

# JPA Java Persistence API

Loïc Nogues  
Novembre 2012

# EJB 3.0

- Java EE 5 (Enterprise Edition) est une plateforme de développement et un ensemble de spécifications pour le développement d'applications d'entreprises multi-tiers
- EJB 3.0 fait partie de Java EE 5 (mai 2006) ; c'est une spécification d'un cadre (framework) pour l'utilisation de composants métier réutilisables par des serveurs d'applications Java

# JPA

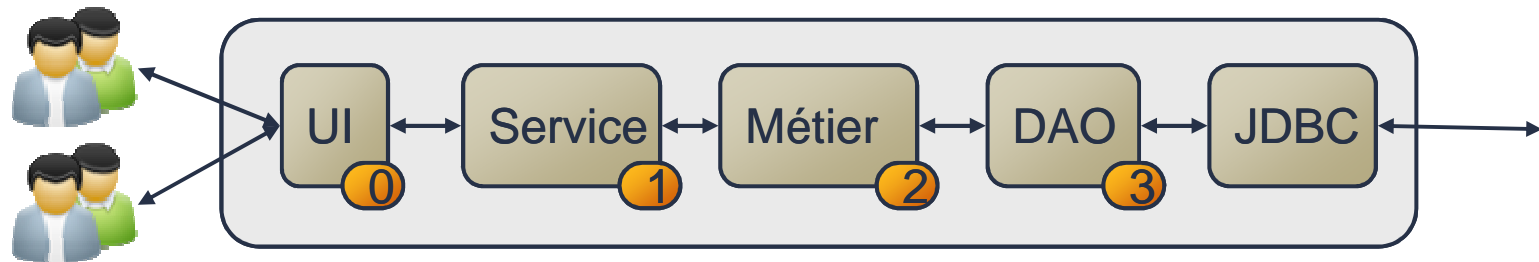
- JPA (Java persistence API) est la partie de la spécification EJB 3.0 qui concerne la persistance des composants dans une base de données relationnelle
- Peut s'appliquer sur toutes les applications Java, même celles qui s'exécutent en dehors d'un serveur d'applications

# JPA - Définition

- Interface Java de référence pour la persistance des mappings O/R.
- Ces mappings permettent d'assurer la transformation de POJO vers la base de données et vice et versa.
- Cette transformation d'objet est effective pour toute action CRUD.
- JPA est basé sur des POJO annotés ainsi que sur un gestionnaire d'entités nommé « EntityManager ».
- C'est l'EntityManager qui est chargé d'assurer les transformations objet/database ainsi que de vérifier l'état des POJO.

# JPA - Définition

- Présentation globales des couches Java EE



0: UI: Interface utilisateur

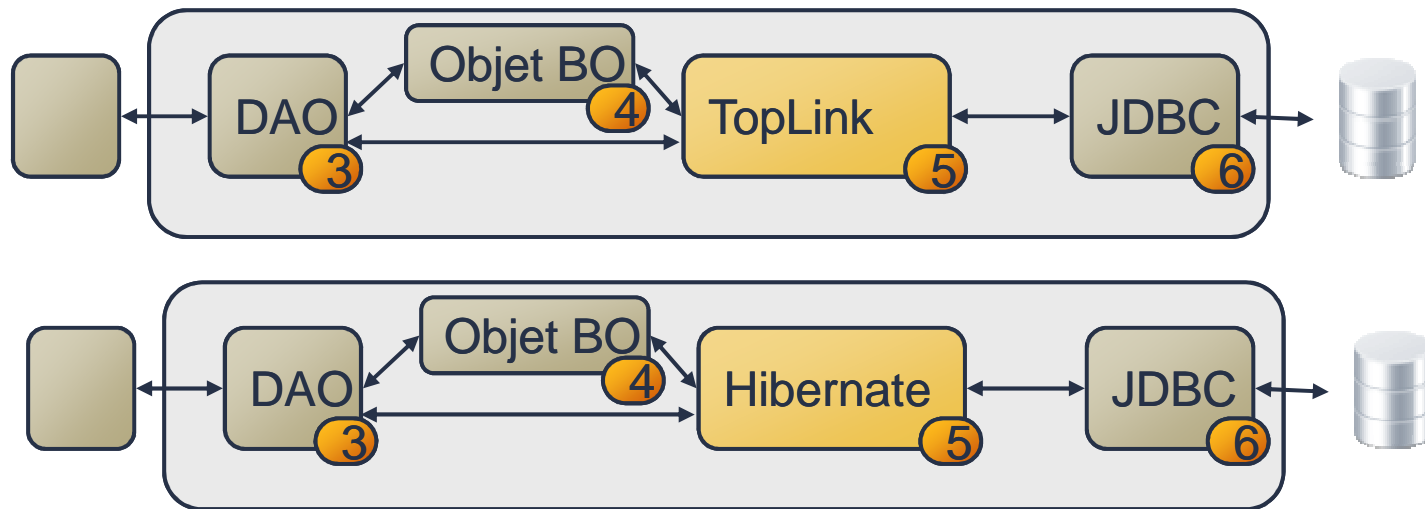
1: La couche service

2: Couche métier dictant les règles métiers (La logique spécifique à l'application)

3: Data Access Object: Couche d'accès aux données (consultation, enregistrement, CRUD en général)

