JPA 2.0 (*Java Persistence API*) Java SE - Partie 1

Université de Nice - Sophia Antipolis
Version 2.14 – 2/1/13
Richard Grin

Plan de la partie 1

- Présentation de JPA
- Entités persistantes
- Gestionnaire d'entités
- Identité des entités
- Associations entre entités
- Compléments sur les associations (persistance et récupération des entités associées, ordre des listes, orphelins)
- Héritage entre entités

Présentation de JPA

JPA

- JPA (*Java persistence API*) est une API qui concerne la persistance des objets dans une base de données relationnelle
- JPA peut être utilisé par toutes les applications Java, Java SE ou Java EE
- C'est l'API ORM (*Object-Relational Mapping*) standard pour la persistance des objets Java

Quelques nouveautés de JPA 2.0 par rapport à JPA 1

- API « criteria » pour écrire des requêtes sans utiliser le langage JPQL, avec vérifications à la compilation
- Listes ordonnées dans la BD
- Collections persistantes d'éléments de type basic (qui ne correspondent pas à une association entre entités) et d'*embeddable*
- Facilités pour des mappings avec des clés primaires qui contiennent des clés étrangères

Principales propriétés de JPA

- Pas de code spécial lié à la persistance à implémenter dans le classes ; les objets POJO (*Plain Old Java Object*) peuvent être persistants
- Des appels simples de haut niveau permettent de gérer la persistance, tels que **`persist(objet)`** pour rendre un objet persistant ; pas d'appel de bas niveau comme avec JDBC
- Les données de la base peuvent être accédées avec une vision « objet » (pas relationnelle)

Spécification JPA

- Pour plus de précisions, lire la spécification à l'adresse http://jcp.org/aboutJava/communityprocess/pfd/jsr317/index.html

Avertissement

- JPA est le plus souvent utilisé dans le contexte d'un serveur d'applications (Java EE)
- Ce cours étudie l'utilisation de JPA par une application autonome, en dehors de tout serveur d'applications
- Quelques informations sur l'utilisation de JPA avec un serveur d'applications sont données dans ce cours mais sans entrer dans les détails
- Les différences sont essentiellement liées à la façon d'obtenir un gestionnaire d'entités et aux transactions ; presque tout le reste est identique

Fournisseur de persistance

- Comme pour JDBC, l'utilisation de JPA nécessite un fournisseur de persistance qui implémente les classes et méthodes de l'API
- GlassFish est l'implémentation de référence de la spécification EJB 3
- *EclipseLink* est l'implémentation de référence de la spécification JPA 2.0 (http://www.eclipse.org/eclipselink/)
- D'autres implémentations : Hibernate Entity Manager, OpenJPA, BEA Kodo

Entités

- Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécification de 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 (importations semblables pour toutes les annotations)

`@Entity`

- Cette annotation peut avoir un attribut **`name`** qui donne le nom de l'entité (rarement utilisé)
- Par défaut, le nom de l'entité est le nom terminal (sans le nom du paquetage) de la classe

Exemple :

```
@Entity(name="dept")
public class Departement {
```


Vocabulaire

- Dans la suite de ce cours et quand il n'y aura pas ambiguïté, « entité » désignera soit une classe entité, soit une instance de classe entité, suivant le contexte

Exemple d'entité – identificateur et attributs

```
@Entity
public class Departement {
  @Id
  @GeneratedValue
  private int id;
  private String nom;
  private String lieu;
```


Exemple d'entité – constructeurs

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

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


Exemple d'entité – une association

L'association inverse dans la classe **Employe**

```
@OneToMany(mappedBy="dept")
private Collection<Employe> employes =
  new ArrayList<Employe>();
public Collection<Employe> getEmployes() {
  return employes;
}
public void addEmploye(Employe emp) {
  employes.add(emp);
}
```


Fichiers de configuration XML

- Les annotations **@Entity** (et toutes les autres annotations JPA) peuvent être remplacées ou/et surchargées (les fichiers XML l'emportent sur les annotations) par des informations enregistrées dans un fichier de configuration XML
- Exemple :

```
<table-generator name="empgen"
 table="ID_GEN" pk-column-value="EmpId"/>
```

- La suite n'utilisera essentiellement que les annotations

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 peuvent 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 partie 2 du cours sur JPA ; section « Configuration d'une unité de persistance »)

Exemple (1/3)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation=
  "http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0
.xsd">
<persistence-unit name="employes"
  transaction-type="RESOURCE_LOCAL">
```

Exemple (2/3)

```
<provider>
  org.eclipse.persistence.jpa.PersistenceProvider
</provider>
<class>jpa.Departement</class>
<class>jpa.Employe</class>
<properties>
  <property
    name="javax.persistence.jdbc.driver"
    value="oracle.jdbc.OracleDriver"/>
  <property
    name="javax.persistence.jdbc.url"
    value="jdbc:oracle:thin:@cl.truc.fr:1521:XE"/>
```


Exemple (3/3)

```
    <property
      name="javax.persistence.jdbc.user"
      value="toto"/>
    <property
      name="javax.persistence.jdbc.password"
      value="xxxxx"/>
  </properties>
</persistence-unit>
</persistence>
```


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.

Exemple de code (1)

```
EntityManagerFactory emf = Persistence.
    createEntityManagerFactory("employes");
EntityManager em =
  emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Dept dept = new Dept("Direction", "Nice");
em.persist(dept);
dept.setLieu("Paris");
tx.commit();
```

sera enregistré dans la base de données

Exemple de code (2)

```
String queryString =
  "SELECT e FROM Employe e "
  + " WHERE e.poste = :poste";
Query query = em.createQuery(queryString);
query.setParameter("poste", "INGENIEUR");
List<Employe> liste =
  query.getResultList();
for (Employe e : liste) {
  System.out.println(e.getNom());
}
em.close();
emf.close()
```


Encadrer par **`try-catch-finally`**

- Remarque : le code précédent (et de nombreux exemples de ce cours) devrait être inclus dans un bloc **`try`** – **`finally`** (les **`close`** finaux dans le **`finally`**), et pourrait contenir éventuellement des blocs **`catch`** pour attraper les exceptions (des **`RuntimeException`**) lancées par les méthodes de **`EntityManager`**
- Ce bloc **`try`** – **`finally`** a été omis pour ne pas alourdir le code

Exemple avec **`try-catch-finally`**

```
EntityManagerFactory emf = null;
EntityManager em = null;
EntityTransaction tx = null;
try {
  emf = Persistence.createEntityManagerFactory("..");
  em = emf.createEntityManager();
  tx = em.getTransaction();
  tx.begin();
  ...
  tx.commit();
} catch(Exception e) { if (tx != null) tx.rollback; }
finally {
  if (em != null) em.close();
  if (emf != null) emf.close();
}
```


`try` avec ressources

- Depuis le JDK 7.0, il est possible d'utiliser un « try avec ressource » qui allège le code
- Une future version de JPA permettra de l'utiliser avec les gestionnaires d'entités et leur fabrique (pas encore le cas pour la version JPA 2.0)

Exemple de **`try`** avec ressources

```
EntityTransaction tx = null;
try (
  EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("..");
  EntityManager em = emf.createEntityManager();) {
  tx = em.getTransaction();
  tx.begin();
  ...
  tx.commit();
} catch(Exception e) { if (tx != null) tx.rollback; }
```

Ce code ne fonctionne pas avec JPA 2.0

Contexte de persistance

- La méthode **`persist(`*`objet`*`)`** de la classe **`EntityManager`** rend persistant un objet
- L'objet est alors géré par le GE
- Toute modification apportée à cet objet sera automatiquement enregistrée dans la base de données
- L'ensemble des entités gérées par un GE s'appelle un contexte de persistance

GE – contexte de persistance

- Dans le cadre d'une application autonome, la relation est simple : un GE possède un contexte de persistance, qui n'appartient qu'à lui et il le garde pendant toute son existence
- Lorsque le GE est géré par un serveur d'applications, la relation est plus complexe ; un contexte de persistance peut se propager d'un GE à un autre et il peut être fermé automatiquement à la fin de la transaction en cours (pas étudié dans ce cours)

Entités

Caractéristiques

- Seules les entités peuvent être
  - passées en paramètre d'une méthode d'un **`EntityManager`** ou d'un **`Query`**
  - renvoyées par une requête (**`Query`**)
  - le but d'une association
  - référencées dans une requête JPQL
- Une classe entité peut utiliser d'autres classes pour conserver des états persistants (*MappedSuperclass* ou *Embeddable* étudiées plus loin ; depuis JPA 2, les 3 derniers points sont aussi vrais pour les *Embeddable*)

Conditions pour une classe entité

- 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`**

Conditions pour une classe entité

- Elle ne doit pas être une classe interne
- Elle peut être une classe abstraite mais elle ne peut pas être une interface
- Pas de contrainte sur le niveau d'accessibilité de la classe (elle peut ne pas être **`public`**)

Méthodes equals et hashcode

- La redéfinition de ces 2 méthodes n'est pas requise pour une classe entité
- Cependant, dans certaines circonstances, ces 2 méthodes sont indispensables pour le bon fonctionnement des applications
- Il est donc recommandé de les redéfinir pour toutes les entités, en utilisant la propriété annotée par **`@Id`** comme critère d'égalité

Exemple de **`equals`** (classe **`C`**)

```
public boolean equals(Object o) {
  if (o == null || o.getClass() != this.getClass()) {
    return false;
  }
  C autreC = (C) o;
  // cas où id pas primitif (Integer par exemple)
  if ((this.id == null && autreC.id != null)
       || (this.id != null
            && !this.id.equals(autreC.id))) {
     return false;
  }
  return true;
}
```


Exemple de **`equals`** (classe **`C`**)

```
public boolean equals(Object o) {
 if (o == null || o.getClass() != this.getClass()) {
    return false;
  }
  C autreC = (C) o;
  // cas où id est primitif (int par exemple)
  return this.id == autreC.id;
}
```

Exemple de **hashcode** (classe **C**)

```
// Cas id non primitif (Integer par exemple)
public int hashCode() {
  return (id != null ? id.hashCode() : 0);
}
```

