Persistance de données : **JPA** avec **Hibernate** 6

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

Plan

Plan

Plan

Limites de l'API **JDBC**

- Nécessité de formuler des requêtes **SQL**
- Logiques relationnelle et objet dans un même fichier
- Le développeur doit tout écrire et gérer : ouverture et fermeture de connexion, gestion de transaction, préparation des requêtes pour faire le **CRUD** : (`Create`, `Read`, `Update`, et `Delete`)...
- Récupération de résultats d'une requête `SELECT` dans un objet `ResultSet` : pour le parcourir, il faut naviguer selon les lignes et les colonnes, case par case.

Limites de l'API **JDBC**

- Nécessité de formuler des requêtes **SQL**
- Logiques relationnelle et objet dans un même fichier
- Le développeur doit tout écrire et gérer : ouverture et fermeture de connexion, gestion de transaction, préparation des requêtes pour faire le **CRUD** : (`Create`, `Read`, `Update`, et `Delete`)...
- Récupération de résultats d'une requête `SELECT` dans un objet `ResultSet` : pour le parcourir, il faut naviguer selon les lignes et les colonnes, case par case.

Une meilleure solution ?

Utiliser un **ORM**...

ORM : **O**bject-**R**elational **M**apping (lien objet-relationnel)

- Couche d'abstraction à la base de données
- Permettant au développeur d'utiliser les tables d'une base de données comme des objets
- Consistant à associer :
  - une ou plusieurs classes à chaque table
  - un attribut de classe à chaque colonne de la table

ORM : avantages

- Plus besoin d'écrire de requête **SQL**
- Suppression d'une très grande partie du code nécessaire avec **JDBC**
  - Gain de temps
  - Simplification du code source
- Code portable indépendant vis-à-vis de **SGBD**

ORM : avantages

- Plus besoin d'écrire de requête **SQL**
- Suppression d'une très grande partie du code nécessaire avec **JDBC**
  - Gain de temps
  - Simplification du code source
- Code portable indépendant vis-à-vis de **SGBD**

ORM : inconvénients

- Complexité (d'apprentissage)
- Problème d'optimalité pour les requêtes complexes modifiant plusieurs tuples

Et **JPA** ?

- **J**ava **P**ersistence **A**PI (ou **J**akarta **P**ersistence **A**PI depuis 2019)
- Proposé par **JSR 220** (**J**ava **S**pecification **R**equests)
- Ce n'est ni **ORM** ni **Framework**
- Spécification : ensemble d'interfaces standardisé par **Oracle**
  - à implémenter par les frameworks **ORM**
  - permettant l'organisation de données
- **JPA** a pris en compte les particularités de la plupart des **ORM** pour standardiser la persistance avec **Java**.

Éléments **JPA**

- Annotations pour définir le lien entre `Entity` et table
- Gestionnaire d'entité (`EntityManager` en anglais) pour persister les données
- Langage de requête adapté aux classes (**JPQL** : **Java** **P**ersistence **Q**uery **L**anguage)

Remarque

JPA peut être vu comme une interface (au sens programmation orientée-objet) qui nécessite d'être implémenté.

Remarque

JPA peut être vu comme une interface (au sens programmation orientée-objet) qui nécessite d'être implémenté.

Quelques frameworks **ORM** implémentant les spécifications **JPA**

- **Hibernate** (**Red Hat**)
- EclipseLink (**Eclipse Foundation**)
- TopLink (**Oracle**)
- Java Data Objects (JDO)
- ...

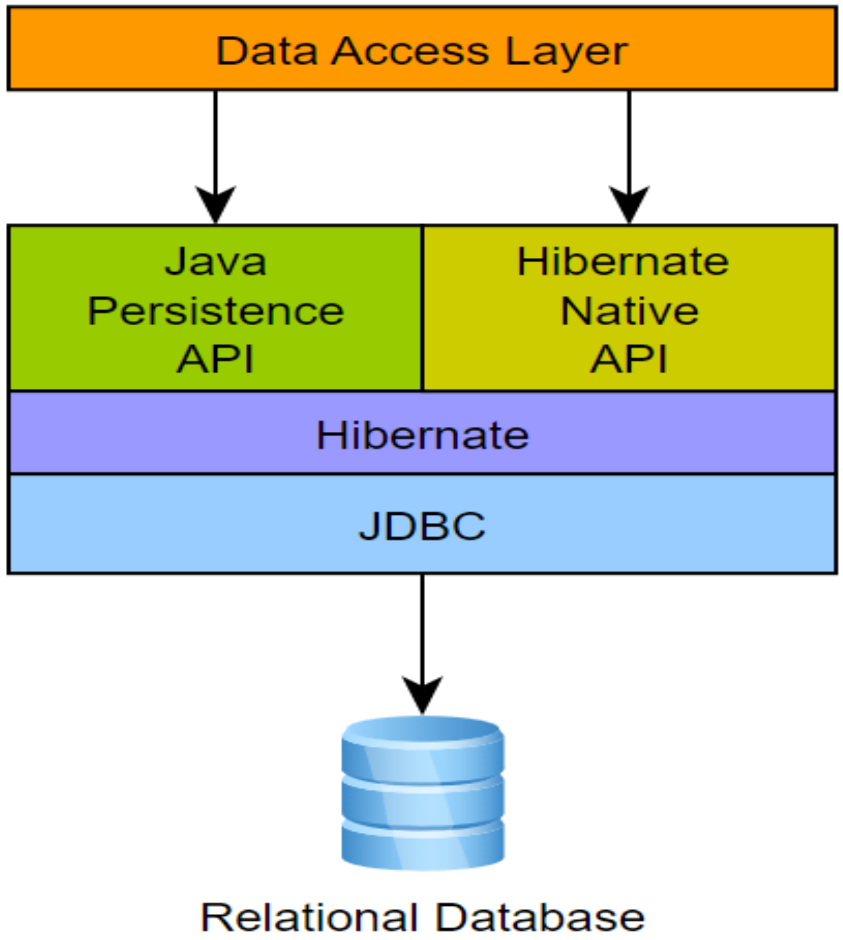
Hibernate

- Framework ORM (le premier) pour **Java** implémentant les spécifications **JPA**
- Open-source et écrit en **Java**
- Créé par **Red Hat** (Entreprise productrice de plusieurs logiciels open-source comme le serveur d'application **JBoss**)
- Possédant une extension **NHibernate** pour la plateforme **.NET** (de **Microsoft**)
- Ayant inspiré plusieurs autres frameworks **ORM** comme **Doctrine** pour **PHP**
- Pouvant être utilisé dans tout type de projet **Java** (Console, Web, Client lourd...)

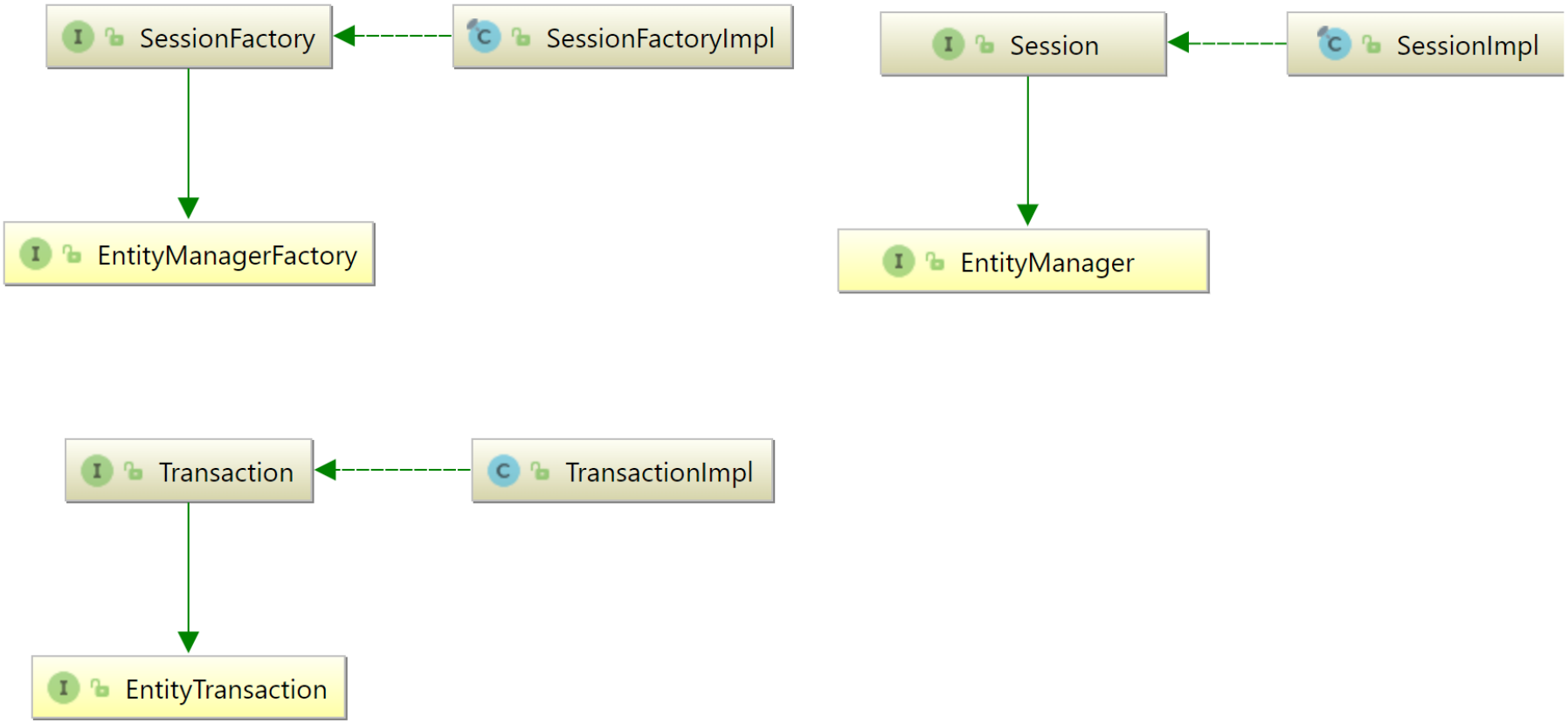
Hibernate vs JPA

- **Hibernate** : Framework ORM qui existe depuis mai 2001.
- **JPA** : Spécification publiée en 2006 pour uniformiser la persistance de données en **Java**.
- **Hibernate** existe bien avant **JPA** et il avait ses propres classes, méthodes et annotations pour la persistance de données : **Hibernate Native API**.
- Depuis la publication de **JPA**, **Hibernate** a dû implémenter les spécifications **JPA**.
- Plusieurs opérations sont réalisables avec des méthodes natives de **Hibernate** ou d'autres imposées par **JPA**.

