

# JPA avec EclipseLink

**Achref El Mouelhi**

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

`elmouelhi.achref@gmail.com`



- 1 Introduction
- 2 Créer une connexion
- 3 Créer un JPA Project
- 4 Entity
  - Création d'une entité
  - Création de tables associées aux entités
  - Création des entités à partir d'une BD existante

- 5 EntityManager
  - Insertion
  - Mise à jour
  - Suppression
  - Consultation
- 6 JPQL : Java Persistence Query Language
- 7 Relation entre entités
  - OneToOne
  - ManyToOne
  - OneToMany
  - ManyToMany
  - Inheritance

- 8 Les classes incorporables
- 9 Les méthodes `callback`
- 10 Utilisation de l'API JPA dans un projet JEE
- 11 Utilisation de l'API JPA dans un projet Maven

# Object-Relational Mapping (lien objet-relationnel)

## Définition

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

# Object-Relational Mapping (lien objet-relationnel)

## Définition

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

Plusieurs ORM proposés pour chaque Langage de POO.

# Object-Relational Mapping

## Pour Java

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

# Object-Relational Mapping

## Pour Java

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

## Quel choix pour PHP ? !

- Doctrine
- pdoMap
- RedBean
- ...

## EclipseLink ?

- est un framework open source de mapping objet-relation
- est dérivé du projet TopLink d'Oracle (un framework d'ORM open source acheté par Oracle en 2002)
- supporte l'API de persistance de données JPA

## JPA : Java Persistence API

- est une interface standardisée par Sun, qui permet l'organisation des données
- a été proposé par JSR (Java Specification Requests)
- s'appuie sur l'utilisation des annotations pour définir le lien entre `Entity` (classe) et table (en base de données relationnelle) et sur le gestionnaire `EntityManager` pour gérer les données (insertion, modification...)

## Étapes

- Création d'une connexion
- Création d'un `JPA Project`
- Création des entités et bien utiliser les annotations
- Établissement de liens entre entités
- Création des tables, à partir des entités, si elles n'existent pas
- Manipulation des entités (ajout, suppression, modification, consultation) avec `EntityManager`

# Créer une connexion

Avant de créer une connexion

créer une base de données appelée `jpa`

# Créer une connexion

## Avant de créer une connexion

créer une base de données appelée `jpa`

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

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

# Création d'une connexion

## Tester la connexion

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

## Création d'un projet JPA

- Aller dans menu `File > New > JPA Project`
- Saisir `FirstJpaProject` comme nom du projet
- Choisir `jrel.8` dans `Target runtime` et cliquer deux fois sur `Next`
- Dans `Platform`, choisir la dernière version d'`EclipseLink (2.5.2)`
- Dans `Type de JPA Implementation`, choisir `User Library`
- Ensuite cocher la case `EclipseLink` et cliquer sur `Download Library...` à droite (pour démarrer le téléchargement)
- Dans `Connection`, choisir la connexion définie précédemment
- Cocher la case `Add driver library to build path` et valider

## Le fichier `persistence.xml`

- est situé dans le répertoire `META-INF` de `src`
- contient les données sur le mapping entre nos entités/ et nos tables dans la base de données

# Une entité (Entity)

## Définition

- correspond à une table d'une base de données relationnelle
- est un objet (Javabeen) contenant quelques informations indispensables (annotations) pour le mapping (faire le lien) avec la base de données

# Une entité

## Pour une meilleure organisation

- Créer un package `org.eclipse.model` sous `src`
- Placer toutes les entités dans ce package

# Une entité

## Pour une meilleure organisation

- Créer un package `org.eclipse.model` sous `src`
- Placer toutes les entités dans ce package

## Création

- Faire un clic droit sur `org.eclipse.model`, aller dans `new` et choisir `JPA Entity`
- Saisir un nom dans `Class name :`, ensuite cliquer sur `Next`
- Avec le bouton `Add...`, ajouter les différents attributs ainsi que leur type
- Cocher la case correspondant à la clé primaire puis cliquer sur `Finish`

Vérifier dans `persistence.xml` que la ligne

`<class>org.eclipse.model.Personne</class>` a bien été ajoutée.