```
// Cas id primitif (int par exemple)
public int hashCode() {
  return id;
}
```


Convention de nommage *JavaBean*

- Un JavaBean possède des propriétés
- Une propriété nommée « prop » est représentée par 2 méthodes pour lire et écrire sa valeur :
  - **`getProp()`** (ou **`isProp()`** si la propriété est de type **`boolean`**) (le « *getter* »)
  - **`setProp(TypeDeProp prop)`**(le « *setter* »)
- Souvent une propriété correspond à une variable d'instance de même nom, mais ça n'est pas obligatoire

2 types d'accès

- L'état persistant d'un objet est constitué par les valeurs de ses attributs
- Le fournisseur de persistance peut accéder à la valeur d'un attribut de 2 façons
  - soit en accédant directement à la variable d'instance ; c'est l'accès « par champ »
  - soit en passant par les accesseurs (*getter* ou *setter*) ; c'est l'accès « par propriété »

Conditions sur les accesseurs

- Si l'accès se fait par champ, la variable d'instance peut ne pas avoir de *getter* et de *setter*
- Si l'accès se fait par propriété, les getter et setter sont tous les deux obligatoires ; ils doivent être **`protected`** ou **`public`**

Accès par propriété

- Les accesseurs peuvent contenir d'autres instructions que le seul code lié à la valeur de la variable sous-jacente
- Ces instructions seront exécutées par le fournisseur de persistance
- Si une exception est levée par un accesseur, la transaction est marquée pour un *rollback* ; les exceptions contrôlées sont enveloppées par une **`PersistenceException`** (non contrôlée, sous **`RuntimeException`**)

Type d'accès par défaut

- Le type d'accès par défaut (par champ ou par attribut) est défini par la position des annotations dans la classe entité
- Il doit être le même pour toute une hiérarchie d'héritage (ne pas mélanger les types d'accès)
- JPA 2.0 permet aussi de choisir le type d'accès pour chaque classe, et même pour chaque attribut dans une classe, en ajoutant une annotation **`Access`** :
  `@Access(AccessType.PROPERTY)` ou **`@Access(AccessType.FIELD)`**

Indiquer un autre mode d'accès à l'intérieur d'une classe (1/2)

- 3 étapes pour indiquer un autre mode d'accès que l'accès par défaut pour un des attribut d'une classe
- Le transparent suivant donne les 3 étapes pour le cas où la classe a un mode d'accès par champ et où on veut un mode d'accès par propriété pour un attribut « nom »
- Pour le cas inverse, il suffit d'inverser les rôles des champs et des getters

Indiquer un autre mode d'accès à l'intérieur d'une classe (2/2)

1. Donner explicitement le mode d'accès de la classe par une annotation (ne pas oublier !) : **`@Entity @Access(AccessType.FIELD) public class …`**
2. Annoter le *getter* par son mode d'accès : **`@Access(AccessType.PROPERTY) String getNom() { …`**
3. Annoter par **`@Transient`** le champ correspondant à la propriété : **`@Transient String nom;`**

`@Access(PROPERTY)`

- Si l'accès par défaut est l'accès par champ et qu'un *getter* est annoté avec **`@Access(AccessType.PROPERTY)`**, il ne faut pas oublier d'annoter le champ correspondant avec **`@Transient`**
- Sinon l'attribut sera rendu persistant dans 2 colonnes de la base de données

Une erreur à ne pas faire

- Tous les attributs d'une hiérarchie de classes qui ne sont pas annotés par **`@Access`** doivent avoir le même type d'accès
- Le mélange d'annotations sur les variables d'instance et sur les *getter* peut causer des erreurs difficiles à comprendre pour un débutant

Quel type d'accès choisir ? (1)

- En programmation objet il est conseillé d'utiliser plutôt les accesseurs que les accès directs aux champs (meilleur contrôle des valeurs)
- Pour JPA c'est différent : l'accès par propriété, oblige à avoir des *setters* et *getters* pour toutes les propriétés, ce qui peut être néfaste et aller à l'encontre du souci d'encapsulation
- Par exemple, les *setters* peuvent permettent à l'utilisateur de ne pas utiliser une méthode ajoutée pour obliger à bien mettre à jour les 2 bouts d'une association bidirectionnelle

Quel type d'accès choisir ? (2)

- L'accès par champ permet de distinguer l'accès aux attributs par JPA et l'accès aux attributs par l'application (effectué par *setters* et *getters*)
- Il est donc conseillé de choisir plutôt l'accès par champ avec JPA
- Pour des cas particuliers il est toujours possible d'utiliser **`@Access(PROPERTY)`** sur certains attributs ; mais attention, un traitement complexe dans un *setter* peut occasionner un problème s'il dépend de la valeur d'autres attributs (pas nécessairement initialisés à ce moment)

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

Types persistants (1)

- Les attributs persistants des entités peuvent être d'un des types suivants :
  - type « *basic* » (appelé ainsi dans la spécification JPA)
  - type « *Embeddable* » (étudié plus loin dans ce cours)
  - collection d'éléments de type *basic* ou *Embeddable* (nouveauté de JPA 2)

Types persistants (2)

- Un attribut persistant peut aussi représenter une association entre entités :
  - entité
  - collection d'entités
- C'est le type déclaré de l'attribut qui détermine la façon de le rendre persistant par le fournisseur de persistance

Types « basic »

- Types primitifs (**`int`**, **`double`**,…)
- **`String`**, **`BigInteger`**, **`BigDecimal`**, classes enveloppes de type primitifs (**`Integer`**,…)
- **`Date`** (des paquetages **`util`** et **`sql`**), **`Calendar`**, **`Time`**, **`Timestamp`**
- Enumérations
- Tableaux de **`byte`**, **`Byte`**, **`char`**, **`Character`**
- Plus généralement **`Serializable`** (un tel attribut sera sauvegardé comme un tout dans la base, dans une seule colonne)

Et les autres tableaux ?

- On remarque que les seuls tableaux « persistants » sont les tableaux de **`byte`** et de **`Character`**
- Si une classe a un attribut tableau d'un autre type, que va-t-il se passer ?
- Si le type des éléments du tableau est **`Serializable`**, le tableau le sera aussi et le tableau sera rendu persistant comme un Serializable, c'est-à-dire comme un tout (Blob ou Clob)

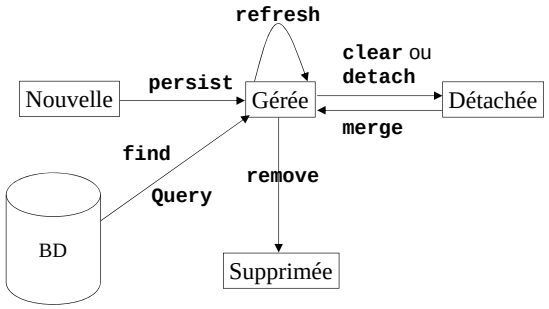
Cycle de vie d'une instance d'entité

- L'instance peut être
  - nouvelle (new) : elle est créée mais pas associée à un contexte de persistance
  - gérée par un gestionnaire de persistance ; elle a une identité dans la base de données (un objet peut devenir géré par la méthode **`persist`**, ou **`merge`** d'une entité détachée ; un objet géré peut aussi provenir d'une requête faite par un gestionnaire d'entités ou d'une navigation à partir d'un objet géré)

Cycle de vie d'une instance d'entité

- détachée : elle a une identité dans la base mais elle n'est plus associée à un contexte de persistance (une entité peut, par exemple, devenir détachée si le contexte de persistance est vidé ou si elle est envoyée dans une autre JVM par RMI)
- supprimée : elle a une identité dans la base ; elle est associée à un contexte de persistance et ce contexte doit la supprimer de la base de données (passe dans cet état par la méthode **`remove`**)

Cycle de vie d'une entité



Tables de la base de données

- Dans les cas simples, une table correspond à une classe
  - le nom de la table est le nom de la classe
  - les noms des colonnes correspondent aux noms des attributs persistants
- Par exemple, les données de la classe **`Departement`** sont enregistrées dans la table **`Departement`** (ou **`DEPARTEMENT`**) dont les colonnes se nomment **`id`**, **`nom`**, **`lieu`**

Configuration « par exception »

- La configuration des classes entités suppose des valeurs par défaut pour le *mapping* de ces classes avec les tables de la base de données
- Il n'est nécessaire d'ajouter des informations de configuration que si ces valeurs par défaut ne conviennent pas
- Par exemple, **`@Entity`** suppose que la table qui contient les données des instances de la classe a le même nom que l'entité

Génération du schéma relationnel

- Lorsque la base de données est créée par l'application JPA, les valeurs par défaut conviennent le plus souvent
- Voir la section « Précisions pour la génération du schéma de la base de données » à la fin de la partie 2 de ce support

Adaptation à un schéma préexistant

- Lorsque le schéma de la base de données existe déjà lorsque l'application qui utilise JPA est écrite, les valeurs par défaut ne conviennent souvent pas, surtout pour ce qui concerne les associations entre entités
- Voir la section « Adaptation à une base relationnelle préexistante » vers la fin de la partie 2 de ce support

Nom de table

- Pour donner à la table un autre nom que le non de la classe, il faut ajouter une annotation **@Table**
- Exemple :

```
@Entity
@Table(name = "AUTRE_NOM")
public class Classe {
  ...
}
```

- La casse indifférente pour les noms de table ou de colonne : AUTRE_NOM = autre_Nom
  (sauf pour certains SGBD comme MySQL)

Nom de colonne

- Pour donner à une colonne de la table un autre nom que le nom de l'attribut correspondant, il faut ajouter une annotation **@Column**
- Cette annotation peut aussi comporter des attributs pour définir plus précisément la colonne
- Exemple :

```
@Column(name="NOM_CLIENT",
updatable=false, length=80)
private String nomClient;
```


Classe *Embeddable* (1)

- Les instances d'une classe annotée **@Embeddable** sont persistantes mais elles n'ont pas une identité dans la base de données
- Des attributs d'une entité peuvent être annotés **@Embedded** pour indiquer que leur type est une classe Embeddable (optionnel ; par défaut si le type de l'attribut est annoté par **@Embeddable**)
- Les données de ces attributs seront sauvegardées dans la table associée à l'entité qui les contient

Exemple

