="PERSONNE">
        <id name="num" type="int">
            <column name="NUM" />
            <generator class="assigned" />
        </id>
        <property name="nom" type="java.lang.String">
            <column name="NOM" />
        </property>
        <property name="prenom" type="java.lang.String">
            <column name="PRENOM" />
        </property>
    </class>
</hibernate-mapping>
```

Remplacer `<generator class="assigned" />` par `<generator class="increment" />` pour que la clé primaire soit auto-incrémentale.

Hibernate

Nouveau contenu du fichier `hibernate.cfg.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory name="">
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.
    Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306
    /hibernate</property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect
    </property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <mapping resource="org/eclipse/model/Personne.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

Hibernate

Étapes

Pour persister des données, il faut

- Créer un objet de configuration d'**Hibernate**
- Utiliser cet objet pour charger le fichier de configuration `hibernate.cfg.xml`
- Enregistrer la classe `Personne` dans l'objet de configuration
- Créer une usine de gestionnaire d'entité (appelée `EntityManagerFactory` par **EclipseLink** et `SessionFactory` par **Hibernate**)
- Créer un gestionnaire d'entité (appelé `EntityManager` par **EclipseLink** et `Session` par **Hibernate**)
- [Démarrer une transaction]
- Utiliser le gestionnaire de données pour persister les données
- [Terminer la stransaction et] fermer les différents flux

Hibernate

Pour créer un objet de configuration

```
Configuration configuration = new Configuration();
```

Hibernate

Pour créer un objet de configuration

```
Configuration configuration = new Configuration();
```

Pour charger le fichier `hibernate.cfg.xml` **(situé dans** `src/main/java`**)**

```
configuration.configure();
```

Hibernate

Pour créer un objet de configuration

```
Configuration configuration = new Configuration();
```

Pour charger le fichier `hibernate.cfg.xml` **(situé dans** `src/main/java`**)**

```
configuration.configure();
```

Si le fichier est situé dans un autre emplacement (par exemple, dans `org.eclipse.model`**), on peut faire**

```
configuration.configure("org/eclipse/model/hibernate.cfg.xml");
```

Hibernate

Pour créer un objet de configuration

```
Configuration configuration = new Configuration();
```

Pour charger le fichier `hibernate.cfg.xml` **(situé dans** `src/main/java`**)**

```
configuration.configure();
```

Si le fichier est situé dans un autre emplacement (par exemple, dans `org.eclipse.model`**), on peut faire**

```
configuration.configure("org/eclipse/model/hibernate.cfg.xml");
```

On peut fusionner les deux étapes précédentes

```
Configuration configuration = new Configuration().configure();
```

Hibernate

Pour créer un objet de configuration

```
Configuration configuration = new Configuration();
```

Pour charger le fichier `hibernate.cfg.xml` **(situé dans** `src/main/java`**)**

```
configuration.configure();
```

Si le fichier est situé dans un autre emplacement (par exemple, dans `org.eclipse.model`**), on peut faire**

```
configuration.configure("org/eclipse/model/hibernate.cfg.xml");
```

On peut fusionner les deux étapes précédentes

```
Configuration configuration = new Configuration().configure();
```

Ajouter l'entité à l'objet de configuration

```
configuration.addClass(Personne.class);
```

Hibernate

Construire l'usine de gestionnaire d'entité à partir de l'objet de configuration

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Hibernate

Construire l'usine de gestionnaire d'entité à partir de l'objet de configuration

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Obtenir le gestionnaire d'entité

```
Session session = sessionFactory.openSession();
```

Hibernate

Construire l'usine de gestionnaire d'entité à partir de l'objet de configuration

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Obtenir le gestionnaire d'entité

```
Session session = sessionFactory.openSession();
```

Démarrer une transaction

```
Transaction transaction = session.beginTransaction();
```

Hibernate

Construire l'usine de gestionnaire d'entité à partir de l'objet de configuration

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Obtenir le gestionnaire d'entité

```
Session session = sessionFactory.openSession();
```

Démarrer une transaction

```
Transaction transaction = session.beginTransaction();
```

Insérer un objet dans la base de données

```
session.persist(personne);
```

Hibernate

Construire l'usine de gestionnaire d'entité à partir de l'objet de configuration

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Obtenir le gestionnaire d'entité

```
Session session = sessionFactory.openSession();
```

Démarrer une transaction

```
Transaction transaction = session.beginTransaction();
```

Insérer un objet dans la base de données

```
session.persist(personne);
```

Terminer la transaction et fermer les flux

```
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Mettons tout ça dans le `main` de `App.java`

```
Personne personne = new Personne();
personne.setNom("travolta");
personne.setPrenom("john");
Configuration configuration = new Configuration().configure();
configuration.addClass(Personne.class);
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Mettons tout ça dans le `main` de `App.java`

```
Personne personne = new Personne();
personne.setNom("travolta");
personne.setPrenom("john");
Configuration configuration = new Configuration().configure();
configuration.addClass(Personne.class);
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Les `import` nécessaires

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
```

Hibernate

Constats

- Une table `personne` a été créée dans la base de données `hibernate`
- Un tuple avec les valeurs `(1,travolta,john)` a été inséré dans la table `personne`

Hibernate

On peut aussi enregistrer la classe à utiliser (ici `Personne`) dans `hibernate.cfg.xml` en ajoutant une nouvelle balise `<mapping class="org.eclipse.model.Personne"/>`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory name="">
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.
    Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306
    /hibernate</property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect
    </property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <mapping resource="org/eclipse/model/Personne.hbm.xml"/>
  <mapping class="org.eclipse.model.Personne"/>
 </session-factory>
</hibernate-configuration>
```

Hibernate

Nous pouvons ainsi supprimer la ligne
`configuration.addClass(Personne.class)`

```
Personne personne = new Personne();
personne.setNom("travolta");
personne.setPrenom("john");
Configuration configuration = new Configuration().
   configure();
//configuration.addClass(Personne.class);
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Chaque entité doit être déclarée

- Soit dans `hibernate.cfg.xml` en ajoutant la ligne `<mapping class="org.eclipse.model.Personne" />`
- Soit lorsqu'on veut persister des données en ajoutant la ligne `configuration.addClass(Personne.class)` ou `configuration.addAnnotatedClass(Personne.class)`

Hibernate

Avant de commencer

- Supprimer le fichier de mapping `Personne.hbm.xml`
- Supprimer la ligne `<mapping resource="org/eclipse/model/Personne.hbm.xml"/>` dans `hibernate.cfg.xml` (Ne pas supprimer `<mapping class="org.eclipse.model.Personne"/>`)
- Supprimer et recréer la base de données `hibernate`

Hibernate

Ajoutons les annotations suivantes dans la classe `Personne`

```
package org.eclipse.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Personne {

  @Id
  @GeneratedValue (strategy=GenerationType.IDENTITY)
  private int num;

  private String nom;
  private String prenom;

  // + getters setters et toString

}
```

Hibernate

Pour ajouter une personne dans la base de données

```
Personne personne = new Personne();
personne.setNom("travolta");
personne.setPrenom("john");
Configuration configuration = new Configuration().
   configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

