



JPA : Java Persistence API

JPA

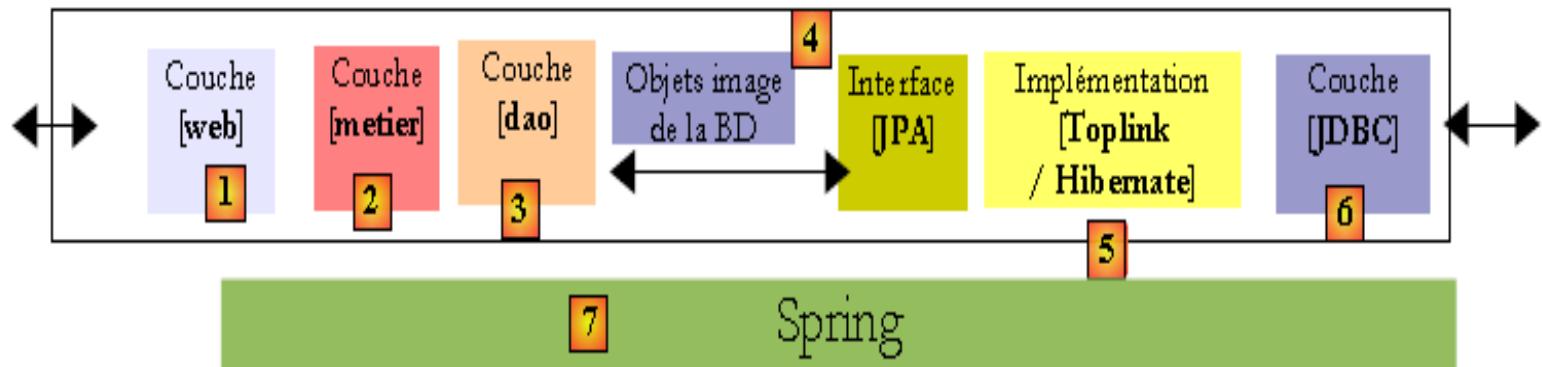
- Introduction
- Les entités
- Le mapping entre une entité et une table
- Le mapping de propriété complexe
- Mapper une entité sur plusieurs tables
- Utilisation d'objets embarqués dans les entités
- Fichier de configuration du mapping
- EntityManager
- Le fichier persistence.xml
- La gestion des relations entre tables dans le mapping
- La gestion de l'héritage

JPA : Introduction

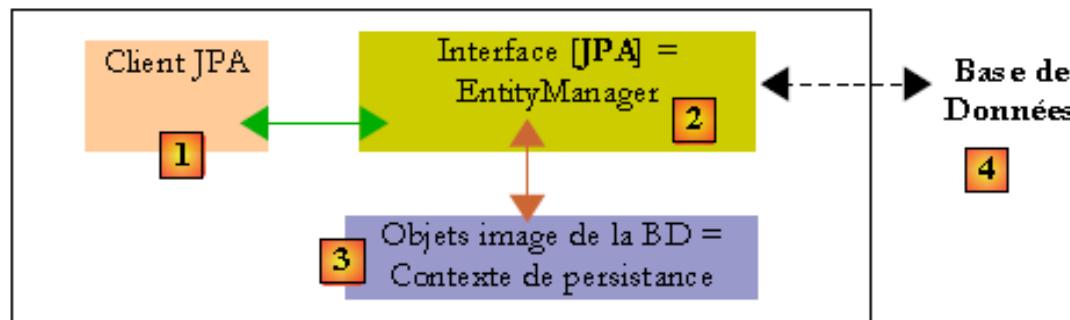
- JPA : Java Persistence API : outil ORM ;
- JPA ne requiert aucune ligne de code mettant en œuvre l'API JDBC ;
- JPA propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données ;
- JPA se base sur les entités (des objets POJO) et sur le gestionnaire de ces entités (EntityManager) ;
- EntityManager est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

JPA : Introduction

➤ Position du JPA dans une architecture 3-tiers :



➤ Le contexte de persistance d'une application :



JPA : Introduction

- L'implémentation de référence de JPA est incluse dans le projet *Glassfish*, disponible à l'URL :

<https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html>

- Cette implémentation de référence repose sur l'outil TopLink d'Oracle dans sa version essential ;
- Pour obtenir les fichiers .jar de JPA il suffit d'exécuter la commande suivante : *java -jar* avec en paramètre le fichier jar téléchargé.

- Exemple : la commande *java -jar glassfish-persistence-installer-v2-b58g.jar* donne comme résultat :



Nom	Date de modificati...	Type
<i>Capacité non spécifiée (5)</i>		
3RD-PARTY-LICENSE.txt	27/08/2009 23:43	Document texte
LICENSE.txt	27/08/2009 23:43	Document texte
README	27/08/2009 23:43	Fichier
toplink-essentials.jar	27/08/2009 23:43	Archive WinRAR
toplink-essentials-agent.jar	27/08/2009 23:43	Archive WinRAR

JPA : Les entités

- Les entités permettent d'encapsuler les données d'une ou plusieurs tables ;
- Les entités sont des simples *POJO (Plain Old Java Object)* ;
- Les entités utilisent les annotations pour mapper un objet JAVA vers une table de la base de données ;
- Un bean entité doit obligatoirement avoir un constructeur sans argument et la classe du bean doit obligatoirement être marquée avec l'annotation `@javax.persistence.Entity` ;
- L'annotation `@javax.persistence.Entity` possède un attribut optionnel nommé *name* qui permet de préciser le nom de l'entité dans les requêtes. Par défaut, ce nom est celui de la classe de l'entité.

JPA : Les entités

- Le bean Entité doit respecter les deux règles suivantes :
 - il doit être annoté par `@javax.persistence.Entity` ;
 - il doit posséder au moins une propriété déclarer comme clé primaire avec l'annotation `@Id`.
- Le bean entity est composé de propriétés qui seront mappés sur les champs de la table de la base de données ;
- Chaque propriété encapsule les données d'un champ d'une table. Ces propriétés sont utilisables au travers des getter et des setter ;
- Une propriété particulière est la clé primaire qui est l'identifiant unique dans la base de données et aussi dans le POJO.

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre table dans le mapping**
- **La gestion de l'héritage**

JPA: Le mapping entre une entité et une table

- La description du mapping entre le bean entité et la table peut être fait de deux façons :
 - Utiliser des annotations ;
 - Utiliser un fichier XML de mapping.
- L'API propose plusieurs annotations pour supporter un mapping O/R assez complet ;

JPA: Le mapping entre une entité et une table

Annotation	Rôle
<code>@javax.persistence.Table</code>	Préciser le nom de la table concernée par le mapping
<code>@javax.persistence.Column</code>	Permet d'associer un champ de la table à la propriété
<code>@javax.persistence.Id</code>	Permet d'associer un champ de la table à la propriété en tant que clé primaire
<code>@javax.persistence.GeneratedValue</code>	Demande la génération automatique de la clé primaire au besoin
<code>@javax.persistence.Basic</code>	Représente la forme de mapping la plus simple. Cette annotation est utilisée par défaut
<code>@javax.persistence.Transient</code>	Demande de ne pas tenir compte du champ lors du mapping

JPA: Le mapping entre une entité et une table

@Table

- L'entité `@Table` possède les attributs suivants :

Attribut	Rôle
<code>name</code>	Nom de la table
<code>schema</code>	Schéma de la table
<code>uniqueConstraints</code>	Contraintes d'unicité sur une ou plusieurs colonnes

JPA: Le mapping entre une entité et une table

@Column

- L'entité @Column possède les attributs suivants :

Attribut	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi-table
unique	Indique si la colonne est unique
nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update
length	Indique la taille d'une colonne de type chaîne de caractères
precision	Indique la taille d'une colonne de type numérique
scale	Indique la précision d'une colonne de type numérique

JPA: Le mapping entre une entité et une table

@GeneratedValue

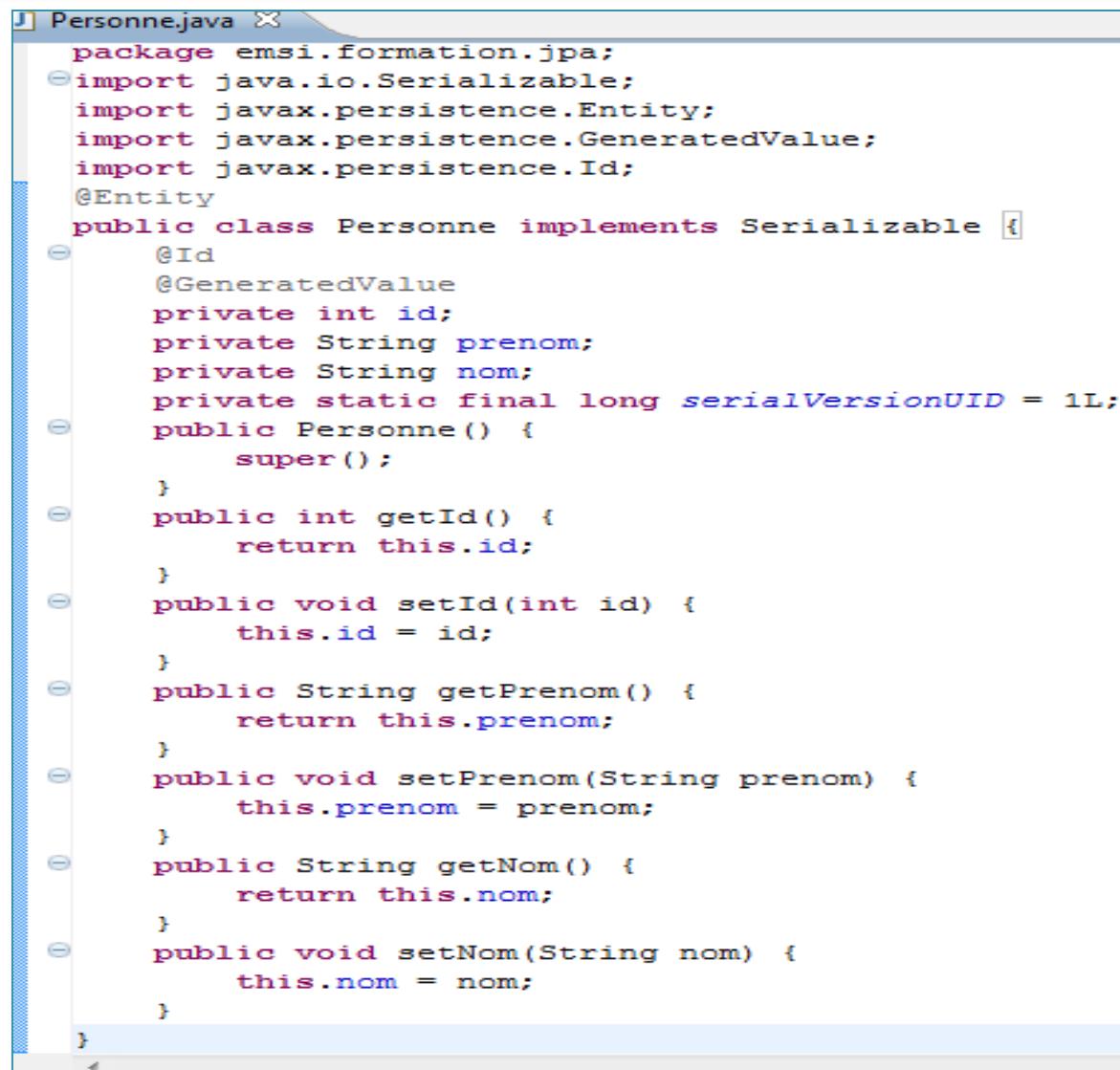
- L'annotation `@GeneratedValue` possède les attributs suivants :

Attribut	Rôle
<code>strategy</code>	Précise le type de générateur à utiliser : TABLE , SEQUENCE , IDENTITY ou AUTO . La valeur par défaut est AUTO
<code>generator</code>	Nom du générateur à utiliser

- Le type **AUTO** est le plus généralement utilisé : il laisse l'implémentation générer la valeur de la clé primaire ;
- Le type **IDENTITY**: La génération se fera à partir d'une Identité propre au SGBD. Il utilise un type de colonne spéciale à la base de données.