```
@Embeddable
public class Adresse {
  private int numero;
  private String rue;
  private String ville;
  . . .
}
```

```
@Entity
public class Employe {
  @Embedded private Adresse adresse;
  ...
```

Optionnel

Classe *Embeddable* (2)

- Elle peut contenir les mêmes annotations que les entités (avec **@Column** par exemple)
- Elle doit remplir les mêmes conditions que les entités (constructeur sans paramètre,…)
- Le type d'accès est le même que l'entité (ou mapped superclass ou classe Embeddable) qui la contient (l'annotation Access peut être utilisé pour changer ce type d'accès)

Classe *Embeddable* (3)

- Les types permis pour les attributs sont les mêmes que les types permis pour les attributs des entités, y compris les Embeddable ou les collections d'Embeddable
- Elle peut contenir une association vers une ou plusieurs entités ; l'association se fera entre l'entité qui contient la classe Embeddable et l'autre entité

Restriction sur les classes Embeddable

- Une instance des ces classes ne peut pas être référencée par plusieurs entités différentes

Utilisation multiple d'une classe Embeddable

- Une classe entité peut référencer plusieurs instances d'une même classe Embeddable
- Par exemple, la classe **`Employe`** peut comporter l'adresse du domicile et l'adresse du travail des employés
- En ce cas, les noms des colonnes dans la table de l'entité ne peuvent être les mêmes pour chacune des utilisations
- L'annotation **`@AttributeOverride`** peut résoudre le problème

`@AttributeOverride(s)`

- Un champ annoté par **`@Embedded`** peut être complété par une annotation **`@AttributeOverride`**, ou plusieurs de ces annotations insérées dans une annotation **`@AttributeOverrides`**
- Ces annotations permettent d'indiquer le nom d'une ou de plusieurs colonnes dans la table de l'entité
- Elles peuvent aussi être utilisées si une classe insérée est référencée par plusieurs classes entités différentes

Exemple

```
@Entity
public class Employe {
  @Embedded
  @AttributeOverrides({
    @AttributeOverride(
      name="ville",
      column=@column(name="ville_travail")),
    @AttributeOverride(...)
  })
  // Adresse du travail
  private Adresse adresseTravail;
```


Annotation pour LOB

- L'annotation **`@Lob`** permet d'indiquer qu'un attribut est un LOB (*Large OBject*) : soit un CLOB (Character LOB, tel un long résumé de livre), soit un BLOB (Binary LOB, tel une image ou une séquence vidéo)
- Le fournisseur de persistance pourra ainsi éventuellement traiter l'attribut de façon spéciale (utilisation de flots d'entrées-sorties par exemple)
- Exemple : `@Lob private byte[] image`

Mode de récupération des attributs

- Un attribut peut être récupéré en mode « retardé » : il n'est chargé en mémoire que si c'est nécessaire, c'est-à-dire si le code désigne explicitement l'attribut
- Par défaut, tous les attributs d'une entité sont chargés en mémoire en même temps que l'entité ; ils sont en mode EAGER

Mode de récupération des attributs

- Si un attribut est d'un type de grande dimension (LOB), il peut être intéressant pour les performance ou l'occupation mémoire de le passer en mode LAZY :

```
@Lob
@Basic(fetch = FetchType.LAZY)
private byte[] cvPdf
```

(à utiliser avec parcimonie car peut générer un accès supplémentaire à la base de données et provoquer des problèmes si l'entité est détachée)

- Cette annotation n'est qu'une suggestion au GE, qu'il peut ne pas suivre

Annotation pour énumération

- Une annotation spéciale n'est pas nécessaire pour un attribut de type énumération si l'énumération est sauvegardée dans la BD sous la forme des numéros des constantes de l'énumération (de 0 à n)
- Si on souhaite sauvegarder les constantes sous la forme de la forme d'une **`String`** qui représente le nom de la valeur de l'énumération, il faut utiliser l'annotation **`@Enumerated`**

Exemple

```
@Enumerated(EnumType.STRING)
private TypeEmploye typeEmploye;
```


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)

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`**

Exemple

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

Collection d'éléments

- JPA 2 a ajouté la possibilité de sauvegarder dans une table à part les attributs qui sont des collections d'éléments de type « basic » ou Embeddable, appelées « collection d'éléments » dans la spécification
- Ces attributs doivent être annotés avec l'annotation **`@ElementCollection`**
- Si l'annotation n'est pas utilisée, l'attribut est considéré comme un Serializable et sauvegardé comme un LOB dans la base

Représentation de la collection dans la base de données

- Les données de la collection sont représentées dans la BD par une table qui contient la clé primaire de l'entité et une ou plusieurs colonnes qui correspondent aux données de type basic ou de type embeddable
- L'annotation **`@CollectionTable`** permet de changer la valeur par défaut du nom de la table (nomEntité_nomAttributCollection)
- Le nom de la colonne peut être modifié avec l'attribut habituel **`@Column`**

Annotation `@ElementCollection`

- Le mode de récupération des éléments de la collection est LAZY par défaut (voir la section sur les compléments sur les associations plus loin dans ce support)
- Il est possible d'indiquer un mode EAGER avec l'annotation **`@ElementCollection`**

Exemples

-
```
@ElementCollection
private Set<String> synonymes
  = new HashSet<String>();
```
-
```
@ElementCollection(fetch=FetchType.EAGER)
private Set<String> synonymes
  = new HashSet<String>();
```


Map de types basic ou embeddable

- L'annotation **`@ElementCollection`** peut aussi être utilisée pour une map dont un des types est basic ou embeddable
- Comme les collections, les maps sont sauvegardées dans une table à part
- La différence est que la table contient une colonne de plus pour les valeurs de la clé de la map
- **`@MapKeyColumn`** permet de changer la valeur par défaut pour le nom de la colonne

Tables multiples

- Il est possible de sauvegarder une entité sur plusieurs tables
- Voir **`@SecondaryTable`** dans la spécification JPA
- C'est surtout utile pour les cas où la base de données existe déjà et ne correspond pas tout à fait au modèle objet

Schéma relationnel

- Dans le cas où le schéma relationnel est construit automatiquement à partir des annotations, il est possible de préciser des informations sur les tables générées ou les colonnes de ces tables
- Par exemple, une contrainte d'unicité, ou « not null », la longueur des colonnes de type varchar, la précision des nombres à virgule, ou même le texte entier qui permet de définir une colonne (en SQL)
- Pour plus de détails voir la fin de la 2ème partie de ce cours sur JPA ou la spécification JPA

Exemple