On peut utiliser `save` pour récupérer la valeur de la clé primaire qui a été attribué au tuple ajouté dans la base de données

```
Personne personne = new Personne();
personne.setNom("travolta");
personne.setPrenom("john");
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
Integer cle = session.save(personne);
transaction.commit();
session.close();
sessionFactory.close();
System.out.println(cle);
```

Hibernate

`save` Vs `persist`

- `save` est une méthode **Hibernate** tant dis que `persist` est une méthode **JPA**
- `save` retourne la valeur de la clé primaire attribuée au tuple
- `persist` n'a pas de valeur de retour
- `save` attribue l'identifiant au tuple immédiatement (à l'exécution d'`insert`) même s'il n'est pas dans une transaction car il s'agit bien de sa valeur de retour
- `persist` attribue l'identifiant au tuple à la fin de la transaction ou lorsqu'on fait un `flush()`
- `persist` peut appliquer la persistance en cascade
- `persist` et `save` permettent aussi de faire la modification si la clé de l'objet à ajouter existe dans la base de données

Hibernate

Si on n'avait pas déclaré l'entité `Personne` **dans** `hibernate.cfg.xml` **(**`<mapping class="org.eclipse.model.Personne"></mapping>`**), on peut, ici, ajouter la ligne** `configuration.addAnnotatedClass(Personne.class)`

```
Personne personne = new Personne();
personne.setNom("travolta");
personne.setPrenom("john");
Configuration configuration = new Configuration().configure();
configuration.addAnnotatedClass(Personne.class);
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Pour chercher une personne (avec `load`)

```
Configuration configuration = new Configuration().
   configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Personne personne1 = session.load(Personne.class, 1);
System.out.println(personne1);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Pour chercher une personne (avec `get`)

```
Configuration configuration = new Configuration().
   configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Personne personne1 = session.get(Personne.class, 1);
System.out.println(personne1);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

`get` vs `load`

- Les deux font la même chose
- Avec `get`, **Hibernate** récupère immédiatement l'objet de la base de données
- Avec `load`, **Hibernate** récupère l'objet de la base de données lors de sa première utilisation
- Si l'objet recherché n'existe pas, `get` retourne `null` tandis que `load` déclenche une `ObjectNotFoundException`

Hibernate

Pour modifier une personne

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Personne personne3 = session.get(Personne.class, 1);
personne3.setNom("Abruzzi");

session.flush();

transaction.commit();
session.close();
sessionFactory.close();
```

La méthode `flush()` ne prend pas de paramètre. Donc elle envoie tous les changements des entités managées dans la base de données

Hibernate

On peut aussi utiliser `persist()` ou `save()` pour enregistrer les modifications

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Personne personne3 = session.get(Personne.class, 2);
personne3.setNom("Abruzzi");

session.persist(personne3);

transaction.commit();
session.close();
sessionFactory.close();
```

Les méthodes `save(obj)` et `persist(obj)` prennent en paramètre l'objet modifié et à sauvegarder dans la base de données

Hibernate

Si l'objet est détaché (n'est pas attaché à une session), le persister entraine sa création et non pas sa modification

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Personne personne4 = new Personne();
personne4.setNom("Denzel");
personne4.setNum(1);
personne4.setPrenom("Washington");

session.persist(personne4);

transaction.commit();
session.close();
sessionFactory.close();
```

Malgré l'existence de l'identifiant 1, un nouveau tuple sera créé avec un identifiant différent de 1 (pareil pour `save()`)

Hibernate

Pour supprimer une personne

```
Configuration configuration = new Configuration().
   configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Personne personne3 = session.get(Personne.class, 1);

session.delete(personne3);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Pour récupérer la liste de toutes les personnes

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Criteria criteria = session.createCriteria(Personne.class);
List<Personne> personnes = (List<Personne>) criteria.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

N'oublions pas d'ajouter `toString()` **dans la classe** `Personne`.

Hibernate

Pour récupérer une liste de personnes selon un critère (ici le `nom`)

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

String string = "travolta";
Criteria criteria = session.createCriteria(Personne.class);
criteria = criteria.add(Restrictions.eq("nom", string));
List<Personne> personnes = (List<Personne>) criteria.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Autres méthodes de la session

- `refresh(entity)` : permet de synchroniser l'état de l'entité passée en paramètre (`entity`) avec son état en base de données.
- `evict(entity)` : permet de détacher l'entité passée en paramètre (`entity`) de l'`EntityManager` qui la gère.
- `merge(entity)` : permet d'attacher l'entité passée en paramètre (`entity`), gérée par un autre `EntityManager`, à l'`EntityManager` courant.

Hibernate

Exemple avec utilisation de `refresh()`

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

//on suppose que John Wick avec un num 1 existe dans la BD
Personne p = session.load(Personne.class, 1);
p.setNom("Travolta");
session.refresh(p);
System.out.println("le nom est " + p.getNom());
// affiche le nom est Wick

transaction.commit();
session.close();
sessionFactory.close();
// si on supprime session.refresh(p); Travolta sera affiché
```

Hibernate

Exemple avec utilisation de `evict()`

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

// on suppose que John Wick avec un num 1 existe dans la BD
Personne p = session.load(Personne.class, 1);
p.setNom("Travolta");
session.evict(p); // p n'est plus géré par session
session.flush();
transaction().commit();
Personne p1 = session.load(Personne.class, 1);
System.out.println("le nom est " + p1.getNom());

// affiche le nom est Wick
session.close();
sessionFactory.close();
```

Hibernate

Autres annotations

Annotation	**désignation**
`@Entity`	permet de qualifier la classe comme entité
`@Id`	indique l'attribut qui correspond à la clé primaire de la table
`@Table`	décrit la table désignée par l'entité
`@Column`	définit les propriétés d'une colonne
`@IdClass`	indique que l'entité annotée contient une clé composée les champs constituant la clé seront annoté par `@Id`
`@GeneratedValue`	s'applique sur les attributs annotés par @Id permet la génération automatique de la clé primaire

Hibernate

Autres annotations

Annotation	désignation
`@Entity`	permet de qualifier la classe comme entité
`@Id`	indique l'attribut qui correspond à la clé primaire de la table
`@Table`	décrit la table désignée par l'entité
`@Column`	définit les propriétés d'une colonne
`@IdClass`	indique que l'entité annotée contient une clé composée les champs constituant la clé seront annoté par `@Id`
`@GeneratedValue`	s'applique sur les attributs annotés par @Id permet la génération automatique de la clé primaire

```
// la classe Personne
@IdClass(PersonnePK.class)
@Entity
public class Personne  {
  @Id
  private String nom;
  @Id
  private String prenom;
// ensuite  getters, setters et
   constructeur
```

```
// la classe PersonnePK
public class PersonnePK {
  @Id
  private String nom;
  @Id
  private String prenom;
// ensuite getters, setters,
   constructeur sans parametres et
   constructeur avec deux
   parametres nom et prenoms
```

Hibernate

Attributs de l'annotation `@Column`

Attribut	**désignation**
`name`	permet de définir le nom de la colonne s'il est différent de celui de l'attribut
`length`	permet de fixer la longueur d'une chaîne de caractères
`unique`	indique que la valeur d'un champ est unique
`nullable`	précise si un champ est null (ou non)
...	...

Hibernate

Attributs de l'annotation `@Table`

Attribut	**désignation**
`name`	permet de définir le nom de la table s'il est différent de celui de l'entité
`uniqueConstraints`	permet de définir des contraintes d'unicité sur un ensemble de colonnes