JPA: Le mapping entre une entité et une table @GeneratedValue

- Exemple :



The screenshot shows a Java code editor window titled "Personne.java". The code defines a class "Personne" that implements "Serializable". It uses annotations from the "javax.persistence" package: "@Entity", "@Id", and "@GeneratedValue". The class has two private fields, "id" and "prenom", and two private static final fields, "nom" and "serialVersionUID". It includes standard getters and setters for "id", "prenom", "nom", and "serialVersionUID". The code is color-coded, with keywords in blue and identifiers in black.

```
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    private String prenom;
    private String nom;
    private static final long serialVersionUID = 1L;
    public Personne() {
        super();
    }
    public int getId() {
        return this.id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getPrenom() {
        return this.prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

JPA: Le mapping entre une entité et une table

@GeneratedValue

- Le type **TABLE** utilise une table dédiée qui stocke les clés des tables générées. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation `@javax.persistence.TableGenerator` ;
- L'annotation `@TableGenerator` possède plusieurs attributs :

Attribut	Rôle
<code>name</code>	Nom identifiant le TableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation <code>@GeneratedValue</code>
<code>table</code>	Nom de la table utilisé
<code>schema</code>	Nom du schéma utiliser
<code>pkColumnName</code>	Nom de la colonne qui précise la clé primaire à générer
<code>valueColumnName</code>	Nom de la colonne qui contient la valeur de la clé primaire générée
<code>allocationSize</code>	Valeur utilisée lors de l'incrémentation de la valeur de la clé primaire

JPA: Le mapping entre une entité et une table

@GeneratedValue

- Le type **SEQUENCE** utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle ;
- L'utilisation de cette stratégie nécessite l'utilisation de l'annotation **@javax.persistence.SequenceGenerator** ;
- L'annotation **@SequenceGenerator** possède plusieurs attributs :

Attribut	Rôle
name	Nom identifiant le SequenceTableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation @GeneratedValue
sequenceName	Nom de la séquence dans la base de données
initialValue	Valeur initiale de la séquence
allocationSize	Valeur utilisée lors l'incrémentation de la valeur de la séquence

JPA: Le mapping entre une entité et une table

@GeneratedValue

- Exemple :

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@SequenceGenerator(name = "PERSONNE_SEQUENCE", sequenceName = "PERSONNE_SEQ")
public class Personne implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PERSONNE_SEQUENCE")
    private int id;
```

JPA: Le mapping entre une entité et une table @GeneratedValue

- Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes. JPA propose deux façons de gérer ce cas de figure :
 - L'annotation `@javax.persistence.IdClass`
 - L'annotation `@javax.persistence.EmbeddedId`
- L'annotation `@IdClass` s'utilise avec une classe qui va encapsuler les propriétés qui composent la clé primaire. Cette classe doit obligatoirement :
 - Être sérialisable ;
 - Posséder un constructeur sans argument ;
 - Fournir une implémentation dédiée des méthodes `equals()` et `hashCode()`.

JPA: Le mapping entre une entité et une table

@GeneratedValue

Exemple : La clé primaire est composée des champs nom et prenom (exemple théorique qui présume que deux personnes ne peuvent avoir le même nom et prénom)

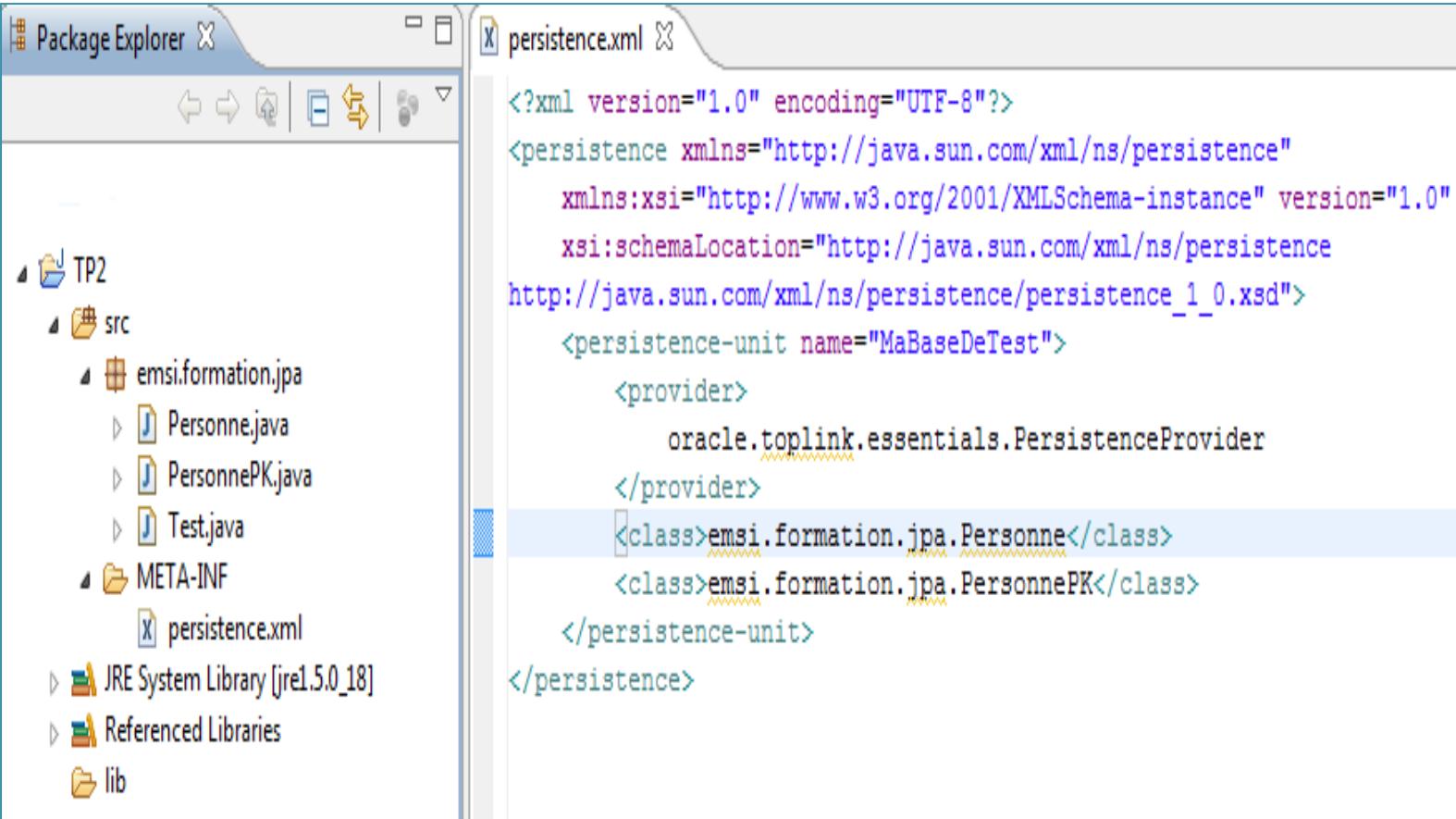
```
PersonnePK.java X
package emsi.formation.jpa;
public class PersonnePK implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private String nom;
    private String prenom;
    public PersonnePK() {
    }
    public PersonnePK(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
}
```

```
PersonnePK.java X
public boolean equals(Object obj) {
    boolean resultat = false;
    if (obj == this) {
        resultat = true;
    } else {
        if (!(obj instanceof PersonnePK)) {
            resultat = false;
        } else {
            PersonnePK autre = (PersonnePK) obj;
            if (!nom.equals(autre.nom)) {
                resultat = false;
            } else {
                if (prenom != autre.prenom) {
                    resultat = false;
                } else {
                    resultat = true;
                }
            }
        }
    }
    return resultat;
}
public int hashCode() {
    return (nom + prenom).hashCode();
}
}
```

JPA: Le mapping entre une entité et une table

@GeneratedValue

- Il est nécessaire de définir la classe de la clé primaire dans le fichier de configuration *persistence.xml* :



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays a project structure for 'TP2' containing 'src' (with 'emsi.formation.jpa' package containing 'Personne.java', 'PersonnePK.java', and 'Test.java'), 'META-INF' (with 'persistence.xml'), and 'lib'. On the right, the editor view shows the XML content of 'persistence.xml'.

```
<?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" version="1.0"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="MaBaseDeTest">
        <provider>
            oracle.toplink.essentials.PersistenceProvider
        </provider>
        <class>emsi.formation.jpa.Personne</class>
        <class>emsi.formation.jpa.PersonnePK</class>
    </persistence-unit>
</persistence>
```

JPA: Le mapping entre une entité et une table

@GeneratedValue

- L'annotation `@IdClass` possède un seul attribut :