```
@Entity
@Table(name="PARTICIPATION2",
  uniqueConstraints =
    @UniqueConstraint(
      columnNames =
        {"EMPLOYE_ID", "PROJET_ID"})
)
public class Participation {
  ...
}
```


Les interfaces (1/2)

- Une interface ne peut être une entité
- Si une classe a une association avec d'autres classes, l'association ne peut donc être traduite en JPA par un attribut de type interface ou une collection d'attributs dont les éléments sont des interfaces
- Par exemple, la classe **`Dessin`** ne peut contenir un attribut **`Collection<Dessinable>`** pour traduire le fait qu'un dessin est composé de figures « dessinables » (**`Dessinable`** est une interface)

Les interfaces (2/2)

- La solution est de s'arranger pour avoir une classe mère entité, avec un id, au-dessus de toutes les classes qui implémentent l'interface
- Si une des classes hérite déjà d'une autre classe, cette solution n'est pas possible
- Il faut alors modifier cette classe et remplacer l'héritage par de la composition ; la classe peut ainsi hériter de la classe mère entité qui contient un id

Gestionnaire d'entités (*Entity Manager*), GE

Principe de base

- La persistance des entités n'est pas transparente
- Une instance d'entité ne devient persistante que lorsque l'application appelle la méthode appropriée du gestionnaire d'entités (**`persist`** ou **`merge`**)
- Cette conception a été voulue par les concepteurs de l'API par souci de flexibilité et pour permettre un contrôle fin sur la persistance des entités

Unité de persistance

- C'est une configuration nommée qui contient les informations nécessaires à l'utilisation d'une base de données
- Elle est associée à un ensemble de classes entités

Configuration d'une unité de persistance

- Les informations sur une unité de persistance sont données dans un fichier **`persistence.xml`** situé dans un sous-répertoire **`META-INF`** d'un des répertoires du *classpath*
- Voir section « Configuration d'une unité de persistance » dans la 2ème partie de ce cours

Contexte de persistance (1)

- Les entités gérées par un gestionnaire d'entités forment un contexte de persistance
- Quand une entité est incluse dans un contexte de persistance (**`persist`** ou **`merge`**), l'état de l'entité est automatiquement sauvegardé dans la base au moment du *commit* de la transaction
- Propriété importante : dans un contexte de persistance il n'existe pas 2 entités différentes qui représentent des données identiques dans la base

Contexte de persistance (2)

- Un contexte de persistance ne peut appartenir qu'à une seule unité de persistance
- Une unité de persistance peut contenir plusieurs contextes de persistance
- C'est la responsabilité de l'application de s'assurer qu'une entité n'appartient qu'à un seul contexte de persistance (afin que 2 entités de 2 contextes de persistance différents ne puissent correspondre à des données identiques dans la base de données)

Contexte de persistance - cache

- Le contexte de persistance joue le rôle de cache et évite ainsi des accès à la base
- Si le code veut récupérer des données (par un **`find`** ou un **`query`**) qui correspondent à une entité du contexte, ce sont les données du cache qui sont renvoyées
- Important : si les données de la base ont été modifiées (et validées) en parallèle sans utiliser ce contexte de persistance, les données récupérées dans le cache ne tiennent pas compte de ces modifications

Contexte de persistance - cache

- Ce fonctionnement peut être bénéfique : meilleures performances, isolation « lecture répétable » sans modifier les paramètres de la base de données
- S'il pose un problème, il est possible de récupérer des modifications effectuées en parallèle sur les données d'une entité en utilisation la méthode **`refresh`** de **`EntityManager`**

Interface **`EntityManager`**

- Elle représente un GE
- Implémentation fournie par le fournisseur de persistance

Types de GE

- GE géré par le container (uniquement disponible dans un serveur d'applications ; pas étudié dans ce cours) ; le contexte de persistance n'existe souvent que le temps d'une seule transaction
- GE géré par l'application (seul type disponible en dehors d'un serveur d'applications) ; le contexte de persistance reste attaché au GE pendant toute son existence

Cycle de vie d'un GE

- En dehors d'un serveur d'applications, c'est l'application qui décide de la durée de vie d'un GE
- La méthode **`createEntityManager()`** de la classe **`EntityManagerFactory`** créé un GE
- Le GE est fermé avec la méthode **`close()`** de la classe **`EntityManager`** ; il ne sera plus possible de l'utiliser ensuite

Fabrique de GE

- La classe **`Persistence`** permet d'obtenir une fabrique de gestionnaire d'entités par la méthode **`createEntityManagerFactory`**
- 2 variantes surchargées de cette méthode :
  - 1 seul paramètre qui donne le nom de l'unité de persistance (définie dans le fichier persistence.xml du répertoire META-INF)
  - Un 2ème paramètre de type **`Map`** (non générique) qui contient des valeurs qui vont écraser les propriétés par défaut contenues dans persistence.xml (très utile si toutes les propriétés ne sont pas connues au moment de l'écriture de l'application)

A savoir

- Une **`EntityManagerFactory`** est « *thread-safe* »
- Un **`EntityManager`** ne l'est pas
- Créer une **`EntityManagerFactory`** est une opération lourde
- Créer un **`EntityManager`** est une opération légère
- Il est donc intéressant de conserver une **`EntityManagerFactory`** entre 2 utilisations

Interface **`EntityManagerFactory`**

- Implémentée par les fabriques de GE renvoyées par la méthode **`createEntityManagerFactory`**
- Elle contient la méthode **`isOpen()`** utilisée pour savoir si une fabrique est ouverte
- et la méthode **`close()`** qui doit être lancée lorsque l'on n'a plus besoin de la fabrique, pour libérer les ressources qu'elle utilise
- La méthode **`close`** rend inutilisables tous les GE créés par la fabrique

Mauvaise configuration

- Si elle rencontre une mauvaise configuration (dans le fichier persistence.xml, dans les annotations, y compris dans la syntaxe des requêtes nommées ou dans les fichiers XML) la méthode **`createEntityManagerFactory`** ne se termine pas correctement et lance une exception

Méthodes de `EntityManager`

- **`void persist(Object entité)`**
- **`<T> T merge(T entité)`**
- **`void remove(Object entité)`**
- **`<T> T find(Class<T> classeEntité, Object cléPrimaire)`**
- **`<T> T getReference(Class<T> classeEntité, Object cléPrimaire)`**
- **`void flush()`**
- **`void setFlushMode(FlushModeType flushMode)`**

Méthodes de `EntityManager`

- **`void lock(Object entité, LockModeType lockMode)`**
- **`void refresh(Object entité)`**
- **`void clear()`**
- **`void detach(Object entité)`**
- **`boolean contains(Object entité)`**
- **`void joinTransaction()`**
- **`void close()`**
- **`boolean isOpen()`**

Méthodes de `EntityManager`

- **`EntityTransaction getTransaction()`**
- **`Query createQuery(String requête)`**
- **`Query createNamedQuery(String nom)`**
- **`Query createNativeQuery(String requête)`**
- **`Query createNativeQuery(String requête, Class classeRésultat)`**

`flush`

- Toutes les modifications effectuées sur les entités du contexte de persistance gérées par le GE sont enregistrées dans la BD lors d'un *flush* du GE (mais il faudra un commit pour que ces modifications soient définitivement enregistrées dans la BD)
- Au moment du *flush*, le GE étudie ce qu'il doit faire pour chacune des entités qu'il gère et il lance les commandes SQL adaptées pour modifier la base de données (INSERT, UPDATE ou DELETE)

`flush`

- Un *flush* est automatiquement effectué au moins à chaque *commit* de la transaction en cours
- Une exception **`TransactionRequiredException`** est levée si la méthode **`flush`** est lancée en dehors d'une transaction

`flush`

- Soit X une des entités gérée, avec une association de X vers une entité Y
- Si cette association est notée avec **`cascade=PERSIST`** ou **`cascade=ALL`**, Y est elle aussi flushée
- Sinon, si Y est *new* ou *removed*, une exception **`IllegalStateException`** est levée et la transaction est marquée pour un *rollback*
- Sinon, si Y est détachée et X possède l'association, Y est *flushée* ; si Y est le bout propriétaire (voir section « associations » plus loint), le comportement est indéfini

Mode de *flush*

- Normalement (mode **`FlushMode.AUTO`**) un *flush* des entités concernées par une requête est effectué avant la requête pour que le résultat tienne compte des modifications effectuées en mémoire sur ces entités
- Il est possible d'éviter ce *flush* avec la méthode **`setFlushMode`** :
  `em.setFlushMode(FlushMode.COMMIT);`
- En ce cas, un *flush* ne sera lancé qu'avant un commit
- Il est possible de modifier ce mode pour une seule requête (voir **`Query`**)

`persist`

- Une entité « nouvelle » devient une entité gérée
- L'état de l'entité sera sauvegardé dans la BD au prochain *flush* ou *commit*
- Aucune autre instruction ne sera nécessaire pour que les modifications effectuées ensuite sur l'entité par l'application soient enregistrées au moment du commit
- En effet le GE conserve toutes les informations nécessaires sur les entités qu'il gère

`persist(A)`

- Si A est nouvelle, elle devient gérée
- Si A était déjà gérée, **`persist`** est ignorée mais l'opération **`persist`** « cascade » sur les entités associées si l'association a l'attribut **`CascadeType.PERSIST`** (voir transparent suivant)
- Si A est supprimée (a été passée en paramètre à **`remove`**), elle devient gérée
- Si A est détachée, une **`IllegalArgumentException`** est lancée (il aurait plutôt fallu appeler la méthode **`merge`**)

Remarque sur `persist`

- Attention, si on récupère une entité par un **`find`** ou un **`query`** et que l'on modifie cette entité en lui ajoutant des objets associés, ceux-ci ne seront pas automatiquement persistants, même s'il y un « **`cascade = cascadeType.PERSIST`** » sur l'association vers ces objets
- La cascade ne marche que pour la méthode **`persist`** ! Voir transparent précédent pour résoudre ce problème

`EntityExistException`

- **`persist`** est destiné aux entités qui n'existent pas déjà dans la base de données : la méthode lance une exception non contrôlée **`EntityExistsException`** si l'entité existe déjà dans la base de données
- L'exception peut être lancée au moment du **`persist`** ou au moment du **`commit`** ou du **`flush`** dans la base

`remove`

- Une entité gérée devient « supprimée »
- Les données correspondantes seront supprimées de la BD

`remove(A)`

- Si A est une entité gérée, elle devient « supprimée » (les données correspondantes de la base seront supprimées de la base au moment du flush du contexte de persistance)
- Ignoré si A est nouvelle ou déjà supprimée
- Si A est détachée, une **`IllegalArgumentException`** est lancée

`refresh`

- Le GE peut synchroniser avec la BD une entité qu'il gère en rafraichissant son état en mémoire avec les données actuellement dans la BD :
  `em.refresh(entite);`
- Les données de la BD sont copiées dans l'entité
- Utiliser cette méthode pour s'assurer que l'entité a les mêmes données que la BD
- Peut être utile pour les transactions longues

`refresh(A)`

- Ignorée si A est nouvelle ou supprimée
- Si A est nouvelle, l'opération « cascade » sur les associations qui ont l'attribut **`CascadeType.REFRESH`**
- Si A est détachée, une **`IllegalArgumentException`** est lancée
- Si A a été supprimée de la base de données, l'exception **`EntityNotFoundException`** est lancée

`clear` et `detach`

- **`clear()`** vide le contexte de persistance
- Toutes les entités gérées de ce contexte deviennent détachées
- **`detach(entité)`** enlève l'entité du contexte de persistance ; l'entité devient détachée ; lance une **`IllegalArgumentException`** si l'argument n'est pas une entité gérée

`find`

- La recherche est polymorphe : l'entité récupérée peut être de la classe passée en paramètre ou d'une sous-classe (renvoie **`null`** si aucune entité n'a l'identificateur passé en paramètre)
- Exemple :
  `Article p = em.find(Article.class, 128);`
  peut renvoyer un article de n'importe quelle sous-classe de **`Article`** (**`Stylo`**, **`Ramette`**,…)

`lock(A)`

- Le fournisseur de persistance gère les accès concurrents aux données de la BD représentées par les entités avec une stratégie optimiste
- **`lock`** permet de modifier la manière de gérer les accès concurrents à une entité A
- Sera étudié plus loin dans la section sur la concurrence

`close`

- Après l'appel de cette méthode, toutes les autres méthodes de **`EntityManager`** lancent une **`IllegalStateException`** (sauf **`isOpen()`** et **`getTransaction()`**)
- Il est impossible de rouvrir un **`EntityManager`**
- Dans un environnement géré (J2EE), la méthode est inutile puisque c'est le serveur d'applications qui se charge de fermer l'**`EntityManager`**

`joinTransaction`

- Pour les applications gérées par un serveur d'applications (donc hors du cadre de ce cours) lorsqu'un contexte de persistance est géré par l'application et que le gestionnaire d'entités a été créé alors qu'aucune transaction JTA n'était en cours
- Synchronise le contexte de persistance avec la transaction pour que le commit de la transaction provoque automatiquement le flush du contexte de persistance dans la base de données

Entité détachée (1)

- Une application distribuée sur plusieurs ordinateurs peut utiliser avec profit des entités détachées
- Une entité gérée par un GE peut être détachée de son contexte de persistance ; par exemple, si le GE est fermé ou si l'entité est transférée sur une autre machine en dehors de la portée du GE

Entité détachée (2)

- Une entité détachée peut être modifiée
- Pour que ces modifications soient enregistrées dans la BD, il est nécessaire de rattacher l'entité à un GE (pas nécessairement celui d'où elle a été détachée) par la méthode **`merge`**

`merge(A)`

- Renvoie une entité gérée A' ; plusieurs cas :
- Si A est une entité détachée, son état est copié dans une entité gérée A' qui a la même identité que A (si A' n'existe pas déjà, il est créé)
- Si A est nouvelle, une nouvelle entité gérée A' est créé et l'état de A est copié dans A' (un id automatique ne sera mis dans A' qu'au commit)
- Si A est déjà gérée, **`merge`** renvoie A ; en plus **`merge`** « cascade » pour tous les associations avec l'attribut **`CascadeType.MERGE`**

merge(*A*)

- Si A a été marquée « supprimée » par la méthode **remove**, une **IllegalArgumentException** est lancée

merge(*A*)

- Attention, la méthode **merge** n'attache pas A
- Elle retourne une entité gérée qui a la même identité dans la BD que l'entité passée en paramètre, mais ça n'est pas le même objet (sauf si A était déjà gérée)
- Après un **merge(A)**, l'application devra donc, sauf cas exceptionnel, ne plus utiliser l'objet A ; on aura souvent ce type de code :