# Une entité

```
package org.eclipse.model;
import java.io.Serializable;
import java.lang.String;
import javax.persistence.*;

/**
 * Entity implementation class for
 * Entity: Personne
 *
 */
@Entity
public class Personne implements
    Serializable {

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

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

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

public int getNum() {
    return this.num;
}

public void setNum(int num) {
    this.num = num;
}

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

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

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

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

# Une entité

## Autres annotations

### Annotation

@Entity

@Id

@Table

@Column

@IdClass

@GeneratedValue

### désignation

permet de qualifier la classe comme entité

indique l'attribut qui correspond à la clé primaire de la table

décrit la table désignée par l'entité

définit les propriétés d'une colonne

indique que l'entité annotée contient une clé composée  
les champs constituant la clé seront annoté par @Id

s'applique sur les attributs annotés par @Id

permet la génération automatique de la clé primaire

# Une entité

## Autres annotations

### Annotation

@Entity

@Id

@Table

@Column

@IdClass

@GeneratedValue

### désignation

permet de qualifier la classe comme entité

indique l'attribut qui correspond à la clé primaire de la table

décrit la table désignée par l'entité

définit les propriétés d'une colonne

indique que l'entité annotée contient une clé composée  
les champs constituant la clé seront annotés par @Id

s'applique sur les attributs annotés par @Id

permet la génération automatique de la clé primaire

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

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

# Une entité

## Attributs de l'annotation @Column

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

# Une entité

## Attributs de l'annotation @Table

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

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

# Une entité

## L'annotation `@Transient`

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

# Création de tables associées aux entités

## Étapes

- Un clic droit sur le projet
- Aller dans JPA Tools
- Choisir `Generate Tables from Entities`

# Création des entités à partir d'une BD existante

## Étapes

- Un clic droit sur le projet
- Aller dans `JPA Tools`
- Choisir `Generate Entities from Tables`
- Sélectionner la connexion ensuite les tables à importer
- Choisir le package dans lequel les entités seront importées
- Valider

# Création des entités à partir d'une BD existante

## Si les entités ne contiennent pas les colonnes

- Supprimer les entités générées
- Utiliser le `Data Source Name` pour se déconnecter et se reconnecter
- (vous pouvez même dans certains cas étendre l'arborescence de chaque table dans le `Data Source Name`)
- Régénérer les entités

# EntityManager

## Pour persister les données, il faut

- ajouter les données concernant la connexion à la base de données dans le fichier `persistence.xml`
- instancier et utiliser un gestionnaire d'entités pour manipuler les données

# EntityManager

## Pour persister les données, il faut

- ajouter les données concernant la connexion à la base de données dans le fichier `persistence.xml`
- instancier et utiliser un gestionnaire d'entités pour manipuler les données

## Gestionnaire d'entités (EntityManager) ?

- est une classe Java
- assure l'interaction entre la base de données et les entités
- permet via les entités, définies dans la section `persistence-unit` du fichier `persistence.xml`, de lire ou écrire des données dans la base de données

# EntityManager

## Comment obtenir une instance d'EntityManager

- il faut utiliser une fabrique de type `EntityManagerFactory`
- Cette dernière a une méthode `createEntityManager()` qui permet d'obtenir un `EntityManager`
- Pour obtenir une instance de la fabrique, il faut utiliser la méthode statique `createEntityManagerFactory()` de la classe `Persistence` qui prend comme paramètre le nom de l'unité de persistence à utiliser (défini dans le fichier `persistence.xml`)

# EntityManager

## Le fichier persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/
  persistence http://xmlns.jcp.org/xml/ns/persistence/
  persistence_2_1.xsd">
  <persistence-unit name="FirstJpaProject">
    <class>org.eclipse.model.Personne</class>
  </persistence-unit>
</persistence>
```

## Obtenir l'EntityManager

```
EntityManagerFactory emf = Persistence.
  createEntityManagerFactory("FirstJpaProject");
EntityManager em = emf.createEntityManager();
```

# EntityManager

## Quelques méthodes de l'EntityManager

- `persist(obj)` : pour stocker `obj` dans la base de données
- `find(NomClasse, id)` : pour chercher un élément de type `NomClasse` ayant l'identifiant `id`
- `flush()` : pour sauvegarder les modifications dans la base de données
- `remove(obj)` : pour supprimer `obj` de la base de données
- ...