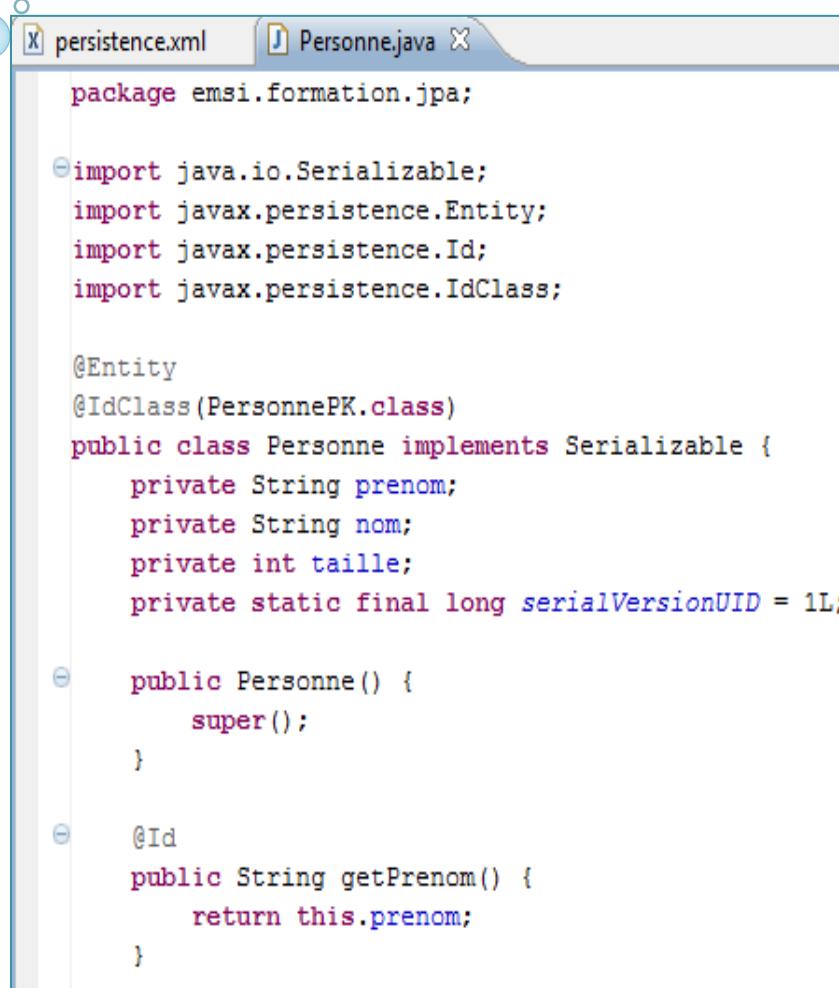
Attribut	Rôle
<code>Class</code>	Classe qui encapsule la clé primaire composée

- Il faut utiliser l'annotation `@IdClass` sur la classe de l'entité ;
- Il est nécessaire de marquer chacune des propriétés de l'entité qui compose la clé primaire avec l'annotation `@Id` ;
- Ces propriétés doivent avoir le même nom dans l'entité et dans la classe qui encapsule la clé primaire ;

JPA: Le mapping entre une entité et une table

@GeneratedValue

Exemple :



The screenshot shows two tabs in an IDE: "persistence.xml" and "Personne.java". The "persistence.xml" tab contains the XML configuration for the persistence unit. The "Personne.java" tab contains the Java code for the entity class.

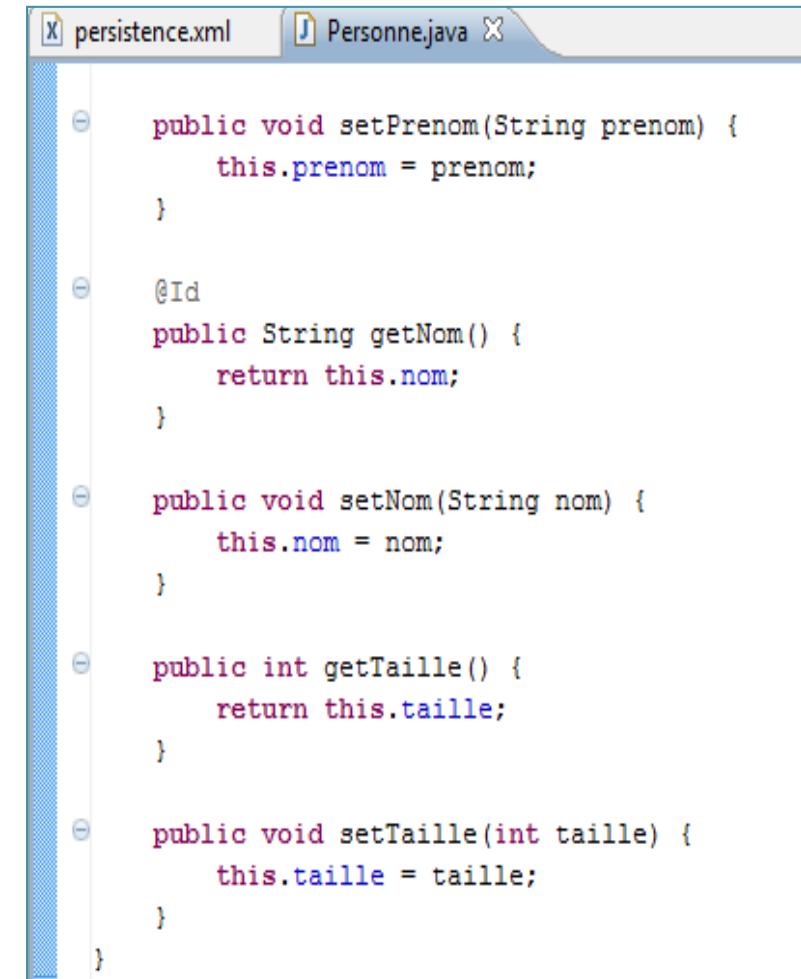
```
package emsi.formation.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {
    private String prenom;
    private String nom;
    private int taille;
    private static final long serialVersionUID = 1L;

    public Personne() {
        super();
    }

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



The screenshot shows two tabs in an IDE: "persistence.xml" and "Personne.java". The "persistence.xml" tab contains the XML configuration for the persistence unit. The "Personne.java" tab contains the Java code for the entity class, showing the generated methods for the @GeneratedValue annotation.

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

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

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

public int getTaille() {
    return this.taille;
}

public void setTaille(int taille) {
    this.taille = taille;
}
```

JPA: Le mapping entre une entité et une table

@GeneratedValue

- Il n'est pas possible de demander la génération automatique d'une clé primaire composée ;
- La classe de la clé primaire est utilisée notamment lors de la recherche ;

Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");  
Personne personne = entityManager.find(Personne.class, clePersonne);
```

JPA: Le mapping entre une entité et une table

@GeneratedValue

- L'annotation `@EmbeddedId` s'utilise avec l'annotation `@javax.persistence.Embeddable`

```
PersonnePK.java X
package emsi.formation.jpa;
import javax.persistence.Embeddable;
@Embeddable
public class PersonnePK implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private String nom;
    private String prenom;
    public PersonnePK() {
    }
    public PersonnePK(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
}
```

```
PersonnePK.java X
public boolean equals(Object obj) {
    boolean resultat = false;
    if (obj == this) {
        resultat = true;
    } else {
        if (!(obj instanceof PersonnePK)) {
            resultat = false;
        } else {
            PersonnePK autre = (PersonnePK) obj;
            if (!nom.equals(autre.nom)) {
                resultat = false;
            } else {
                if (prenom != autre.prenom) {
                    resultat = false;
                } else {
                    resultat = true;
                }
            }
        }
    }
    return resultat;
}
public int hashCode() {
    return (nom + prenom).hashCode();
}
}
```

JPA: Le mapping entre une entité et une table @GeneratedValue



```
PersonnePK.java Personne.java
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
@Entity
public class Personne implements Serializable {
    @EmbeddedId
    private PersonnePK clePrimaire;
    private int taille;
    private static final long serialVersionUID = 1L;
    public Personne() {
        super();
    }
    public PersonnePK getClePrimaire() {
        return clePrimaire;
    }
    public void setClePrimaire(PersonnePK clePrimaire) {
        this.clePrimaire = clePrimaire;
    }
    public int getTaille() {
        return taille;
    }
    public void setTaille(int taille) {
        this.taille = taille;
    }
}
```

JPA: Le mapping entre une entité et une table

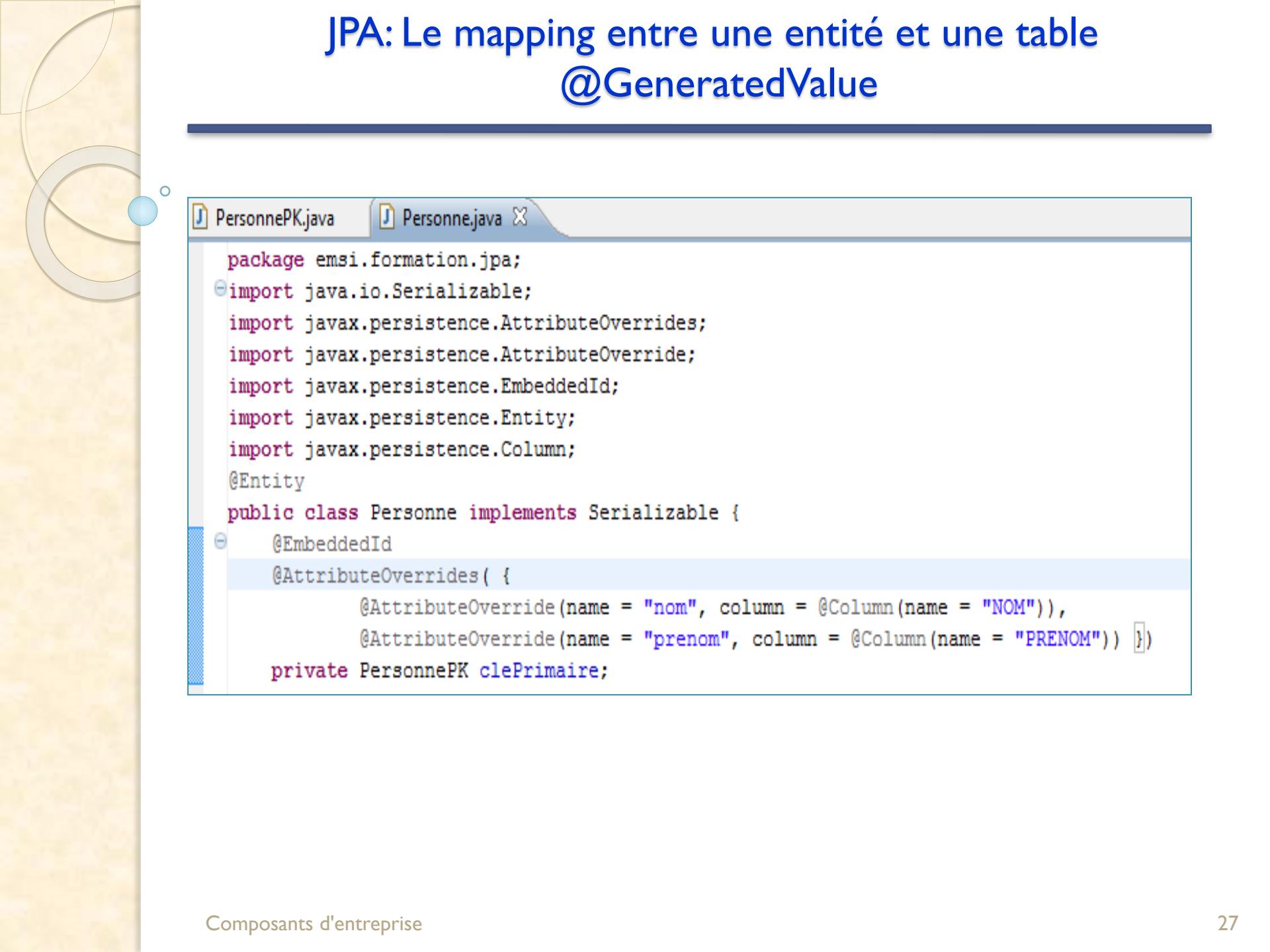
@GeneratedValue

- L'annotation `@AttributeOverrides` est une collection d'attributs `@AttributeOverride` ;
- Ces annotations permettent de ne pas avoir à utiliser l'annotation `@Column` dans la classe de la clé ou de modifier les attributs de cette annotation dans l'entité qui la met en œuvre ;
- L'annotation `@AttributeOverride` possède plusieurs attributs :

Attribut	Rôle
<code>name</code>	Précise le nom de la propriété de la classe imbriquée
<code>column</code>	Précise la colonne de la table à associer à la propriété

JPA: Le mapping entre une entité et une table

@GeneratedValue