```
a = em.merge(a);
```

l'objet anciennement pointé par **a** ne sera plus référencé

merge – une erreur à ne pas faire

- ~~**em.merge(a);**~~
  a.setMachin(truc);
  ~~**em.commit();**~~
- La modification de l'objet référencé par a ne sera pas enregistrée dans la base car cet objet n'est pas géré ; il faut utiliser l'objet renvoyé par **merge** :

```
a = em.merge(a);
a.setMachin(truc);
em.commit();
```


En dehors d'une transaction (1)

- Les méthodes suivantes (*read only*) peuvent être lancées en dehors d'une transaction : **find**, **getReference**, **refresh** et requêtes (**query**)
- Les méthodes **flush**, **lock** et modifications de masse (**executeUpdate**) ne peuvent être lancées en dehors d'une transaction

En dehors d'une transaction (2)

- Dans le cadre d'une application autonome (avec un GE géré par l'application, le cas étudié dans ce cours), les méthodes **persist**, **remove**, **merge** peuvent être exécutées en dehors d'une transaction ; les modifications sur les objets gérés seront enregistrées par un flush dès qu'une transaction sera active

En dehors d'une transaction (3)

- Pour les applications d'entreprise, les transactions peuvent être gérées par le container (pas étudié dans ce cours)
- Dans ce cas, lorsque le contexte de persistance est fermé à la fin de la transaction (contexte de persistance non « étendu », lié à une transaction), les méthodes **persist**, **remove**, **merge** ne peuvent être lancées que dans le cadre d'une transaction (elles ne peuvent avoir un sens que si un contexte de persistance existe)

En dehors d'une transaction (4)

- Certains SGBD sont mis en mode autocommit lorsque des modifications sont effectuées sur des entités gérées en dehors d'une transaction, ce qui peut poser de sérieux problèmes (en cas de rollback de la transaction par l'application, ces modifications ne seront pas invalidées)
- Il est donc conseillé de tester le comportement du SGBD

Transaction non terminée

- Il ne faut jamais oublier de terminer une transaction par **`commit()`** ou **`rollback()`** car, si on le fait pas, le résultat dépend du fournisseur de persistance et du SGBD

Identité des entités

Clé primaire

- Une entité doit avoir un attribut qui correspond à la clé primaire dans la table associée
- La valeur de cet attribut ne doit jamais être modifiée par l'application dès que l'entité correspond à une ligne de la base
- Cet attribut doit être défini dans l'entité racine d'une hiérarchie d'héritage (uniquement à cet endroit dans toute la hiérarchie d'héritage)
- Une entité peut avoir une clé primaire composite (pas recommandé)

Annotation

- L'attribut clé primaire est désigné par l'annotation **`@Id`**
- Pour une clé composite on utilise **`@EmbeddedId`** ou **`@IdClass`**

Type de la clé primaire

- Le type de la clé primaire (ou des champs d'une clé primaire composée) doit être un des types suivants :
  - type primitif Java (ne pas utiliser les types numériques non entiers)
  - classe qui enveloppe un type primitif
  - **`java.lang.String`**
  - **`java.util.Date`**
  - **`java.sql.Date`**

Génération automatique de clé

- Si la clé est de type numérique entier, l'annotation **@GeneratedValue** indique que la clé primaire sera générée automatiquement par le SGBD
- Cette annotation peut avoir un attribut **strategy** qui indique comment la clé sera générée (il prend ses valeurs dans l'énumération **GeneratorType**)
- Si la clé est générée automatiquement, le code Java ne doit pas y faire référence ; par exemple dans le constructeur de la classe

Types de génération

- **AUTO** : le type de génération est choisi par le fournisseur de persistance, selon le SGBD (séquence, table,…) ; valeur par défaut
- **SEQUENCE** : utilise une séquence est utilisée
- **IDENTITY** : une colonne de type IDENTITY est utilisée
- **TABLE** : une table qui contient la prochaine valeur de l'identificateur est utilisée
- On peut aussi préciser le nom de la séquence ou de la table avec l'attribut **generator**

Précisions sur la génération

- Les annotations **@SequenceGenerator** et **@TableGenerator** permettent de donner plus de précisions sur la séquence ou la table qui va permettre de générer la clé
- Elles définissent des générateurs d'identificateurs qui pourront être utilisés dans toute l'unité de persistance, et pas seulement dans la classe dans laquelle elles sont

Générateurs d'identificateurs

- Les annotations **@SequenceGenerator** et **@TableGenerator** peuvent annoter l'identificateur de l'entité ou l'entité elle-même qui les utilise, ou même une autre entité
- Si le générateur ne sert que dans une seule classe, il vaut mieux mettre l'annotation avec l'annotation **@Id** de la classe
- Sinon, une entrée dans un des fichiers XML de configuration de JPA, plutôt qu'une annotation, peut être un bon emplacement

Exemple de générateur

```
@SequenceGenerator(
name = "emp_seq",
sequence_name = "emp_seq",
allocation_size = 10,
initialValue = 600)
```


Exemple d'utilisation du générateur

```
@Id
@GeneratedValue(
  strategy = GenerationType.SEQUENCE,
  generator = "emp_seq")
private long id;
```

Identificateur du générateur

Valeur d'incrément d'une séquence

- Si une séquence utilisée par JPA est créée en dehors de JPA, il faut que la valeur de pré-allocation de JPA (égale à 50 par défaut) corresponde à la valeur d'incrémentation de la séquence ; si ça n'est pas le cas, on aura alors ce type d'annotation :

```
@SequenceGenerator(
name="seq3", sequenceName="seq3",
initialValue="125, allocationSize="20")
```


persist et id automatique

- La spécification n'impose rien sur le moment où la valeur de l'identificateur est mise dans l'objet géré
- La seule assurance est qu'après un flush dans la base de données (donc un commit) l'identificateur aura déjà reçu sa valeur
- Avec EclipseLink, Oracle et MySQL (et peut-être avec d'autres produits), la valeur de l'identificateur est mise dès l'appel de **persist**, sans attendre le commit, mais il est risqué pour la portabilité de l'utiliser

Clé composite

- Pas recommandé, mais une clé primaire peut être composée de plusieurs colonnes
- Peut arriver quand la BD existe déjà, en particulier quand la classe correspond à une table association (association M:N, cas étudié dans la section suivante sur les associations)
- 2 possibilités :
  - **@IdClass**
  - **@EmbeddedId** et **@Embeddable**

Classe pour la clé composite

- Dans les 2 cas, la clé primaire doit être représentée par une classe Java dont les attributs correspondent aux composants de la clé primaire
- La classe doit être **public**, posséder un constructeur sans paramètre, être sérialisable et redéfinir **equals** et **hashcode** (2 instances doivent être égales si elles ont les mêmes valeurs)

@EmbeddedId

- La classe clé primaire est annoté par **@Embeddable**
- La clé primaire est représentée dans l'entité par un seul attribut, du type de la classe « *embeddable* » et annoté par **@EmbeddedId**
- Le type d'accès (par champs ou propriétés) de la classe « *embeddable* » doit être le même que celui de l'entité dont la clé primaire est définie

Exemple avec **@EmbeddedId**

```
@Entity
public class Employe {
  @EmbeddedId
  private EmployePK employePK;
  ...
}
```

```
@Embeddable
public class EmployePK {
  private String nom;
  private Date dateNaissance;
  ...
}
```

@IdClass

- **@IdClass** correspond au cas où la classe entité comprend plusieurs attributs annotés par **@Id**
- La classe entité est annotée par **@IdClass** qui prend en paramètre le nom de la classe clé primaire
- La classe clé primaire n'est pas annotée ; ses attributs ont les mêmes noms et mêmes types que les attributs annotés **@Id** dans la classe entité

Exemple avec @IdClass

```
@Entity
@IdClass(EmployePK)
public class Employe {
  @Id private String nom;
  @Id private Date dateNaissance;
  ...
}
```

```
public class EmployePK {
  private String nom;
  private Date dateNaissance;
  ...
}
```


Quelle solution choisir ?

- **@IdClass** existe pour assurer une compatibilité avec la spécification EJB 2.0
- C'est une affaire de goût, mais **@EmbeddedId** est plus simple car on ne répète pas 2 fois les mêmes définitions pour les attributs des clés

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 l'attribut (ou les attributs) correspondant, pour que JPA puisse les gérer correctement ; par exemple :

```
@ManyToOne
private Departement departement
```


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>**

Types à utiliser

- Le plus souvent **`Collection`** sera utilisé
- **`Set`** peut être utile pour éliminer les doublons
- **`List`** peut être utilisé pour conserver un ordre mais nécessite quelques précautions (voir section « Compléments sur les associations »)
- Map permet d'avoir un accès rapide à une entité associée par l'intermédiaire d'une clé

Types à ne pas utiliser

- Le type déclaré ne doit pas être un type concret, tels que **`HashSet`** ou **`ArrayList`** pour les entités gérées (ce qui permet au fournisseur de persistance d'utiliser son propre type concret)
- Il faut évidemment initialiser l'attribut avec une classe concrète telle que **`HashSet`** ou **`ArrayList`** (utiliser des types génériques de préférence), mais les types de déclaration doivent être des interfaces

Association bidirectionnelle

- 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, appelé aussi bout « inverse ») 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
private Departement departement;
```

- Dans la classe **`Departement`** :

```
@OneToMany(mappedBy = "departement")
private Collection<Employe> employes;
```

Quel est le bout propriétaire ?