# EntityManager

Commençons par ajouter les informations sur la connexion dans `persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="FirstJpaProject">
    <class>org.eclipse.model.Personne</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpa" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
      <property name="javax.persistence.target-database" value="MySQL5" />
      <property name="javax.persistence.ddl-generation" value="drop-and-create-tables" />
      <property name="javax.persistence.ddl-generation.output-mode" value="both" />
      <property name="javax.persistence.logging.level" value="FINE" />
    </properties>
  </persistence-unit>
</persistence>
```

`ddl` : data definition language (ou langage de définition de données (LDD)).

# EntityManager

## Valeurs possibles pour `ddl-generation`

- `drop-and-create-tables` : EclipseLink essaiera de supprimer toutes les tables, puis de les recréer toutes. (très utile pour la phase de développement pendant les tests ou lorsque les données existantes doivent être effacées.
- `create-or-extend-tables` : EclipseLink essaiera de créer des tables. Si la table existe, il ajoutera les colonnes manquantes.
- `create-tables` : EclipseLink essaiera d'exécuter `CREATE TABLE` pour chaque table. Si la table existe déjà, une exception est levée et la table n'est pas créée. la table existante sera utilisée. EclipseLink poursuivra ensuite avec la prochaine instruction.
- `none` : Aucun LDD généré (aucune `CREATE TABLE`). Aucun schéma généré.

# EntityManager

## Valeurs possibles pour `ddl-generation.output-mode`

- `both` : Le LDD sera généré et écrit à la fois dans la base de données et dans un fichier.
- `database` : (Valeur par défaut) Le LDD sera généré et écrit uniquement dans la base de données.
- `sql-script` : DDL sera généré et écrit uniquement dans un fichier.

Ce mode dépend évidemment de la configuration précédente (la valeur choisie pour `ddl-generation`).

# L'insertion

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
// instantiation de l'entité  
Personne p = new Personne();  
p.setPrenom("John");  
p.setNom("Wick");  
EntityTransaction transaction = em.getTransaction();  
transaction.begin(); // début transaction  
em.persist(p); // persister dans la base de données  
transaction.commit(); // valider transaction  
// fermer l'EntityManager et la factorie  
em.close();  
emf.close();
```

# L'insertion

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
// instantiation de l'entité  
Personne p = new Personne();  
p.setPrenom("John");  
p.setNom("Wick");  
EntityTransaction transaction = em.getTransaction();  
transaction.begin(); // début transaction  
em.persist(p); // persister dans la base de données  
transaction.commit(); // valider transaction  
// fermer l'EntityManager et la factorie  
em.close();  
emf.close();
```

```
transaction.rollback() pour annuler la transaction
```

# La mise à jour

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
// chercher l'élément a modifier  
Personne p = em.find(Personne.class, 10); // on indique la  
    classe et la valeur de la clé primaire  
EntityTransaction transaction = em.getTransaction();  
transaction.begin();  
if (p == null) {  
    System.out.println("Personne non trouvée");  
}  
else {  
    p.setPrenom("Bob"); // faire une modification  
    em.flush(); // enregistrer la modification  
}  
// valider la transaction  
transaction.commit();  
em.close();  
emf.close();
```

# La suppression

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
// chercher l'élément à supprimer  
Personne p = em.find(Personne.class, 10);  
EntityTransaction transaction = em.getTransaction();  
transaction.begin();  
if (p == null) {  
    System.out.println("Personne non trouvée");  
}  
else {  
    // supprimer l'élément  
    em.remove(p);  
}  
// valider la transaction  
transaction.commit();  
em.close();  
emf.close();
```

# La recherche

## Fonctions de recherche

- `find()` : cherche et retourne un élément selon la valeur de la clé primaire. Retourne `null` si la valeur n'existe pas
- `getReference()` : fonctionne comme `find()`. Mais elle déclenche une exception si la valeur n'existe pas

# La recherche

## Fonctions de recherche

- `find()` : cherche et retourne un élément selon la valeur de la clé primaire. Retourne `null` si la valeur n'existe pas
- `getReference()` : fonctionne comme `find()`. Mais elle déclenche une exception si la valeur n'existe pas

On peut aussi définir nos fonctions de recherche en utilisant des requêtes SQL

# La consultation

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
// definir la requete  
String str = "select * from Personne";  
// executer la requete et preciser l'entite concernee  
Query query = em.createNativeQuery(str, org.eclipse.model  
    .Personne.class);  
// recuperer le resultat  
List <Personne> personnes = (List <Personne> ) query.  
    getResultList();  
for (Personne p : personnes) {  
    System.out.println("nom = "+ p.getNom());  
}  
em.close();  
emf.close();
```

# La recherche

## Création de requêtes

- `createNativeQuery()` : permet d'exécuter une requête SQL
- `createNamedQuery()` : permet d'exécuter une requête JPQL définie par des annotations
- `createQuery()` : permet d'exécuter une requête JPQL

# La recherche

## Création de requêtes

- `createNativeQuery()` : permet d'exécuter une requête SQL
- `createNamedQuery()` : permet d'exécuter une requête JPQL définie par des annotations
- `createQuery()` : permet d'exécuter une requête JPQL

## Récupération du résultat

- `getResultList()` : permet de récupérer le résultat sous forme d'une liste
- `getSingleResult()` : permet de récupérer un seul élément de la sélection
- `getFirstResult()` : permet de récupérer le premier élément de la sélection