Hibernate



Source : *documentation officielle*



Source : *documentation officielle*

Hibernate

Pour persister des données : deux solutions possibles

1. En utilisant les classes/interfaces/méthodes de **JPA**
2. En utilisant les classes/interfaces/méthodes natives de **Hibernate** (et qui héritent/implémentent les classes/interfaces **JPA**)

Hibernate

Quelques classes **JPA**

- `Persistence`
- `EntityManagerFactory`
- `EntityManager`
- `EntityTransaction`
- ...

Quelques classes **Hibernate**

- `Configuration`
- `SessionFactory`
- `Session`
- `Transaction`
- ...

Quelques classes **JPA**

- `Persistence`
- `EntityManagerFactory`
- `EntityManager`
- `EntityTransaction`
- ...

Quelques classes **Hibernate**

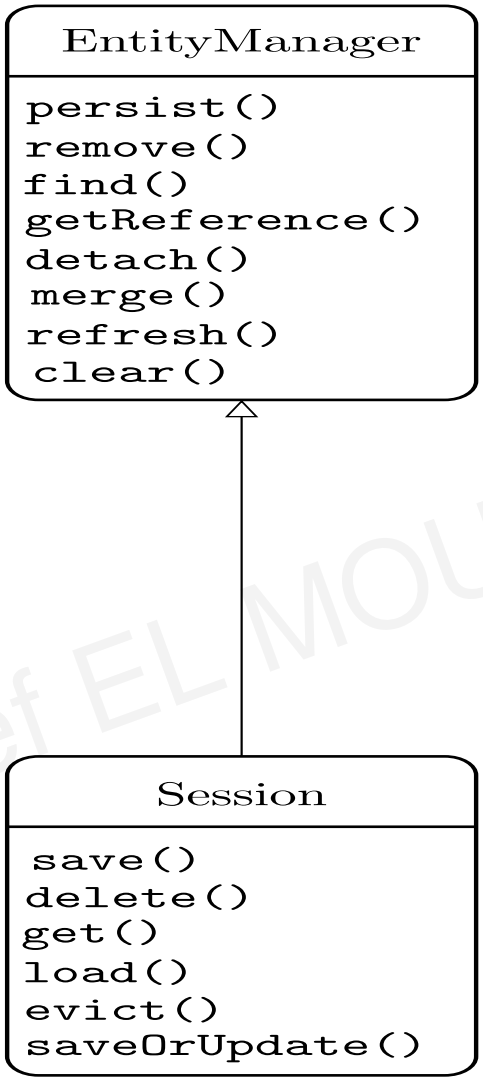
- `Configuration`
- `SessionFactory`
- `Session`
- `Transaction`
- ...

Données de configuration de **JPA**

`Persistence.xml` : à placer dans `src/main/resources/META-INF`

Données de configuration de **Hibernate**

`hibernate.cfg.xml` : : à placer dans `src/main/java`



Répartition de quelques méthodes de persistance JPA/Hibernate

JDK 8 : téléchargement

https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

JDK 11 : téléchargement

https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html

JDK 17 : téléchargement

https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html

Eclipse : téléchargement

`https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2022-09/R/eclipse-inst-jre-win64.exe`

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**

Avant de commencer, installer **MySQL** (s'il n'est pas installé)

- Aller à `https://dev.mysql.com/downloads/mysql/` et choisir la version à télécharger selon le système d'exploitation
- Lancer l'installation du fichier MSI sous Windows (fichier pkg de l'archive DMG sous MAC)

Télécharger **JDBC**

- Aller sur `https://dev.mysql.com/downloads/file/?id=480292`
- Télécharger et Décompresser l'archive `.zip`

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 (par exemple `mysql_hibernate`) à cette connexion dans `Name`
- Cliquer sur `New Driver Definition` devant la liste déroulante de `Drivers`

Spécification du driver

- Choisir le dernier `MySQL JDBC DRIVER`
- Dans la rubrique `JAR List`, préciser son emplacement (chemin vers le Driver téléchargé et décompressé) et supprimer l'existant
- Dans la rubrique `Properties`, modifier les données de connexion
  - `jdbc:mysql://localhost:3306/cours_hibernate` dans `Connection URL`,
  - `hibernate` dans `DataBase Name`
  - `com.mysql.cj.jdbc.Driver` dans `Driver class`
  - `root` dans `Password`
  - `root` dans `User ID`
- Valider puis cliquer sur `Test Connection`
- Cliquer sur `Finish`

Tester (démarrer) la connexion (deuxième méthode)

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

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

Créer un projet **Maven**

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

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

```
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-
   core -->
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.6.13.Final</version>
</dependency>
```

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

```
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-
   core -->
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.6.13.Final</version>
</dependency>
```

Ajouter une deuxième pour le driver de MySQL

```
<!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.0.0</version>
</dependency>
```

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

```
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-
   core -->
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.6.13.Final</version>
</dependency>
```

Ajouter une deuxième pour le driver de MySQL

```
<!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.0.0</version>
</dependency>
```

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

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`

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

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`

Si on veut remplir le formulaire champ par champ

- Choisir **MySQL** de la liste `Database dialect`
- Sélectionner le driver **MySQL**, `com.mysql.cj.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/cours_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

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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
 </session-factory>
</hibernate-configuration>
```

Pour autoriser Hibernate à créer la base de données si elle n'existe pas, on ajoute

```
<?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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
 </session-factory>
</hibernate-configuration>
```

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).

Pour afficher les requêtes **SQL** exécutées par **Hibernate** dans la console

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

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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
 </session-factory>
</hibernate-configuration>
```

Pour bien formater les requêtes SQL affichées

```
<?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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <property name="format_sql">true</property>
 </session-factory>
</hibernate-configuration>
```

POJO

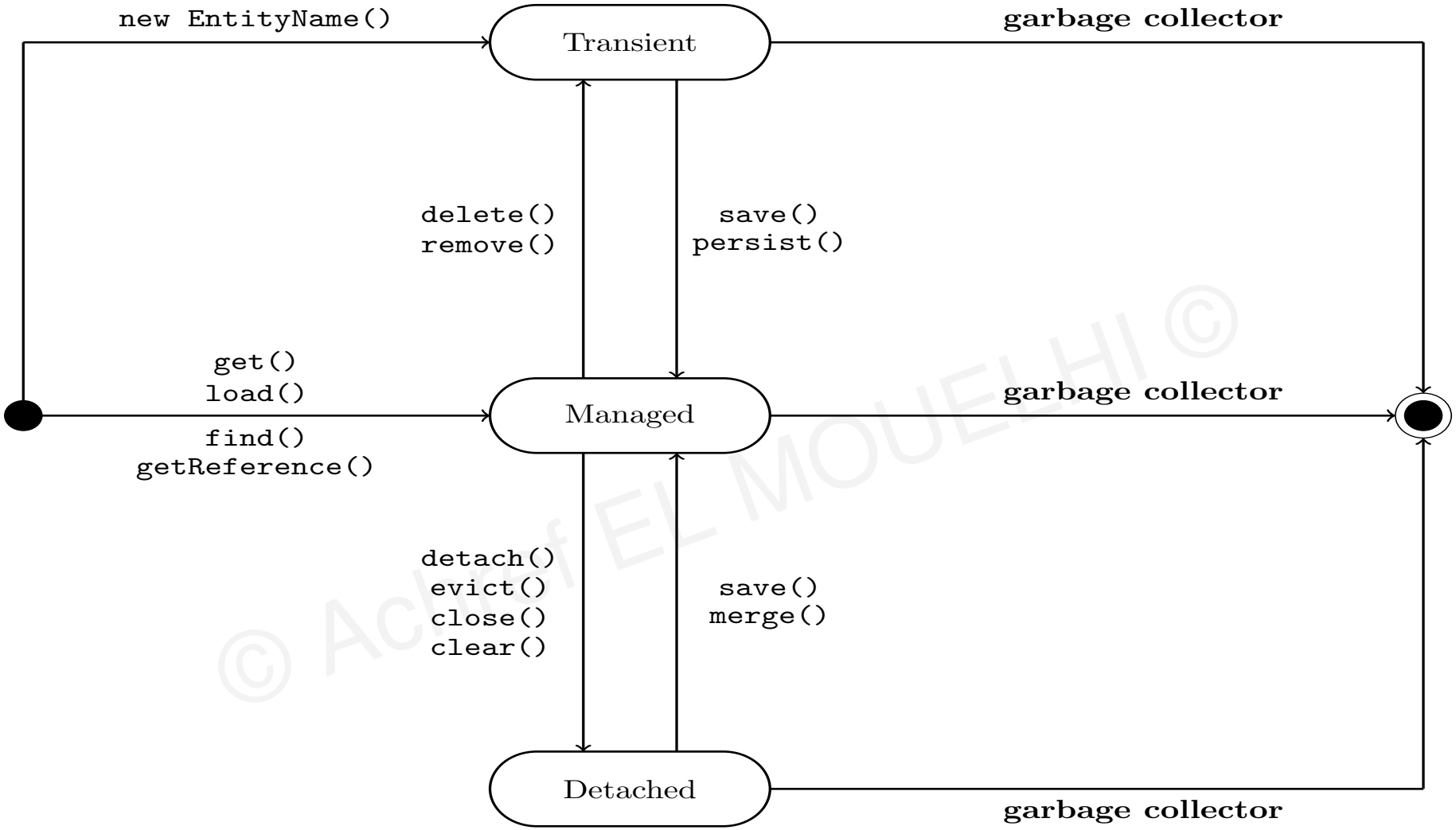
- **P**lain **O**ld **J**ava **O**bject : traduit en français en `bon vieil objet Java`
- Terme inventé par **Rebecca Parsons** et **Josh MacKenzie** en septembre 2000
- Faisant référence aux classes :
  - Avec un constructeur sans paramètres
  - Pouvant contenir d'autres constructeurs
  - Ayant un getter et setter pour chaque attribut privé ou protégé
  - N'implémentant pas une interface prédéfinie (`EventListener`...)
  - Ne héritant pas d'une classe prédéfinie (`HttpServlet`...)
- L'équivalent en **.NET** est **POCO** (**P**lain **O**ld **C**LR **O**bject), en **PHP** est **POPO** (**P**lain **O**ld **P**HP **O**bject).