```
PersonnePK.java Personne.java
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.AttributeOverrides;
import javax.persistence.AttributeOverride;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Column;
@Entity
public class Personne implements Serializable {
    @EmbeddedId
    @AttributeOverrides( {
        @AttributeOverride(name = "nom", column = @Column(name = "NOM")),
        @AttributeOverride(name = "prenom", column = @Column(name = "PRENOM")) })
    private PersonnePK clePrimaire;
```

JPA: Le mapping entre une entité et une table

@Transient

- Par défaut, toutes les propriétés sont mappées sur la colonne correspondante dans la table.
- L'annotation `@javax.persistence.Transient` permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.

JPA: Le mapping entre une entité et une table

@Basic

- L'annotation `@javax.persistence.Basic` représente la forme de mapping la plus simple.
- `@Basic` est optionnelle et est celle utilisée par défaut ;
- Ce mapping concerne :
 - les types primitifs ;
 - les wrappers de type primitifs ;
 - les tableaux de ces types ;
 - les types `java.math.BigInteger` ;
 - `java.math.BigDecimal` ;
 - `java.util.Date` ;
 - `java.util.Calendar` ;
 - `java.sql.Date` ;
 - `java.sql.Time` et `java.sql.Timestamp`.

JPA: Le mapping entre une entité et une table

@Basic

- L'annotation @Basic possède plusieurs attributs :

Attribut	Rôle
fetch	<p>Permet de préciser comment la propriété est chargée selon deux mode :</p> <ul style="list-style-type: none">■ LAZY : la valeur est chargée uniquement lors de son utilisation ;■ EAGER : la valeur est toujours chargée (valeur par défaut). <p>Cette fonctionnalité permet de limiter la quantité de données obtenues par une requête.</p>
optionnal	Indique que la colonne est nullable.

JPA: Le mapping entre une entité et une table

@Basic

➤ Exemple :

```
Personne.java X
package emsi.formation.jpa;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

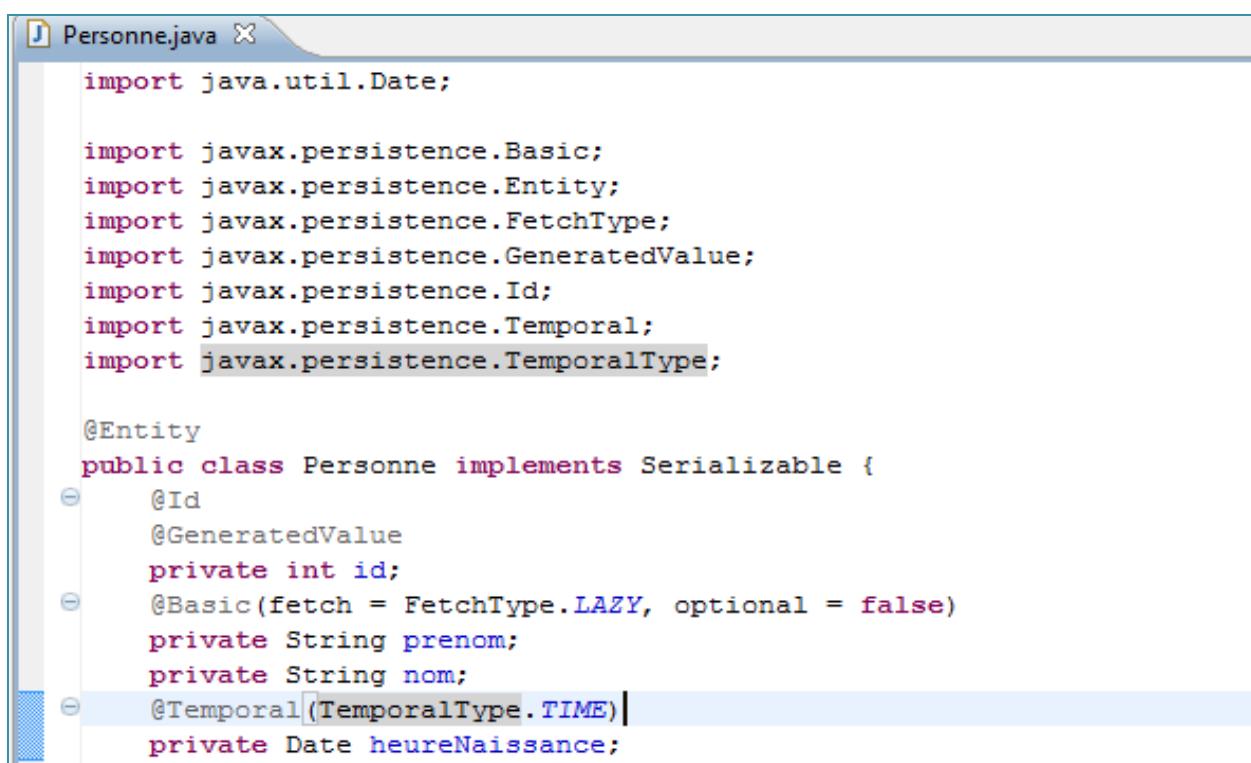
@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    private static final long serialVersionUID = 1L;

    public int getId() {
        return id;
    }
}
```

JPA: Le mapping entre une entité et une table

@Temporal

- L'annotation `@javax.persistence.Temporal` permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (Date et Calendar) ;
- La valeur par défaut est `timestamp` ;
- Exemple :



The screenshot shows a Java code editor with a file named "Personne.java". The code defines an entity "Personne" with various fields and annotations. The "heureNaissance" field is annotated with `@Temporal(TemporalType.TIME)`.

```
Personne.java
import java.util.Date;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    @Temporal(TemporalType.TIME)
    private Date heureNaissance;
}
```

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre table dans le mapping**
- **La gestion de l'héritage**

JPA: Le mapping de propriété complexe

- JPA permet de mapper les champs de types Blob et Clob ;
- L'annotation `@javax.persistence.Lob` permet mapper une propriété sur une colonne de type Blob ou Clob selon le type de la propriété :
 - Blob pour les tableaux de byte ou Byte ou les objets sérialisables ;
 - Clob pour les chaînes de caractères et les tableaux de caractères char ou Char.

JPA: Le mapping de propriété complexe @Lob

➤ exemple :

```
Personne.java X
package emsi.formation.jpa;
import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
import com.sun.imageio.plugins.jpeg.JPEG;
@Entity
public class Personne implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    @Lob
    @Basic(fetch = FetchType.LAZY)
    private JPEG photo;
    private static final long serialVersionUID = 1L;
```

```
Personne.java X
public Personne() {
    super();
}
public int getId() {return this.id;}
public void setId(int id) {
    this.id = id;
}
public JPEG getPhoto() {return photo;}
public void setPhoto(JPEG photo) {this.photo = photo;}
public String getPrenom() {return prenom;}
public void setPrenom(String prenom) {
    this.prenom = prenom;
}
public String getNom() {return nom;}
public void setNom(String nom) {this.nom = nom;}
}
```

JPA: Le mapping de propriété complexe @Enumerated

- L'annotation `@javax.persistence.Enumerated` permet d'associer une propriété de type énumération à une colonne de la table sous la forme d'un numérique ou d'une chaîne de caractères ;
- Cette forme est précisée en paramètre de l'annotation grâce à l'énumération `EnumType` qui peut avoir comme valeur `EnumType.ORDINAL` (valeur par défaut) ou `EnumType.STRING`.

➤ exemple :

```
public enum Genre {  
    HOMME,  
    FEMME,  
    INCONNU  
}
```

```
@Entity  
public class Personne implements Serializable {  
    @Id  
    @GeneratedValue  
    private int id;  
  
    @Basic(fetch = FetchType.LAZY, optional = false)  
    private String prenom;  
  
    private String nom;  
  
    @Enumerated(EnumType.STRING)  
    private Genre genre;
```

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre table dans le mapping**
- **La gestion de l'héritage**

JPA: Mapper une entité sur plusieurs tables

- L'annotation `@javax.persistence.SecondaryTable` permet de préciser qu'une autre table sera utilisée dans le mapping ;
- Pour utiliser cette fonctionnalité, la seconde table doit posséder une jointure entre sa clé primaire et une ou plusieurs colonnes de la première table.
- L'annotation `@SecondaryTable` possède plusieurs attributs :

Attribut	Rôle
Name	Nom de la table
Schema	Nom du schéma
pkJoinColumns	Collection des clés primaire de la jointure sous la forme d'annotations de type <code>@PrimaryKeyJoinColumn</code>
uniqueConstraints	

JPA: Mapper une entité sur plusieurs tables

@SecondaryTable

- L'annotation `@PrimaryKeyJoinColumn` permet de préciser une colonne qui compose la clé primaire de la seconde table et entre dans la jointure avec la première table. Elle possède plusieurs attributs :

Attribut	Rôle
<code>name</code>	Nom de la colonne
<code>referencedColumnName</code>	Nom de la colonne dans la première table (obligatoire si les noms de colonnes sont différents entre les deux tables)
<code>columnDefinition</code>	

- Il est nécessaire pour chaque propriété de l'entité qui est mappée sur la seconde table de renseigner le nom de la table dans l'attribut `table` de l'annotation `@Column` ;

JPA: Mapper une entité sur plusieurs tables @SecondaryTable

➤ Exemple :

```
J PersonneAdresse.java X
package emsi.forum;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@SecondaryTable(name = "ADRESSE", pkJoinColumns = { @PrimaryKeyJoinColumn(name = "ID_ADRESSE") })
public class PersonneAdresse implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Basic(fetch = FetchType.LAZY, optional = false)
    private String prenom;
    private String nom;
    @Column(name = "RUE", table = "ADRESSE")
    private String rue;
    @Column(name = "CODEPOSTAL", table = "ADRESSE")
    private String codePostal;
    @Column(name = "VILLE", table = "ADRESSE")
    private String ville;
    private static final long serialVersionUID = 1L;
```

Database **mabasedetest** - Table **ADRESSE** running on **localhost**

Field	Type	Attributes	Null	Default	Extra	Action
ID_ADRESSE	int(11)		No	0		
RUE	varchar(50)		No			
CODEPOSTAL	varchar(7)		No			
VILLE	varchar(50)		No			

JPA: Mapper une entité sur plusieurs tables

@SecondaryTables

- Si le mapping d'une entité utilise plusieurs tables, il faut utiliser l'annotation `@javax.persistence.SecondaryTables` qui est une collection d'annotation `@SecondaryTable` ;
- Exemple :