# La consultation

**Les requêtes natives : exemple avec `getSingleResult()` en utilisant les requêtes paramétrées**

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
Query query = em.createNativeQuery("select * from  
    Personne where num = ?", org.eclipse.model.  
    Personne.class);  
query.setParameter(1, 5);  
Personne p = (Personne) query.getSingleResult();  
System.out.println("nom = " + p.getNom());  
em.close();  
emf.close();
```

# La consultation

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

```
/**
 * Entity implementation class for Entity: Personne
 *
 */
@Entity
@NamedQuery(
    name="findByNomPrenom",
    query="SELECT p FROM Personne p WHERE p.nom = :
        nom and p.prenom = :prenom"
)
public class Personne implements Serializable {
    ...
}
```

# La consultation

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

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
Query query = em.createNamedQuery("findByNomPrenom")  
    ;  
query.setParameter("nom", "Wick");  
query.setParameter("prenom", "John");  
List <Personne> personnes = (List <Personne> ) query  
    .getResultList();  
for (Personne p : personnes) {  
    System.out.println("nom = "+ p.getNom());  
}  
em.close();  
emf.close();
```

# La consultation

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 implements Serializable {
  ...
}
```

# EntityManager

## Les méthodes utilisées de l'EntityManager

- `persist(entity)` : permet de persister une entité. Un `commit` engendrera l'enregistrement de cette entité dans la base de données.
- `remove(entity)` : permet de rendre une entité non persistante. Un `commit` engendrera la suppression de cette entité de la base de données.
- `flush()` : permet de persister les modifications. Un `commit` engendrera l'enregistrement de ces modifications dans la base de données.

# EntityManager

## Les méthodes utilisées de l'EntityManager

- `persist(entity)` : permet de persister une entité. Un commit engendrera l'enregistrement de cette entité dans la base de données.
- `remove(entity)` : permet de rendre une entité non persistante. Un commit engendrera la suppression de cette entité de la base de données.
- `flush()` : permet de persister les modifications. Un commit engendrera l'enregistrement de ces modifications dans la base de données.

### Attention

La méthode `flush()` ne prend pas de paramètre

# EntityManager

## Autres méthodes de l'EntityManager

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

# EntityManager

## Exemple avec utilisation de `refresh()`

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
//on suppose que John Wick avec un num 10 existe  
    dans la BD  
Personne p = em.find(Personne.class, 10);  
p.setNom("Travolta");  
em.refresh(p);  
System.out.println("le nom est " + p.getNom());  
// imprime Wick  
em.close();  
emf.close();  
// si on supprime em.refresh(p); Travolta sera  
    affiche
```

# EntityManager

## Exemple avec utilisation de `detach()`

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
//on suppose que John Wick avec un num 10 existe  
    dans la BD  
Personne p = em.find(Personne.class, 10);  
p.setNom("Travolta");  
em.detach(p); // p n'est plus gere par em  
em.getTransaction().begin();  
em.flush();  
em.getTransaction().commit();  
Personne p1 = em.find(Personne.class, 10);  
System.out.println("le nom est " + p1.getNom());  
// affiche le nom est Wick
```

# JPQL

## Définition

- JPQL est un langage de requêtes adapté à la spécification JPA
- Inspiré du langage SQL (et HQL : Hibernate Query Language) mais adapté aux entités JPA
- Permet de manipuler les entités JPA et pas les tables d'une base de données
- Supporte des requêtes de type `select`, `update` et `delete`

# JPQL

## Définition

- JPQL est un langage de requêtes adapté à la spécification JPA
- Inspiré du langage SQL (et HQL : Hibernate Query Language) mais adapté aux entités JPA
- Permet de manipuler les entités JPA et pas les tables d'une base de données
- Supporte des requêtes de type `select`, `update` et `delete`

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

# JPQL

## Exemple :

```
String str= "Select p from Personne as p";
Query query = em.createQuery(str);
List <Personne> personnes = (List <Personne> ) query
    .getResultList();
for (Personne p : personnes) {
    System.out.println(p.getNom() + " " + p.
        getPrenom());
}
```

## Explication

- On sélectionne des objets p de type `Personne`
- On peut aussi écrire "Select p from Personne p"

## Remarques

- La requête JPQL précédente permet de sélectionner un objet
- Il est tout de même possible de sélectionner seulement quelques attributs d'un objet
- Dans ce cas là, le résultat est un tableau contenant les champs attributs sélectionnés
- Et il est impossible de modifier (ou supprimer) les valeurs de ces attributs sélectionnées

# JPQL

Comme avec SQL, on peut faire des jointures

```
"select p from Personne p  
join p.sports s  
where s.nom= 'football'"
```

# JPQL

## Comme avec SQL, on peut faire des jointures

```
"select p from Personne p  
join p.sports s  
where s.nom= 'football'"
```

## Équivalent en SQL