Entité

- Représentant d'une (ou plusieurs) table·s d'une base de données relationnelle dans le modèle objet.
- Classe Persistante : **POJO** + méta-données indispensables pour le mapping
- Ne pouvant pas être une interface ou une énumération
- Pouvant être une classe concrète ou abstraite
- Pouvant étendre des classes non-entités ou des classes entités

Récapitulons

- Une entité $\simeq$ une table
- Un attribut simple $\simeq$ une colonne
- Un attribut objet ou une collection $\simeq$ une table + une clé étrangère
- Une clé primaire avec méta-donné `id` $\simeq$ clé primaire



Cycle de vie d'une entité JPA

Hibernate propose deux solutions pour les méta-données indispensables

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

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

```
public class Personne {

  private int num;
  private String nom;
  private String prenom;

  // Ajouter un constructeur sans paramètre et un avec tous les paramètres sauf num

  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;
  }

  // et un toString
}
```

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

- Faire un clic droit sur la classe `Personne` et aller dans `New > Other`
- Chercher `Hibernate`
- Sélectionner `Hibernate XML Mapping File (hbm.xml)`
- Cliquer sur `Next` puis valider et aller vérifier les données dans le fichier **XML** généré

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">
<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.

Déclarons `Personne.hbm.xml` **dans** `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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <property name="format_sql">true</property>
  <mapping resource="org/eclipse/model/Personne.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

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 si elle n'est pas déclarée dans `hibernate.cfg.xml`
- Créer une usine de gestionnaire d'entité (appelée `EntityManagerFactory` par **JPA** et `SessionFactory` par **Hibernate**)
- Créer un gestionnaire d'entité (appelé `EntityManager` par **JPA** et `Session` par **Hibernate**)
- [Démarrer une transaction]
- Utiliser le gestionnaire de données pour persister les données
- [Terminer la transaction et] fermer les différents flux

Pour créer un objet de configuration

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

Pour créer un objet de configuration

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

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

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

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 `hibernate.cfg.xml` **est situé dans un autre emplacement (par exemple, dans** `org.eclipse.model`**), il faut préciser son chemin**

```
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 `hibernate.cfg.xml` est situé dans un autre emplacement (par exemple, dans `org.eclipse.model`), il faut préciser son chemin

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

On peut fusionner les deux étapes précédentes

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

Les données de la configuration peuvent aussi être chargées sans passer par le fichier `hibernate.cfg.xml`

```
Configuration configuration = new Configuration()
    .setProperty("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver")
    .setProperty("hibernate.connection.username", "root")
    .setProperty("hibernate.connection.password", "root")
    .setProperty("hibernate.connection.url", "jdbc:mysql://localhost:3306/
      cours_hibernate?createDatabaseIfNotExist=true")
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect")
    .setProperty("hibernate.show_sql", "true")
    .setProperty(Environment.HBM2DDL_AUTO, "update");

configuration.addResource("org/eclipse/model/Personne.hbm.xml");

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

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

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

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();
```

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();
```

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.save(personne);
```

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.save(personne);
```

Terminer la transaction et fermer les flux

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

Plaçons toutes les instructions précédentes dans le `main` de `App.java`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class App {
 public static void main( String[] args ) {
   Personne personne = new Personne("Wick", "John");
   Configuration configuration = new Configuration().configure();
   SessionFactory sessionFactory = configuration.buildSessionFactory();
   Session session = sessionFactory.openSession();
   Transaction transaction = session.beginTransaction();

   session.save(personne);

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

Constats

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

Hibernate avec annotations **JPA**

- Dans `hibernate.cfg.xml`,
  remplacer `<mapping resource="org/eclipse/model/Personne.hbm.xml"/>`
  par `<mapping class="org.eclipse.model.Personne" />`

- Supprimer la base de données `cours_hibernate`

Ajoutons les annotations suivantes dans la classe `Personne`

```
package org.eclipse.model;

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

@Entity
public class Personne {

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

  // + constructeurs + getters + setters + toString

}
```

Re-testons le `main` précédent

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class App {
 public static void main( String[] args ) {
   Personne personne = new Personne("Wick", "John");
   Configuration configuration = new Configuration().configure();
   SessionFactory sessionFactory = configuration.buildSessionFactory();
   Session session = sessionFactory.openSession();
   Transaction transaction = session.beginTransaction();

   session.save(personne);

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

Hibernate

On peut remplacer `<mapping class="org.eclipse.model.Personne" />` **de** `hibernate.cfg.xml` **par** `configuration.addAnnotatedClass(Personne.class)`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne personne = new Personne("Wick", "John");
        Configuration configuration = new Configuration().configure();
        configuration.addAnnotatedClass(Personne.class);
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        session.save(personne);

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

Remarque

Dans la suite de ce cours, nous continuons à déclarer les entitées annotées dans `hibernate.cfg.xml`.

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

```
Personne personne = new Personne("Wick", "John");
Configuration configuration = new Configuration().configure();
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Integer cle = (Integer) session.save(personne);

transaction.commit();
session.close();
sessionFactory.close();
System.out.println(cle);
```

Remarque

Par défaut, avec **Hibernate**, il faut exécuter toutes les opérations d'écriture en mode transaction.

Pour permettre d'exécuter les opérations d'écriture hors transaction, il daut ajouter la propriété suivante dans `hibernate.cfg.xml`

```
<property name="hibernate.allow_update_outside_transaction">
    true
</property>
```

Et pour tester (n'oublions pas d'ajouter `flush`)

```
Personne personne = new Personne("Maggio", "Sophie");
Configuration configuration = new Configuration().configure();
configuration.addAnnotatedClass(Personne.class);
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();

session.save(personne);
session.flush();

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

Pour chercher une personne (avec `load`)

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

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

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

Ou

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

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

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

Pour chercher une personne (avec `get`)

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

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

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

`get` vs `load` [dépréciée depuis **Hibernate 6**]

- Les deux permettent la recherche selon l'identifiant
- 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` lance une `ObjectNotFoundException`

Vérifiez que la requête de `get` a été exécutée mais pas celle de `load`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

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

        Personne personne1 = session.load(Personne.class, 1);
        System.out.println("*****************************************");
        Personne personne2 = session.get(Personne.class, 1);
        session.close();
        sessionFactory.close();
    }
}
```

Pour chercher une personne (avec `load`)

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

List<Personne> personnes = session
    .byMultipleIds(Personne.class)
    .multiLoad(1, 2, 3);
personnes.forEach(System.out::println);

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

Pour modifier une personne

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

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

session.flush();

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

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

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

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

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

session.persist(personne3);

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

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();

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

session.save(personne4);

8session.close();
sessionFactory.close();
```

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

Pour supprimer une personne, on peut utiliser `delete` [dépréciée depuis Hibernate 6]

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

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

session.delete(personne3);
session.flush();

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

Remarque

- Depuis **Hibernate 6**, la classe `Criteria` a été supprimée.
- La sélection se fait avec les `createQuery` (à voir dans la section **SQL et HQL**).

Commençons par définir un fichier `Persistence.xml` **dans** `src/main/resources/META-INF`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns
     /persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="jpa-config">
    <description>Exemple avec Hibernate et JPA </description>
    <class>org.eclipse.model.Personne</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"
         />
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/
        cours_hibernate?createDatabaseIfNotExist=true" />
      <property name="jakarta.persistence.jdbc.user" value="root" />
      <property name="jakarta.persistence.jdbc.password" value="" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

Commençons par créer un objet de configuration et charger les données

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
```

Commençons par créer un objet de configuration et charger les données

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
```

Ensuite, obtenir le gestionnaire d'entité

```
EntityManager entityManager = emf.createEntityManager();
```

Puis, créer une transaction

```
EntityTransaction transaction = entityManager.getTransaction();
```

Puis, créer une transaction

```
EntityTransaction transaction = entityManager.getTransaction();
```

Et la démarrer

```
transaction.begin();
```

Puis, créer une transaction

```
EntityTransaction transaction = entityManager.getTransaction();
```

Et la démarrer

```
transaction.begin();
```

Insérer un objet dans la base de données

```
entityManager.persist(personne);
```

Puis, créer une transaction

```
EntityTransaction transaction = entityManager.getTransaction();
```

Et la démarrer

```
transaction.begin();
```

Insérer un objet dans la base de données

```
entityManager.persist(personne);
```

Terminer la transaction et fermer les flux

```
transaction.commit();
entityManager.close();
```

Plaçons toutes les instructions précédentes dans `main` de `Main.java`

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
        EntityManager entityManager = emf.createEntityManager();
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        Personne personne = new Personne("Dalton", "Jack");
        entityManager.persist(personne);
        transaction.commit();

        System.out.println(personne);
        entityManager.close();
    }
}
```

Remarques

- La méthode `save` de la classe `Session` de **Hibernate** n'existe pas dans `EntityManager` de **JPA**.
- La méthode `persist` de la classe `EntityManager` de **JPA** est implémentée par la classe `Session` de **Hibernate**.