# JPA - Couches DAO

- Présentation des implémentations classiques des couches DAO



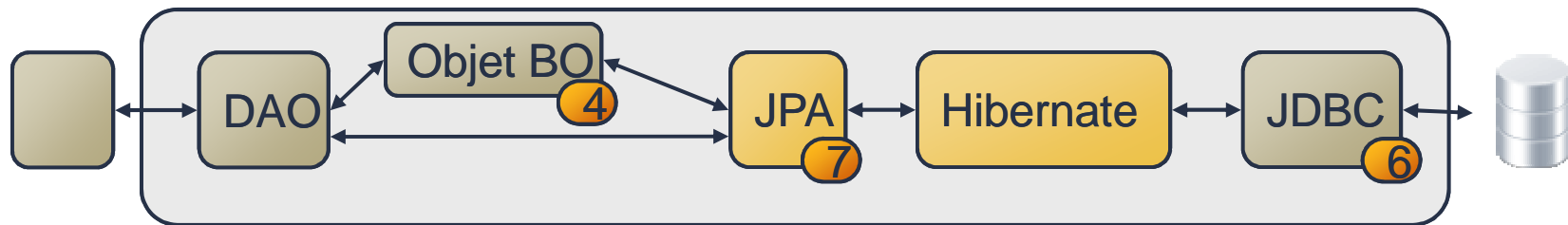
4: Objets BO représentation une ligne de BDD

5: Couche d'implémentation du mapping Objet / BDD (dans l'ex: Hibernate ou TopLink)

6: Pilote JDBC du SGBD (Oracle, Postgres, MySQL...)

# JPA - Son rôle

## ■Présentation de couches avec JPA



- La couche JPA [7] doit faire un pont entre le monde relationnel de la base de données [6] et le monde objet [4] manipulé par les programmes J2EE
- Ce pont est fait par configuration et il y a deux façons de le faire :
  - Par fichiers XML. C'était quasiment l'unique façon de faire jusqu'à l'avènement du JDK 1.5
  - Par annotations Java depuis le JDK 1.5

# Avertissement

- JPA est le plus souvent utilisé dans le contexte d'un serveur d'application
- Dans ce cours on étudiera JPA en dehors de tout serveur d'application



# Entités

# Entités

- Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécification JPA
- Le développeur indique qu'une classe est une entité en lui associant l'annotation `@Entity`
- Ne pas oublier d'importer `javax.persistence.Entity` dans les classes entités

# Entités

- La table liée à l'entité aura par défaut le même nom.
- Si on souhaite préciser le nom de la table, on peut utiliser la balise @Table
- Exemple  
@Table (name="nom\_de\_table")

# Exemple d'entité – les champs

```
@Entity
public class Departement {
    private int id;
    private String nom;
    private String lieu;
    private Collection<Employe> employes =
        new List<Employe>();
}
```

# Une propriété

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

# Propriété

- La colonne liée à la propriété aura par défaut le même nom.
- Si on souhaite préciser le nom de la colonne, on peut utiliser la balise @Column
- Attribut possible
  - name
  - length
  - nullable
  - unique

# Les constructeurs

```
/**
 * Constructeur sans paramètre
 * obligatoire.
 */
public Departement() { }

public Departement(String nom, String lieu) {
    this.nom = nom;
    this.lieu = lieu;
}
```

# L'identificateur

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

public void setId(int id) {
    this.id = id;
}
```



# L'identificateur

- On peut lier l'id à une séquence base de donnée.

```
@Id
@SequenceGenerator (name="SeqPersonne",
sequenceName="nomSequence")
@GeneratedValue (strategy = GenerationType.SEQUENCE
, generator="SeqPersonne")
public int getId() {
    return id;
}
```

# Conditions pour les classes entités

- Elle doit posséder un attribut qui représente la clé primaire dans la BD (@Id)
- Une classe entité doit avoir un constructeur sans paramètre protected ou public
- Elle ne doit pas être final
- Aucune méthode ou champ persistant ne doit être final
- Si une instance peut être passée par valeur en paramètre d'une méthode comme un objet détaché, elle doit implémenter Serializable

## Types temporels

- ❑ Lorsqu'une classe entité a un attribut de type temporel (**Calendar** ou **Date** de **java.util**), il est **obligatoire** d'indiquer de quel type temporel est cet attribut par une annotation **@Temporal**
- ❑ Cette indication permettra au fournisseur de persistance de savoir comment déclarer la colonne correspondante dans la base de données : une date (un jour), un temps sur 24 heures (heures, minutes, secondes à la milliseconde près) ou un *timeStamp* (date + heure à la microseconde près)

R. Grin

JPA

page 55

## Annotation pour les types temporels

- ❑ 3 types temporels dans l'énumération **TemporalType** : **DATE**, **TIME**, **TIMESTAMP**
- ❑ Correspondent aux 3 types de SQL ou du paquetage **java.sql** : **Date**, **Time** et **Timestamp**

R. Grin

JPA

page 56

## Exemple

```
@Temporal(TemporalType.DATE)
private Calendar dateEmb;
```

# Attributs persistants

- Par défaut, tous les attributs d'une entité sont persistants
- L'annotation `@Basic` indique qu'un attribut est persistant mais elle n'est donc indispensable que si on veut préciser des informations sur cette persistance (par exemple, une récupération retardée)
- Seuls les attributs dont la variable est `transient` ou qui sont annotés par `@Transient` ne sont pas persistants

# Configuration de la connexion

- Il est nécessaire d'indiquer au fournisseur de persistance comment il peut se connecter à la base de données
- Les informations doivent être données dans un fichier `persistence.xml` situé dans un répertoire `META-INF` dans le classpath
- Ce fichier peut aussi comporter d'autres informations ; il est étudié en détails dans la suite du cours

# Exemple - Simple

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
  <persistence-unit name="Employes">
    <class>jpa.Departement</class>
    <class>jpa.Employe</class>
    <properties>
      <property name="toplink.jdbc.driver"
        value="oracle.jdbc.OracleDriver"/>
      <property name="toplink.jdbc.url"
        value="jdbc:oracle:thin:@cl.truc.fr:1521:XE"/>
      <property name="toplink.jdbc.user"
        value="toto"/>
      <property name="toplink.jdbc.password"
        value="xxxxxx"/>
    </properties>
  </persistence-unit>
</persistence>
```

# Exemple - Complet

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
    <!-- provider -->
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <!-- Classes persistantes -->
      <property name="hibernate.archive.autodetection" value="class, hbm" />
      <!-- logs SQL -->
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="use_sql_comments" value="true"/>