```
"select *  
from Sport s, Personne_Sport ps, Personne p  
where s.nom = ps.nom and p.num = ps.num  
and s.nom = 'football'"
```

# JPQL

## Comme avec SQL, on peut faire des jointures

```
"select p from Personne p
join p.sports s
where s.nom= 'football'"
```

## Équivalent en SQL

```
"select *
from Sport s, Personne_Sport ps, Personne p
where s.nom = ps.nom and p.num = ps.num
and s.nom = 'football'"
```

## Explication

- La jointure se fait avec `join p.sports s`
- Le reste, c'est du SQL classique

## On peut également utiliser

- group by
- order by
- having
- distinct
- les fonctions d'agrégation count, avg...
- les requêtes imbriquées
- les mots-clés all, (not) in, like, between, (not) null, (not) exists...

## Attention aux collections

```
String str= "select p from Personne p  
where p.sports.nom= 'football'";  
Query query = em.createQuery(str);
```

- `p.sports` est une collection
- cette requête génère une erreur

# Relation entre entités

## 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

# Création de l'entité Adresse

```
package org.eclipse.model;

/**
 * Entity implementation class for
 * Entity: Adresse
 */
@Entity
public class Adresse implements
    Serializable {

    @Id
    private String rue;
    private String codePostal;
    private String ville;
    private static final long
        serialVersionUID = 1L;

    public Adresse() {
        super();
    }
    public String getRue() {
        return this.rue;
    }
}
```

```
    }
    public void setRue(String rue) {
        this.rue = rue;
    }

    public String getCodePostal() {
        return this.codePostal;
    }

    public void setCodePostal(String
        codePostal) {
        this.codePostal = codePostal;
    }

    public String getVille() {
        return this.ville;
    }

    public void setVille(String
        ville) {
        this.ville = ville;
    }
}
```

# Modifier l'entité Personne

```
@Entity
public class Personne implements Serializable {

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

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

    // les getters/setters de chaque attribut
```

# Modifier l'entité Personne

```
@Entity
public class Personne implements Serializable {

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

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

    // les getters/setters de chaque attribut
```

## Notation

- Personne : entité propriétaire
- Adresse : entité inverse

# Relation entre entités

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

## Explication

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

# Relation entre entités

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

## Explication

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

# Testons l'ajout d'une personne

```
EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("FirstJpaProject");  
EntityManager em = emf.createEntityManager();  
Adresse a = new Adresse();  
a.setRue("Lyon");  
a.setCodePostal("13015");  
a.setVille("Marseille");  
Personne p = new Personne();  
p.setAdresse(a);  
p.setNom("Ego");  
p.setPrenom("Paul");  
EntityTransaction t =em.getTransaction();  
transaction.begin();  
em.persist(p);  
transaction.commit();  
System.out.println("insertion reussie ");  
em.close();  
emf.close();
```

# Testons l'ajout d'une personne

## Sans la précision suivante

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

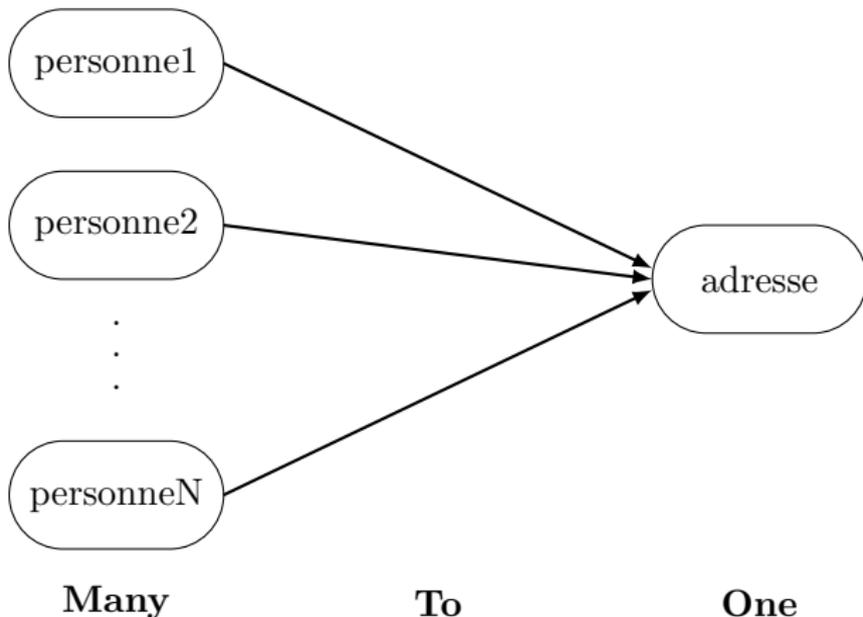
Il fallait persister l'objet adresse avant l'objet personne