`save` [dépréciée depuis **Hibernate 6**] 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

On peut chercher une personne selon l'identifiant avec la méthode `find`

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
        EntityManager entityManager = emf.createEntityManager();
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        Personne personne1 = entityManager.find(Personne.class, 1);
        System.out.println(personne1);

        transaction.commit();
        entityManager.close();
    }

}
```

On peut aussi utiliser la méthode `getReference`

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {
       EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
       EntityManager entityManager = emf.createEntityManager();
       EntityTransaction transaction = entityManager.getTransaction();
       transaction.begin();

       Personne personne1 = entityManager.getReference(Personne.class, 1);
       System.out.println(personne1);

       transaction.commit();
       entityManager.close();
    }

}
```

`find` vs `getReference`

- Les deux permettent la recherche selon l'identifiant
- `find` récupère immédiatement l'objet de la base de données
- `getReference` récupère l'objet de la base de données lors de sa première utilisation
- Si l'objet recherché n'existe pas, `find` retourne `null` tandis que `getReference` lance une `EntityNotFoundException`

Vérifiez que la requête de `find` a été exécutée mais pas celle de `getReference`

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config")
           ;
        EntityManager entityManager = emf.createEntityManager();
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        Personne personne1 = entityManager.getReference(Personne.class, 1);
        System.out.println("****************************************");
        Personne personne2 = entityManager.find(Personne.class, 1);

        transaction.commit();
        entityManager.close();
    }

}
```

Pour modifier une personne

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
        EntityManager entityManager = emf.createEntityManager();
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        Personne personne1 = entityManager.getReference(Personne.class, 1);
        personne1.setNom("Abruzzi");
        entityManager.flush();

        transaction.commit();
        entityManager.close();
    }
}
```

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

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
        EntityManager entityManager = emf.createEntityManager();
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        Personne personne1 = entityManager.getReference(Personne.class, 1);
        personne1.setNom("Travolta");
        entityManager.persist(personne1);

        transaction.commit();
        entityManager.close();
    }
}
```

Pour supprimer une personne, on peut utiliser `remove`

```
package org.eclipse.main;

import org.eclipse.model.Personne;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class Main {

    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
        EntityManager entityManager = emf.createEntityManager();
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();

        Personne personne1 = entityManager.getReference(Personne.class, 1);
        entityManager.remove(personne1);

        transaction.commit();
        entityManager.close();
    }
}
```

Pour obtenir une `Session` depuis un `EntityManager`

```
package org.eclipse.main;

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

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

public class Main {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
        EntityManager entityManager = emf.createEntityManager();

        Session session = entityManager.unwrap(Session.class);

        Transaction transaction = session.beginTransaction();

        Personne personne = new Personne("Wick", "John");
        session.persist(personne);

        transaction.commit();

        session.close();
        sessionFactory.close();
        entityManager.close();
    }
}
```

Pour obtenir un `EntityManager` depuis une `Session`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy;
import org.hibernate.cfg.Configuration;

import jakarta.persistence.EntityManager;

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

        EntityManager em = session.getEntityManagerFactory().createEntityManager();

        Personne personne = new Personne("Dalton", "Jack");

        em.getTransaction().begin();
        em.persist(personne);
        em.getTransaction().commit();

        entityManager.close();
        session.close();
        sessionFactory.close();
    }
}
```

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)` [**Hibernate**] ou `detach(entity)` [**JPA**] : permet de détacher l'entité passée en paramètre (`entity`) de l'`EntityManager` qui la gère.
- `merge(entity)` : permet d'insérer l'entité passée en paramètre (`entity`) dans la base de données s'il n'y est pas, sinon elle le met à jour. Le paramètre (`entity`) est non persistant.

Exemple avec `refresh()`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {

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

        // on suppose que John Wick avec un num 1 existe dans la BD
        Personne p = session.find(Personne.class, 1)
        ;
        System.out.println(p.getNom());
        // affiche Wick

        p.setNom("Travolta");
        session.refresh(p);

        System.out.println(p.getNom());
        // affiche Wick

        session.close();
        sessionFactory.close();
    }
}
```

Exemple avec `detach()`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {

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

        // on suppose que John Wick avec un num 1 existe dans la BD
        Personne p = session.find(Personne.class, 1);
        System.out.println(p.getNom());
        // affiche Wick

        session.detach(p); // p n'est plus géré par session
        p.setNom("Travolta");
        session.flush();

        Personne p1 = session.find(Personne.class, 1);
        System.out.println(p1.getNom());
        // affiche Wick

        session.close();
        sessionFactory.close();
    }
}
```

Exemple avec `merge()`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        // on suppose que John Wick avec un num 1 existe dans la BD
        Personne p = new Personne(52, "Segal", "Steven");
        session.getTransaction().begin();
        session.merge(p);
        session.getTransaction().commit();
        Personne p1 = session.find(Personne.class, 52);
        System.out.println(p1.getNom());
        // affiche Segal
        p1.setNom("Austin");
        session.getTransaction().begin();
        session.merge(p1);
        session.getTransaction().commit();
        Personne p2 = session.find(Personne.class, 52);
        System.out.println(p2.getNom());
        // affiche Austin
        session.close();
        sessionFactory.close();
    }
}
```

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 contient une clé composée
`@GeneratedValue`	permet la génération de la clé pour les attributs annotés par `@Id`
`@Convert`	indique comment convertir la valeur avant de la persister
`@@Enumerated`	indique que les valeurs possibles sont définies dans une énumération

Classe `PersonnePK`

```
public class PersonnePK {

  @Id
  private String nom;

  @Id
  private String prenom;

  // + getters, setters,
     constructeur sans paramètres
     et constructeur avec deux
     paramètres nom et prénom
```

Classe `Personne`

```
@IdClass(PersonnePK.class)
@Entity
public class Personne  {

  @Id
  private String nom;

  @Id
  private String prenom;

  // + getters, setters et
     constructeur
```

Attributs de l'annotation `@Column`

Attribut	**désignation**
`name`	définit un nom de colonne 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)
...	...

Attributs de l'annotation `@Table`

Attribut	**désignation**
`name`	permet de donner un nom à la table différent de celui de l'entité
`uniqueConstraints`	définit une contrainte d'unicité sur un ensemble de colonnes

Attributs de l'annotation `@Table`

Attribut	désignation
`name`	permet de donner un nom à la table différent de celui de l'entité
`uniqueConstraints`	définit une contrainte d'unicité sur un ensemble de colonnes

Exemple

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

`@Transient`

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

Commençons par créer l'énumération `Genre`

```
package org.eclipse.enums;

public enum Genre {
    HOMME,
    FEMME
}
```

Modifions l'entité `Personne`

```java
package org.eclipse.model;

import org.eclipse.enums.Genre;
import org.hibernate.type.YesNoConverter;

import jakarta.persistence.Convert;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Personne {

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

    @Enumerated(EnumType.STRING)
    private Genre genre;

    @Convert(converter = YesNoConverter.class)
    private boolean donneurOrgane;

    // + constructeurs + getters + setters + toString

}
```

Remarque

- `EnumType.STRING` permet de persister une chaîne de caractère `HOMME` ou `FEMME`
- `EnumType.ORDINAL` permet de persister un entier `0` (pour `HOMME`) ou `1` (pour `FEMME`)

Et pour tester

```
package org.eclipse.main;

import org.eclipse.enums.Genre;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne personne = new Personne("Baggio", "Robert", Genre.HOMME, false);
        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();

        session.persist(personne);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

Annotations **Hibernate**

Annotation	**désignation**
`@Formula`	définit une colonne virtuelle calculée en fonction d'autres
`@NaturalId`	indique que l'attribut est unique
`@ColumnDefault`	indique la valeur par défaut de l'attribut
`@CurrentTimestamp`	permet d'initialiser un attribut de type date
`@ColumnTransformer`	transforme la valeur avant sa lecture et/ou son écriture

Exemple avec `@Formula` (de la documentation officielle)

```
@Entity
public static class Account {

        @Id
        private Long id;

        private Double credit;

        private Double rate;

        @Formula(value = "credit * rate")
        private Double interest;

        // + Getters and setters

}
```

Exemple avec `@Formula` (de la documentation officielle)

```
@Entity
public static class Account {

        @Id
        private Long id;

        private Double credit;

        private Double rate;

        @Formula(value = "credit * rate")
        private Double interest;

        // + Getters and setters

}
```

Il n'y aura pas de colonne `interest` en base de données.

Modifions l'entité `Personne`

```java
package org.eclipse.model;

import java.time.Instant;

import org.eclipse.enums.Genre;
import org.hibernate.annotations.CurrentTimestamp;
import org.hibernate.annotations.NaturalId;
import org.hibernate.tuple.GenerationTiming;
import org.hibernate.type.YesNoConverter;

import jakarta.persistence.Convert;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Personne {

    // les attributs précédents

    @CurrentTimestamp(timing = GenerationTiming.INSERT)
    public Instant createdAt;

    @NaturalId
    private int numCNI;

    // + constructeurs + getters + setters + toString

}
```

Remarque

- `GenerationTiming.INSERT` génère la valeur uniquement à l'insertion
- `GenerationTiming.ALWAYS` génère la valeur à l'ajout et la modification

Et pour tester

```
package org.eclipse.main;

import org.eclipse.enums.Genre;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne personne = new Personne("Baggio", "Robert", Genre.HOMME, false, 1000);
        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();

        session.persist(personne);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

Pour la suite, nous considérons le contenu suivant pour la classe `Personne`

```
package org.eclipse.model;

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

@Entity
public class Personne {

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

  // + constructeurs + getters + setters + toString

}
```

Remarque

Hibernate nous offre la possibilité d'exécuter des requêtes **SQL**

Remarque

Hibernate nous offre la possibilité d'exécuter des requêtes **SQL**

SQL

- **S**tructured **Q**uery **L**anguage
- Langage de requête basé sur l'algèbre relationnel

Remarque

Hibernate nous offre la possibilité d'exécuter des requêtes **SQL**

SQL

- **S**tructured **Q**uery **L**anguage
- Langage de requête basé sur l'algèbre relationnel

Pour conclure

SQL est un langage de requête qui nous permet de manipuler des tables d'une base de données relationnelle.

Remarque

JPA et **Hibernate** fournissent un langage de requête permettant de manipuler les entités : **JPQL** et **HQL**.

Remarque

JPA et **Hibernate** fournissent un langage de requête permettant de manipuler les entités : **JPQL** et **HQL**.

JPQL et **HQL**

- **JPQL** : **JP**A **Q**uery **L**anguage
- **HQL** : **H**ibernate **Q**uery **L**anguage

HQL

- Langage de requêtes pour les entités **JPA** définies par **Hibernate**
- Inspiré du langage **SQL** et adapté aux entités **JPA**
- Permettant de manipuler des entités et pas les tables
- Implémentant **JPQL** :
  - Une requête **JPQL** est toujours une requête **HQL** valide.
  - La réciproque est fausse.

JPQL

- Supportant `SELECT`, `UPDATE` et `DELETE`
- Clauses `SELECT` et `FROM` obligatoires dans une requête `SELECT`
- N'autorise pas la concaténation dans `SELECT` avec `||`
- ...

HQL

- Supportant `SELECT`, `INSERT`, `UPDATE` et `DELETE`
- Seule la clause `FROM` est obligatoire dans une requête `SELECT`
- Autorise la concaténation dans `SELECT` avec `||`
- ...

Structure d'une requête JPQL

```
select_clause
from_clause
[where_clause]
[groupby_clause]
[having_clause]
[orderby_clause]
```

Structure d'une requête HQL