```
PersonneAdresse.java X
package emsi.formation.jpa;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.SecondaryTables;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@SecondaryTables( {
    @SecondaryTable(name = "ADRESSE", pkJoinColumns = { @PrimaryKeyJoinColumn(name = "ID_ADRESSE") }),
    @SecondaryTable(name = "INFO_PERS", pkJoinColumns = { @PrimaryKeyJoinColumn(name = "ID_INFO_PERS") }) })
public class PersonneAdresse implements Serializable {

    @Id
    @GeneratedValue
    private int id;
```

- Travaux pratiques : Réaliser le TPI relatif à l'API JPA.

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre table dans le mapping**
- **La gestion de l'héritage**

JPA: Utilisation d'objets embarqués dans les entités

- JPA permet d'utiliser dans les entités des objets Java qui ne sont pas des entités mais qui sont agrégés dans l'entité et dont les propriétés seront mappées sur les colonnes correspondantes dans la table ;
- La mise en œuvre de cette fonctionnalité est similaire à celle utilisée avec l'annotation `@EmbeddedId` pour les clés primaires composées ;
- La classe embarquée est un simple *POJO* qui doit être marquée avec l'annotation `@javax.persistence.Embeddable` ;
- **Travaux pratiques** : Réaliser le **TP2** relatif à l'API JPA.

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre tables dans le mapping**
- **La gestion de l'héritage**

JPA: Fichier de configuration du mapping

- JPA offre la possibilité de définir le mapping dans un fichier XML ;
- Le nom du fichier est par défaut *orm.xml* stocké dans le répertoire **META-INF** ;
- L'élément racine est le tag `<entity-mappings>` ;
- Pour chaque entité, il faut utiliser un tag fils `<entity>`. Ce tag possède deux attributs :
 - *Class* : qui permet préciser le nom pleinement qualifié de la classe de l'entité ;
 - *Access* : qui permet de préciser le type d'accès aux données (*PROPERTY* pour un accès via les getter/setter ou *FIELD* pour un accès via les champs).
- La déclaration de la clé primaire se fait dans un tag `<id>` fils d'un tag `<attributes>`.
- Le tag `<id>` possède un attribut nommé *name* qui permet de préciser le nom du champ qui est la clé primaire.

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **EntityManager**
 - **Le fichier persistence.xml**
 - **La gestion des relations entre tables dans le mapping**
 - **La gestion de l'héritage**

JPA: EntityManager

- Les interactions entre la base de données et les beans entité sont assurées par un objet de type *javax.persistence.EntityManager* ;
- *EntityManager* permet de lire, rechercher des données et de les mettre à jour (*ajout,modification, suppression*) ;
- *EntityManager* assure les interactions avec le gestionnaire de transactions ;
- *EntityManager* gère un ensemble défini de beans entité nommé *persistence unit* ;
- La définition d'un *persistence unit* est assurée dans un fichier de description nommé *persistence.xml*.

JPA: EntityManager

1. Etape n°1 : Création de la session factory.
2. Etape n°2 : création de la session.
3. Etape n°3 : Ouverture d'une transaction.
4. Etape n°4 : Exécution de la requête.
5. Etape n°5 : validation de la transaction.
6. Etape n°6 : Fermeture de la session.
7. Etape n°7 : fermeture de la factory.

JPA: EntityManager

1. Etape n°1 : Création de la session factory.

```
package ma.formations.jpa.dao;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
public class SessionBuilder {
    private static EntityManagerFactory sessionFactory;
    private SessionBuilder() {
    }
    public static EntityManagerFactory getSessionFactory() {
        try {
            if (sessionFactory == null)
                sessionFactory=Persistence.createEntityManagerFactory("unite1");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sessionFactory;
    }
}
```

JPA: EntityManager

Etape n°2 : création de la session.

```
EntityManager session =SessionBuilder.getSessionFactory().createEntityManager();
```

Etape n°3 : Ouverture d'une transaction

```
EntityTransaction tx = session.getTransaction();
tx.begin();
```

JPA: EntityManager

- **La méthode persist**

La méthode **persist** ne se contente pas d'enregistrer une entité en base, elle positionne également la valeur de l'attribut représentant la clé de l'entité. La détermination de la valeur de la clé dépend de la stratégie spécifiée par [@GeneratedValue](#). L'insertion en base ne se fait pas nécessairement au moment de l'appel à la méthode **persist** (on peut toutefois forcer l'insertion avec la méthode **EntityManager.flush()**). Cependant, l'[EntityManager](#) garantit que des appels successifs à sa méthode **find** permettront de récupérer l'instance de l'entité.

- **La méthode find**

La méthode **EntityManager.find (Class<T>, Object)** permet de rechercher une entité en donnant sa clé primaire. Un appel à cette méthode ne déclenche pas forcément une requête **SELECT** vers la base de données.

En effet, un [EntityManager](#) agit également comme un cache au-dessus de la base de données. Ainsi, il garantit l'unicité des instances des objets. Si la méthode **find** est appelée plusieurs fois sur la même instance d'un [EntityManager](#) avec une clé identique, alors l'instance retournée est toujours **la même**.

JPA: Les méthodes EntityManager

- **La méthode refresh**

La méthode `EntityManager.refresh(Object)` annule toutes les modifications faites sur l'entité durant la transaction courante et recharge son état à partir des valeurs en base de données.

- **La méthode merge**

La méthode `EntityManager.merge(T)` est parfois considérée comme la méthode permettant de réaliser les `UPDATE` des entités en base de données. Il n'en est rien et la sémantique de la méthode `merge` est très différente. En fait, il **n'existe pas** à proprement parlé de méthode pour réaliser la mise à jour d'une entité. Un `EntityManager` surveille les entités dont il a la charge et réalise les mises à jour si nécessaire au commit de la transaction. Par exemple le code ci-dessous suffit à déclencher une requête SQL UPDATE

JPA: Les méthodes EntityManager

- **La méthode detach**

Comme son nom l'indique, la méthode `EntityManager.detach(Object)` détache une entité, c'est-à-dire que l'instance passée en paramètre ne sera plus gérée par l'`EntityManager`. Ainsi, lors du commit de la transaction, les modifications faites sur l'entité détachée ne seront pas prises en compte.

- **La méthode remove**

La méthode `EntityManager.remove(Object)` supprime une entité. Si l'entité a déjà été persistée en base de données, cette méthode entraînera une requête SQL DELETE.

JPA: EntityManager

➤ Exemple :

Exemple :

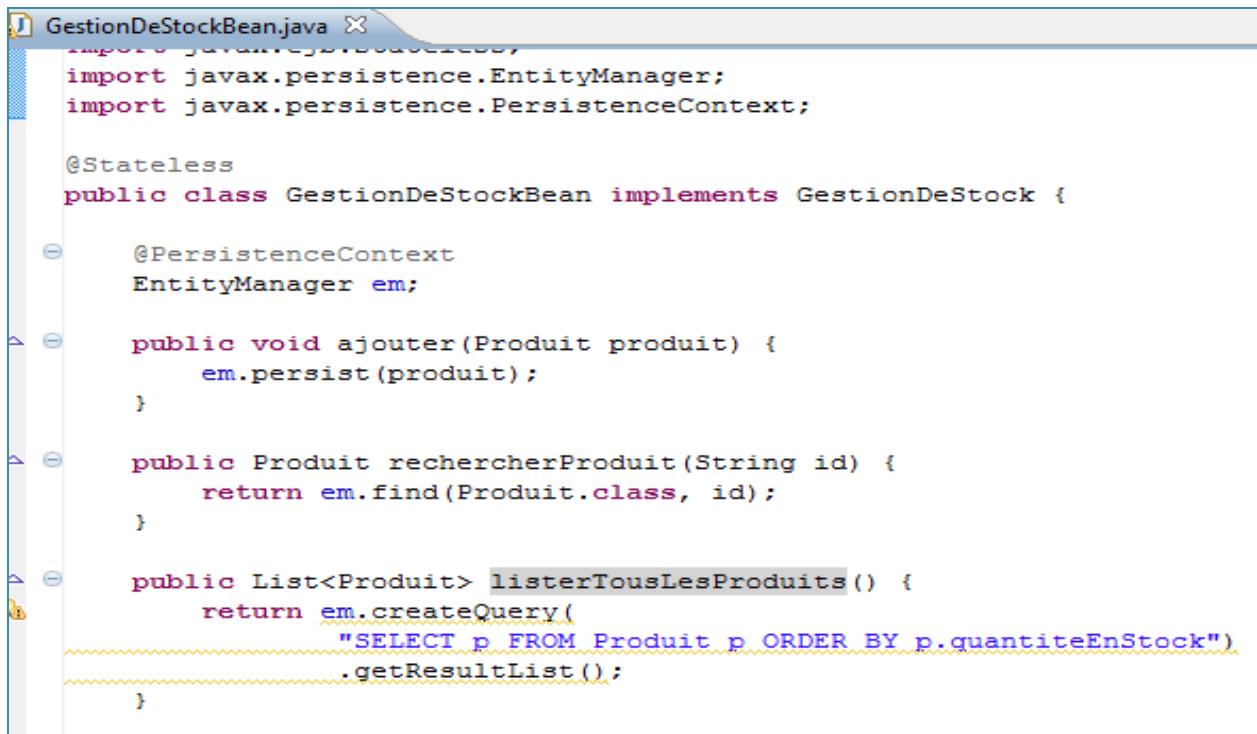
```
@PersistenceUnit(unitName="MaBaseDeTestPU")  
private EntityManagerFactory factory;
```

- Il est possible d'utiliser la fabrique pour obtenir un objet de type *EntityManager*.
- Pour associer ce contexte à la transaction courante, il faut utiliser la méthode *joinTransaction()* ;
- La méthode *close()* est automatiquement appelée par le conteneur : il ne faut pas utiliser cette méthode dans un conteneur sinon une exception de type *IllegalStateException* est levée.

JPA: EntityManager

- L'annotation `@javax.persistence.PersistenceContext` sur un champ de type `EntityManager` permet d'injecter un contexte de persistance ;
- Cette annotation possède un attribut `unitName` qui précise le nom de l'unité de persistance ;

➤ Exemple :



The screenshot shows a Java code editor with a file named "GestionDeStockBean.java". The code defines a stateless bean class "GestionDeStockBean" that implements an interface "GestionDeStock". The class contains three methods: "ajouter", "rechercherProduit", and "listerTousLesProduits". The "ajouter" method takes a "Produit" object and calls "em.persist(produit)". The "rechercherProduit" method takes a string "id" and returns a "Produit" object found by "em.find(Produit.class, id)". The "listerTousLesProduits" method returns a list of "Produit" objects obtained by executing a query: "SELECT p FROM Produit p ORDER BY p.quantiteEnStock". The code uses annotations such as @Stateless, @PersistenceContext, and @PersistenceContext(unitName = "MyPU").

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContext(unitName = "MyPU");

@Stateless
public class GestionDeStockBean implements GestionDeStock {

    @PersistenceContext
    EntityManager em;

    public void ajouter(Produit produit) {
        em.persist(produit);
    }

    public Produit rechercherProduit(String id) {
        return em.find(Produit.class, id);
    }

    public List<Produit> listerTousLesProduits() {
        return em.createQuery(
            "SELECT p FROM Produit p ORDER BY p.quantiteEnStock")
            .getResultList();
    }
}
```

JPA: EntityManager

➤ La méthode *contains()* :

Permet de savoir si une instance fournie en paramètre est gérée par le contexte. Dans ce cas, elle renvoie *true*, sinon elle renvoie *false* ;

➤ La méthode *close()* : le contexte de persistance est fermé. Force la synchronisation du contexte de persistance avec la base de données :

- si un objet du contexte n'est pas présent dans la base, il y est mis par une opération SQL *INSERT* ;
- si un objet du contexte est présent dans la base et qu'il a été modifié depuis qu'il a été lu, une opération SQL *UPDATE* est faite pour persister la modification ;
- si un objet du contexte a été marqué comme " supprimé " à l'issue d'une opération *remove* sur lui, une opération SQL *DELETE* est faite pour le supprimer de la base.