```
@Table(
    name="personne",
    uniqueConstraints={
        @UniqueConstraint(name="nom_prenom", columnNames
           ={"nom", "prenom"})
    }
)
```

Hibernate

L'annotation `@Transient`

L'attribut annoté par `@Transient` n'aura pas de colonne associée dans la table correspondante à l'entité en base de données.

Hibernate

Création de requêtes

- `createSQLQuery()` : permet d'exécuter une requête **SQL**
- `createNamedQuery()` : permet d'exécuter une requête **HQL** définie par des annotations
- `createQuery()` : permet d'exécuter une requête **HQL**

Pour exécuter une requête SQL

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

String sqlRequete = "select * from Personne";
SQLQuery query = session.createSQLQuery(sqlRequete);
query.addEntity(Personne.class);

List<Personne> personnes = (List<Personne>) query.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

Sans `query.addEntity(Personne.class)`**, on ne peut faire le cast** `List <Personne> personnes = (List <Personne>) query.list()`

Hibernate

On peut aussi exécuter une requête SQL paramétrée

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

String sqlRequete = "select * from Personne where nom = :nom";
SQLQuery query = session.createSQLQuery(sqlRequete);
query.addEntity(Personne.class);
query.setParameter("nom", "Abruzzi");

List<Personne> personnes = (List<Personne>) query.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Les requêtes nommées : exemple (on commence tout d'abord par définir la requête dans l'entité)

```
@Entity
@NamedQuery(
     name="findByNomPrenom",
     query="SELECT p FROM Personne p WHERE p.nom = :
        nom and p.prenom = :prenom"
)
public class Personne {
    // le code précédent
}
```

On peut importer `NamedQuery` **de** `javax.persistence.NamedQuery`

Hibernate

Les requêtes nommées : exemple (ensuite nous l'utiliserons)

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Query query = session.getNamedQuery("findByNomPrenom");
query.setParameter("nom", "Abruzzi");
query.setParameter("prenom", "John");

List<Personne> personnes = (List<Personne>) query.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

Il faut importer `Query` **de** `org.hibernate.Query`

Hibernate

Et si on veut définir plusieurs requêtes nommées

```
@Entity
@NamedQueries({
@NamedQuery(
    name="findByNomPrenom",
    query="SELECT p FROM Personne p WHERE p.nom = :nom and p.
       prenom = :prenom"
),
@NamedQuery(
    name="findByPrenom",
    query="SELECT p FROM Personne p WHERE p.prenom = :prenom"
),
})
public class Personne {
    // le code précédent
}
```

Hibernate

Et si on veut définir plusieurs requêtes nommées

```
@Entity
@NamedQueries({
@NamedQuery(
    name="findByNomPrenom",
    query="SELECT p FROM Personne p WHERE p.nom = :nom and p.
       prenom = :prenom"
),
@NamedQuery(
    name="findByPrenom",
    query="SELECT p FROM Personne p WHERE p.prenom = :prenom"
),
})
public class Personne {
    // le code précédent
}
```

C'est quoi le langage utilisé dans la chaîne `query` ?

Hibernate

HQL : Hibernate Query Language

- HQL est un langage de requêtes pour les entités JPA défini par Hibernate
- Inspiré du langage SQL mais adapté aux entités JPA
- Permet de manipuler des entités et pas les tables d'une base de données
- Supporte des requêtes de type `select`, `update` et `delete`

Hibernate

HQL : Hibernate Query Language

- HQL est un langage de requêtes pour les entités JPA défini par Hibernate
- Inspiré du langage SQL mais adapté aux entités JPA
- Permet de manipuler des entités et pas les tables d'une base de données
- Supporte des requêtes de type `select`, `update` et `delete`

On manipule des entités et non pas des tables. Le nom des entités est sensible à la casse.

Hibernate

Utiliser HQL pour récupérer une liste de personnes ayant un `nom = travolta`

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

String hql = "select p from Personne p where nom = :nom";
String string = "travolta";
Query query = session.createQuery(hql);
query.setParameter("nom", string);
List<Personne> personnes = (List<Personne>) query.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Simplifier la requête précédente

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

String hql = "from Personne where nom = :nom";
String string = "travolta";
Query query = session.createQuery(hql);
query.setParameter("nom", string);
List<Personne> personnes = (List<Personne>) query.list();
for(Personne personne : personnes)
    System.out.println(personne);

transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Hibernate : documentation officielle (en français)

`https://docs.jboss.org/hibernate/orm/3.6/reference/fr-FR/html/index.html`

Hibernate

Quatre (ou trois) relations possibles

- `OneToOne` : chaque objet d'une première classe est en relation avec un seul objet de la deuxième classe
- `OneToMany` : chaque objet d'une première classe peut être en relation avec plusieurs objets de la deuxième classe (la réciproque est `ManyToOne`)
- `ManyToMany` : chaque objet d'une première classe peut être en relation avec plusieurs objets de la deuxième classe et inversement

Hibernate

Avant de commencer

- Supprimer et recréer la base de données `hibernate`

Hibernate

L'entité `Adresse` **dans** `org.eclipse.model`

```
@Entity
public class Adresse {

  @Id
  private String rue;
  private String codePostal;
  private String ville;

  // + getters, setters et toString

}
```

Hibernate

Modifier l'entité Personne

```
@Entity
public class Personne {

    @Id
    private int num;
    private String nom;
    private String prenom;

    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    @JoinColumn(name="rue", referencedColumnName="rue", nullable=false)
        private Adresse adresse;

        // + les getters/setters de chaque attribut

}
```

Hibernate

Modifier l'entité Personne

```
@Entity
public class Personne {

    @Id
    private int num;
    private String nom;
    private String prenom;

    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    @JoinColumn(name="rue", referencedColumnName="rue", nullable=false)
        private Adresse adresse;

        // + les getters/setters de chaque attribut

}
```

Appellation

- `Personne` : entité propriétaire
- `Adresse` : entité inverse

Hibernate

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.
    REMOVE})
```

Explication

- `cascade` : ici on `cascade` les deux opérations `PERSIST` et `REMOVE` qu'on peut faire de l'entité propriétaire à l'entité inverse
- On peut cascader d'autres opérations telles que `DETACH`, `MERGE`, et `REFRESH`...
- on peut cascader toutes les opérations avec `ALL`

Hibernate

```
@JoinColumn(name="rue", referencedColumnName="rue",
   nullable=false)
```

Explication

- Pour désigner la colonne dans `Adresse` qui permet de faire la jointure
- Pour dire que chaque personne doit avoir une adresse (donc on ne peut avoir une personne sans adresse)