```
[select_clause]
from_clause
[where_clause]
[groupby_clause]
[having_clause]
[orderby_clause]
```

Trois méthodes pour l'exécution de requêtes

- `createNativeQuery()` : 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();

String sqlRequete = "SELECT * FROM personne";
NativeQuery<Personne>  query = session.createNativeQuery(sqlRequete,
   Personne.class);

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

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

Ou avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-
   config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

String sqlRequete = "SELECT * FROM personne";
Query  query = entityManager.createNativeQuery(sqlRequete, Personne.
   class);

List<Personne> personnes = query.getResultList();
personnes.forEach(System.out::println);

transaction.commit();
entityManager.close();
```

On peut aussi exécuter une requête SQL paramétrée

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

String sqlRequete = "SELECT * FROM personne WHERE nom = :nom";
NativeQuery<Personne>  query = session.createNativeQuery(sqlRequete,
   Personne.class);
query.setParameter("nom", "Wick");

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

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

Ou avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-
   config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

String sqlRequete = "SELECT * FROM personne WHERE nom = :nom";

Query query = entityManager.createNativeQuery(sqlRequete, Personne.
   class);
query.setParameter("nom", "Wick");
List<Personne> personnes = query.getResultList();
personnes.forEach(System.out::println);

transaction.commit();
entityManager.close();
```

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
}
```

Il faut importer `NamedQuery` **de** `jakarta.persistence.NamedQuery`

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

```
package org.eclipse.main;

import java.util.List;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

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

        Query<Personne> query = session.createNamedQuery("findByNomPrenom", Personne.
          class);
        query.setParameter("nom", "Wick");
        query.setParameter("prenom", "John");

        List<Personne> personnes = (List<Personne>) query.list();
        personnes.forEach(System.out::println);
        session.close();
        sessionFactory.close();
    }
}
```

Solution avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

String sqlRequete = "SELECT * FROM personne WHERE nom = :nom";

Query query = entityManager.createNamedQuery("findByNomPrenom", Personne.class);
query.setParameter("nom", "Wick");
query.setParameter("prenom", "John");
List<Personne> personnes = query.getResultList();
personnes.forEach(System.out::println);

transaction.commit();
entityManager.close();
```

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
}
```

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` ?

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

```java
package org.eclipse.main;

import java.util.List;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

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

        String hql = "SELECT p FROM Personne p WHERE p.nom = :nom";
        String string = "Wick";
        Query<Personne> query = session.createQuery(hql, Personne.class);
        query.setParameter("nom", string);
        List<Personne> personnes = query.list();

        personnes.forEach(System.out::println);

        session.close();
        sessionFactory.close();
    }
}
```

Simplifier la requête précédente

```
package org.eclipse.main;

import java.util.List;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

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

        String hql = "FROM Personne p WHERE p.nom = :nom";
        String string = "Wick";
        Query<Personne> query = session.createQuery(hql, Personne.class);
        query.setParameter("nom", string);
        List<Personne> personnes = query.list();

        personnes.forEach(System.out::println);

        session.close();
        sessionFactory.close();
    }
}
```

Solution avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

String hql = "SELECT p FROM Personne p WHERE p.nom = :nom";
String string = "Wick";

Query query = entityManager.createQuery(hql, Personne.class);
query.setParameter("nom", string);
List<Personne> personnes = query.getResultList();
personnes.forEach(System.out::println);

transaction.commit();
entityManager.close();
```

API Criteria de Hibernate

- Dépréciée depuis la version 5.2
- Supprimée dans la 6.0

API Criteria de Hibernate

- Dépréciée depuis la version 5.2
- Supprimée dans la 6.0

Quelle solution pour les requêtes personnalisées et/ou complexes ?

- Soit avec les requêtes **SQL** et **HQL**
- Soit avec **JPA Criteria API**

JPA Criteria API

- Introduite dans **JPA 2.0**
- Ajout de `UPDATE` et `DELETE` dans **JPA 2.1**
- Objectif : remplacer les requêtes **SQL** ou **HQL** de type `String` par des requêtes de type objet

Classes et interfaces indispensables

- `CriteriaBuilder` : interface utilisée pour créer un `CriteriaQuery`
- `CriteriaQuery` : interface utilisée par `EntityManager` pour construire une requête de type `SELECT`
- `CriteriaUpdate` : interface utilisée par `EntityManager` pour construire une requête de type `UPDATE`
- `CriteriaDelete` : interface utilisée par `EntityManager` pour construire une requête de type `DELETE`
- `Query` : interface utilisée pour exécuter la requête et récupérer le résultat
- `Root` : interface utilisée pour spécifier l'élément de la clause `from` d'une requête **SQL**

Pour créer un objet de type `CriteriaBuilder`

```
CriteriaBuilder cb = session.getCriteriaBuilder();
```

Pour créer un objet de type `CriteriaBuilder`

```
CriteriaBuilder cb = session.getCriteriaBuilder();
```

Utilisons `CriteriaBuilder` **pour créer l'objet** `CriteriaQuery`

```
CriteriaQuery<Personne> cr = cb.createQuery(Personne.class);
```

Pour créer un objet de type `CriteriaBuilder`

```
CriteriaBuilder cb = session.getCriteriaBuilder();
```

Utilisons `CriteriaBuilder` **pour créer l'objet** `CriteriaQuery`

```
CriteriaQuery<Personne> cr = cb.createQuery(Personne.class);
```

Ensuite l'objet `Root`

```
Root<Personne> root = cr.from(Personne.class);
```

Pour créer un objet de type `CriteriaBuilder`

```
CriteriaBuilder cb = session.getCriteriaBuilder();
```

Utilisons `CriteriaBuilder` **pour créer l'objet** `CriteriaQuery`

```
CriteriaQuery<Personne> cr = cb.createQuery(Personne.class);
```

Ensuite l'objet `Root`

```
Root<Personne> root = cr.from(Personne.class);
```

Spécifions ce qu'il faut sélectionner et retourner par la requête

```
cr.select(root);
```

Enfin, il suffit d'exécuter la requête et de récupérer le résultat

```
Query<Personne> query = session.createQuery(cr);

List<Personne> results = query.getResultList();
results.forEach(System.out::println);
```

Voici tout le `main`

```java
package org.eclipse.main;

import java.util.List;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Root;

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

        CriteriaBuilder cb = session.getCriteriaBuilder();
        CriteriaQuery<Personne> cr = cb.createQuery(Personne.class);
        Root<Personne> root = cr.from(Personne.class);
        cr.select(root);
        Query<Personne> query = session.createQuery(cr);
        List<Personne> results = query.getResultList();
        results.forEach(System.out::println);

        session.close();
        sessionFactory.close();
    }
}
```

Quelques méthodes de `Query`

- `getResultList()` : retourne le résultat sous forme d'un tableau d'objet
- `getSingleResult()` : retourne le résultat sous forme d'un seul d'objet
- `getFirstResult()` : retourne le premier élément de la sélection sous forme d'un objet
- ...

Pour sélectionner quelques personnes selon un critère (nom = Abruzzi)

```
cr.select(root).where(cb.equal(root.get("nom"), "Abruzzi"));
```

Pour sélectionner quelques personnes selon un critère (nom = Abruzzi)

```
cr.select(root).where(cb.equal(root.get("nom"), "Abruzzi"));
```

Pour sélectionner quelques personnes selon plusieurs critères (ET logique)

```
Predicate[] predicates = new Predicate[2];
predicates[0] = cb.between(root.get("num"), 1, 200);
predicates[1] = cb.like(root.get("nom"), "%i%");
cr.select(root).where(predicates);
```

Pour sélectionner quelques personnes selon un critère (nom = Abruzzi)

```
cr.select(root).where(cb.equal(root.get("nom"), "Abruzzi"));
```

Pour sélectionner quelques personnes selon plusieurs critères (ET logique)

```
Predicate[] predicates = new Predicate[2];
predicates[0] = cb.between(root.get("num"), 1, 200);
predicates[1] = cb.like(root.get("nom"), "%i%");
cr.select(root).where(predicates);
```

Ou

```
Predicate numPredicate = cb.between(root.get("num"), 1, 200);
Predicate nomPredicate = cb.like(root.get("nom"), "%i%");
cr.select(root).where(cb.and(numPredicate, nomPredicate));
```

Pour sélectionner quelques personnes selon plusieurs critères (OU logique)

```
Predicate numPredicate = cb.between(root.get("num"), 1, 200);
Predicate nomPredicate = cb.like(root.get("nom"), "%i%");
cr.select(root).where(cb.or(numPredicate, nomPredicate));
```

Pour ordonner le résultat

```
cr.orderBy(
     cb.asc(root.get("nom")),
     cb.desc(root.get("prenom"))
);
```

Exemple avec une fonction d'agrégation : le nombre de tuples

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Long> cr = cb.createQuery(Long.class);
Root<Personne> root = cr.from(Personne.class);

cr.select(cb.count(root));

Query<Long> query = session.createQuery(cr);
List<Long> nombre = query.getResultList();

System.out.println(nombre);
```

Exemple avec une fonction d'agrégation : la moyenne d'une colonne

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Double> cr = cb.createQuery(Double.class);
Root<Personne> root = cr.from(Personne.class);

cr.select(cb.avg(root.get("num")));

Query<Double> query = session.createQuery(cr);
List moyenne = query.getResultList();

System.out.println(moyenne);
```

Exemple de mise à jour (JPA 2.1)

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.MutationQuery;

import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaUpdate;
import jakarta.persistence.criteria.Root;

public class App {
    public static void main(String[] args) {
        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        CriteriaBuilder cb = session.getCriteriaBuilder();
        CriteriaUpdate<Personne> cu = cb.createCriteriaUpdate(Personne.class);
        Root<Personne> root = cu.from(Personne.class);
        cu.set("nom", "Doe");
        cu.where(cb.equal(root.get("prenom"), "John"));
        Transaction transaction = session.beginTransaction();
        MutationQuery query = session.createMutationQuery(cu);
        int nombre = query.executeUpdate();
        System.out.println(nombre);
        transaction.commit();
        session.close();
        sessionFactory.close();
    }
}
```

Exemple de suppression (JPA 2.1)