JPA: EntityManager

- La méthode *clear()* :
permet de détacher toutes les entités gérées par le contexte. Dans ce cas, toutes les modifications apportées aux entités sont perdues ;
 - il est préférable d'appeler la méthode *flush()* avant la méthode *clear()*.

- La méthode *flush()* :
Le contexte de persistance est synchronisé avec la base de données de la façon décrite pour *close()*.

JPA: EntityManager

- Le mode de synchronisation est géré par les méthodes suivantes de l'interface EntityManager :

`void setFlushMode(FlushModeType flushMode)` : Il y a deux valeurs possibles pour flushmode :

- `FlushModeType.AUTO` (défaut) : la synchronisation a lieu avant chaque requête SELECT faite sur la base ;
- `FlushModeType.COMMIT` : la synchronisation n'a lieu qu'à la fin des transactions sur la base.

`FlushModeType getFlushMode()` : rend le mode actuel de synchronisation

JPA: EntityManager

Insertion dans la BDD

- Pour insérer des données dans la BDD il faut :
 - Instancier une occurrence de la classe de l'entité ;
 - Initialiser les propriétés de l'entité ;
 - Définir les relations de l'entité avec d'autres entités au besoin ;
 - Utiliser la méthode `persist()` de l'EntityManager en passant en paramètre l'entité.

```
private void initEntityManager() {  
    emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);  
    em = emf.createEntityManager();  
}  
  
private void closeEntityManager() {  
    em.close();  
    emf.close();  
}  
  
private void addPersons() {  
    em.getTransaction().begin();  
    Personne p1 = new Personne("prenom1", "nom1");  
    Personne p2 = new Personne("prenom2", "nom2");  
    em.persist(p1);  
    em.persist(p2);  
    em.getTransaction().commit();  
}
```

JPA: EntityManager

Recherche des occurrences

➤ Pour effectuer des recherches de données, l'EntityManager propose deux mécanismes :

- La recherche à partir de la clé primaire ;
 - `find()` et `getReference()` ;
- La recherche à partir d'une requête utilisant une syntaxe dédiée.

➤ Exemple :

```
private Personne search(int id) {  
    Personne personne = em.find(Personne.class, id);  
    return personne;  
}
```

JPA: EntityManager

Recherche par requête

La recherche par requête repose sur des méthodes dédiées de la classe EntityManager :

- `createQuery()`, `createNamedQuery()` et `createNativeQuery()` ;
- et sur un langage de requête spécifique nommé JPQL (Java Persistence Query Language). Une requête JPQL est analogue à une requête SQL (utilisation des objets plutôt que des tables).

➤ Exemple :

```
private List<Personne> searchByName(String nom) {  
    Query query = em.createQuery("select p from Personne p where p.nom='"+ nom +"'");  
    return query.getResultList();  
}
```

JPA: EntityManager

Recherche par requête

- L'objet *Query* gère aussi des paramètres nommés dans la requête. Le nom de chaque paramètre est préfixé par « `:` » dans la requête. La méthode `setParameter()` permet de fournir une valeur à chaque paramètre ;
- Exemple :

```
private List<Personne> searchByName2(String nom) {  
    Query query = em.createQuery("select p from Personne p where p.nom=:nom");  
    query.setParameter("nom", nom);  
    return query.getResultList();  
}
```

JPA: EntityManager

Modifier une occurrence

- Pour modifier une entité existante dans la base de données, il faut :
 - Obtenir une instance de l'entité à modifier (par recherche sur la clé primaire ou l'exécution d'une requête)
 - Modifier les propriétés de l'entité
 - Selon le mode de synchronisation des données de l'EntityManager, il peut être nécessaire d'appeler la méthode flush() explicitement

➤ Exemple :

```
private void updatePerson(int i) {  
    em.getTransaction().begin();  
    Personne personne=em.find(Personne.class, i);  
    personne.setNom("Nom modifié");  
    em.flush();  
    em.getTransaction().commit();  
}
```

JPA: EntityManager merge et remove

➤ Exemple pour merge :

```
private void merge(int i) {  
    em.getTransaction().begin();  
    Personne personne=em.find(Personne.class, i);  
  
    Personne personne2=new Personne();  
    personne2.setId(personne.getId());  
    personne2.setNom("Nom Remodifié");  
    em.merge(personne2);  
    em.getTransaction().commit();  
}
```

➤ Exemple pour remove :

```
private void remove(int i) {  
    em.getTransaction().begin();  
    Personne personne=em.find(Personne.class, i);  
    em.remove(personne);  
    em.getTransaction().commit();  
}
```

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **Utilisation du bean entité**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des transactions hors Java EE**
- **La gestion des relations entre tables dans le mapping**
- **Les callbacks d'événements**

JPA: Le fichier persistence.xml

- Ce fichier persistence.xml contient la configuration de base pour le mapping notamment en fournissant les informations sur la connexion à la base de données à utiliser ;
- Le fichier persistence.xml doit être stocké dans le répertoire *META-INF*;
- La racine du document XML du fichier *persistence.xml* est le tag `<persistence>` ;
 - Il contient un ou plusieurs tags `<persistence-unit>` ;
 - Ce tag possède deux attributs : *name* (obligatoire) qui précise le nom de l'unité et qui servira à y faire référence et *transaction-type* (optionnel) qui précise le type de transaction utilisée (ceci dépend de l'environnement d'exécution : Java SE ou Java EE).

JPA: Le fichier persistence.xml

- Le tag `<persistence-unit>` peut avoir les tags fils suivants :

Tag	Rôle
<code><description></code>	Description purement informative de l'unité de persistance(optionnel)
<code><provider></code>	Nom pleinement qualifié d'une classe de type javax.persistence.PersistenceProvider (optionnel). Généralement fournie par le fournisseur de l'implémentation de l'API : une utilisation de ce tag n'est requise que pour des besoins spécifiques.
<code><jta-data-source></code>	Nom JNDI de la DataSource utilisée dans un environnement avec support de JTA (optionnel)
<code><non-jta-data-source></code>	Nom JNDI de la DataSource utilisée dans un environnement sans support de JTA (optionnel)
<code><mapping-file></code>	Précise un fichier de mapping supplémentaire (optionnel)
<code><jar-file></code>	Précise un fichier jar qui contient des entités à inclure dans l'unité de persistance : le chemin précisé est relatif par rapport au fichier persistence.xml (optionnel)

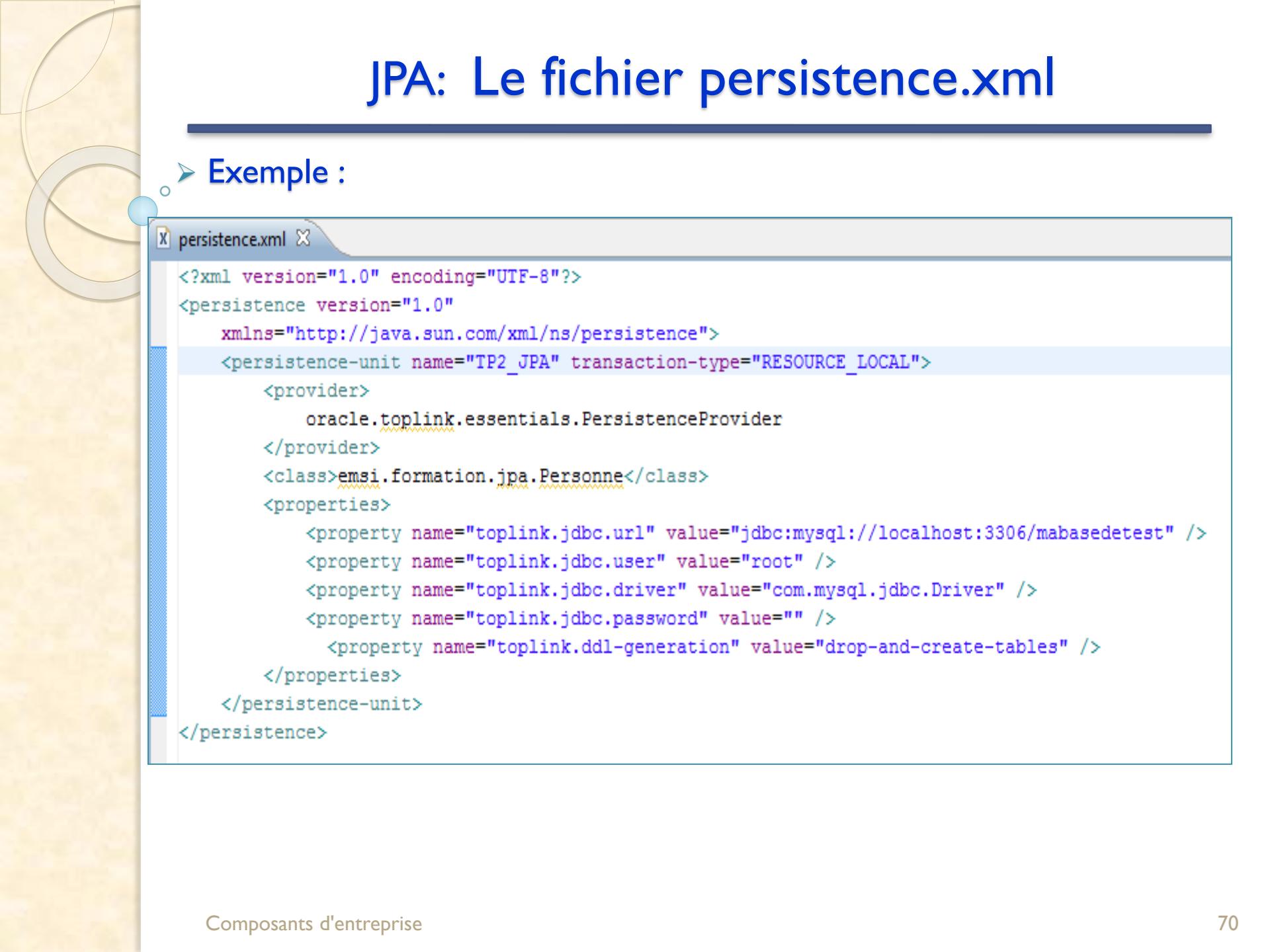
JPA: Le fichier persistence.xml

Tag	Rôle
<class>	Précise une classe d'une entité qui sera incluse dans l'unité de persistence (optionnel)
<properties>	Fournir des paramètres spécifiques au fournisseur. Comme Java SE ne propose pas de serveur JNDI, c'est fréquemment via ce tag que les informations concernant la source de données sont définis (optionnel)
<exclude-unlisted-classes>	Inhibition de la recherche automatique des classes des entités (optionnel)