Hibernate

Testons l'ajout d'une personne

```
/* Adresse */
Adresse adresse = new Adresse();
adresse.setRue("Lyon");
adresse.setCodePostal("13015");
adresse.setVille("Marseille");
/* Personne */
Personne personne = new Personne();
personne.setAdresse(adresse);
personne.setNom("Ego");
personne.setPrenom("Paul");
/* Persistance */
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

N'oublions pas de déclarer l'entité `Adresse` dans `hibernate.cfg.xml`

Hibernate

Constats

- Deux tables ont été créées :
  - `adresse` avec les colonnes `rue, codePostal, ville`
  - `personne` avec les colonnes `num, nom, prenom, #rue`
- Deux tuples ont été insérés :
  - `(Lyon, 13015, Marseille)` dans `adresse`
  - `(1, Ego, Paul, Lyon)` dans `personne`

Hibernate

Constats

- Deux tables ont été créées :
  - `adresse` avec les colonnes `rue, codePostal, ville`
  - `personne` avec les colonnes `num, nom, prenom, #rue`
- Deux tuples ont été insérés :
  - `(Lyon, 13015, Marseille)` dans `adresse`
  - `(1, Ego, Paul, Lyon)` dans `personne`

Remplacer `persist()` par `save()` et vérifier que ça ne marche pas

Hibernate

Testons l'ajout avec `save`

```
Adresse adresse = new Adresse();
adresse.setRue("Paradis");
adresse.setCodePostal("13015");
adresse.setVille("Marseille");

Personne personne = new Personne();
personne.setAdresse(adresse);
personne.setNom("Wick");
personne.setPrenom("John");

Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.save(adresse);
session.save(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Sans la précision suivante

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.
    REMOVE})
```

Hibernate

Sans la précision suivante

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.
    REMOVE})
```

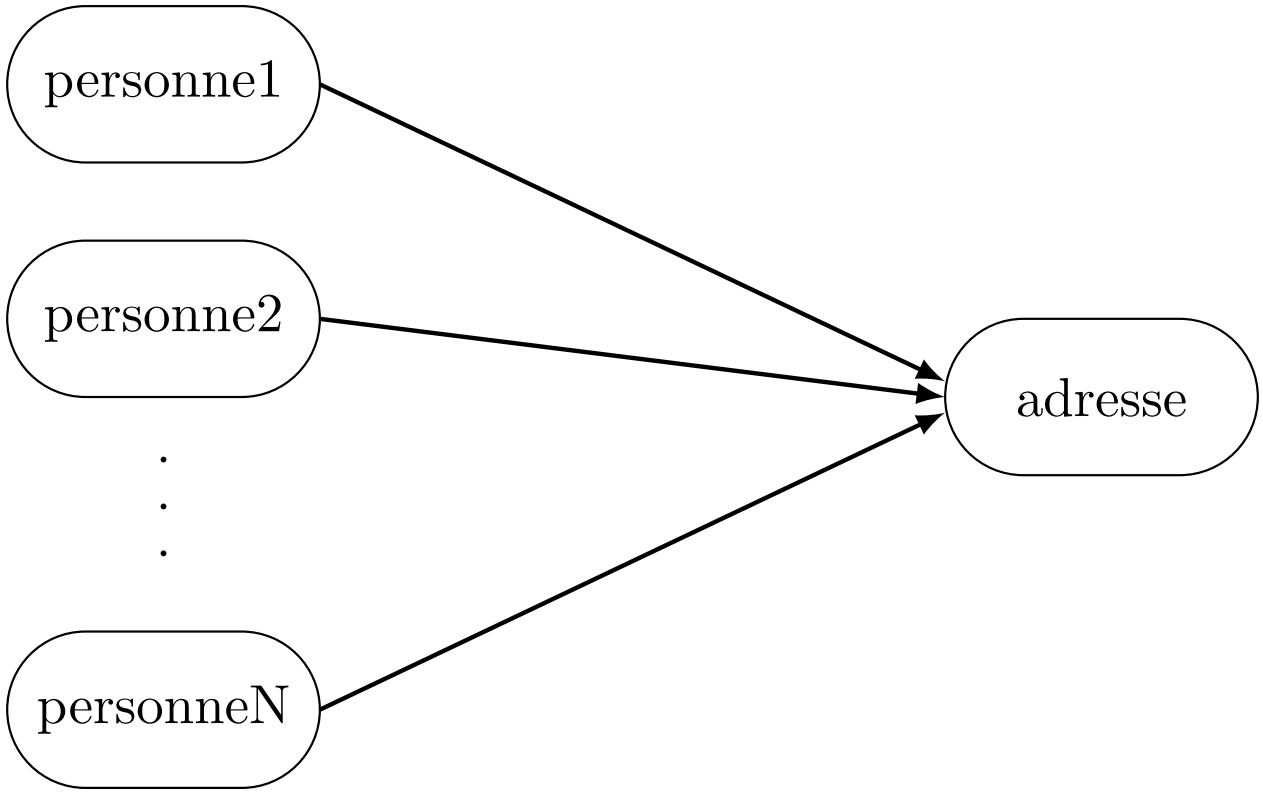
Il fallait persister l'objet adresse avant l'objet personne

```
Transaction transaction = session.beginTransaction();
session.persist(adresse);
session.persist(personne);
transaction.commit();
```

Hibernate

Exemple

Si on suppose que plusieurs personnes peuvent avoir la même adresse



Hibernate

Il faut juste changer

```
@ManyToOne(cascade={CascadeType.PERSIST, CascadeType
    .REMOVE})
@JoinColumn(name="rue", referencedColumnName="rue",
    nullable=false)
```

Hibernate

Il faut juste changer

```
@ManyToOne(cascade={CascadeType.PERSIST, CascadeType
    .REMOVE})
@JoinColumn(name="rue", referencedColumnName="rue",
    nullable=false)
```

Remarque

Le schéma de la base de données ne change pas.