```
EntityTransaction t =em.getTransaction();  
transaction.begin();  
em.persist(a);  
em.persist(p);  
transaction.commit();
```

# ManyToOne

## Exemple

Si on suppose que plusieurs personnes peuvent avoir la même adresse



# ManyToOne

Il faut juste changer

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

# ManyToOne

Il faut juste changer

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

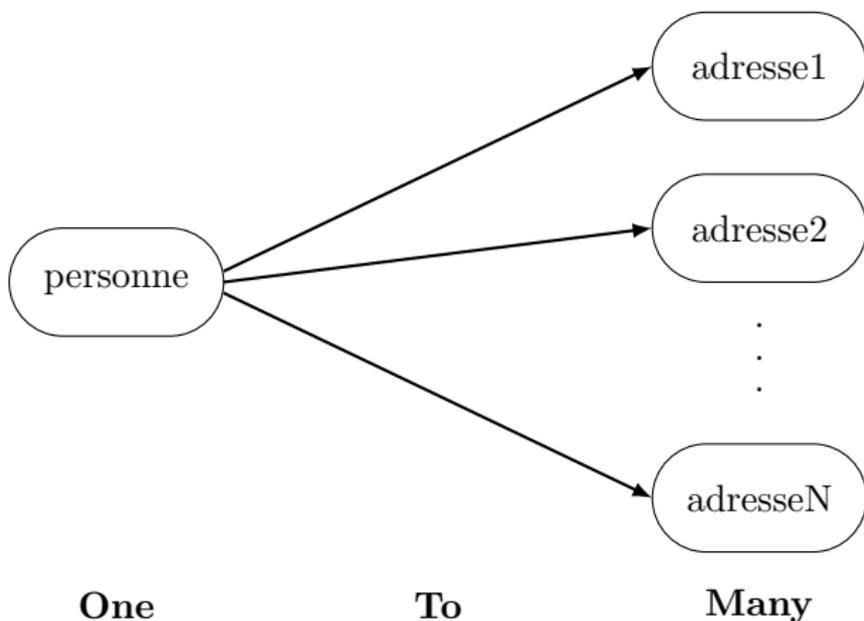
Le reste

C'est pareil

# OneToMany

## Exemple

Si on suppose qu'une personne peut avoir plusieurs adresses



# OneToMany

Il faut changer

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

On peut aussi définir quand les objets de l'entité inverse (ici Adresse) seront chargés dans l'entité propriétaire (ici Personne)

# OneToMany

Il faut ajouter dans l'annotation l'attribut `fetch`

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

# OneToMany

Il faut ajouter dans l'annotation l'attribut `fetch`

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

Deux valeurs possibles pour `CONSTANTE`

- **EAGER** : les objets de l'entité `Adresse` en relation avec un objet `personne` de l'entité `Personne` seront chargés au même temps.
- **LAZY** (par défaut) : les objets de l'entité `Adresse` en relation avec un objet `personne` de l'entité `Personne` seront chargés seulement quand on fait `personne.getAdresses'`).

# OneToMany

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

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

# OneToMany

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

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

## 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`).
- Modifier le nom de la méthode `add` en `addAdresse` et `remove` en `removeAdresse`

# OneToMany

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

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

## 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`).
- Modifier le nom de la méthode `add` en `addAdresse` et `remove` en `removeAdresse`

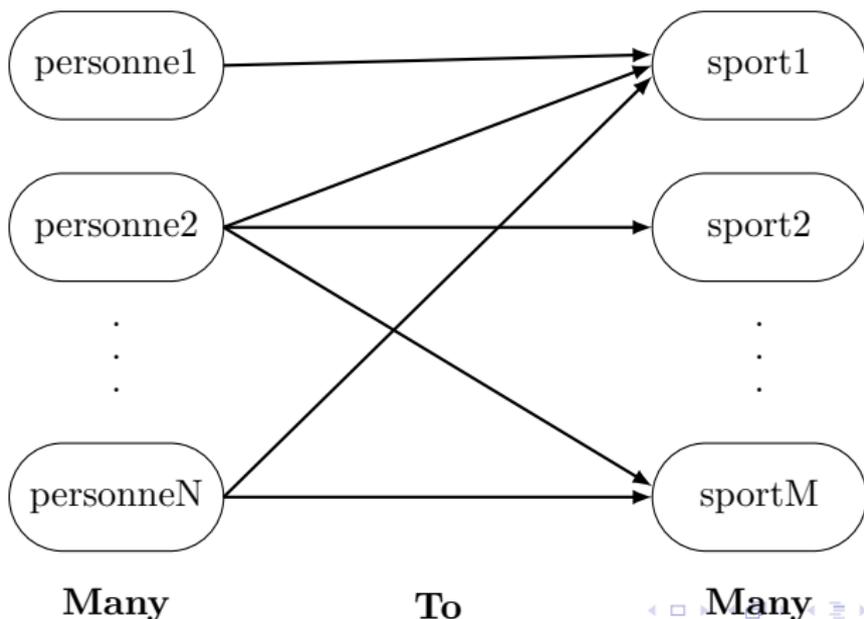
# Testons l'ajout d'une personne

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("
    FirstJpaProject");
EntityManager em = emf.createEntityManager();
Adresse a1 = new Adresse();
a1.setRue("Lyon");
a1.setCodePostal("13015");
a1.setVille("Marseille");
Adresse a2 = new Adresse();
a2.setRue("Paradis");
a2.setCodePostal("13003");
a2.setVille("Marseille");
Personne p1 = new Personne();
p1.setNom("Wick");
p1.setPrenom("John");
p1.addAdresse(a1);
p1.addAdresse(a2);
EntityTransaction t =em.getTransaction();
transaction.begin();
em.persist(p1);
transaction.commit();
System.out.println("insertion reussie ");
em.close();
emf.close();
```