- L'ensemble des classes des entités qui compose l'unité de persistance peut être spécifié explicitement dans le fichier persistence.xml ou déterminé dynamiquement à l'exécution par recherche de toutes les classes possédant une annotation `@javax.persistence.Entity` ;
- Par défaut, la liste de classes explicites est complétée par la liste des classes issue de la recherche dynamique. Pour empêcher la recherche dynamique, il faut utiliser le tag `<exclude-unlisted-classes>`

JPA: Le fichier persistence.xml

➤ Exemple :



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="TP2_JPA" transaction-type="RESOURCE_LOCAL">
        <provider>
            oracle.toplink.essentials.PersistenceProvider
        </provider>
        <class>emsi.formation.jpa.Personne</class>
        <properties>
            <property name="toplink.jdbc.url" value="jdbc:mysql://localhost:3306/mabasedetest" />
            <property name="toplink.jdbc.user" value="root" />
            <property name="toplink.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="toplink.jdbc.password" value="" />
            <property name="toplink.ddl-generation" value="drop-and-create-tables" />
        </properties>
    </persistence-unit>
</persistence>
```

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **Utilisation du bean entité**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre tables dans le mapping**
- **La gestion de l'héritage**

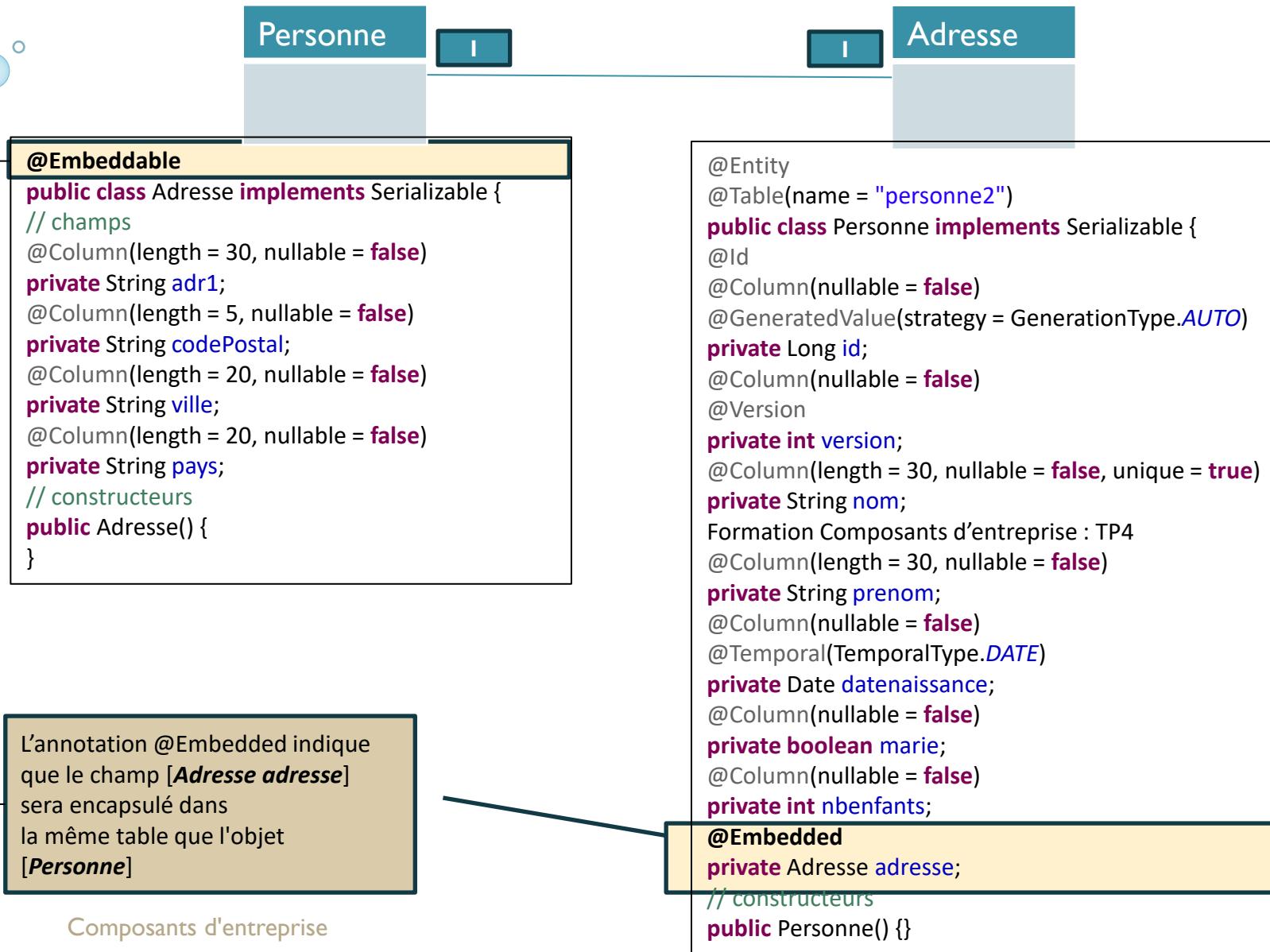
JPA: La gestion des relations entre tables dans le mapping

➤ Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations :

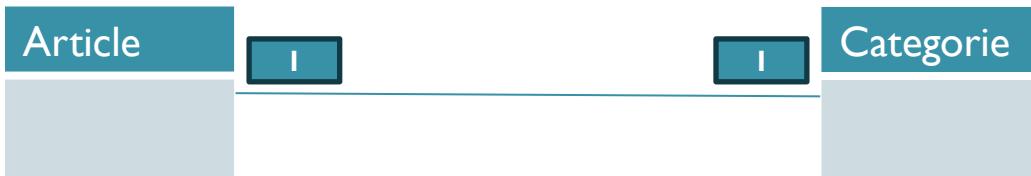
- 1-1 (one-to-one) ;
- 1-n (one-to-many) ;
- n-1 (many-to-one) ;
- n-n (many-to-many).

➤ **Travaux pratiques** : Réaliser les **TPs 3,4 et 5** relatif à l'API JPA.

JPA: La gestion des relations entre tables dans le mapping (one-to-one avec une inclusion)



JPA: La gestion des relations entre tables dans le mapping (one-to-one via une clé étrangère (FK))

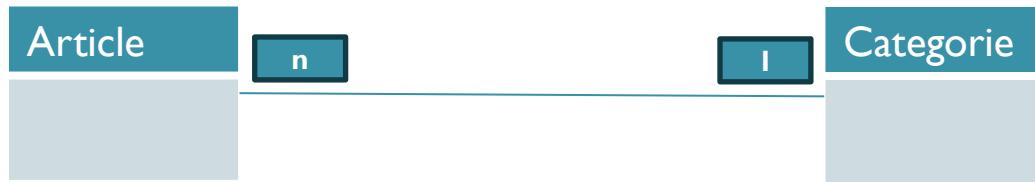


- Une personne à une seule adresse
- **fetch** est l'opération de chargement de la catégorie depuis la base de données :
- **eager** : l'adresse doit être chargée systématiquement lorsque la personne est chargé. Cette stratégie est appliquée par défaut pour @OneToOne et @ManyToOne
- **Lazy**: l'adresse ne sera chargée qu'à la demande (par exemple lorsque la méthode getAdresse() sera appelée). Cette stratégie est appliquée par défaut pour @OneToMany et @ManyToMany

La relation est implémentée par la clé étrangère `adresse_id` dans article non nul

```
@Entity  
@Table(name = "personne3")  
public class Personne implements Serializable{  
    @Id  
    @Column(nullable = false)  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    @Column(length = 30, nullable = false, unique = true)  
    private String nom;  
    @Column(length = 30, nullable = false)  
    private String prenom;  
    @Column(nullable = false)  
    @Temporal(TemporalType.DATE)  
    private Date datenaissance;  
    @Column(nullable = false)  
    private boolean marie;  
    @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)  
    // @OneToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST,  
    // CascadeType.REFRESH, CascadeType.REMOVE},  
    @JoinColumn(name = "adresse_id", unique = true, nullable = false)  
    private Adresse adresse;  
    // constructeurs  
    public Personne() {  
    }  
    public Personne(Str
```

JPA: La gestion des relations entre tables dans le mapping (one-to-many)

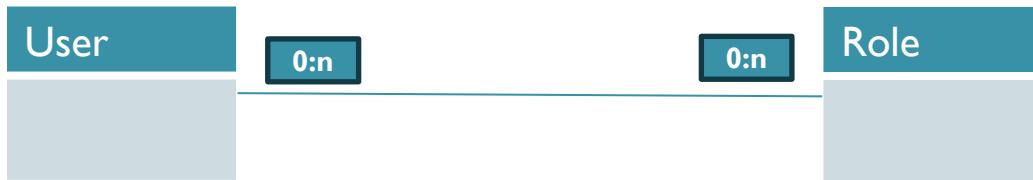


- Un article à une seule catégorie,
- **fetch** est l'opération de chargement de la catégorie depuis la base de données :
- **eager** : la catégorie doit être chargée systématiquement lorsque l'article est chargé. Cette stratégie est appliquée par défaut pour @Basic, @OneToOne et @ManyToOne
- **Lazy**: la catégorie ne sera chargée qu'à la demande (par exemple lorsque la méthode `getCategoria()` sera appelée). Cette stratégie est appliquée par défaut pour @OneToMany et @ManyToMany

La relation est implémentée par la clé étrangère `categoria_id` dans article non nul

```
@Entity  
@Table(name="article")  
public class Article implements Serializable {  
    // champs  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    @Version  
    private int version;  
    @Column(length = 30)  
    private String nom;  
    // relation principale Article (many) -> Category (one)  
    @ManyToOne(fetch=FetchType.LAZY)  
    // implémentée par une clé étrangère (categoria_id) dans Article  
    // 1 Article a nécessairement 1 Categorie (nullable=false)  
    @JoinColumn(name = "categoria_id", nullable = false)  
    private Categorie categorie;  
    // constructeurs  
    public Article() {  
    }  
    // getters et setters  
}
```

JPA: La gestion des relations entre tables dans le mapping (many-to-one)



```
@Entity  
@Table(name="T_Users")  
public class User {  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int idUser;  
    private String login;  
    private String password;  
    private int connectionNumber;  
  