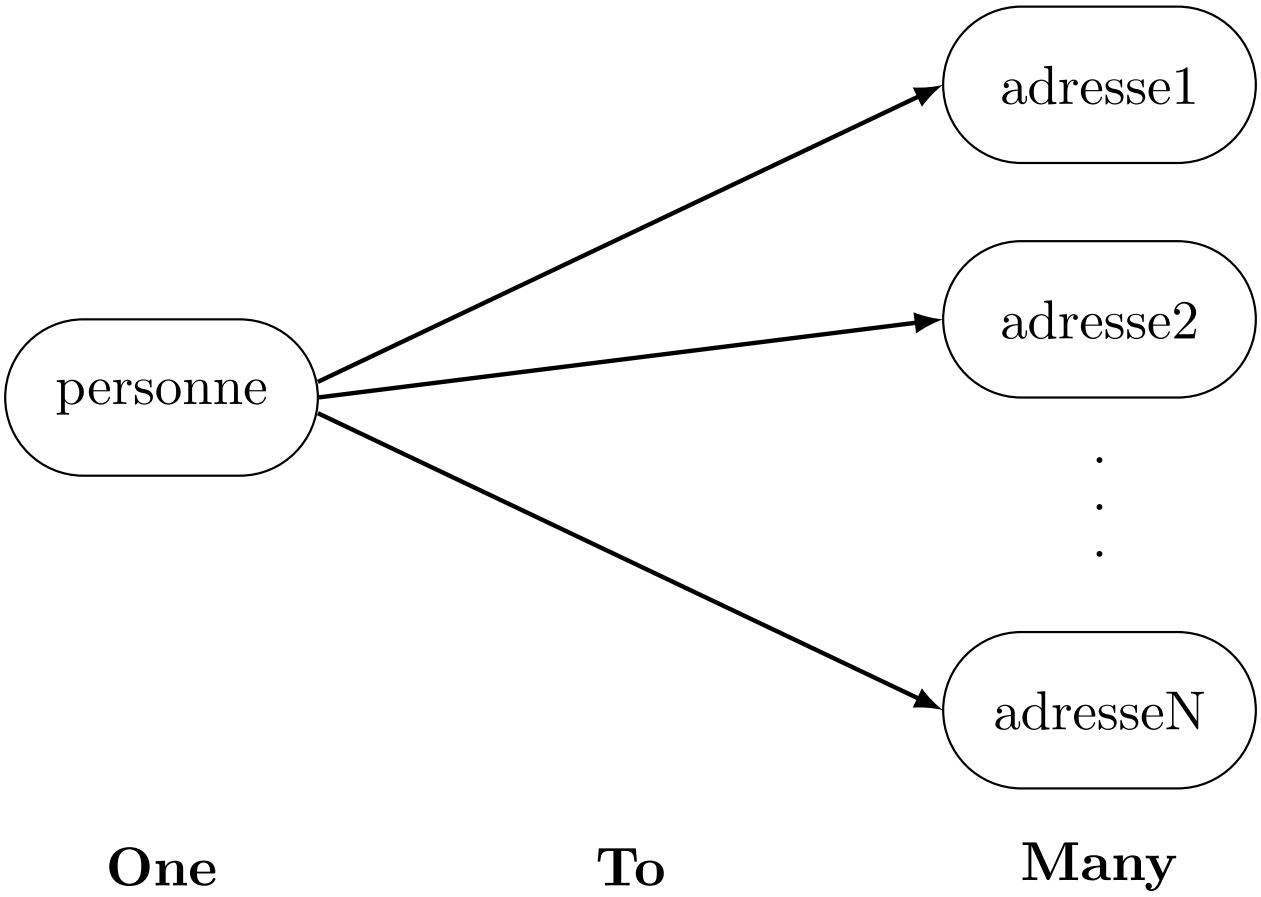
Hibernate

```
Adresse adresse = new Adresse();
adresse.setRue("New York");
adresse.setCodePostal("13015");
adresse.setVille("Marseille");
Personne personne = new Personne();
personne.setAdresse(adresse);
personne.setNom("Messi");
personne.setPrenom("Thiago");
Personne personne2 = new Personne();
personne2.setAdresse(adresse);
personne2.setNom("Messi");
personne2.setPrenom("Leo");
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
session.persist(personne2);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Exemple

Si on suppose qu'une personne peut avoir plusieurs adresses



Hibernate

Il faut changer l'attribut `adresse` par une liste d'adresses

```
@OneToMany(cascade={CascadeType.PERSIST,
   CascadeType.REMOVE})
private List <Adresse> adresses = new ArrayList <
   Adresse> ();
```

Hibernate

Il faut changer l'attribut `adresse` par une liste d'adresses

```
@OneToMany(cascade={CascadeType.PERSIST,
   CascadeType.REMOVE})
private List <Adresse> adresses = new ArrayList <
   Adresse> ();
```

N'oublions pas de supprimer ce code de l'entité `Personne`

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.
   REMOVE})
@JoinColumn(name="rue", referencedColumnName="rue",
   nullable=false)
private Adresse adresse;
// les getters/setters de chaque attribut
```

Hibernate

Ensuite

- il faut générer le getter et le setter d'`adresses`
- il faut aussi générer la méthode `add` et `remove` qui permettent d'ajouter ou de supprimer une adresse pour un objet personne (Dans `Source`, choisir `Generate Delegate Methods`).
- Renommer `add` en `addAdresse` et `remove` en `removeAdresse`

Hibernate

Avant de tester

Supprimer et recréer la base de données

Hibernate

```
Adresse a1 = new Adresse();
a1.setRue("Estaque");
a1.setCodePostal("13016");
a1.setVille("Marseille");
Adresse a2 = new Adresse();
a2.setRue("Merlan");
a2.setCodePostal("13013");
a2.setVille("Marseille");
Personne p1 = new Personne();
p1.setNom("Wick");
p1.setPrenom("John");
p1.addAdresse(a1);
p1.addAdresse(a2);
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
transaction.commit();
session.close();
sessionFactory.close();
```

Hibernate

Constats

- Trois tables ont été créées :
  - `adresse` avec les colonnes `rue, codePostal, ville`
  - `personne` avec les colonnes `num, nom, prenom`
  - `personne_adresse` avec les colonnes `#personne_num, #adresses_rue`
- Cinq tuples ont été insérés :
  - `(Estaque, 13016, Marseille)` et `(Merlan, 13013, Marseille)` dans `adresse`
  - `(1, Wick, John)` dans `personne`
  - `(1, Estaque)` et `(1, Merlan)` dans `personne_adresse`

Hibernate

On peut aussi définir si les objets de l'entité inverse (ici `Adresse`) seront chargés dans l'entité propriétaire (ici `Personne`). Il faut ajouter dans l'annotation l'attribut `fetch` :

```
@OneToMany(cascade={CascadeType.PERSIST,  CascadeType.REMOVE},
   fetch = FetchType.CONSTANTE)
private List <Adresse> adresses = new ArrayList <Adresse> ();
```

Hibernate

On peut aussi définir si les objets de l'entité inverse (ici `Adresse`) seront chargés dans l'entité propriétaire (ici `Personne`). Il faut ajouter dans l'annotation l'attribut `fetch` :

```
@OneToMany(cascade={CascadeType.PERSIST,  CascadeType.REMOVE},
   fetch = FetchType.CONSTANTE)
private List <Adresse> adresses = new ArrayList <Adresse> ();
```

Deux valeurs possibles pour `CONSTANTE`

- `EAGER` : les objets de l'entité `Adresse` en relation avec un objet `personne` de l'entité `Personne` seront chargés immédiatement.
- `LAZY` (par défaut) : les objets de l'entité `Adresse` en relation avec un objet `personne` de l'entité `Personne` seront chargés seulement lorsqu'ils sont demandés (quand on fait `personne.getAdresses()`).