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.MutationQuery;

import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaDelete;
import jakarta.persistence.criteria.Root;

public class App {
    public static void main(String[] args) {
        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        CriteriaBuilder cb = session.getCriteriaBuilder();
        CriteriaDelete<Personne> cd = cb.createCriteriaDelete(Personne.class);
        Root<Personne> root = cd.from(Personne.class);
        cd.where(cb.greaterThan(root.get("num"), 100));
        MutationQuery query = session.createMutationQuery(cd);
        Transaction transaction = session.beginTransaction();
        int nombre = query.executeUpdate();
        System.out.println(nombre);
        transaction.commit();
        session.close();
        sessionFactory.close();
    }
}
```

JPA Criteria API vs JPQL et HQL

- Les requêtes **JPQL** ou **HQL** sont définies comme des chaînes, de la même manière que **SQL**.
- Les requêtes de **JPA Criteria API** sont définies par une suite d'instanciation d'objets **Java** qui représentent des éléments de la requête.
- Avec **JPA Criteria API**, les erreurs seront détectées à la compilation, à l'exécution avec **JPQL** et **HQL**.
- Les requêtes **JPQL** ou **HQL** sont préférées lorsqu'elles ne sont pas complexes.
- Les requêtes **JPQL** ou **HQL** et les requêtes **JPA Criteria API** sont équivalentes en terme de performance et efficacité.

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

Avant de commencer

Commençons par supprimer et recréer la base de données `cours_hibernate`.

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

```
@Entity
public class Adresse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String rue;
    private String codePostal;
    private String ville;

    // + getters + setters + constructeurs et toString

}
```

Sans oublier de déclarer `Adresse` **dans** `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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <property name="format_sql">true</property>
  <mapping class="org.eclipse.model.Personne" />
  <mapping class="org.eclipse.model.Adresse" />
 </session-factory>
</hibernate-configuration>
```

Déclarons `Adresse` **dans** `Personne`

```
@Entity
public class Personne {

    // attributs précédents

    private Adresse adresse;

    // + getter + setter + constructeur (nom, prenom, adresse)

}
```

Pour pouvoir attacher une `adresse` à une `Personne` et les persister, il faut ajouter le type d'association : `@OneToOne`

```
@Entity
public class Personne {

    // attributs précédents

        @OneToOne
        private Adresse adresse;

        // méthodes précédentes

}
```

Pour pouvoir attacher une `adresse` à une `Personne` et les persister, il faut ajouter le type d'association : `@OneToOne`

```
@Entity
public class Personne {

    // attributs précédents

        @OneToOne
        private Adresse adresse;

        // méthodes précédentes

}
```


Appellation

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

On peut aussi ajouter l'annotation `@JoinColumn` pour spécifier plus de détails sur la colonne de jointure

```
@JoinColumn(name="id", referencedColumnName="id", 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)

Testons l'ajout d'une personne

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Ego", "Paul", adresse);

        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.persist(adresse);
        session.persist(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

Testons l'ajout d'une personne

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Ego", "Paul", adresse);

        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.persist(adresse);
        session.persist(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

Il faut persister `adresse` avant `personne`

Constats

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

Solution avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
Personne personne = new Personne("Ego", "Paul", adresse);

entityManager.persist(adresse);
entityManager.persist(personne);
entityManager.flush();

transaction.commit();
entityManager.close();
```

Pour persister `adresse` et `personne` avec un seul `persist`, on ajoute l'attribut `cascade` dans l'annotation `@OneToOne`

```
@Entity
public class Personne {

    // attributs précédents

    @OneToOne(cascade = { CascadeType.PERSIST })
    private Adresse adresse;

    // méthodes précédentes
}
```

Pour persister `adresse` et `personne` avec un seul `persist`, on ajoute l'attribut `cascade` dans l'annotation `@OneToOne`

```
@Entity
public class Personne {

    // attributs précédents

    @OneToOne(cascade = { CascadeType.PERSIST })
    private Adresse adresse;

    // méthodes précédentes
}
```

Explication

- `CascadeType.PERSIST` permet de persister une `Personne` avec une `Adresse` sans avoir besoin de persister `Adresse` auparavant.
- On peut cascader plusieurs autres opérations telles que `REMOVE`, `REFRESH`...
- On peut aussi tout cascader avec `ALL`

Testons avec un seul `persist`

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Lock", "John", adresse);

        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.persist(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

Testons avec un seul `persist`

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Lock", "John", adresse);

        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.persist(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

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

La seule solution avec `save` est de le répéter

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Lock", "John", adresse);

        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.save(adresse);
        session.save(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

Remarque

- En base de données relationnelle, on utilise souvent le **Snake Case** pour nommer les tables, les colonnes...
- Il est possible de demander à **Hibernate** de respecter cette stratégie

Modifions la stratégie de nommage

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Lock", "John", adresse);

        Configuration configuration = new Configuration().configure();
        configuration.setPhysicalNamingStrategy(new
           CamelCaseToUnderscoresNamingStrategy());
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.save(adresse);
        session.save(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

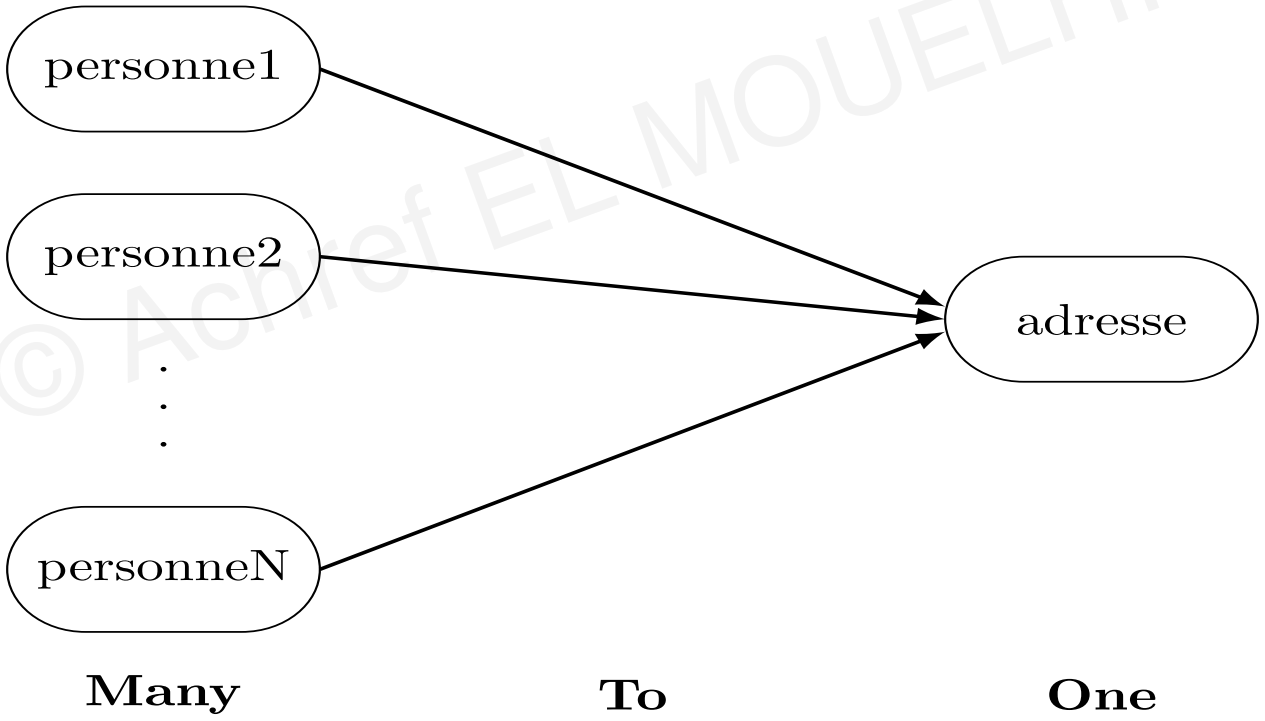
Modifions la stratégie de nommage

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Adresse adresse = new Adresse("Lyon", "13015", "Marseille");
        Personne personne = new Personne("Lock", "John", adresse);

        Configuration configuration = new Configuration().configure();
        configuration.setPhysicalNamingStrategy(new
           CamelCaseToUnderscoresNamingStrategy());
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.save(adresse);
        session.save(personne);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

Vérifier que l'attribut `codePostal` a été remplacé par la colonne `code_postal`.

Hypothèse

Si plusieurs personnes pouvaient avoir la même adresse.



Il faut juste remplacer `@ManyToOne` **par** `@OneToOne`

```
@ManyToOne(cascade = { CascadeType.PERSIST })
private Adresse adresse;
```

Il faut juste remplacer `@ManyToOne` **par** `@OneToOne`

```
@ManyToOne(cascade = { CascadeType.PERSIST })
private Adresse adresse;
```


Remarque

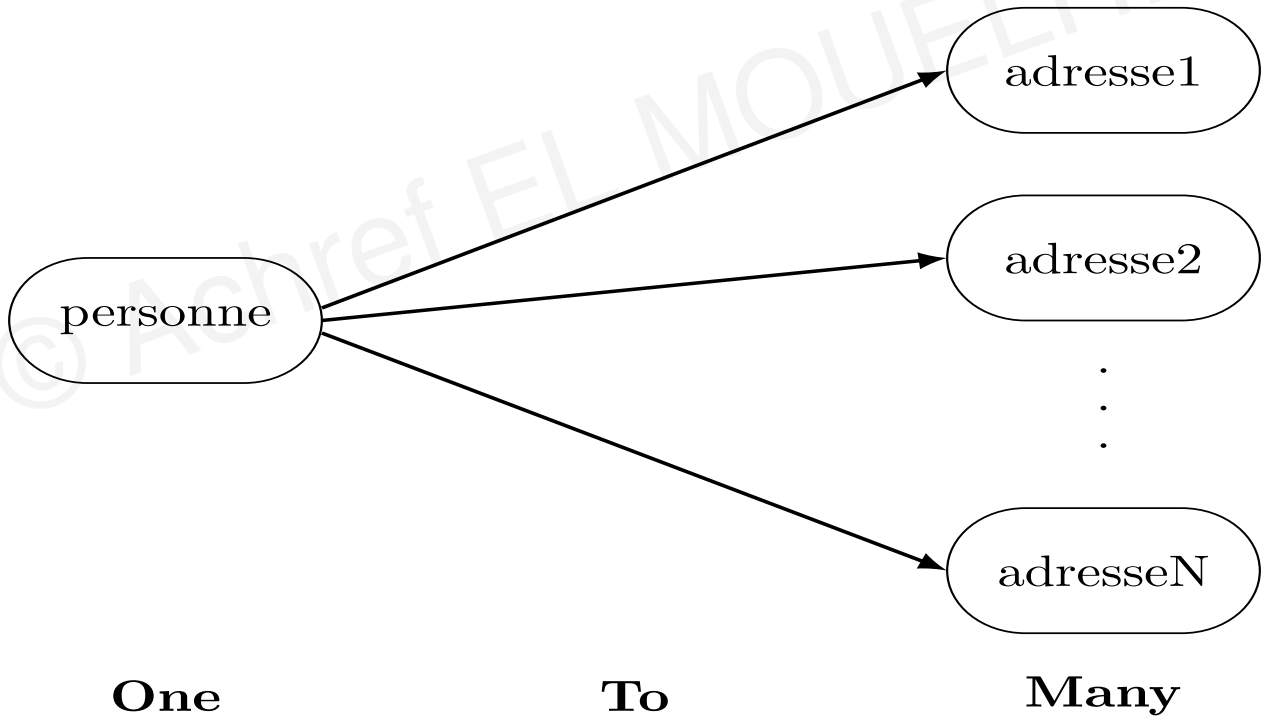
Le schéma de la base de données ne change pas.