    @ManyToMany  
    @JoinTable( name = "T_Users_Roles_Associations", joinColumns = @JoinColumn( name = "idUser" ),  
    inverseJoinColumns = @JoinColumn( name = "idRole" ) )  
    private List<Role> roles = new ArrayList<>(); public User() { }
```

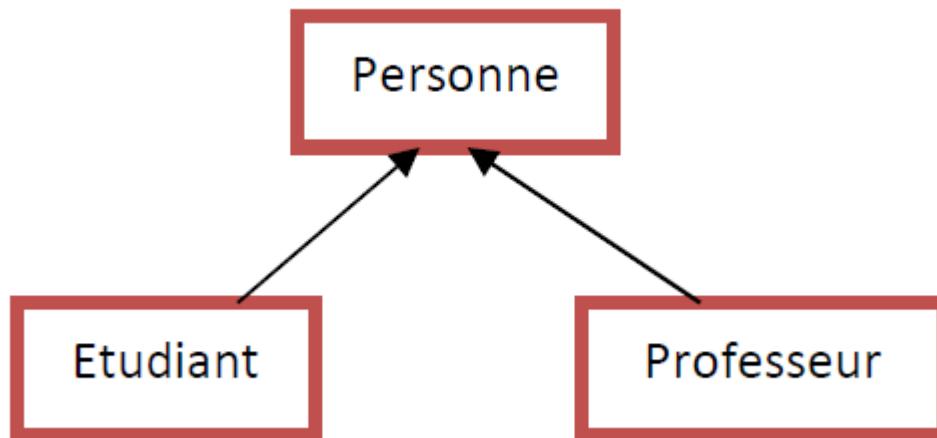
```
@Entity  
@Table(name="T_Roles")  
public class Role {  
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int idRole;  
    private String roleName;  
  
    @ManyToMany  
    @JoinTable(  
        name = "T_Users_Roles_Associations",  
        joinColumns = @JoinColumn( name = "idRole" ),  
        inverseJoinColumns = @JoinColumn( name = "idUser" ) )  
    private List<User> users = new ArrayList<>();
```

JPA

- **Introduction**
- **Les entités**
- **Le mapping entre une entité et une table**
- **Le mapping de propriété complexe**
- **Mapper une entité sur plusieurs tables**
- **Utilisation d'objets embarqués dans les entités**
- **Fichier de configuration du mapping**
- **Utilisation du bean entité**
- **EntityManager**
- **Le fichier persistence.xml**
- **La gestion des relations entre tables dans le mapping**
- **La gestion de l'héritage**

JPA: La gestion des relations entre tables dans le mapping – Héritage

Considérant les trois classes ci-dessous:



JPA: La gestion des relations entre tables dans le mapping – Héritage (Stratégie : SINGLE_TABLE)

```
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name="descriminateur")  
@DiscriminatorValue(value="p")  
public class Personne {  
    @Id  
    @GeneratedValue  
    private int identifiant;  
    private String pseudo;  
    private String mail;  
    private String codePostal;  
    public Personne() {  
        super();  
    }  
}
```

```
@Entity  
@DiscriminatorValue("etud")  
public class Etudiant extends Personne{  
    private Date dateInscription;  
  
    public Etudiant() {  
    }  
}
```

```
@Entity  
@DiscriminatorValue("prof")  
public class Professeur extends Personne{  
    private String diplome;  
    public Professeur() {  
        super();  
    }  
}
```

Resultset 1					
SQL Query Area					
* 1 SELECT * FROM "formation"."personne"					
descriminateur	identifiant	codePostal	mail	pseudo	dateInscription
p	1	10000	user01@gmail.com	user01	2016-01-23 22:28:26
p	2	20000	user02@gmail.com	user02	2016-01-23 22:28:26
p	3	30000	user03@gmail.com	user03	2016-01-23 22:28:26
p	4	40000	user04@gmail.com	user04	2016-01-23 22:28:26
prof	5	50000	user05@gmail.com	user05	2016-01-23 22:28:26
prof	6	60000	user06@gmail.com	user06	2016-01-23 22:28:26
prof	7	70000	user07@gmail.com	user07	2016-01-23 22:28:26
prof	8	80000	user08@gmail.com	user08	2016-01-23 22:28:26
etud	9	90000	user09@gmail.com	user09	2016-01-23 22:28:26
etud	10	100000	user10@gmail.com	user10	2016-01-23 22:28:26
etud	11	110000	user11@gmail.com	user11	2016-01-23 22:29:26

JPA: La gestion des relations entre tables dans le mapping – Héritage (Stratégie : JOINED)

```
@Entity  
@Inheritance(strategy = InheritanceType.JOINED)  
public class Personne {  
    @Id  
    @GeneratedValue  
    private int identifiant;  
    private String pseudo;  
    private String mail;  
    private String codePostal;  
    public Personne() {  
        super();  
    }  
}
```

Table Name: personne Database: formation

Columns and Indices Table Options Advanced Options

Column Name	Datatype
identifiant	INT(11)
codePostal	VARCHAR(255)
mail	VARCHAR(255)
pseudo	VARCHAR(255)

```
@Entity  
@PrimaryKeyJoinColumn(name="id")  
public class Etudiant extends Personne{  
    private Date dateInscription;  
  
    public Etudiant() {  
    }  
}
```

Table Name: etudiant Database: formation Comment: InnoDB free

Columns and Indices Table Options Advanced Options

Column Name	Datatype
dateInscription	DATETIME
id	INT(11)

Indices Foreign Keys Column Details

Index Name	Key Name	Ref. Table	Column
FKC55D557C59E10859	FKC55D557C59E10859	personne	id

Foreign Key Settings

On Delete	On Update
Restrict	Restrict

```
@Entity  
@DiscriminatorValue("prof")  
public class Professeur extends Personne{  
    private String diplome;  
    public Professeur() {  
        super();  
    }  
}
```

Table Name: professeur Database: formation Comment: InnoDB free: 4096

Columns and Indices Table Options Advanced Options

Column Name	Datatype
diplome	VARCHAR(255)
id	INT(11)

Indices Foreign Keys Column Details

Index Name	Key Name	Ref. Table	Column
FK2EEAED659E1085C	FK2EEAED659E1085C	personne	id

Foreign Key Settings

On Delete	On Update
Restrict	Restrict

JPA: La gestion des relations entre tables dans le mapping – Héritage (Stratégie : JOINED)

```
@Entity  
@Inheritance(strategy = InheritanceType.JOINED)  
public class Personne {  
  
    @Id  
    @GeneratedValue  
    private int identifiant;  
    private String pseudo;  
    private String mail;  
    private String codePostal;  
    public Personne() {  
        super();  
    }  
}
```

SQL Query Area				
*	1	SELECT * FROM `formation`.`personne`		
▶		identifiant	codePostal	mail
▶	1	10000		user01@gmail.com
▶	2	20000		user02@gmail.com
▶	3	30000		user03@gmail.com
▶	4	40000		user04@gmail.com
▶	5	50000		user05@gmail.com
▶	6	60000		user06@gmail.com
▶	7	70000		user07@gmail.com
▶	8	80000		user08@gmail.com
▶	9	90000		user09@gmail.com
▶	10	100000		user10@gmail.com
▶	11	110000		user11@gmail.com
▶				pseudo
▶				user01
▶				user02
▶				user03
▶				user04
▶				user05
▶				user06
▶				user07
▶				user08
▶				user09
▶				user10
▶				user11

```
@Entity  
@PrimaryKeyJoinColumn(name="id")  
public class Etudiant extends Personne{  
private Date dateInscription;  
  
public Etudiant() {  
}  
}
```

SQL Query Area			
*	1	SELECT * FROM `formation`.`etudiant`	
▶		dateInscription	id
▶	2016-01-23 18:14:45		9
▶	2016-01-23 18:14:45		10
▶	2016-01-23 18:14:45		11

```
@Entity  
@DiscriminatorValue("prof")  
public class Professeur extends Personne{  
private String diplome;  
public Professeur() {  
super();  
}}
```

SQL Query Area			
*	1	SELECT * FROM `formation`.`professeur`	
▶		diplome	id
▶	INGENIEUR D'ETAT		5
▶	DOCTEUR D'ETAT		6
▶	phd		7
▶	INGENIEUR D'ETAT		8

JPA: La gestion des relations entre tables dans le mapping – Héritage (Stratégie : TABLE_PER_CLASS)

```
@Entity  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
public class Personne {  
    @Id  
    @GeneratedValue  
    private int identifiant;  
    private String pseudo;  
    private String mail;  
    private String codePostal;  
    public Personne() {  
        super();  
    }  
}
```

Table Name:	personne	Database:	format
Columns and Indices	Table Options	Advanced Options	
Column Name	Datatype		
identifiant	INT(11)		
codePostal	VARCHAR(255)		
mail	VARCHAR(255)		
pseudo	VARCHAR(255)		

```
@Entity  
public class Etudiant extends Personne{  
    private Date dateInscription;  
  
    public Etudiant() {  
    }  
}
```

Table Name:	etudiant	Database:	format
Columns and Indices	Table Options	Advanced Options	
Column Name	Datatype		
identifiant	INT(11)	NOT NULL	<input checked="" type="checkbox"/>
codePostal	VARCHAR(255)		
mail	VARCHAR(255)		
pseudo	VARCHAR(255)		
dateInscription	DATETIME		

```
@Entity  
public class Professeur extends Personne{  
    private String diplome;  
    public Professeur() {  
        super();  
    }  
}
```

Table Name:	professeur	Database:	format
Columns and Indices	Table Options	Advanced Options	
Column Name	Datatype		
identifiant	INT(11)	NOT NULL	<input checked="" type="checkbox"/>
codePostal	VARCHAR(255)		
mail	VARCHAR(255)		
pseudo	VARCHAR(255)		
diplome	VARCHAR(255)		

JPA: La gestion des relations entre tables dans le mapping – Héritage (Stratégie : TABLE_PER_CLASS)

```
@Entity  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
public class Personne {  
    @Id  
    @GeneratedValue  
    private int identifiant;  
    private String pseudo;  
    private String mail;  
    private String codePostal;  
    public Personne() {  
        super();  
    }  
}
```

SQL Query Area

```
1 SELECT * FROM `formation`.`personne`
```

identifiant	codePostal	mail	pseudo
1	10000	user01@gmail.com	user01
2	20000	user02@gmail.com	user02
3	30000	user03@gmail.com	user03
4	40000	user04@gmail.com	user04

```
@Entity  
public class Etudiant extends Personne{  
    private Date dateInscription;  
  
    public Etudiant() {  
    }  
}
```

SQL Query Area

```
1 SELECT * FROM `formation`.`etudiant`
```

identifiant	codePostal	mail	pseudo	dateInscription
9	90000	user09@gmail.com	user09	2016-01-23 22:48:39
10	100000	user10@gmail.com	user10	2016-01-23 22:48:39
11	110000	user11@gmail.com	user11	2016-01-23 22:48:39

```
@Entity  
public class Professeur extends Personne{  
    private String diplome;  
    public Professeur() {  
        super();  
    }  
}
```

SQL Query Area

```
1 SELECT * FROM `formation`.`professeur`
```

identifiant	codePostal	mail	pseudo	diplome
5	50000	user05@gmail.com	user05	INGENIEUR D'ETAT
6	60000	user06@gmail.com	user06	DOCTEUR D'ETAT
7	70000	user07@gmail.com	user07	phd
8	80000	user08@gmail.com	user08	INGENIEUR D'ETAT