Valeurs par défaut

- Des valeurs par défaut sont supposées pour, par exemple, le nom de la colonne clé étrangère qui traduit l'association (DEPARTEMENT_ID pour l'exemple du transparent précédent)
- Si ces valeurs ne conviennent pas, ajouter des annotations dans le bout propriétaire
- Par exemple, pour indiquer le nom de la colonne clé étrangère :

```
@JoinColumn(name="DEPT_ID")
```

Annotation @**JoinColumn**

- Cette annotation donne le nom de la colonne clé étrangère qui représente l'association dans le modèle relationnel
- Elle doit être mise du côté propriétaire (celui qui contient la clé étrangère)
- Sans cette annotation, le nom est défini par défaut :
  <entité_but>_<clé_primaire_entité_but>
  par exemple, **departement_id**

Exemple

```
@ManyToOne
@JoinColumn(name="numero_departement")
private Departement departement;
```


Annotation @**JoinColumns**

- L'annotation **@JoinColumns** permet d'indiquer le nom des colonnes qui constituent la clé étrangère dans le cas où il y en a plusieurs (si la clé primaire référencée contient plusieurs colonnes)
- En ce cas, les annotations **@JoinColumn** doivent nécessairement comporter un attribut **referencedColumnName** pour indiquer quelle colonne est référencée (parmi les colonnes de la clé primaire référencée)

Exemple

```
@JoinColumns({
  @JoinColumn(name = "n1",
              referencedColumnName = "c1"),
  @JoinColumn(name = "n2",
              referencedColumnName = "c2")
})
```


Méthode de gestion de l'association

- Pour faciliter la gestion des 2 bouts d'une association bidirectionnelle, il est commode d'ajouter une méthode qui effectue tout le travail

Exemple

- 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) {
  Departement d = e.getDept();
  if (d != null)
    d.employes.remove(e);
  this.employes.add(e);
  employe.setDept(this);
}
```

Et si on oublie un des 2 bouts ?

- Si les entités sont utilisées dans la suite du programme, on va travailler avec des données incohérentes
- Par exemple, un employé est dans la liste des employés du département 10 alors que sa variable d'instance indique qu'il est dans le département 20
- De plus, les données peuvent ne pas être enregistrées dans la base au commit

Et si on oublie un des 2 bouts ?

- Si une association bidirectionnelle est modifiée par le bout propriétaire, la modification est enregistrée dans la base de données
- Si elle est modifiée par le bout non propriétaire, elle n'est pas enregistrée dans la base
- Exemple : si un employé est ajouté à la liste des employés d'un département sans que le département de l'employé soit modifié, la modification **n'est pas** enregistrée dans la base !

Et pour une association unidirectionnelle ?

- Si une entité e1 a une association unidirectionnelle vers une entité e2
- Si on fait un persist de e1, l'association avec e2 est bien enregistrée dans la base de données au commit

Association 1:1

- Annotation **@OneToOne**
- Représentée par une clé étrangère ajoutée dans la table qui correspond au côté propriétaire
- Exemple :

```
@OneToOne
private Adresse adresse;
```


Association 1:1 optionnelle

- L'annotation **@OneToOne** est optionnelle ; on peut ne pas la mettre si les valeurs par défaut conviennent
- En effet, tout attribut de type entité dans une entité est considéré comme représentant une association 1:1

Association 1:1 sur les clés

- 2 classes peuvent être reliées par leur identificateur : 2 entités sont associées si elles ont les mêmes clés
- L'annotation **@PrimaryKeyJoinColumn** permet d'indiquer qu'il ne faut pas représenter l'association par une clé étrangère à part
- Attention, c'est au développeur de s'assurer que les entités associées ont bien les mêmes clés
- Depuis JPA 2.0 il vaut mieux utiliser les identités dérivées (étudiées avec les associations M:N ; voir exemple du transparent suivant)

Exemple

- Voici une partie du code d'une entité **`Parking`** qui est en association 1-1 avec l'entité **`Employe`** ; la place de parking d'un employé a la même clé primaire que l'employé
- Avec **`PrimaryKeyJoinColumn`** :

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

- Avec entité dérivée (le mieux depuis JPA 2.0) :

```
@Id @OneToOne
private Employe employe;
```


Associations 1:N et N:1

- Annotations **`@OneToMany`** et **`@ManyToOne`**
- Représentée par une clé étrangère dans la table qui correspond au côté propriétaire (obligatoirement le côté « Many ») pour les associations bidirectionnelles
- Par défaut un attribut qui est une collection d'entités correspond à une association 1:N (**`@OneToMany`** est optionnel) unidirectionnelle (sinon il faudrait un attribut d'annotation **`mappedBy`**)

Exemple

```
class Employe {
  @ManyToOne
  private Departement departement
  ...
}
```

```
class Departement {
  ...
  @OneToMany(mappedBy = "departement")
  private List<Employe> employes();
  ...
}
```


Association 1:N unidirectionnelle

- Par défaut, une association unidirectionnelle 1:N est traduite pas une table association
- Elle n'est pas traduite par une clé étrangère dans la table du côté N comme c'est le cas pour une association bidirectionnelle (ou une association N:1 unidirectionnelle)
- L'annotation **`@JoinTable`** est optionnelle ; elle permet seulement de modifier les valeurs par défaut pour les noms de table association et de colonnes clés étrangères

1:N unidirectionnelle

```
public class Dept {
  ...
  @OneToMany
  @JoinTable(name="DEPT_EMP",
    joinColumns=@JoinColumn(name="DEPT_ID"),
    inverseJoinColumns=
      @JoinColumn(name="EMP_ID"))
  private Collection<Employe> employes;
  ...
}
```


Clé étrangère pour traduire une association 1:N unidirectionnelle

- JPA 2.0 permet de se passer d'une table association en ajoutant à côté de l'annotation **`@OneToMany`** (celle qui contient la collection) l'annotation **`@JoinColumn`** qui précise le nom de la colonne clé étrangère dans la table qui correspond au bout « N » (l'autre bout)

Table association pour une association N:1

- Pour se conformer à une structure préexistante on peut vouloir traduire une association N:1 (uni ou bidirectionnelle) par une table association
- Il suffit d'annoter l'association par **`@JoinTable`** (étudiée en détails dans la suite du cours) :

```
@ManyToOne
@JoinTable(name="EMP_DEPT")
private Departement departement;
```


Association M:N

- Traduite par 1 ou 2 collections (suivant directionnalité) dans les classes qui participent à l'association, annotées par **`@ManyToMany`**
- Le développeur peut choisir le côté propriétaire d'une association M:N bidirectionnelle ; l'autre côté comporte l'attribut **`mappedBy`**
- Représentée par une table association dans la base de données relationnelle

Exemple

- Les employés d'une entreprise peuvent participer à des projets
- Dans l'entité **`Employe`** :

```
@ManyToMany
private Collection<Projet> projets;
```

- Dans l'entité **`Projet`** :

```
@ManyToMany(mappedBy = "projets")
private Collection<Employe> employes;
```


Association M:N – valeurs par défaut

- Le nom de la table association est la concaténation des 2 tables (celle de l'entité propriétaire en 1er), séparées par « _ »
- Les noms des colonnes clés étrangères sont les concaténations du nom de l'attribut qui pointe vers « l'autre entité », de « _ » et de la colonne « Id » de la table référencée
- Si les valeurs par défaut ne conviennent pas, le côté propriétaire doit comporter une annotation **`@JoinTable`**

`@JoinTable`

- Donne des informations sur la table association qui va représenter l'association
- Attribut **`name`** donne le nom de la table
- Attribut **`joinColumns`** donne les noms des colonnes de la table qui référencent les clés primaires du côté propriétaire de l'association
- Attribut **`inverseJoinColumns`** donne les noms des colonnes de la table qui référencent les clés primaires du côté qui n'est pas propriétaire de l'association

Exemple (classe `Employe`)

```
@ManyToMany
@JoinTable(
  name = "EMP_PROJET",
  joinColumns = @JoinColumn(name = "matr"),
  inverseJoinColumns =
    @JoinColumn(name = "code_Projet")
)
private Collection<Projet> projets;
```

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

- L'association M:N est traduite par une classe association qui a 2 liens annotés **`@ManyToOne`** vers les classes qui participent à l'association
- Selon la directionnalité de l'association, ces classes peuvent aussi contenir chacune une collection de la classe association, annotée par **`@OneToMany`**

Classes **`Employe`** et **`Projet`**

```
@Entity public class Employe {
  @Id private int id;
  @OneToMany(mappedBy="employe")
  private Collection<Participation>
    participations;
  . . .
}
```

```
@Entity public class Projet {
  @Id private int id;
  @OneToMany(mappedBy="projet")
  private Collection<Participation>
    participations;
  . . .
}
```


2 cas pour la classe association

- 2 possibilités pour cette classe, suivant qu'elle contient ou non un attribut identificateur (**`@Id`**) unique non significatif ou que son identificateur est constitué des références vers les classes associées
- Le plus simple est de n'avoir qu'un seul attribut identificateur

Exemple avec 1 attribut 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`**

Code classe association (début)

```
@Entity
public class Participation {
  @Id @GeneratedValue
  private int id;

  @ManyToOne
  private Employe employe;
  @ManyToOne
  private Projet projet;

  private String fonction;
```

Code classe association (suite)

```
public Participation() { }

public Participation(Employe employe,
    Projet projet, String fonction) {
  this.employe = employe;
  this.projet = projet;
  this.fonction = fonction;
}
```


Code classe association (fin)