Et pour tester

```
package org.eclipse.main;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {

        Adresse adresse = new Adresse("Plantes", "75014", "Paris");
        Personne personne1 = new Personne("Shephard", "Jack", adresse);
        Personne personne2 = new Personne("Shephard", "Kate", adresse);
        Configuration configuration = new Configuration().configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = sessionFactory.openSession();

        session.persist(personne1);
        session.persist(personne2);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

Hypothèse

Et si une personne pouvait avoir plusieurs adresses.



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

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

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

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

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

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

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`

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`

N'oublions pas de supprimer la base de données.

Pour tester

```
package org.eclipse.main;

import java.util.List;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {

        Adresse adr1 = new Adresse("Plantes", "75014", "Paris");
        Adresse adr2 = new Adresse("Paradis", "13006", "Marseille");
        Personne personne = new Personne("Shephard", "Jack", List.of(adr1, adr2));

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

        session.persist(personne);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

Constats

- Trois tables créées :
  - `adresse` avec les colonnes `id`, `rue`, `codePostal`, `ville`
  - `personne` avec les colonnes `num`, `nom`, `prenom`
  - `personne_adresse` avec les colonnes `#personne_num`, `#adresses_id`
- Cinq tuples insérés :
  - `(2, Paradis, 13006, Marseille)` et `(1, Plantes, 75014, Paradis)` dans `adresse`
  - `(1, Shephard, Jack)` dans `personne`
  - `(1, 1)` et `(1, 2)` dans `personne_adresse`

Solution avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

Adresse adr1 = new Adresse("Plantes", "75014", "Paris");
Adresse adr2 = new Adresse("Paradis", "13006", "Marseille");
Personne personne = new Personne("Shephard", "Jack", List.of(adr1, adr2));

entityManager.persist(personne);

entityManager.flush();

transaction.commit();
entityManager.close();
```

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 }, fetch = FetchType.
   CONSTANTE)
private List <Adresse> adresses;
```

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 }, fetch = FetchType.
   CONSTANTE)
private List <Adresse> adresses;
```

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()`).

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();

Personne personne = session.find(Personne.class, 1);
System.out.println(personne.getNom());

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

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();

Personne personne = session.find(Personne.class, 1);
System.out.println(personne.getNom());

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


Remarque

Vérifier dans la console d'**Eclipse** que la requête **SQL** générée par **Hibernate** ne sélectionne pas les adresses.

L'adresse sera chargée uniquement sur demande

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

Personne personne = session.find(Personne.class, 1);
System.out.println(personne.getAdresses());

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

L'adresse sera chargée uniquement sur demande

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

Personne personne = session.find(Personne.class, 1);
System.out.println(personne.getAdresses());

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


Remarque

Vérifier dans la console d'**Eclipse** que la requête **SQL** générée par **Hibernate** sélectionne les adresses.

Modifions le chargement d`adresses` dans `Personne` (`EAGER`)

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

Modifions le chargement d`adresses` dans `Personne` (`EAGER`)

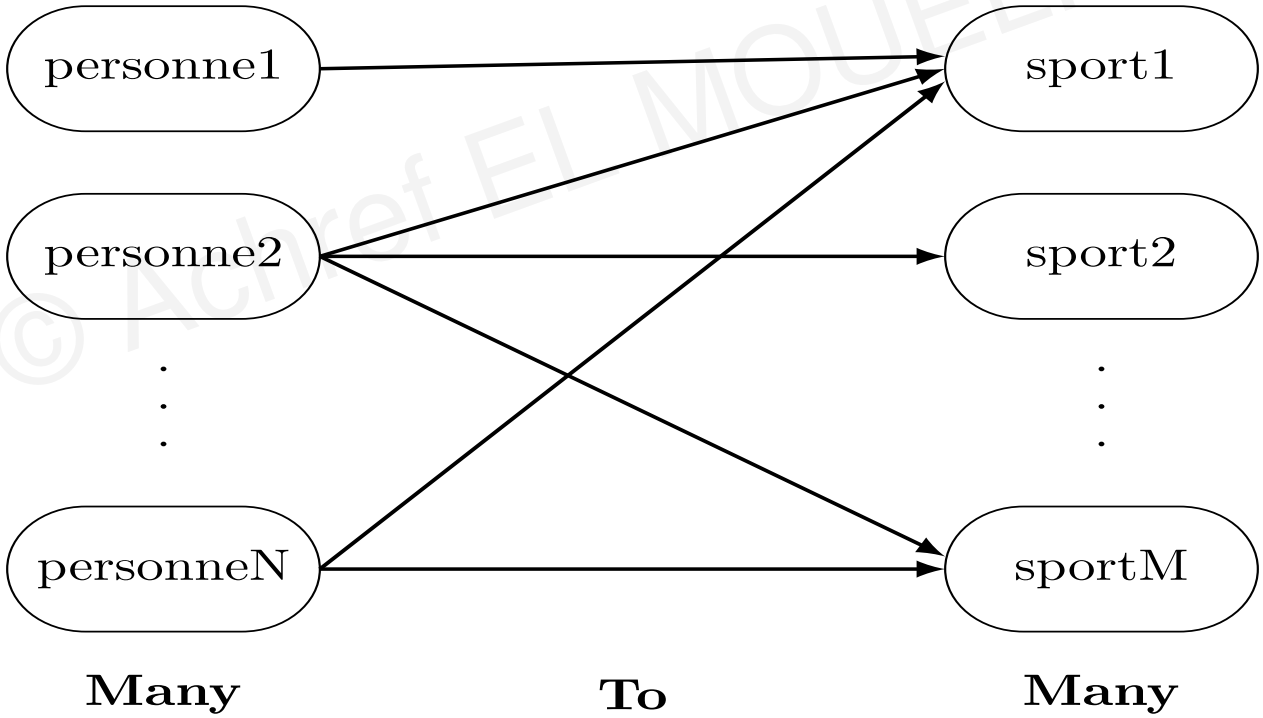
```
@OneToMany(cascade = { CascadeType.PERSIST }, fetch = FetchType.EAGER)
private List <Adresse> adresses;
```

Remarque

Exécuter le `main` précédent et vérifier dans la console d'**Eclipse** que la requête **SQL** générée par **Hibernate** sélectionne les adresses.

Exemple

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



Ce qu'il faut faire

- Créer une entité `Sport`
- Définir la relation `ManyToMany` (exactement comme les deux relations précédentes) soit dans `Personne` soit dans `Sport`

Création de l'entité Sport

```
@Entity
public class Sport {
        @Id
        @Column(length = 20)
        private String nom;
        private String type;

        // + constructeurs + getters + setters +
           toString
}
```

Sans oublier de déclarer `Sport` **dans** `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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <property name="format_sql">true</property>
  <mapping class="org.eclipse.model.Personne" />
  <mapping class="org.eclipse.model.Adresse" />
  <mapping class="org.eclipse.model.Sport" />
 </session-factory>
</hibernate-configuration>
```

Ajoutons le `ManyToMany` dans la classe `Personne`

```
@Entity
public class Personne {

  // code précédent

  @ManyToMany(cascade = { CascadeType.PERSIST })
  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);
  }
}
```

Pour tester

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.eclipse.model.Sport;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne p1 = new Personne("Voight", "Hank");
        Personne p2 = new Personne("Atwater", "Kevin");
        Sport s1 = new Sport("football", "collectif");
        Sport s2 = new Sport("tennis", "individuel");
        Sport s3 = new Sport("box", "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();
        session.persist(p1);
        session.persist(p2);
        session.flush();
        session.close();
        sessionFactory.close();
    }
}
```

Constats

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

Solution avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

Personne p1 = new Personne("Voight", "Hank");
Personne p2 = new Personne("Atwater", "Kevin");
Sport s1 = new Sport("football", "collectif");
Sport s2 = new Sport("tennis", "individuel");
Sport s3 = new Sport("box", "collectif ou individuel");
p1.addSport(s1);
p1.addSport(s3);
p2.addSport(s1);
p2.addSport(s2);
p2.addSport(s3);

entityManager.persist(p1);
entityManager.persist(p2);
entityManager.flush();
transaction.commit();
entityManager.close();
```

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

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`

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();`

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

Modifier l'entité inverse `Sport`

```
@Entity
public class Sport {

  @Id
  private String nom;
  private String type;

  @ManyToMany(mappedBy="sports")
  private List <Personne> personnes = new ArrayList <Personne> ();

  // ajouter les getter, setter, add et remove
}
```

Modifier l'entité inverse `Sport`

```
@Entity
public class Sport {

  @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`

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 `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().remove(this);
}
```

Ainsi, on peut faire

```
public class App {
    public static void main(String[] args) {
        Personne p1 = new Personne("Voight", "Hank");
        Personne p2 = new Personne("Atwater", "Kevin");
        Sport s1 = new Sport("football", "collectif");
        Sport s2 = new Sport("tennis", "individuel");
        Sport s3 = new Sport("box", "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();
        session.persist(p1);
        session.persist(p2);
        session.flush();
        Sport sport = session.find(Sport.class, "football");
        for (Personne personne : sport.getPersonnes()) {
            System.out.println(personne.getNom());
        }
        session.close();
        sessionFactory.close();
    }
}
```