# ManyToMany

## Exemple

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



# ManyToMany

## Il faut juste changer

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

# Création de l'entité Sport

```
public class Sport implements Serializable {

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

## Ajoutons le ManyToMany dans la classe Personne

```
@Entity
public class Personne implements Serializable {
    @Id
    private int num;
    private String nom;
    private String prenom;
    @ManyToMany(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    private List <Sport> sports = new ArrayList <Sport> ();

    // les getters/setters et constructeur

    public List<Sport> getSports() {
        return sports;
    }
    public void setSports(List<Sport> sports) {
        this.sports = sports;
    }
    public boolean addSport(Sport arg0) {
        return sports.add(arg0);
    }
    public boolean removeSport(Object arg0) {
        return sports.remove(arg0);
    }
}
```

# Testons tout ça

```
EntityManagerFactory emf =
    Persistence.
        createEntityManagerFactory("
            FirstJpaProject");
EntityManager em = emf.
    createEntityManager();
Personne p1 = new Personne();
Personne p2 = new Personne();
p1.setNom("Wick");
p1.setPrenom("John");
p2.setNom("Bob");
p2.setPrenom("Joe");
Sport s1 = new Sport();
Sport s2 = new Sport();
Sport s3 = new Sport();
s1.setNom("football");
s2.setNom("tennis");
s3.setNom("box");
s1.setType("collectif");
```

```
s2.setType("individuel");
s3.setType("collectif ou
    individuel");
p1.addSport(s1);
p1.addSport(s3);
p2.addSport(s1);
p2.addSport(s2);
p2.addSport(s3);
EntityManager t = em.
    getTransaction();
transaction.begin();
em.persist(p1);
em.persist(p2);
transaction.commit();
System.out.println("insertion
    reussie ");
em.close();
emf.close();
```

# Cas particulier

## Si la table 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

# Cas particulier

## Si la table 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

## 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

## 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

# Relations bidirectionnelles : exemple

## Modifier l'entité inverse Sport

```
/**
 * Entity implementation class for Entity: Sport
 *
 */
@Entity
public class Sport implements Serializable {

    @Id
    private String nom;
    private String type;
    private static final long serialVersionUID = 1L;

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

### mappedBy

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

# Relations bidirectionnelles : exemple

## Modifier aussi l'entité propriétaire `Personne`

```
@Entity
public class Personne implements Serializable {
    ...
    public void add(Sport s) {

        // ajouter this a la liste des personnes de ce sport

        s.addPersonne(this);
        sports.add(s);
    }

    public void remove(Sport s) {

        // supprimer this de la liste des personnes de ce sport

        s.removePersonne(this);
        sports.remove(s);
    }
}
```

La même chose à faire dans `Sport` pour la liste `personnes`. **Attention aux boucles infinies.**

# Relations bidirectionnelles : exemple

Ainsi, on peut faire :

```
p1.add(s1);  
p1.add(s3);  
p2.add(s1);  
p2.add(s2);  
p2.add(s3);  
EntityTransaction t = em.getTransaction();  
transaction.begin();  
em.persist(p1);  
em.persist(p2);  
transaction.commit();  
System.out.println(s1.getPersonnes().get(0).getNom()  
    );
```