```
  public Employe getEmploye() {
    return employe;
  }
  public Projet getProjet() {
    return projet;
  }
}
```


Code classe association (complément)

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

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


Exemple avec 2 identificateurs

- Si la base de données existe déjà, il est 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
- En ce cas, la solution est plus complexe

Difficulté avec 2 identificateurs

- La difficulté vient de l'écriture de la classe **`Participation`**
- En effet, ça ne marcherait pas d'annoter comme composants de la clé, les 2 attributs **`employe`** et **`projet`**
- Leur type sont des entités et ne correspondent pas aux entiers qui constituent la clé primaire de la table **`PARTICIPATION`** de la BD

Identité dérivée

- JPA 2 prend en compte cette situation avec la notion d'identité dérivée (JPA 1 avait une solution plus complexe et moins portable)
- Il existe plusieurs variantes qui sont décrites en détails dans la spécification de JPA 2 (section 2.4.1.2), selon que les classes qui participent à l'association ont des clés simples ou composées de plusieurs attributs, et selon l'utilisation de **`@IdClass`** ou **`@EmbeddedId`**
- Les transparents suivants reprennent l'exemple précédent de classe association

Classe **Participation** (2 variantes)

- Les identificateurs sont dans une classe à part
- Les attributs de la classe des identificateurs ont le type des identificateurs des entités qui participent à l'association (par exemple **int** pour **Employe**)
- 2 variantes pour les représenter :
  - avec **@EmbeddedId**
  - avec **@IdClass**

Classe **Participation** (variante avec **@Embeddable**)

- La clé composite de **Participation** est représentée par une classe Embeddable **ParticipationId**
- L'annotation **@MapsId** se place sur un attribut de l'identificateur qui participe à une association ; elle indique la correspondance entre l'attribut de Participation et le champ de la classe **Embeddable ParticipationId**

Classe **Participation**

```
@Entity
public class Participation {
  @EmbeddedId
  private ParticipationId id;
  @ManyToOne @MapsId("employePK")
  private Employe employe;
  @ManyToOne @MapsId("projetPK")
  private Projet projet;
  private String fonction;
 ...
}
```

Nom de l'attribut de **ParticipationId**

Nom de l'attribut de **ParticipationId**

Classe Embeddable des identificateurs

```
@Embeddable
public class ParticipationId
    implements Serializable {
  private int employePK;
  private int projetPK;
  // Redéfinition de equals et hashcode
  ...
  // Reste éventuel de la classe
  ...
}
```

type de l'identificateur de l'entité **Projet**

Classe **Participation** (variante avec **@IdClass**)

- Le code suivant utilise une classe **IdClass** comme type de l'identificateur
- Les noms des champs identificateurs doivent être les mêmes dans la classe **ParticipationId** et dans la classe **Participation**
- Cependant les types sont différents : dans **ParticipationId** les types sont ceux de l'identificateur des entités qui participent à l'association

Classe **Participation**

```
@Entity
@IdClass(ParticipationId.class)
public class Participation {
  @Id
  @ManyToOne
  private Employe employe;
  @Id
  @ManyToOne
  private Projet projet;
  private String fonction;
 ...
}
```

Classe **IdClass** des identificateurs

```
public class ParticipationId
    implements Serializable {
  private int employe;
  private int projet;
  // Redéfinition de equals et hashcode
  ...
  // Reste de la classe
  ...
}
```

Même type que l'id de **Projet**

Même nom que dans Participation

Gérer les 2 bouts de l'association

- Il faut éviter la possibilité qu'un bout seulement d'une association soit gérée
- Pour cela on ajoute souvent une méthode dans une des classes participante, qui gère les 2 bouts de l'association
- Dans ce cas particulier, c'est le constructeur de **Participation** qui gère l'association

Participation (constructeurs)

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

public Participation(Employe employe,
                     Projet projet,
                     String fonction) {
  this.employe = employe;
  this.projet = projet;
  employe.getParticipations.add(this);
  projet.getParticipations.add(this);
  this.fonction = fonction;
}
```


Map pour association

- A la place d'une collection il est possible d'utiliser une *map* pour les associations OneToMany ou ManyToMany
- Les valeurs de la *map* correspondent aux entités associées
- Important : la classe de la clé doit avoir sa méthode **hashCode** compatible avec sa méthode **equals**

Map pour association

- L'association peut être annotée avec **@MapKey** qui indique que la clé de la *map* est l'identificateur de l'entité associée (de la classe « clé primaire » si l'entité a une clé primaire composite)
- La clé de la map peut aussi être un autre attribut de l'entité associée, de type basic, Embedded ou Entity ; en ce cas il faut indiquer le nom de l'attribut avec l'attribut name de l'annotation : **@MapKey(name="nom")**

Map pour association

- Attention l'annotation **@MapKey** est obligatoire même si la valeur de l'attribut name est la valeur par défaut
- En effet, si on ne met pas cette annotation, JPA considérera que la clé de la map est une valeur supplémentaire et il l'associera à une colonne autre que l'identificateur de l'entité associée, dans la base de données

Exemple

- Les employés d'un département peuvent être enregistrés dans une *map* dont les clés sont les noms des employés (on suppose que 2 employés n'ont pas le même nom)

```
public class Departement {
  ...
  @OneToMany(mappedBy = "departement")
  @MapKey(name = "nom")
  public Map<String,Employe> getEmployes(){
    ...
  }
```


Autre exemple

- La clé doit identifier la valeur dans le contexte de l'instance de la classe qui contient la map
- Les numéros de téléphone d'un employé sont rangés dans une map dont la clé est le type du numéro (mobile, domicile, fixe travail,...)
-

```
public class Employe {
  @OneToMany
  @MapKeyColumn(name="TYPE_NUMERO")
  private Map<String,Telephone> tels;
...
tels.put("domicile", "0492....");
```


Compléments sur les associations

- persistance des entités associées
- attribut « cascade »
- récupération des entités associées
- suppression des orphelins
- ordre des éléments d'une liste
- association bidirectionnelle avec un Embeddable

Persistance par transitivité

- Un service de persistance implémente la persistance par transitivité (*reachability*) si une instance devient automatiquement persistante lorsqu'elle est référencée par une instance déjà persistante
- C'est un comportement logique : un objet n'est pas vraiment persistant si une partie des valeurs de ses propriétés n'est pas persistante

Pas si simple

- Maintenir une cohérence automatique des valeurs persistantes n'est pas si simple
- Par exemple, si un objet devient non persistant, faut-il aussi rendre non persistants tous les objets qu'il a rendu persistants par transitivité ?
- De plus, le service de persistance doit alors examiner toutes les références des objets qui sont rendus persistants, et ce, de façon récursive, ce qui peut nuire aux performances

Le choix de JPA

- Par défaut, JPA n'effectue pas de persistance par transitivité automatique : rendre persistant un objet ne suffit pas à rendre automatiquement et immédiatement persistants tous les objets qu'il référence
- Comme la cohérence n'est pas gérée automatiquement, c'est le code de l'application qui se doit de conserver cette cohérence, au moins au moment du commit

Cohérence des données

- Si le code a mal géré cette cohérence, une exception est lancée
- Par exemple, si un département est rendu persistant alors que la collection des employés du département contient des employés non gérés par le contexte de persistance, une **`IllegalStateException`** va être lancée au moment du commit, et la transaction va sans doute être invalidée (rollback) dans un bloc **`catch(IllegalStateException`**)

Persistance automatique

- Afin de faciliter le maintien de cette cohérence, il est possible d'indiquer à JPA que les objets associés à un objet persistant doivent être automatiquement rendus persistants
- Pour cela il suffit d'ajouter un attribut « **`cascade`** » dans les informations de mapping de l'association

Attribut **`cascade`**

- Les annotations qui décrivent les associations entre objets peuvent avoir un attribut **`cascade`** pour indiquer que certaines opérations du GE doivent être appliquées aux objets associés
- Ces opérations sont **`PERSIST`**, **`REMOVE`**, **`REFRESH`**, **`DETACH`** et **`MERGE`** ; **`ALL`** correspond à toutes ces opérations
- Par défaut, aucune opération n'est appliquée transitivement

Exemples

-
```
@OneToMany(
  cascade = CascadeType.PERSIST)
  private Collection<Employe> employes;
```
-
```
@OneToMany(
    cascade = { CascadeType.PERSIST,
                CascadeType.MERGE },
    mappedBy = "client")
  private Collection<Facture> factures;
```


Récupération des entités associées

- Lorsqu'une entité est récupérée depuis la base de données par une requête (**`Query`**) ou par un **`find`**, est-ce que les entités associées doivent être elles aussi récupérées ?
- Si elles sont récupérées, est-ce que les entités associées à ces entités doivent elles aussi être récupérées ?
- On voit que le risque est de récupérer un très grand nombre d'entités qui ne seront pas utiles pour le traitement en cours

EAGER ou LAZY

- JPA laisse le choix de récupérer ou non immédiatement les entités associées, suivant les circonstances
- Il suffit de choisir le mode de récupération de l'association (**`LAZY`** ou **`EAGER`**)
- Une requête sera la même, quel que soit le mode de récupération
- Dans le mode **`LAZY`** les données associées ne sont récupérées que lorsque c'est vraiment nécessaire

Récupération retardée (LAZY)

- Dans le cas où une entité associée n'est pas récupérée immédiatement, JPA remplace l'entité par un « proxy », objet qui permettra de récupérer l'entité plus tard si besoin est
- Ce proxy contient la clé primaire qui correspond à l'entité non immédiatement récupérée
- Il est possible de lancer une requête avec une récupération immédiate, même si une association est en mode LAZY (join fetch de JPQL étudié plus loin)

Comportement par défaut de JPA

- Par défaut, JPA ne récupère immédiatement que les entités associées par des associations dont le but est « One » (une seule entité à l'autre bout) : OneToOne et ManyToOne (mode EAGER)
- Pour les associations dont le but est « Many » (une collection à l'autre bout), OneToMany et ManyToMany, par défaut, les entités associées ne sont pas récupérées immédiatement (mode LAZY)

Indiquer le type de récupération des entités associées

- L'attribut **`fetch`** d'une association permet d'indiquer une récupération immédiate des entités associées (**`FetchType.EAGER`**) ou une récupération retardée (**`FetchType.LAZY`**) si le comportement par défaut ne convient pas
- Exemple :

```
@OneToMany(mappedBy = "departement",
           fetch = FetchType.EAGER)
private Collection<Employe> employes;
```


Suppression des « orphelins »

- Certaines entités ne peuvent exister qu'en association avec d'autres entités
- Par exemple des lignes de commandes ne peuvent exister sans leur commande
- Pour ces cas, JPA 2 a ajouté la possibilité de suppression automatique des orphelins dès qu'ils sont enlevés de la collection qui les contient : dès qu'une ligne de commande est enlevée de la collection contenue dans la commande, elle est supprimée (dans la BD)