Pour définir une relation bidirectionnelle entre deux entités

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

Trois solutions pour la transformation de l'héritage

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

Exemple

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

Exemple

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

La classe `Etudiant`

```
@Entity
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
public class Enseignant extends Personne {

  private int salaire;

  public int getSalaire() {
    return salaire;
  }

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

Pour indiquer la solution choisie pour la transformation de l'héritage

On utilise l'annotation `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`

Pour indiquer la solution choisie pour la transformation de l'héritage

On utilise l'annotation `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`

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`.

Pour indiquer la solution choisie pour la transformation de l'héritage

On utilise l'annotation `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`

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 `PERS`, `ETU` ou `ENS`.

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;
  }
}
```

Sans oublier de déclarer `Sport` **dans** `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>
  <property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
  </property>
  <property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/cours_hibernate?createDatabaseIfNotExist=true
  </property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
  <property name="hbm2ddl.auto">update</property>
  <property name="show_sql">true</property>
  <property name="format_sql">true</property>
  <mapping class="org.eclipse.model.Personne" />
  <mapping class="org.eclipse.model.Adresse" />
  <mapping class="org.eclipse.model.Sport" />
  <mapping class="org.eclipse.model.Enseignant" />
  <mapping class="org.eclipse.model.Etudiant" />
 </session-factory>
</hibernate-configuration>
```

Et pour tester

```
package org.eclipse.main;

import org.eclipse.model.Enseignant;
import org.eclipse.model.Etudiant;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne personne = new Personne("Guardiola", "Pep");
        Enseignant enseignant = new Enseignant("Ferguson", "Sir", 10000);
        Etudiant etudiant = new Etudiant("Mourinho", "Jose", "Ligue 1");

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

        session.persist(personne);
        session.persist(enseignant);
        session.persist(etudiant);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

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

Solution avec Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa-config");
EntityManager entityManager = emf.createEntityManager();
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

Personne personne = new Personne("Guardiola", "Pep");
Enseignant enseignant = new Enseignant("Ferguson", "Sir", 10000);
Etudiant etudiant = new Etudiant("Mourinho", "Jose", "Ligue 1");

entityManager.persist(personne);
entityManager.persist(enseignant);
entityManager.persist(etudiant);
entityManager.flush();
entityManager.flush();
transaction.commit();
entityManager.close();
```

TABLE_PER_CLASS

Commençons par supprimer l'annotation `@DiscriminatorValue` dans `Etudiant` et `Enseignant`.

TABLE_PER_CLASS

Commençons par supprimer l'annotation `@DiscriminatorValue` dans `Etudiant` et `Enseignant`.

La classe `Etudiant`

```
@Entity
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
public class Enseignant extends Personne {

  private int salaire;

  public int getSalaire() {
    return salaire;
  }

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

Pour la classe `Personne`**, remplaçons** `SINGLE_TABLE` **par** `TABLE_PER_CLASS` **et supprimons** `@DiscriminatorColumn` **et** `@DiscriminatorValue`

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Personne {

  //  le code précédent

}
```

Pour la clé primaire, remplaçons `GenerationType.IDENTITY` **par** `GenerationType.TABLE`

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Personne {

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

    //  + le reste du code précédent

}
```

Rien à changer dans `main` pour tester

```
package org.eclipse.main;

import org.eclipse.model.Enseignant;
import org.eclipse.model.Etudiant;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne personne = new Personne("Guardiola", "Pep");
        Enseignant enseignant = new Enseignant("Ferguson", "Sir", 10000);
        Etudiant etudiant = new Etudiant("Mourinho", "Jose", "Ligue 1");

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

        session.persist(personne);
        session.persist(enseignant);
        session.persist(etudiant);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

Constats

- Trois tables créées :
  - `Personne` avec les colonnes : `num`, `nom` et `prenom`
  - `Enseignant` avec les colonnes : `num`, `nom`, `prenom` et `salaire`
  - `Etudiant` avec les colonnes : `num`, `nom`, `prenom` et `niveau`
- Trois tuples insérés
  - un dans la table `Personne`
  - un dans la table `Enseignant`
  - un dans la table `Etudiant`

Pour la solution `JOINED`**, modifions la classe** `Personne`

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Personne {

  // +  le code précédent

}
```

La classe `Etudiant`

```
@Entity
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
public class Enseignant extends Personne {

  private int salaire;

  public int getSalaire() {
    return salaire;
  }

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

Rien à changer dans `main` pour tester

```
package org.eclipse.main;

import org.eclipse.model.Enseignant;
import org.eclipse.model.Etudiant;
import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class App {
    public static void main(String[] args) {
        Personne personne = new Personne("Guardiola", "Pep");
        Enseignant enseignant = new Enseignant("Ferguson", "Sir", 10000);
        Etudiant etudiant = new Etudiant("Mourinho", "Jose", "Ligue 1");

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

        session.persist(personne);
        session.persist(enseignant);
        session.persist(etudiant);
        session.flush();

        session.close();
        sessionFactory.close();
    }
}
```

Constats

- Trois tables créées :
  - `Personne` avec les colonnes : `num`, `nom` et `prenom`
  - `Enseignant` avec les colonnes : `num` et `salaire`
  - `Etudiant` avec les colonnes : `num` et `niveau`
- Cinq tuples insérés
  - trois dans la table `Personne`
  - un dans la table `Enseignant`
  - un dans la table `Etudiant`

Les classes incorporables

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

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 + "]";
    }
}
```

Utilisons `NomComplet` **dans** `Personne`

```
@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;
    }
```

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);

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

session.persist(personne);

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

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);

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

session.persist(personne);

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

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

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()`)

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

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

Comme les triggers en **SQL**.

Les 7 annotations **JPA** disponibles

- `@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.

Exemple : l'entité `Personne`

```
public class Personne {

  // les attributs précédents

  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 méthodes
}
```

Et pour tester

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

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

        Personne p1 = new Personne("Wick", "John");
        session.persist(p1);
        session.flush();
        System.out.println("nbrMAJ = " + p1.getNbrMAJ());
        // affiche nbrMAJ = 0
        p1.setNom("Travolta");
        session.flush();
        System.out.println("nbrMAJ = " + p1.getNbrMAJ());
        // affiche nbrMAJ = 1
        p1.setNom("Abruzzi");
        session.flush();
        System.out.println("nbrMAJ = " + p1.getNbrMAJ());
        // affiche nbrMAJ = 2

        session.close();
        sessionFactory.close();
    }
}
```

UUID (**U**niversally **U**nique **I**dentifier)

- identifiant unique universel utilisé en informatique pour identifier de manière unique des informations sans la nécessité de coordonner les identifiants centralement
- codé sur 128 bits divisé en plusieurs segments avec des rôles spécifiques, garantissant l'unicité à travers des systèmes distribué
- supporté depuis la version 5 d'**Hibernate**

Format général de UUID : `xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx`

- `xxxxxxxx` : 8 caractères hexadécimaux (32 bits), généré aléatoirement.
- `xxxx` : 4 caractères hexadécimaux suivants (16 bits), généré aléatoirement.
- `Mxxx` : les 4 caractères hexadécimaux suivants (16 bits). Le premier caractère (M) indique la version de l'**UUID**.
- `Nxxx` : les 4 caractères hexadécimaux suivants (16 bits). Les 2 premiers bits (N) indiquent la variante de l'**UUID**.
- `xxxxxxxxxxxx` : les 12 derniers caractères hexadécimaux (48 bits).

Exemple

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID num;

    // + les autres attributs et méthodes
}
```

Exemple

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID num;

    // + les autres attributs et méthodes
}
```


Pour définir une clé primaire comme une **UUID** en **Hibernate** et **JPA**, il suffit d'utiliser l'annotation `@GeneratedValue` avec `GenerationType.AUTO` sur un champ de type `UUID`.

Organisation du code

- Créer une classe `HibernateUtils` 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 `HibernateUtils` 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`

La classe `HibernateUtils`

```
package org.eclipse.config;

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

public class HibernateUtils {

        private static SessionFactory sessionFactory;

        static {
                sessionFactory = new Configuration().configure().
                  buildSessionFactory();
        }

        public static SessionFactory getSessionFactory() {
                return sessionFactory;
        }
}
```

La classe `GenericDao`

```
public class GenericDao<ENTITY, PRIMARY_KEY> {

        private Class<ENTITY> clazz;

        public GenericDao(Class<ENTITY> clazz) {
                this.clazz = clazz;
        }

        public <R> R runInTransaction(Function<Session, R> action) {
                Transaction t = null;
                R result;
                try (var s = HibernateUtils.getSessionFactory().openSession()) {
                        t = s.beginTransaction();
                        result = action.apply(s);
                        t.commit();
                        return result;
                } catch (Exception e) {
                        try {
                                if (t != null)
                                        t.rollback();
                        } catch (Exception e1) {
                                e.addSuppressed(e1);
                        }
                        throw e;
                }
        }
```

La classe `GenericDao` (suite)

```
        public Collection<ENTITY> findAll() {
                try (var s = HibernateUtils.getSessionFactory().openSession()) {
                        return s.createQuery("from " + clazz.getSimpleName(), clazz).
                          getResultList();
                }
        }

        public Optional<ENTITY> findById(PRIMARY_KEY id) {
                try (var s = HibernateUtils.getSessionFactory().openSession()) {
                        return Optional.ofNullable(s.get(clazz, id));
                }
        }

        public ENTITY save(ENTITY u) {
                return runInTransaction(s -> {
                        s.persist(u);
                        return u;
                });
        }

        public ENTITY update(ENTITY u) {
                return runInTransaction(s -> s.merge(u));
        }

        public void deleteById(PRIMARY_KEY id) {
                delete(findById(id).get());
        }
}
```

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

```
public class Personne {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Integer num;
        private String nom;
        private String prenom;

        // + le reste du code

}
```

La classe `PersonneDao`

```
package org.eclipse.dao;

import org.eclipse.model.Personne;

public class PersonneDao extends GenericDao<Personne,
   Integer> {

        public PersonneDao() {
                super(Personne.class);
        }

}
```

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 {
    Personne personne = new Personne();
    personne.setNom("Ferdinand");
    personne.setPrenom("Rio");
    PersonneDao personneDao = new PersonneDao();
    System.out.println(personneDao.save(personne));
    Personne personne2 = personneDao.findById(3);
    personne2.setNom("Turing");
    personneDao.update(personne2);
  }
}
```