Hibernate

Par défaut, c'est `LAZY` (rien à modifier dans l'entité `Personne`)

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Personne.class);
List<Personne> personnes = (List<Personne>) criteria.list();
for(Personne personne : personnes) {
    System.out.println(personne);
}
session.close();
sessionFactory.close();
```

Hibernate

Par défaut, c'est `LAZY` (rien à modifier dans l'entité `Personne`)

```
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.
   buildSessionFactory();
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Personne.class);
List<Personne> personnes = (List<Personne>) criteria.list();
for(Personne personne : personnes) {
    System.out.println(personne);
}
session.close();
sessionFactory.close();
```

Mettre le chargement à `EAGER` et tester

```
@OneToMany(cascade={CascadeType.PERSIST,  CascadeType.REMOVE},
   fetch = FetchType.EAGER)
private List <Adresse> adresses = new ArrayList <Adresse> ();
```

Hibernate

Remarque

- Possible d'avoir des tuples dupliqués avec le chargement `EAGER`
- En effet, **Hibernate** fait une jointure interne sur la table de jointure (`personne_adresse`) ce qui cause des éventuelles duplications (par exemple si une personne avait plusieurs adresses)

Hibernate

Remarque

- Possible d'avoir des tuples dupliqués avec le chargement `EAGER`
- En effet, **Hibernate** fait une jointure interne sur la table de jointure (`personne_adresse`) ce qui cause des éventuelles duplications (par exemple si une personne avait plusieurs adresses)

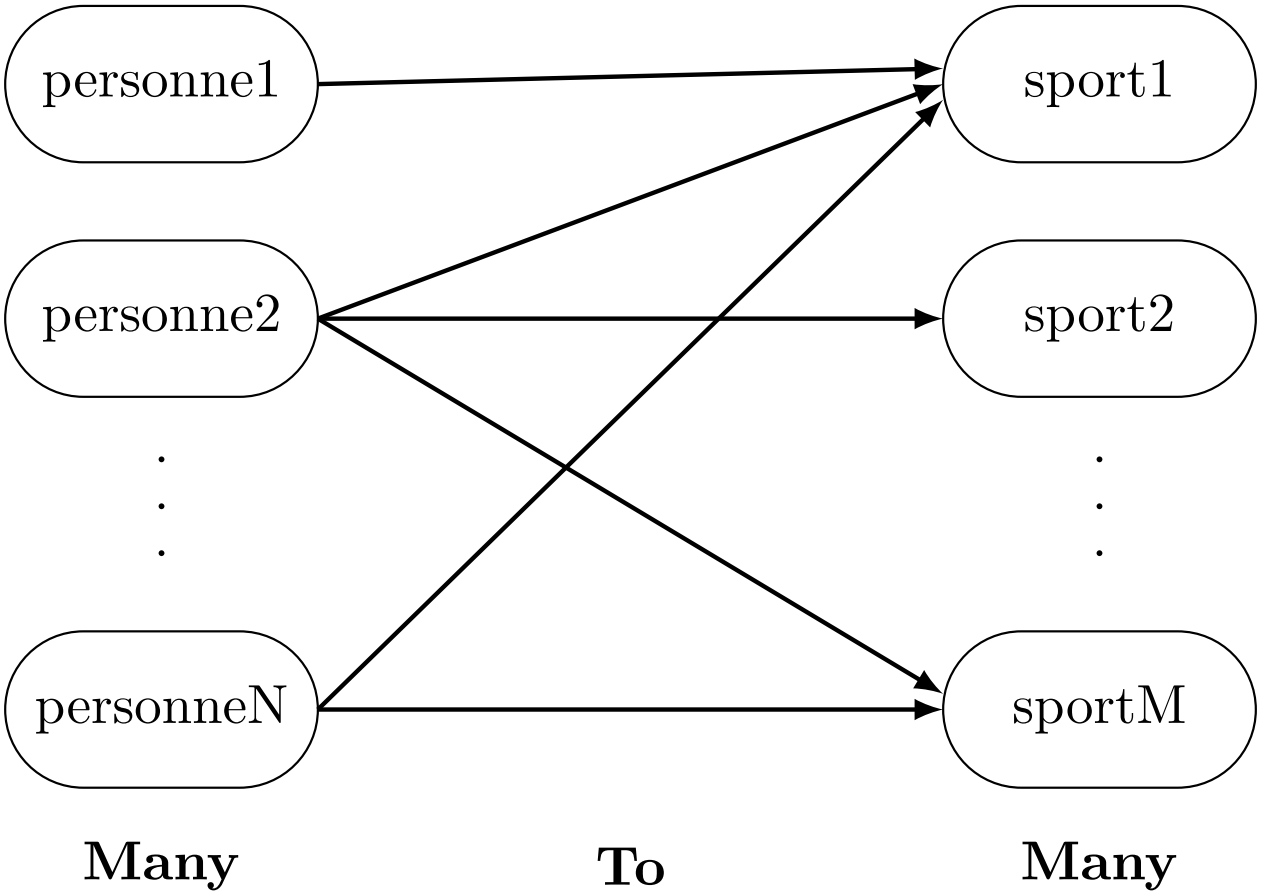
Nous pouvons rectifier le problème de duplication en ajoutant l'annotation suivante

```
@OneToMany(cascade={CascadeType.PERSIST,
   CascadeType.REMOVE}, fetch = FetchType.EAGER)
@Fetch(FetchMode.SELECT)
private List <Adresse> adresses = new ArrayList <
   Adresse> ();
```

Hibernate

Exemple

- Une personne peut pratiquer plusieurs sports
- Un sport peut être pratiqué par plusieurs personnes



Hibernate

Ce qu'il faut faire :

- commencer par créer une entité `Sport`
- définir la relation `ManyToMany` (exactement comme les deux relations précédentes) soit dans `Personne` soit dans `Sport`

Hibernate

Création de l'entité Sport

```
public class Sport {

        @Id
        private String nom;
        private String type;
        private static final long serialVersionUID = 1L;
        public Sport() {
                super();
        }
        public String getNom() {
                return nom;
        }
        public void setNom(String nom) {
                this.nom = nom;
        }
        public String getType() {
                return type;
        }
        public void setType(String type) {
                this.type = type;
        }
}
```

Hibernate

Ajoutons le `ManyToMany` dans la classe `Personne`

```
@Entity
public class Personne implements Serializable {

  // code précédent

  @ManyToMany(cascade={CascadeType.PERSIST,  CascadeType.REMOVE})
  private List <Sport> sports = new ArrayList <Sport> ();

  public List<Sport> getSports() {
    return sports;
  }
  public void setSports(List<Sport> sports) {
    this.sports = sports;
  }
  public boolean addSport(Sport sport) {
    return sports.add(sport);
  }
  public boolean removeSport(Sport sport) {
    return sports.remove(sport);
  }
}
```

Hibernate

```
Personne p1 = new Personne();
Personne p2 = new Personne();
p1.setNom("Voight");
p1.setPrenom("Bill");
p2.setNom("Bob");
p2.setPrenom("Joe");
Sport s1 = new Sport();
Sport s2 = new Sport();
Sport s3 = new Sport();
s1.setNom("football");
s2.setNom("tennis");
s3.setNom("box");
s1.setType("collectif");
s2.setType("individuel");
s3.setType("collectif ou individuel");
p1.addSport(s1);
p1.addSport(s3);
p2.addSport(s1);
p2.addSport(s2);
p2.addSport(s3);
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(p1);
session.persist(p2);
transaction.commit();
session.close();
sessionFactory.close();
```

N'oublions pas de déclarer l'entité `Sport` **dans** `hibernate.cfg.xml`

Hibernate

Constats

- Deux tables ont été créées (sans compter les tables trois précédentes) :
  - `sport` avec les colonnes `nom, type`
  - `personne_sport` avec les colonnes `#personne_num, #sports_nom`
- Dix tuples ont été insérés :
  - `(box, collectif ou individuel), (football, collectif)` et `(tennis, individuel)` dans `sport`
  - `(2, Voight, Bill)` et `(3, Bob, Joe)` dans `personne`
  - `(2, football), (2, box), (3, football), (3, tennis)` et `(3, box)` dans `personne_sport`

Hibernate

Si la classe association est porteuse de données

- Par exemple : la relation (ArticleCommande) entre Commande et Article
- Il faut préciser la quantité de chaque article dans une commande

Hibernate

Si la classe association est porteuse de données

- Par exemple : la relation (ArticleCommande) entre Commande et Article
- Il faut préciser la quantité de chaque article dans une commande

Solution

- Créer trois entités `Article`, `Commande` et `ArticleCommande`
- Définir la relation `OneToMany` entre `Article` et `ArticleCommande`
- Définir la relation `ManyToOne` entre `ArticleCommande` et `Commande`

Hibernate

Remarques

- Les relations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `personne.getSports();`
- Mais on ne peut faire `sport.getPersonnes();`

Hibernate

Remarques

- Les relations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `personne.getSports();`
- Mais on ne peut faire `sport.getPersonnes();`

Solution

Rendre les relations bidirectionnelles

Hibernate

Modifier l'entité inverse `Sport`