      <!-- connexion JDBC -->
      <property name="hibernate.connection.driver_class" value="org.postgresql.Driver" />
      <property name="hibernate.connection.url" value="jdbc:postgresql://xx.xx.xx.xxx:5432/test-j2ee" />
      <property name="hibernate.connection.username" value="USER" />
      <property name="hibernate.connection.password" value="MDP" />
      <!-- création automatique du schéma -->
      <property name="hibernate.hbm2ddl.auto" value="create" />
    -->
    <!-- Dialecte -->
    <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
    <!-- Cache configuration -->
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
    <property name="cache.provider_class" value="org.hibernate.cache.EhCacheProvider"/>
  </properties>
</persistence-unit>
</persistence>
```



# Les associations



# Généralités

- Une association peut être uni ou bidirectionnelle
- Elle peut être de type 1:1, 1:N, N:1 ou M:N
- Les associations doivent être indiquées par une annotation sur la propriété correspondante, pour que JPA puisse les gérer correctement

# Association 1:1

- Annotation @OneToOne
- Représentée par une clé étrangère ajoutée dans la table qui correspond au côté propriétaire
- Exemple :
  - Ajouter un lien vers un objet adresse sur l'objet Personne
  - Référencer la personne sur l'objet adresse (bi-directionnalité)

# Association 1:1

- Au niveau BDD

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Trigg
Name	Type	Length	Decimals	Allow Null		
id	numeric	10	0	<input type="checkbox"/>		1
version	numeric	5	0	<input checked="" type="checkbox"/>		
nom	varchar	30	0	<input checked="" type="checkbox"/>		
prenom	varchar	30	0	<input checked="" type="checkbox"/>		
datenaissance	date	0	0	<input checked="" type="checkbox"/>		
nbenfants	numeric	3	0	<input checked="" type="checkbox"/>		
marie	bool	0	0	<input checked="" type="checkbox"/>		
adresse_id	numeric	30	0	<input checked="" type="checkbox"/>		

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Trigg
Name	Type	Length	Decimals	Allow Null		
id	numeric	30	0	<input type="checkbox"/>		1
version	numeric	30	0	<input type="checkbox"/>		
adresse	varchar	250	0	<input checked="" type="checkbox"/>		
cp	varchar	10	0	<input checked="" type="checkbox"/>		
ville	varchar	150	0	<input checked="" type="checkbox"/>		

1: Table t\_personne  
2: Table t\_adresse  
3: FK pointant sur adresse

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Triggers	Privileges	Options	Comment	SQL Preview
Name	Field Names	Reference Schema	Reference Table	Foreign Field Names	On Delete	On Update				
FK\$PERSONNE\$ADRESSE	adresse_id	public	t_adresse	id	NO ACTION	NO ACTION				

- Sur l'objet Personne

```

59 @OneToOne( cascade=CascadeType.ALL, fetch=FetchType.LAZY)
60 @JoinColumn( name="adresse_id", unique=true, nullable=true )
61 private Adresse adresse;
--

```

- Sur l'objet Adresse

```

39 @OneToOne(mappedBy = "adresse", fetch=FetchType.LAZY)
40 private Personne personne;
--

```

# Association 1:1 sur les clés

- 2 classes peuvent être reliées par leur identificateur : 2 entités sont associées ssi elles ont les mêmes clés
- L'annotation `@PrimaryKeyJoinColumn` doit alors être utilisée pour indiquer au fournisseur de persistance qu'il ne faut pas utiliser une clé étrangère à part pour représenter l'association
- Attention, c'est au développeur de s'assurer que les entités associées ont bien les mêmes clés

# Exemple

```
@OneToOne
@PrimaryKeyJoinColumn
private Employe employe
```

# Représentation des associations

## 1:N et M:N

- Elles sont représentées par des collections ou maps qui doivent être déclarées par un des types interface suivants (de `java.util`) :
  - `Collection`
  - `Set`
  - `List`
  - `Map`
- Les variantes génériques sont conseillées ;  
par exemple `Collection<Employe>`

# Associations 1:N et N:1

- Objectif
  - Un objet « enfant »
  - Mapper la collection sur Personne
- Annotations @OneToMany et @ManyToOne
- Représentée par une clé étrangère dans la table qui correspond au côté propriétaire (obligatoirement le côté « Many »)

# Exemple

- L'objet Enfant :

```
19 @Entity
20 @Table( name="t_enfant" )
21 public class Enfant implements Serializable {
22
23     @Id
24     @Column(name="ID", nullable=false)
25     @SequenceGenerator( name = "SeqPersonne", sequenceName = "seq_personne" )
26     @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "SeqPersonne" )
27     private Integer id;
28
29     @Column(name = "VERSION", nullable = false)
30     @Version
31     private int version;
32
33     @Column(name = "prenom", length = 30, nullable = false, unique = true)
34     private String prenom;
35 }
```



# Exemple

- Au niveau BDD

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules
Name	Type	Length	Decimals	Allow Null	
<b>id</b>	numeric	30	0	<input type="checkbox"/>	1
version	numeric	30	0	<input checked="" type="checkbox"/>	
prenom	varchar	50	0	<input checked="" type="checkbox"/>	
parent_id	numeric	30	0	<input checked="" type="checkbox"/>	

1: Table t\_enfant  
2: FK pointant sur personne

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Triggers	Privileges	Options	Comment	SQL Preview
Name	Field Names	Reference Schema	Reference Table	Foreign Field Names	On Delete	On Update				
<b>FK\$ENFANT\$PERSONNE</b>	parent_id	public	t_personne	id	NO ACTION	NO ACTION				

- Sur l'objet Personne

```

63 @OneToMany(mappedBy = "parent", cascade=CascadeType.ALL)
64 private List<Enfant> enfants = new ArrayList<Enfant>();
65
66 /**
67  * Ajouter un enfant.
68  * @param pEnfant (@link Enfant)
69  */
70 public void addEnfant( final Enfant pEnfant ) {
71     enfants.add(pEnfant);
72     pEnfant.setParent( this );
73 }

```

- Sur l'objet Enfant