Attribut `orphanRemoval`

- Si une annotation **`@OneToMany`** a l'attribut **`orphanRemoval`** à **`true`**, les entités supprimées de la collection se verront appliquer l'opération **`remove`** par le gestionnaire d'entité (les données correspondantes seront supprimées de la BD)
- L'annotation **`@OneToOne`** peut aussi avoir cet attribut ; la suppression de l'orphelin a lieu lorsque la référence vers lui est mise à **`null`** dans l'entité qui lui est associée

Attribut `orphanRemoval`

- La suppression a lieu au moment du flush
- L'attribut **`orphanRemoval`** implique « **`cascade=REMOVE`** », ce qui correspond bien à la sémantique de l'attribut : si l'entité qui les « contient » est enlevée de la base de données, les « orphelins » le sont aussi

Exemple

```
@Entity
public class Facture {
  ...
  @OneToMany(mappedBy="facture",
    cascade=ALL, orphanRemoval=true)
  private Collection<LigneFacture> lignes;
  ...
}
```


Ordre dans les listes

- Par défaut, l'ordre d'une liste Java n'est pas nécessairement préservé dans la base de données
- De plus, l'ordre en mémoire de la liste doit être maintenu par le code (ça n'est pas automatique)

Annotations pour l'ordre dans les listes

- L'annotation **`@OrderBy`** impose un ordre pour récupérer les entités associées dans la liste lors d'une requête (mais cet ordre n'est pas dans la base de données ; c'est seulement au moment de la récupération que les données sont ordonnées)
- Depuis JPA 2.0, l'annotation **`@OrderColumn`** fait maintenir l'ordre de la liste dans la base de données (en ajoutant une colonne dans la base)

`@OrderBy`

- Il faut préciser un ou plusieurs attributs (séparés par une virgule) de type basic, qui déterminent l'ordre
- Chaque attribut peut être précisé par **ASC** ou **DESC** (ordre ascendant ou descendant) ; **ASC** par défaut
- Si aucun attribut n'est précisé, l'ordre est
  - celui de la clé primaire si la collection correspond à une association entre entités
  - celui de la valeur pour les collections d'éléments de type basic

Exemples

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

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


Cas des embeddables

- Si la colonne qui sert à ordonner appartient à un embeddable, la notation pointée peut être utilisée pour la désigner :
  `@OrderBy("adresse.codePostal")`

@OrderColumn

- Il peut arriver que l'ordre dans une liste ne corresponde pas à l'ordre d'une valeur dans les éléments de la liste
- Par exemple, une liste peut être triée par ordre d'ajout dans la liste
- Si on veut conserver l'ordre de la liste dans la base de données, il est nécessaire d'ajouter une colonne dans la table qui conserve la liste
- C'est le but de cette annotation introduite par JPA 2.0

@OrderColumn

- Cette annotation peut être ajoutée aux associations OneToMany, ManyToMany et aux collections d'éléments
- L'annotation doit être mise du côté de l'attribut qui référence la liste ordonnée ; attention, pour une association 1-N bidirectionnelle c'est donc le côté **non** propriétaire

Colonne pour ordonner

- La colonne est dans la table association (s'il y en a une) ou dans la table qui contient la clé étrangère ou dans la table des éléments pour une collection d'éléments
- Les valeurs de la colonne sont des nombres entiers ; ils commencent à 0 et sont contigus (ce qui peut nuire aux performances à cause des insertions-suppressions en milieu de liste)
- L'attribut **name** de l'annotation peut servir à donner le nom de la colonne (par défaut, elle se nomme « <nom attribut>_order »)

Colonne uniquement dans la BD

- La colonne ajoutée n'est pas visible dans le modèle objet (aucun attribut qui lui correspond dans les classes Java)
- Si l'ordre dépend des valeurs d'une colonne visible dans le modèle objet, il vaut mieux utiliser **@OrderBy** vu précédemment (pour des raisons de performance si les numéros d'ordre ne sont pas stables ; à tester dans chaque cas réel)

Utilisation dans les requêtes

- Cette colonne sert aussi à imposer un ordre lors de la récupération par une requête (pas besoin d'annotation **@OrderBy**)
- La fonction **INDEX** peut être utilisée dans une requête JPQL pour désigner la valeur de la colonne qui conserve l'ordre (voir section sur les requêtes dans la 2ème partie de ce support sur JPA)

Exemple

```
@Entity
public class Facture {
  ...
  @OneToMany(mappedBy = "facture")
  @OrderColumn(name = "numero_ligne")
  private List<LigneCommande> lignes;
  ...
}
```

Association bidirectionnelle avec les classes Embeddable

- Soit une association bidirectionnelle entre une entité E1 et une classe Embeddable Emb qui est insérée dans une entité E2
- Dans l'entité E1, il faut désigner l'entité E2 et dans la classe Embeddable il faut désigner l'entité E1

Exemple d'association dans Embeddable

```
@Entity class Employe {
   private Embedded Contact contact;
@Embeddable class Contact {
   @OneToMany(mappedBy="employe")
   private Set<Telephone> telephones;
@Entity class Telephone {
   @ManyToOne private Employe employe;
```

Pas **`Contact`** !

Héritage

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`**)

Annotation `@Inheritance`

- Le choix de la stratégie se fait avec l'annotation **`@Inheritance`**
- C'est la classe racine de la hiérarchie d'héritage qui doit être annotée
- Si on choisit la stratégie par défaut (**`SINGLE_TABLE`**), l'annotation est optionnelle : si la classe racine est une entité (annotée **`@Entity`**) et n'a pas d'annotation **`@Inheritance`**, la stratégie d'héritage est supposée être **`SINGLE_TABLE`**

Une table par hiérarchie

- Sans doute la stratégie la plus utilisée
- Valeur par défaut de la stratégie de traduction de l'héritage
- Performante et permet le polymorphisme
- Mais beaucoup de valeurs NULL dans les colonnes si la hiérarchie est complexe
- De plus, l'ajout d'un nouveau type d'entité après le démarrage de l'application oblige à modifier la définition de la table qui contient les données

Exemple

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

Dans la classe racine de la hiérarchie ; optionnel

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

« Employe » par défaut

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 (1)

- 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 (2)

- L'annotation **`@DicriminatorColumn`** 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 l'entité (le nom de la classe si pas d'attribut **`name`** pour **`@Entity`**)

Une table par classe (1)

- Toutes les classes, même les classes abstraites, sont représentées par une table
- Nécessite des jointures pour retrouver les propriétés d'une instance d'une classe
- Toutes les tables qui correspondent aux sous-classes ont une clé étrangère qui référence la clé primaire de la table qui correspond à la racine de la hiérarchie (voir cours sur le mapping objet-relationnel)

Une table par classe (2)

- La table qui correspond à la classe racine de la hiérarchie d'héritage doit comporter une colonne discriminatrice
- Cette colonne permet de simplifier certaines requêtes ; par exemple, pour retrouver les noms de tous les employés (classe **`Employe`** qui hérite de la classe **`Personne`** à la racine de la hiérarchie d'héritage)
- Voir la colonne discriminatrice de la stratégie « une seule table par hiérarchie » ; même valeur par défaut

Exemple

```
@Entity
@Inheritance(strategy=
    InheritanceType.JOINED)
@DiscriminatorValue("P")
public abstract class Personne {...}
```

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


`@PrimaryKeyJoinColumn`

- Par défaut, la colonne clé étrangère a le même nom que la colonne clé primaire référencée
- Si ça n'est pas le cas, il faut utiliser l'annotation **`@PrimaryKeyJoinColumn(`**
  `name=<`*`nom colonne clé étrangère`*`>)`
- Une annotation **`@PrimaryKeyJoinColumns`** peut être utilisée en cas de clé étrangère composée de plusieurs colonnes

Une table par classe concrète

- Stratégie seulement optionnelle
- Pas recommandé car le polymorphisme est plus complexe à obtenir (voir cours sur le mapping objet-relationnel)
- Performante car 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=EMPLOYE)
  public class Employe extends Personne {
    ...
  }
  ```

Entité abstraite

- Une classe abstraite peut être une entité (annotée par **`@Entity`**)
- Son état sera persistant et sera utilisé par les sous-classes entités
- Comme toute entité, elle pourra désigner le type retour d'une requête (*query*) pour une requête polymorphe

Compléments

- L'annotation **`@Entity`** ne s'hérite pas
- Les sous-classes entités d'une entité doivent être annotée par **`@Entity`**
- Une entité peut avoir une classe mère qui n'est pas une entité ; en ce cas, l'état de cette classe mère ne sera pas persistant

Classe mère persistante

- Une entité peut aussi avoir une classe mère dont l'état est persistant, sans que cette classe mère ne soit une entité
- En ce cas, la classe mère doit être annotée par **`@MappedSuperclass`**
- Aucune table ne correspondra à cette classe mère dans la base de données ; l'état de cette classe mère sera rendu persistant dans les tables associées à ses classes entités filles

Classe mère persistante

- Cette classe mère n'est pas une entité
- Donc elle ne pourra pas être renvoyée par une requête, ne pourra pas être passée en paramètre d'une méthode d'un **`EntityManager`** ou d'un **`Query`** et ne pourra être le but d'une association

Exemple

- Si toutes les entités ont des attributs pour enregistrer la date de la dernière modification et le nom de l'utilisateur qui a effectué cette modification, il peut être intéressant d'avoir une classe abstraite **`Base`**, mère de toutes les entités qui contient ces attributs
- Cette classe mère sera annotée avec **`@MappedSuperclass`**

Code de l'exemple

```
@MappedSuperclass
public abstract class Base {
  @Id @GeneratedValue
  private Long Id;
  @Version
  private Integer version;
  @ManyToOne
  private User user;
  @Temporal(value = TemporalType.TIMESTAMP)
  private Date dateModif;
  ...
}
```

Classe mère « non persistante »

- Une classe entité peut aussi hériter d'une classe mère dont l'état n'est pas persistant
- En ce cas, l'état hérité de la classe mère ne sera pas persistant
- La classe mère ne comportera aucune annotation particulière liée à la persistance, ni **`@Entity`**, ni **`@MappedSuperclass`**