```
@Entity
public class Sport implements Serializable {

  @Id
  private String nom;
  private String type;

  @ManyToMany(mappedBy="sports")
  private List <Personne> personnes = new ArrayList <Personne>
     ();
  // ajouter les getter, setter, add et remove
}
```

`mappedBy`

- fait référence à l'attribut `sports` de la classe `Personne`

Hibernate

Nouveau code des méthodes `addSport` et `removeSport` de la classe `Personne`

```
public void addSport(Sport sport) {
  sports.add(sport);
  sport.getPersonnes().add(this);
}

public void removeSport(Sport sport) {
  sports.remove(sport);
  sport.getPersonnes().remove(this);
}
```

Hibernate

Nouveau code des méthodes `addSport` et `removeSport` de la classe `Personne`

```
public void addSport(Sport sport) {
  sports.add(sport);
  sport.getPersonnes().add(this);
}

public void removeSport(Sport sport) {
  sports.remove(sport);
  sport.getPersonnes().remove(this);
}
```

Nouveau code des méthodes `addPersonne` et `removePersonne` de la classe `Sport`

```
public void addPersonne(Personne personne) {
  personnes.add(personne);
  personne.getSports().add(this);
}

public void removePersonne(Personne personne) {
  personnes.remove(personne);
  personne.getSports().add(this);
}
```

Hibernate

Ainsi, on peut faire : (Supprimer et recréer la base de données)

```
// tout le code précédent du main +
for (Personne personne : s1.getPersonnes())
  System.out.println(personne.getNom());
```

Hibernate

Ainsi, on peut faire : (Supprimer et recréer la base de données)

```
// tout le code précédent du main +
for (Personne personne : s1.getPersonnes())
  System.out.println(personne.getNom());
```

affiche

```
Voight
Bob
```

Hibernate

Pour définir une relation bidirectionnelle entre deux entités

- si dans l'entité propriétaire la relation définie est `OneToMany`, alors dans l'entité inverse la relation sera `ManyToOne`, et inversement.
- si dans l'entité propriétaire la relation définie est `OneToOne`, alors dans l'entité inverse la relation sera aussi `OneToOne`.
- si dans l'entité propriétaire la relation définie est `ManyToMany`, alors dans l'entité inverse la relation sera aussi `ManyToMany`.

Hibernate

Trois possibilités avec l'héritage

- `SINGLE_TABLE`
- `TABLE_PER_CLASS`
- `JOINED`

Hibernate

Trois possibilités avec l'héritage

- `SINGLE_TABLE`
- `TABLE_PER_CLASS`
- `JOINED`

Exemple

- Une classe mère `Personne`
- Deux classes filles `Etudiant` et `Enseignant`

Hibernate

Pour indiquer comment transformer les classes mère et filles en tables

Il faut utiliser l'annotation `@Inheritance`

Hibernate

Tout dans une seule table

Dans la classe mère on ajoute

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

Hibernate

Exemple

Et pour distinguer un étudiant d'un enseignant, d'une personne

- `@DiscriminatorColumn(name="TYPE_PERSONNE")` dans la classe mère,
- `@DiscriminatorValue(value="PERS")` dans la classe `Personne`,
- `@DiscriminatorValue(value="ETU")` dans la classe `Etudiant` et
- `@DiscriminatorValue(value="ENS")` dans la classe `Enseignant`.

Dans la table `personne`, on aura une colonne `TYPE_PERSONNE` qui aura comme valeur soit `PERS`, soit `ETU` soit `ENS`.

Hibernate

La classe `Personne`

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_PERSONNE")
@DiscriminatorValue(value="PERS")
public class Personne {

  // + tout le code précédent

}
```

La classe `Etudiant`

```
@Entity
@DiscriminatorValue(value="ETU")
public class Etudiant extends Personne {

  private String niveau;

  public String getNiveau() {
    return niveau;
  }

  public void setNiveau(String niveau) {
    this.niveau = niveau;
  }
}
```

La classe `Enseignant`

```
@Entity
@DiscriminatorValue(value="ENS")
public class Enseignant extends Personne {

  private int salaire;

  public int getSalaire() {
    return salaire;
  }

  public void setSalaire(int salaire) {
    this.salaire = salaire;
  }
}
```

Hibernate

Et pour tester

```
/* Personne */
Personne personne = new Personne();
personne.setNom("Guardiola");
personne.setPrenom("Pep");
/* Enseignant */
Enseignant enseignant = new Enseignant();
enseignant.setNom("Ferguson");
enseignant.setPrenom("Sir");
enseignant.setSalaire(10000);
/* Étudiant */
Etudiant etudiant = new Etudiant();
etudiant.setNom("Mourinho");
etudiant.setPrenom("Jose");
etudiant.setNiveau("Ligue 1");
/* Persistance */
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
session.persist(personne);
session.persist(etudiant);
session.persist(enseignant);
transaction.commit();
session.close();
sessionFactory.close();
```

N'oublions pas de déclarer les entités `Etudiant` **et** `Enseignant` **dans** `hibernate.cfg.xml`

Hibernate

Constats

- Trois colonnes ajoutées à la table `personne` :
  - `TYPR_PERSONNE`
  - `niveau`
  - `salaire`
- Trois tuples ajoutés avec les valeurs suivantes :

TYPE_PERSONNE	num	nom	prenom	niveau	salaire
PERS	1	Guardiola	Pep	NULL	NULL
ETU	2	Mourinho	Jose	Ligue 1	NULL
ENS	3	Ferguson	Sir	NULL	10000

Hibernate

Exemple avec une table pour chaque entité

- `@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)` :
  Chaque entité sera transformée en table.