```

39 @ManyToOne(fetch=FetchType.LAZY)
40 @JoinColumn(name = "parent_id", nullable = false)
41 private Personne parent;

```

# Association M:N

- Annotations @ManyToMany
- Représentée par une table association

# Association M:N

- Les valeurs par défaut :
  - le nom de la table association est la concaténation des 2 tables, séparées par « \_ »
  - les noms des colonnes clés étrangères sont les concaténations de la table référencée, de « \_ » et de la colonne « Id » de la table référencée

# Association M:N

- Si les valeurs par défaut ne conviennent pas, le côté propriétaire doit comporter une annotation `@JoinTable`
- L'autre côté doit toujours comporter l'attribut `mappedBy`

# Exemple

## ■ Au niveau BDD

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Triggers	Privilege
Name	Type	Length	Decimals	Allow Null			
personne_id	numeric	30	0	<input type="checkbox"/>			1
▶ activite_id	numeric	30	0	<input type="checkbox"/>			2

1: FK pointant sur personne  
2: FK pointant sur activite

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Triggers	Privileges	Options	Comment	SQL Preview
Name		Field Names								
FK\$PERSONNE_ACTIVITE\$PERSONNE		personne_id								
▶ FK\$PERSONNE_ACTIVITE\$ACTIVITE		activite_id								

## ■ Sur l'objet Personne

```

66 @ManyToMany(cascade={CascadeType.PERSIST})
67 @JoinTable(name="t_personne_activite",
68             joinColumns = @JoinColumn(name = "personne_id"),
69             inverseJoinColumns = @JoinColumn(name = "activite_id"))
70 private List<Activite> activites = new ArrayList<Activite>();
71

```

## ■ Sur l'objet Activite

```

30 // relation inverse Activite -> Personne
31 @ManyToMany(mappedBy = "activites")
32 private List<Personne> personnes = new ArrayList<Personne>();

```

# Types à utiliser pour association N

Le plus souvent `Collection` sera utilisé

`Set` peut être utile pour éliminer les doublons

Les types concrets, tels que `HashSet` ou `ArrayList`, ne peuvent être utilisés que pour des entités « nouvelles » ; dès que l'entité est gérée, les types interfaces doivent être utilisés (ce qui permet au fournisseur de persistance d'utiliser son propre type concret)

`List` peut être utilisé pour conserver un ordre mais nécessite quelques précautions

# Ordre dans les collections

- L'ordre d'une liste n'est pas nécessairement préservé dans la base de données
- De plus, l'ordre en mémoire doit être maintenu par le code (pas automatique)
- Tout ce qu'on peut espérer est de récupérer les entités associées dans la liste avec un certain ordre lors de la récupération dans la base, en utilisant l'annotation `@OrderBy`

# @OrderBy

- Cette annotation indique dans quel ordre sont récupérées les entités associées
- Il faut préciser un ou plusieurs attributs qui déterminent l'ordre
- Chaque attribut peut être précisé par ASC ou DESC (ordre ascendant ou descendant); ASC par défaut
- Les différents attributs sont séparés par une virgule
- Si aucun attribut n'est précisé, l'ordre sera celui de la clé primaire



# Exemple

```
@Entity
public class Departement {
    ...
    @OneToMany(mappedBy="departement")
    @OrderBy("nomEmploye")
    public List<Employe> getEmployes() {
        ...
    }
}
```

```
@OrderBy("poste DESC, nomEmploye ASC")
```

# Associations bidirectionnelles

- Le développeur est responsable de la gestion correcte des 2 bouts de l'association
- Par exemple, si un employé change de département, les collections des employés des départements concernés doivent être modifiées
- Un des 2 bouts est dit « propriétaire » de l'association

# Bout propriétaire

- Pour les associations autres que M:N ce bout correspond à la table qui contient la clé étrangère qui traduit l'association
- Pour les associations M:N le développeur peut choisir arbitrairement le bout propriétaire
- L'autre bout (non propriétaire) est qualifié par l'attribut `mappedBy` qui donne le nom de l'attribut dans le bout propriétaire qui correspond à la même association

# Exemple

## Dans la classe Employe :

```
@ManyToOne
public Departement getDepartement() {
    return departement;
}
```

## Dans la classe Departement :

```
@OneToMany(mappedBy="departement")
public Collection<Employe> getEmployes(){
    return employes;
}
```

# Méthode de gestion de l'association

Pour faciliter la gestion des 2 bouts d'une association le code peut comporter une méthode qui effectue tout le travail

En particulier, dans les associations 1-N, le bout « 1 » peut comporter ce genre de méthode (dans la classe Departement d'une association département-employé) :

```
public void ajouterEmploye(Employe e) {  
    this.employees.add(e);  
    employe.setDept(this);  
}
```

# Association M:N avec information portée par l'association

- Une association M:N peut porter une information
- Exemple :  
Association entre les employés et les projets  
Un employé a une (et une seule) fonction dans chaque projet auquel il participe
- En ce cas, il n'est pas possible de traduire l'association en ajoutant 2 collections (ou maps) comme il vient d'être décrit

# Classe association pour une association M:N

- L'association sera traduite par une classe association
- 2 possibilités pour cette classe, suivant qu'elle contient ou non un attribut identificateur (@Id) unique
- Le plus simple est de n'avoir qu'un seul attribut identificateur

# Exemple – 1 identificateur

- Association entre les employés et les projets
- Cas d'un identificateur unique : la classe association contient les attributs `id (int)`, `employe (Employe)`, `projet (Projet)` et `fonction (String)`
- L'attribut `id` est annoté par `@Id`
- Les attributs `employe` et `projet` sont annotés par `@ManyToOne`



# Exemple – 1 identificateur

Si le schéma relationnel est généré d'après les informations de mapping par les outils associés au fournisseur de persistance, on peut ajouter une contrainte d'unicité sur (EMPLOYEE\_ID, PROJET ID) qui traduit le fait qu'un employé ne peut avoir 2 fonctions dans un même projet :