# Relations bidirectionnelles : exemple

Ainsi, on peut faire :

```
p1.add(s1);  
p1.add(s3);  
p2.add(s1);  
p2.add(s2);  
p2.add(s3);  
EntityTransaction t = em.getTransaction();  
transaction.begin();  
em.persist(p1);  
em.persist(p2);  
transaction.commit();  
System.out.println(s1.getPersonnes().get(0).getNom()  
    );
```

affiche Wick

# Remarques

## Pour définir une relation bidirectionnelle entre deux entités

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

# L'association d'héritage

## Trois possibilités avec l'héritage

- SINGLE\_TABLE
- TABLE\_PER\_CLASS
- JOINED

# L'association d'héritage

## Trois possibilités avec l'héritage

- SINGLE\_TABLE
- TABLE\_PER\_CLASS
- JOINED

## Exemple

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

# L'association d'héritage

Pour indiquer comment transformer les classes mère et filles en tables

Il faut utiliser l'annotation `@Inheritance`

# L'association d'héritage

## Tout dans une seule table

Dans la classe mère on ajoute

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

# L'association d'héritage

## Exemple

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

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

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

# L'association d'héritage

## Exemple avec une table pour chaque entité

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

## Les classes incorporables : pas de table correspondante en BD

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 + "];"
    }
}
```

## Les classes incorporables : pas de table correspondante en BD

L'entité `Personne` devient ainsi :

```
@Entity
public class Personne {

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

    private NomComplet nomComplet;

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

## Les classes incorporables : pas de table correspondante en BD

**Pour persister une personne**

```
Personne personne = new Personne();  
NomComplet nomComplet = new NomComplet();  
nomComplet.setNom("travolta");  
nomComplet.setPrenom("john");  
personne.setNomComplet(nomComplet);  
EntityTransaction t = em.getTransaction();  
transaction.begin();  
em.persist(personne);  
transaction.commit();
```

## Les classes incorporables : pas de table correspondante en BD

**Pour persister une personne**

```
Personne personne = new Personne();  
NomComplet nomComplet = new NomComplet();  
nomComplet.setNom("travolta");  
nomComplet.setPrenom("john");  
personne.setNomComplet(nomComplet);  
EntityTransaction t = em.getTransaction();  
transaction.begin();  
em.persist(personne);  
transaction.commit();
```

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

# Les évènements

## 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()`)

# Les évènements

## 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

# Les évènements

## Une méthode `callback`

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

C'est comme les triggers en SQL

# Les annotations

- `@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.

# Les méthodes callback

## Exemple : l'entité `Personne`

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

# Les méthodes callback

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

# JEE & JPA

## Étapes

- Créer un nouveau projet JEE (en Allant dans `File`, ensuite `New` et chercher `Dynamic Web Project`)
- Saisir le nom du projet (dans `Project name` :)
- Dans l'onglet `Configuration`, cliquer sur `Modify...`
- Cocher la case `JPA` et choisir la dernière version de JPA
- Cliquer sur `Next` et configurer les options de JPA
- Générer le `web.xml` du projet JEE
- Valider

# JEE & JPA

## Remarques

- Ainsi, on a créé un projet JEE dans lequel on peut créer, utiliser et manipuler des entités JPA
- Sans cette configuration, nous ne pourrions pas utiliser l'API JPA

# Maven & JPA

## Étapes

- Créer un nouveau projet Maven
- Ajouter les liens et les répertoires qui manquent à ce projet
- Ajouter JPA

# Maven & JPA

## Étapes

- Créer un nouveau projet Maven
- Ajouter les liens et les répertoires qui manquent à ce projet
- Ajouter JPA

## Création d'un projet Maven

- Aller dans `File > New` et chercher `Maven Project`
- Choisir un projet `maven-archetype-webapp`
- Remplir le champs `Group Id` par `org.eclipse`
- Remplir le champs `Artifact Id` par `JeeJpaMavenProject`
- Valider

# JEE & JPA

## Si `index.jsp` est signalé en rouge

- Faire clic droit sur le nom du projet
- Choisir `Properties`
- Chercher puis sélectionner `Targeted Runtimes`
- Cocher la case `Apache Tomcat vX.X` puis cliquer sur `Apply and Close`

# JEE & JPA

## S'il n'y a pas de `src/main/java` dans Java Resources

- Faire clic droit sur le nom du projet
- Aller dans Build Path > Configure Build Path...
- Cliquer sur Order and Export
- Cocher les trois case Maven Dependencies, Apache Tomcat vX.X et JRE System Library
- Cliquer sur Apply and Close

# JEE & JPA

## Ajouter JPA

- Faire un clic droit sur le nom du projet et aller dans `Properties`
- Chercher `Project Facets`
- Cocher la case `JPA`
- Cliquer sur le lien qui apparaît (`Further configuration available...`) pour ajouter les données sur `EclipseLink` et la connexion
- Cliquer sur `Apply and Close`