Hibernate

Les classes incorporables

- Pas de table correspondante dans la base de données
- Utilisée généralement par des entités

Hibernate

Et si on déplace les attributs `nom` et `prénom` dans une nouvelle classe `NomComplet`

```
@Embeddable
public class NomComplet {
    private String nom;
    private String prenom;
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    @Override
    public String toString() {
    return "NomComplet [nom=" + nom + ", prenom=" + prenom + "]";
    }
}
```

Hibernate

L'entité `Personne` devient ainsi :

```
@Entity
public class Personne {

    @Id
    @GeneratedValue (strategy=GenerationType.IDENTITY)
    private int num;

    private NomComplet nomComplet;

    public int getNum() {
        return num;
    }
    public void setNum(int num) {
        this.num = num;
    }
    public NomComplet getNomComplet() {
        return nomComplet;
    }
    public void setNomComplet(NomComplet nomComplet) {
        this.nomComplet = nomComplet;
    }
```

Hibernate

Les classes incorporables : pas de table correspondante en BD

```
Personne personne = new Personne();
NomComplet nomComplet = new NomComplet();
nomComplet.setNom("travolta");
nomComplet.setPrenom("john");
personne.setNomComplet(nomComplet);
Transaction transaction = session.beginTransaction();
transaction.begin();
session.persist(personne);
transaction.commit();
```

Hibernate

Les classes incorporables : pas de table correspondante en BD

```
Personne personne = new Personne();
NomComplet nomComplet = new NomComplet();
nomComplet.setNom("travolta");
nomComplet.setPrenom("john");
personne.setNomComplet(nomComplet);
Transaction transaction = session.beginTransaction();
transaction.begin();
session.persist(personne);
transaction.commit();
```

Il n'y aura pas de table `NomComplet`, les attributs `nom` et `prénom` seront transformés en colonne dans la table `Personne`

Hibernate

Cycle de vie d'une entité

Le cycle de vie de chaque objet d'une entité JPA passe par trois événements principaux

- création (avec `persist()`)
- mise à jour (avec `flush()`)
- suppression (avec `remove()`)

Hibernate

Une méthode `callback`

- Une méthode `callback` est une méthode qui sera appelée avant ou après un évènement survenu sur une entité
- On utilise les annotations pour spécifier quand la méthode callback sera appelée

Hibernate

Une méthode `callback`

- Une méthode `callback` est une méthode qui sera appelée avant ou après un évènement survenu sur une entité
- On utilise les annotations pour spécifier quand la méthode callback sera appelée

C'est comme les triggers en SQL

Hibernate

- `@PrePersist` : avant qu'une nouvelle entité soit persistée.
- `@PostPersist` : après l'enregistrement de l'entité dans la base de données.
- `@PostLoad` : après le chargement d'une entité de la base de données.
- `@PreUpdate` : avant que la modification d'une entité soit enregistrée en base de données.
- `@PostUpdate` : après que la modification d'une entité est enregistrée en base de données.
- `@PreRemove` : avant qu'une entité soit supprimée de la base de donnée.
- `@PostRemove` : après qu'une entité est supprimée de la base de donnée.

Hibernate

Exemple : l'entité `Personne`

```
public class Personne implements Serializable {

  @Id
  private int num;
  private String nom;
  private String prenom;
  private int nbrMAJ=0; // pour calculer le nombre de
     modification
  public int getNbrMAJ() {
    return nbrMAJ;
  }
  public void setNbrMAJ(int nbrMAJ) {
    this.nbrMAJ = nbrMAJ;
  }
  @PostUpdate
  public void updateNbrMAJ() {
    this.nbrMAJ++;
  }
  // les autres getters, setters et constructeur
```

Hibernate

```
Personne p1 = new Personne();
p1.setNom("Wick");
p1.setPrenom("John");
session.getTransaction().begin();
session.persist(p1);
session.getTransaction().commit();
System.out.println("nbrMAJ = " + p1.getNbrMAJ());
// affiche nbrMAJ = 0
p1.setNom("Travolta");
session.getTransaction().begin();
session.flush();
session.getTransaction().commit();
p1.setNom("Abruzzi");
session.getTransaction().begin();
session.flush();
session.getTransaction().commit();
System.out.println("nbrMAJ = " + p1.getNbrMAJ());
// affiche nbrMAJ = 2
```

Hibernate

Organisation du code

- Créer une classe `HibernateUtil` qui se charge de lire le fichier `hibernate.cfg.xml` et de construire un objet de `SessionFactory`
- Créer une deuxième classe générique `GenericDao` qui récupère l'objet de `SessionFactory` construit par `HibernateUtil` et l'utilise pour faire le CRUD pour l'entité passé comme paramètre générique
- Pour chaque entité, on crée une classe `Dao` qui étend `GenericDao`

Hibernate

La classe `HibernateUtil`

```
package org.eclipse.config;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

  private static  SessionFactory sessionFactory;

  public static SessionFactory getSessionFactory() {

    Configuration configuration = new Configuration().configure
       ();
    return configuration.buildSessionFactory();

  }
}
```

La classe `GenericDao`

```
package org.eclipse.dao;

import java.util.List;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class GenericDao <T,P> {

        protected Session session;
        private Class<T> entity;

        public GenericDao(Class<T> entity, Session session) {
                this.session = session;
                this.entity = entity;
        }
        public Session getSession() {
                return session;
        }
        public P save(T obj) throws Exception {
                Transaction tx = null;
                P result;
                try {
                        tx = session.beginTransaction();
                        result = (P) session.save(obj);
                        tx.commit();

                } catch (Exception e) {
                        if (tx != null)
                                tx.rollback();
                        throw e;
                }
                return result;
        }
```

La classe `GenericDao` (suite)

```
        public void update(T obj) throws Exception {
                Transaction tx = null;
                try {
                        tx = session.beginTransaction();
                        session.saveOrUpdate(obj);
                        tx.commit();
                } catch (Exception e) {
                        if (tx != null)
                                tx.rollback();
                        throw e;
                }
        }
        public void delete(T obj) throws Exception {
                Transaction tx = null;
                try {
                        tx = session.beginTransaction();
                        session.delete(obj);
                        tx.commit();
                } catch (Exception e) {
                        if (tx != null)
                                tx.rollback();
                        throw e;
                }
        }
        public T findById(int id) {
                return session.get(entity, id);
        }
        public List<T> findAll() {
                return (List<T>) session.createQuery("from " + entity.getName()).list();
        }
}
```

Pour les clés primaires des entités, on utilise que les types objets

```
public class Personne {
  private Integer num;
  private String nom;
  private String prenom;

  public Integer getNum() {
    return num;
  }
  public void setNum(Integer num) {
    this.num = num;
  }
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

Hibernate

La classe `PersonneDao`

```
package org.eclipse.dao;

import org.eclipse.model.Personne;
import org.hibernate.Session;

public class PersonneDao extends GenericDao<Personne,
   Integer> {

        public PersonneDao(Session session) {
                super(Personne.class, session);
        }
}
```

Hibernate

Pour tester tout ça dans le `main` de `App.java`

```
package org.eclipse;

import org.eclipse.config.HibernateUtil;
import org.eclipse.dao.PersonneDao;
import org.eclipse.model.Personne;
import org.hibernate.Session;

public class App {
  public static void main(String[] args) throws Exception {
    Session session = HibernateUtil.getSessionFactory().openSession();
    Personne personne = new Personne();
    personne.setNom("Ferdinand");
    personne.setPrenom("Rio");
    PersonneDao personneDao = new PersonneDao(session);
    int cle = personneDao.save(personne);
    System.out.println(cle);
    Personne personne2 = personneDao.findById(3);
    personne2.setNom("Turing");
    personneDao.saveOrUpdate(personne2);
  }
}
```