```
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={ "EMPLOYEE_ID", "PROJET_ID" }))
public class Participation {
```

# Exemple – 2 identificateurs

- Si la base de données existe déjà, il sera fréquent de devoir s'adapter à une table association qui contient les colonnes suivantes (pas de colonne id):
  - `employe_id`, clé étrangère vers EMPLOYE
  - `projet_id`, clé étrangère vers PROJET
  - `fonction`
- et qui a (`employe_id`, `projet_id`) comme clé primaire

# Exemple – 2 identificateurs

- En ce cas, la solution est plus complexe, et pas toujours portable dans l'état actuel de la spécification JPA
- La solution donnée dans les transparents suivants convient pour TopLink Essentials et Hibernate, les 2 fournisseurs de persistance les plus utilisés ; elle n'a pas été testée sur d'autres fournisseurs

# Exemple – 2 identificateurs

- La difficulté vient de l'écriture de la classe `Participation`
- L'idée est de dissocier la fonction d'identificateur des attributs `employeId` et `projetId` de leur rôle dans les associations avec les classes `Projet` et `Employe`

# Exemple – 2 identificateurs

- Pour éviter les conflits au moment du flush, les colonnes qui correspondent aux identificateurs sont marqués non modifiables ni insérables (pas de persistance dans la BD)
- En effet, leur valeur sera mise par le mapping des associations 1-N vers `Employe` et `Projet` qui sera traduite par 2 clés étrangères

# Classes Employe et Projet

```
@Entity public class Employe {  
    @Id public int getId() { ... }  
    @OneToMany(mappedBy="employe")  
    public Collection<Participation>  
        getParticipations() { ... }  
    . . .  
}
```

```
@Entity public class Projet {  
    @Id public int getId() { ... }  
    @OneToMany(mappedBy="projet")  
    public Collection<Participation>  
        getParticipations() { ... }  
    . . .  
}
```

# Classe Participation (2 choix pour Id)

- On peut utiliser une classe « Embeddable » ou une « IdClass » pour représenter la clé composite de Participation
- Le code suivant utilise une « IdClass »

# Classe Participation (Id)

```
@Entity
@IdClass(ParticipationId.class)
public class Participation {
    // Les identificateurs "read-only"
    @Id
    @Column(name="EMPLOYEE_ID",
            insertable="false", updatable="false")
    public int getEmployeeId() { ... }

    @Id
    @Column(name="PROJET_ID",
            insertable="false", updatable="false")
    public int getProjetId() { ... }
```



# Classe Participation (champs)

```
private Employe employe;  
private long employeId;  
private Projet projet;  
private long projetId;  
private String fonction;
```

# Classe Participation (constructeurs)

```
public Participation() { }

// Constructeur pour faciliter une
// bonne gestion des liens
public Participation(Employe e, Projet p) {
    this.employe = e;
    this.projet = p;
    e.getParticipations().add(this);
    p.getParticipations().add(this);
}
```

# Etablir une association

- Il faut éviter la possibilité qu'un bout seulement de l'association soit établie et pour cela, il faut qu'une seule classe s'en charge
- Pour cela, on peut faire gérer l'association entière par `Projet`, par `Employe` ou par `Participation`
- Le transparent suivant montre comment la faire gérer par le constructeur de `Participation`

# Classe Participation (constructeurs)

```
public Participation() { } // Pour JPA

public Participation(Employe employe, Projet projet,
                    String fonction) {

    this.employe = employe;
    this.employeId = employe.getId();
    this.projet = projet;
    this.projetId = projet.getId();
    employe.getParticipations.add(this);
    projet.getParticipations.add(this);
    this.fonction = fonction;
}
```

# Classe Participation (associations)

```
// Les associations
@ManyToOne
public Employe getEmploye() {
    return employe;
}

@ManyToOne
public Projet getProjet() {
    return projet;
}

. . .
}
```

# Classe ParticipationId

```
public class ParticipationId implements Serializable {  
    private int employeeId;  
    private int projetId;  
  
    public int getEmployeeId() { ... }  
  
    public void setEmployeeId(int employeeId)  
    { ... }  
  
    public int getProjetId() { ... }  
  
    public void setProjetId(int projetId)  
    { ... }  
  
    // Redéfinir aussi equals et hashCode  
}
```

# Gestionnaire d'entité

# Gestionnaire d'entités

- Classe `javax.persistence.EntityManager`
- Le gestionnaire d'entités (GE) est l'interlocuteur principal pour le développeur
- Il fournit les méthodes pour gérer les entités : les rendre persistantes, les supprimer de la base de données, retrouver leurs valeurs dans la base, etc.



# Création d'entité

```
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("test");  
EntityManager em = emf.createEntityManager();  
EntityTransaction transac = em.getTransaction();  
transac.begin();  
  
Personne nouvellePersonne = new Personne();  
nouvellePersonne.setId(4);  
nouvellePersonne.setNom("nom4");  
nouvellePersonne.setPrenom("prenom4");  
em.persist(nouvellePersonne);  
transac.commit();  
  
em.close();  
emf.close();
```

# Chargement d'entité

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

Personne personne = em.find(Personne.class, 4);
if (personne != null) {
    System.out.println("Personne.nom=" +
        personne.getNom());
}
em.close();
emf.close();}
```

# Recherche d'entité 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.
- L'objet Query encapsule et permet d'obtenir les résultats de son exécution. La méthode getSingleResult() permet d'obtenir un objet unique retourné par la requête.

```
Query query = em.createQuery("select p from Personne p  
where p.nom='nom2'");  
    Personne personne = (Personne) query.getSingleResult();  
    if (personne == null) {  
        System.out.println("Personne non trouvée");  
    } else {  
        System.out.println("Personne.nom=" +  
personne.getNom());  
    }
```

# Recherche d'entité par requête

- La méthode `getResultList()` renvoie une collection qui contient les éventuelles occurrences retournées par la requête..

```
Query query = em.createQuery("select p.nom from Personne p  
where p.id > 2");  
List noms = query.getResultList();  
for (Object nom : noms) {  
    System.out.println("nom = "+nom);  
}
```

# Recherche d'entité 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.

```
Query query = em.createQuery("select p.nom from Personne p  
where p.id > :id");  
    query.setParameter("id", 1);  
    List noms = query.getResultList();  
    for (Object nom : noms) {  
        System.out.println("nom = "+nom);  
    }
```

# Langage de requête

- Langage nommé JQL ou HQL
- Syntaxe proche du SQL (au lieu de tables et de colonnes on requête sur des objets et des attributs)
- Clause SELECT optionnelle : « FROM Personne p WHERE p.nom=:nom »
- Pas la peine de faire de jointure pour accéder au contenu de sous objet
- Exemple
  - « From Personne p where p.adresse.ville.cp=59000 »

# Modifier une entité

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

EntityTransaction transac = em.getTransaction();
transac.begin();

Query query = em.createQuery("select p from Personne p where
p.nom='nom2'");
Personne personne = (Personne) query.getSingleResult();
if (personne == null) {
    System.out.println("Personne non trouvée");
} else {
    System.out.println("Personne.prenom=" + personne.getPrenom());

    personne.setPrenom("prenom2 modifie");
    em.flush();

    personne = (Personne) query.getSingleResult();
    System.out.println("Personne.prenom=" + personne.getPrenom());
}

transac.commit();
```

# Supprimer une entité

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

EntityTransaction transac = em.getTransaction();
transac.begin();

Personne personne = em.find(Personne.class, 4);
if (personne == null) {
    System.out.println("Personne non trouvée");
} else {
    em.remove(personne);
}

transac.commit();

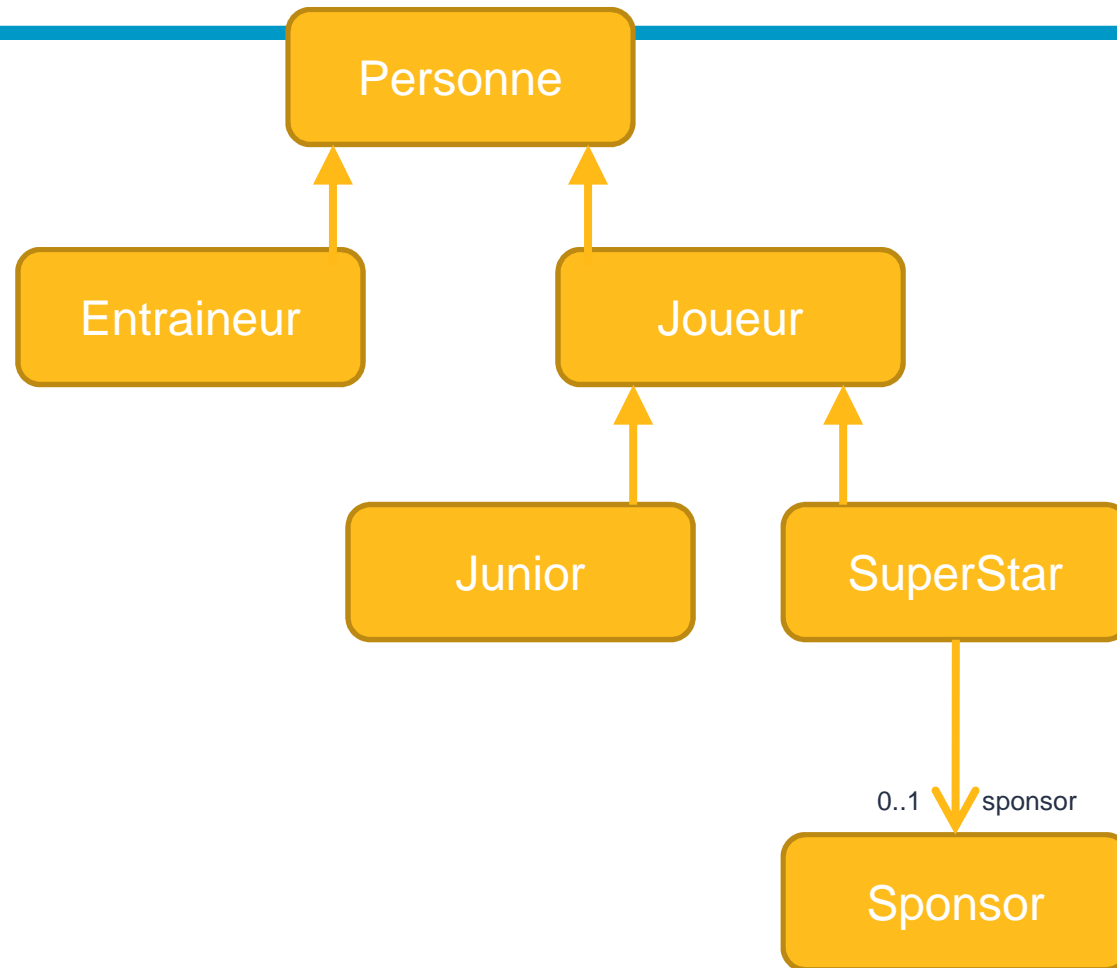
em.close();
emf.close();
```





# Héritage et polymorphisme

# Exemple



# Stratégies

- A ce jour, les implémentations de JPA doivent obligatoirement offrir 2 stratégies pour la traduction de l'héritage :
  - une seule table pour une hiérarchie d'héritage (SINGLE\_TABLE)
  - une table par classe ; les tables sont jointes pour reconstituer les données (JOINED)
- La stratégie « une table distincte par classe concrète » est seulement optionnelle (TABLE\_PER\_CLASS)

# Une table par hierarchie

- Sans doute la stratégie la plus utilisée
- Valeur par défaut de la stratégie de traduction de l'héritage
- Elle est performante et permet le polymorphisme
- Mais elle induit beaucoup de valeurs NULL dans les colonnes si la hiérarchie est complexe

# Exemple

A mettre dans la  
classe racine de la  
hiérarchie

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Personne {...}
```

« Employe » par  
défaut

```
@Entity
@DiscriminatorValue("E")
public class Employe extends Personne {
    ...
}
```

# Nom de la table

- Si on choisit la stratégie « une seule table pour une arborescence d'héritage » la table a le nom de la table associée à la classe racine de la hiérarchie

# Colonne discriminatrice

- Une colonne de la table doit permettre de différencier les lignes des différentes classes de la hiérarchie d'héritage
- Cette colonne est indispensable pour le bon fonctionnement des requêtes qui se limitent à une sous-classe
- Par défaut, cette colonne se nomme `DTYPE` et elle est de type `Discriminator.STRING` de longueur 31 (autres possibilités pour le type : `CHAR` et `INTEGER`)

# Colonne discriminatrice

- L'annotation `@DiscriminatorColumn` permet de modifier les valeurs par défaut
- Ses attributs :
  - `name`
  - `discriminatorType`
  - `columnDefinition` fragment SQL pour créer la colonne
  - `length` longueur dans le cas où le type est `STRING` (31 par défaut)



# Exemple

```
@Entity
@Inheritance
@DiscriminatorColumn(
    name="TRUC",
    discriminatorType="STRING",
    length=5)

public class Machin {
    ...
}
```

# Valeur discriminatrice

- Chaque classe est différenciée par une valeur de la colonne discriminatrice
- Cette valeur est passée en paramètre de l'annotation `@DiscriminatorValue`
- Par défaut cette valeur est le nom de la classe

# Exemple complet

```
20 @Entity
21
22
23 @Table( name="t_animal")
24 @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
25 @DiscriminatorColumn( name="typeanimal", discriminatorType=DiscriminatorType.STRING )
26 @DiscriminatorValue("ANI")
27 public class Animal implements Serializable {
28     @Id
29     @Column(name="ID", nullable=false)
30     @SequenceGenerator( name = "SeqAnimal", sequenceName = "seq_animal" )
31     @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "SeqAnimal" )
32     private Integer id;
33
34     @Column(name = "NOM", length = 30, nullable = false)
35     private String nom;
36
37     @Column(name = "MAMMIFERE", nullable = false)
38     private Boolean mammifere;
39
40     @Column(name = "NBPATTES", nullable = false)
41     private Integer nbPattes;
42 }
```

# Exemple complet

- Chat extends Animal


```
11 @Entity
12
13 @DiscriminatorValue("CHAT")
14
15 public class Chat extends Animal {
16
17     @Column(name = "NBMOUSTACHES", nullable = false)
18     private Integer nbMoustaches;
19 }
20
```

- Rhinoceros extends Animal

```
11 @Entity
12
13 @DiscriminatorValue("RHIN")
14
15 public class Rhinoceros extends Animal {
16
17     @Column(name = "NBCORNES", nullable = false)
18     private Integer nbCornes;
19 }
```

# Exemple complet

- En base

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Trigg
Name	Type	Length	Decimals	Allow Null		
id	numeric	30	0	<input type="checkbox"/>		1
nom	varchar	30	0	<input checked="" type="checkbox"/>		
nbpattes	numeric	2	0	<input type="checkbox"/>		
mammifere	bool	0	0	<input type="checkbox"/>		
typeanimal	varchar	4	0	<input type="checkbox"/>		
nbmoustaches	numeric	4	0	<input checked="" type="checkbox"/>		
nbcornes	numeric	4	0	<input checked="" type="checkbox"/>		

- Ex d'enregistrement

id	nom	nbpattes	mammifere	typeanimal	nbmoustaches	nbcornes
1400	Hermine	4	t	CHAT	8	(Null)
1450	pharyngite	4	t	RHIN	(Null)	2
1500	ani1	3	f	ANI	(Null)	(Null)
1501	ani2	3	f	ANI	(Null)	(Null)
1502	ani3	3	f	ANI	(Null)	(Null)
1503	pito	4	t	CHAT	8	(Null)
1504	...	4	t	CHAT	8	(Null)

# Une table par classe concrète

- Stratégie seulement optionnelle (fonctionne sous Hibernate et TopLink)
- Le polymorphisme est plus complexe à obtenir
- Chaque classe concrète correspond à une seule table totalement séparée des autres tables
- Toutes les propriétés de la classe, même celles qui sont héritées, se retrouvent dans la table

# Exemple

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

```
@Entity
@Table(name=EMPLOYEE)
public class Employe extends Personne {
    ...
}
```

# Une table par classe

- Toutes les classes, même les classes abstraites, sont représentées par une table
- Les données communes sont portés par la table mère
- Nécessite des jointures pour retrouver les propriétés d'une instance d'une classe
- Une colonne discriminatrice est ajoutée dans la table qui correspond à la classe racine de la hiérarchie d'héritage
- Cette colonne permet de simplifier certaines requêtes ; par exemple, pour retrouver les noms de tous les employés (classe Personne à la racine de la hiérarchie d'héritage)



# Exemple

- Un objet Vehicule

```
18 @Entity
19 @Table(name="t_vehicule")
20 @Inheritance(strategy=InheritanceType.JOINED)
21
22 public class Vehicule implements Serializable {
23     @Id
24     @Column(name="ID", nullable=false)
25     @SequenceGenerator( name = "SeqVehicule", sequenceName = "seq_vehicule" )
26     @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "SeqVehicule" )
27     private Integer id;
28
29     @Column(name = "prix", nullable = true)
30     private Integer prix;
31
32     @Column(name = "carburant", nullable = true)
33     private String carburant;
34 }
```

# Exemple

## ■ Un objet Avion héritant de Vehicule

```
12 @Entity
13 @Table(name="t_avion")
16 @PrimaryKeyJoinColumn(name="VEHICULE_ID")
17 public class Avion extends Vehicule {
18
19     @Column(name = "helice", nullable = true)
20     private Boolean helice;
21
22     @Column(name = "nbmoteurs", nullable = true)
23     private Integer nbMoteurs;
24 }
```

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules
Name	Type	Length	Decimals	Allow Null	
▶ vehicule_id	numeric	30	0	<input type="checkbox"/>	1
helice	bool	0	0	<input checked="" type="checkbox"/>	
nbmoteurs	numeric	4	0	<input checked="" type="checkbox"/>	

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Triggers	Privileges	Options	Comment	SQL Preview
Name		Field Names	Reference Schema	Reference Table	Foreign Field Names	On Delete	On Update			
▶ FK\$AVION\$VEHICULE		vehicule_id	public	t_vehicule	id	NO ACTION	NO ACTION			


# Exemple

- Un objet Voiture héritant de Vehicule

```






















12 @Entity
13 @Table(name="t_voiture")
16 @PrimaryKeyJoinColumn(name="VEHICULE_ID")
17 public class Voiture extends Vehicule {
18     @Column(name = "nbroues", nullable = true)
19     private Integer nbRoues;
20
21     @Column(name = "marque", nullable = true)
22     private String marque;
23
24     @Column(name = "modele", nullable = true)
25     private String modele;
26 }

```

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules
Name	Type	Length	Decimals	Allow Null	
vehicule_id	numeric	30	0	<input type="checkbox"/>	 1
nbroues	numeric	4	0	<input checked="" type="checkbox"/>	
marque	varchar	50	0	<input checked="" type="checkbox"/>	
modele	varchar	50	0	<input checked="" type="checkbox"/>	

Fields	Indexes	Foreign Keys	Uniques	Checks	Rules	Triggers	Privileges	Options	Comment	SQL Preview
Name	Field Names		Reference Schema	Reference Table	Foreign Field Names	On Delete	On Update			
FK\$VOITURE\$VEHICULE	vehicule_id		public	t_vehicule	id	NO ACTION	NO ACTION			

# Bilan

	Single table	Table per Class	Joined
Colonnes répétées	 Colonnes des classes filles cumulées + discriminant	 Colonnes des classes mères cumulées	
Clés étrangères	 Relation sur la classe mère uniquement	 Relation sur la classe fille uniquement	
Unicité		 Unicité à cheval sur plusieurs tables	
Non Nullité	 Les colonnes d'une classe fille sont laissées nulle chez ses soeurs		
Ecritures (Insert, Update, Delete)			 Plusieurs écritures pour une seule instance
Recherche sur classe mère		 Une union sur plusieurs tables	 Des jointures sur toutes les tables
Recherche sur classe fille			 Beaucoup de jointures

# Transactions

# 2 types de transactions

- Les transactions locales à une ressource, fournies par JDBC sont attachées à une seule base de données
- Les transactions JTA, ont plus de fonctionnalités que les transactions JDBC ; en particulier elles peuvent travailler avec plusieurs bases de données

# Transactions dans Java SE (sans serveur d'applications)

- D'après la spécification JPA, dans Java SE, les fournisseurs de persistance doivent supporter les transactions locales à une ressource mais ne sont pas obligés de supporter les transactions JTA
- La démarcation des transactions est choisie par le développeur
- Les contextes de persistance peuvent couvrir plusieurs transactions

# EntityTransaction

En dehors d'un serveur d'applications, une application doit utiliser l'interface `javax.persistence.EntityTransaction` pour travailler avec des transactions locales à une ressource

Une instance de `EntityTransaction` peut s'obtenir par la méthode `getTransaction()` de `EntityManager`



# EntityTransaction

```
public interface EntityTransaction {  
    public void begin();  
    public void commit();  
    public void rollback();  
    public void setRollbackOnly();  
    public boolean getRollbackOnly();  
    public void isActive();  
}
```

# Exemple

```
EntityManager em;  
...  
try {  
    em.getTransaction().begin()  
    ...  
    em.getTransaction().commit();  
} finally {  
    em.close();  
}
```

# Rollback

- En cas de rollback,
  - rollback dans la base de données
  - le contexte de persistance est vidé ; toutes les entités deviennent détachées

# Rollback

- Les instances d'entités Java gardent les valeurs qu'elles avaient au moment du `rollback`
- Mais ces valeurs sont le plus souvent fausses
- Il est donc rare d'utiliser ces entités en les rattachant par un `merge` à un GE
- Le plus souvent il faut relancer des requêtes pour récupérer des entités avec des valeurs correctes

# Transaction et contexte de persistance

- Quand un GE n'est pas géré par un container le contexte de persistance n'est pas fermé à la fin d'une transaction
- Quand un GE est géré par un container et que le contexte de persistance n'est pas de type « étendu », le contexte est fermé à la fin d'une transaction

# Synchronisation d'un GE avec une transaction

- Synchronisation d'un GE avec une transaction : le GE est enregistré auprès de la transaction ; un commit de la transaction provoquera alors automatiquement un flush du GE (le GE est averti lors du commit)
- En dehors d'un serveur d'applications (avec Java SE), un GE est obligatoirement synchronisé avec les transactions (qu'il a lancées par la méthode `begin()` de `EntityTransaction`)

# Synchronisation d'un GE avec une transaction

- Les modifications effectuées sur les entités gérées sont enregistrées dans la base de données au moment d'un flush du contexte de persistance
- Si le GE est synchronisé à la transaction en cours, le commit de la transaction enregistre donc les modifications dans la base
- Les modifications sont enregistrées dans la base, même si elles ont été effectuées avant le début de la transaction (avant `tx.begin()` dans Java SE)

# Limiter le nombre de connections

- Hibernate

```
<property name="hibernate.connection.pool_size" value="3" />
```

- Toplink

```
<property name="toplink.jdbc.read-connection.max" value="3" />
```

```
<property name="toplink.jdbc.write-connection.max" value="3" />
```



# Questions

---

?