SIR

Cours n°2 Architecture de Java EE
-
L'API de persistence : JPA

Olivier Barais
ISTIC
Université de Rennes 1
barais@irisa.fr

Cours bâti à partir de :

- Cours de Richard Grin (Université de Nice)
- Livre Hibernate 3.0 (Eyrolles)

JPA

- JPA est devenu un standard pour la persistance des objets Java
- Pour plus de précisions, lire la spécification à l'adresse

  http://jcp.org/aboutJava/communityprocess/pfd/jsr220/index.html

Avertissement

- JPA est le plus souvent utilisé dans le contexte d'un serveur d'application

- Ici, nous verrons JPA dans le cadre d'une application autonome, en dehors de tout serveur d'application

Plan

- Présentation générale
- Notion d’entités persistantes
- Compléments sur les entités : identité, associations, héritage
- Gestionnaire de persistance
- Langage d’interrogation

Présentation générale

Entités

- Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécification de JPA (héritage des EJBs)
- 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 (idem pour toutes les annotations)

Exemple d'entité

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
/**
 * Define an employee with an id and a name.
 */
@Entity
@Table(name = "EMPLOYEES")
public class Employee implements java.io.Serializable
    {
   /**
    * Id for serializable class.
    */
   private static final long serialVersionUID = -
      2366200007462547454L;
   /**
    * Id of this employee.
    */
   private int id;
   /**
    * Name of the employee.
    */
   private String name;
   /**
    * Gets the Id of the employee.
    * @return the id of the employee.
    */
   @Id
   public int getId() {
      return id;
   }
/** Sets id of the employee.
    * @param id the id's employee
    */
   public void setId(final int id) {
      this.id = id;
   }
   /**Sets the name.
    * @param name of employee.
    */
   public void setName(final String name) {
      this.name = name;
   }
   /** Gets the name of the employee.
    * @return name of the employee.
    */
   public String getName() {
      return name;
   }
   /*Computes a string representation of this
      employee.
    * @return string representation.
    */
   @Override
   public String toString() {
      StringBuilder sb = new StringBuilder();
sb.append("Employee[id=").append(id).append(",
      name=").append(getName()).append("]");
      return sb.toString();
   }}
```

Exemple d'entité – les champs

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
/**
 * Define an employee with an id and a
     name.
 */
@Entity
@Table(name = "EMPLOYEES")
public class Employee implements
     java.io.Serializable {
   /**
    * Id for serializable class.
    */
   private static final long
     serialVersionUID = -
     2366200007462547454L;

private String name;
```

```
/**
 * Sets the name.
 * @param name of employee.
 */
public void setName(final String
  name) {
    this.name = name;
}
/**
 * Gets the name of the employee.
 * @return name of the employee.
 */
public String getName() {
    return name;
}
/*Computes a string representation of this
  employee.
 * @return string representation.
 */
@Override
public String toString() {
   StringBuilder sb = new StringBuilder();

   sb.append("Employee[id=").append(id).appen
   d(", name=").append(getName()).append("]");
   return sb.toString();
}}
```

Exemple d'entité – l'identificateur

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
/**
 * Define an employee with an id and a name.
 */
@Entity
@Table(name = "EMPLOYEES")
public class Employee implements
      java.io.Serializable {
   /**
    * Id for serializable class.
    */
   private static final long serialVersionUID = -
      23662000074625474 54L;
   /**
    * Id of this employee.
    */
    private int id;

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

```
/** Sets id of the employee.
    * @param id the id's employee
    */
    public void setId(final int id) {
       this.id = id;
    }

/*Computes a string representation of this
      employee.
    * @return string representation.
    */
   @Override
   public String toString() {
      StringBuilder sb = new StringBuilder();

      sb.append("Employee[id=").append(id).appen
      d(", name=").append(getName()).append("]");
      return sb.toString();
   }}
```

Exemple d'entité – une association

```
@Entity
@NamedQuery(name="tousLesEmployees",query="SELE
CT p FROM Employee p")
public class Employee implements
      java.io.Serializable{
      …
      private long id;
      private String name;
      private Entreprise ent;
      …
      public Participant() {}

      public Participant(String name) {
            setName(name);
      }


      @Id @GeneratedValue
   (strategy=GenerationType.AUTO)
      public long getId() {
            return this.id;
      }
```

L'association inverse

dans la classe **Entreprise**

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

@ManyToOne
@JoinColumn(name="Ecole_id
   ");
public Entreprise getEnt()
   {
   return ent;
}

public void setEnt
   (Entreprise ent) {
   this.ent = ent;
   }
…
}
```

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 que les annotations

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…

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 l'objet sera enregistrée dans la base de données par le GE
- L'ensemble des entités gérées par un GE s'appelle un contexte de persistance

Exemple

```
{ EntityManagerFactory emf =
  Persistence.
createEntityManagerFactory("employe
  s");
EntityManager em =
emf.createEntityManager();
EntityTransaction tx =
  em.getTransaction();
tx.begin();
Employee e1 = new Employee();
em.persist(e1);
e1.setName("Dupont");
tx.commit();
```

Exemple (suite)

```
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();
```

Notion d’entités persistantes

Caractéristiques

- Seules les entités peuvent être
  - renvoyées par une requête (**Query**)
  - passées en paramètre d'une méthode d'un **EntityManager** ou d'un **Query**
  - la fin d'une association
  - référencées dans une requête JPQL
- Une classe entité peut utiliser d'autres classes pour conserver des états persistants (*MappedSuperclass* ou *Embedded* étudiées plus loin)

Conditions pour les classes entités

- Une classe entité doit avoir un constructeur sans paramètre, et doit être **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**
- Elle doit posséder un attribut qui représente la clé primaire dans la BD

Convention de nommage *JavaBean*

- Les entités doivent suivre la convention de nommage des propriétés du modèle JavaBean pour celles qui seront sauvegardées dans la base de données

- Si une variable d'instance s'appelle **var**, les modificateurs (*getters*) et accesseurs (*setters*) doivent s'appeler **setVar** et **getVar** (ou **isVar** si la variable a le type **boolean**)

- Les getters et setters doivent être **protected** ou **public**

2 types d’accès

- Le fournisseur de persistance accédera à la valeur d’une variable d’instance
  - soit en accédant directement à la variable d’instance (par introspection)
  - soit en passant par ses accesseurs (*getter* ou *setter*)
- Le type d’accès est déterminé par l’emplacement des annotations (au dessus de la variable d’instance ou au dessus du *getter*)

Vocabulaire JPA

- Un champ désigne une variable d'instance
- JPA parle de propriété lorsque l'accès se fait en passant par les accesseurs (getter ou setter)
- Lorsque le type d'accès est indifférent, JPA parle d'attribut

Choix du type d'accès

- Le choix doit être le même pour toutes les classes d'une hiérarchie d'héritage (interdit de mélanger les 2 façons)

- En programmation objet il est conseillé d'utiliser plutôt les accesseurs que les accès directs aux champs (meilleur contrôle des valeurs) ; c'est aussi le cas avec JPA

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 (null) 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

Cycle de vie d'une instance d'entité

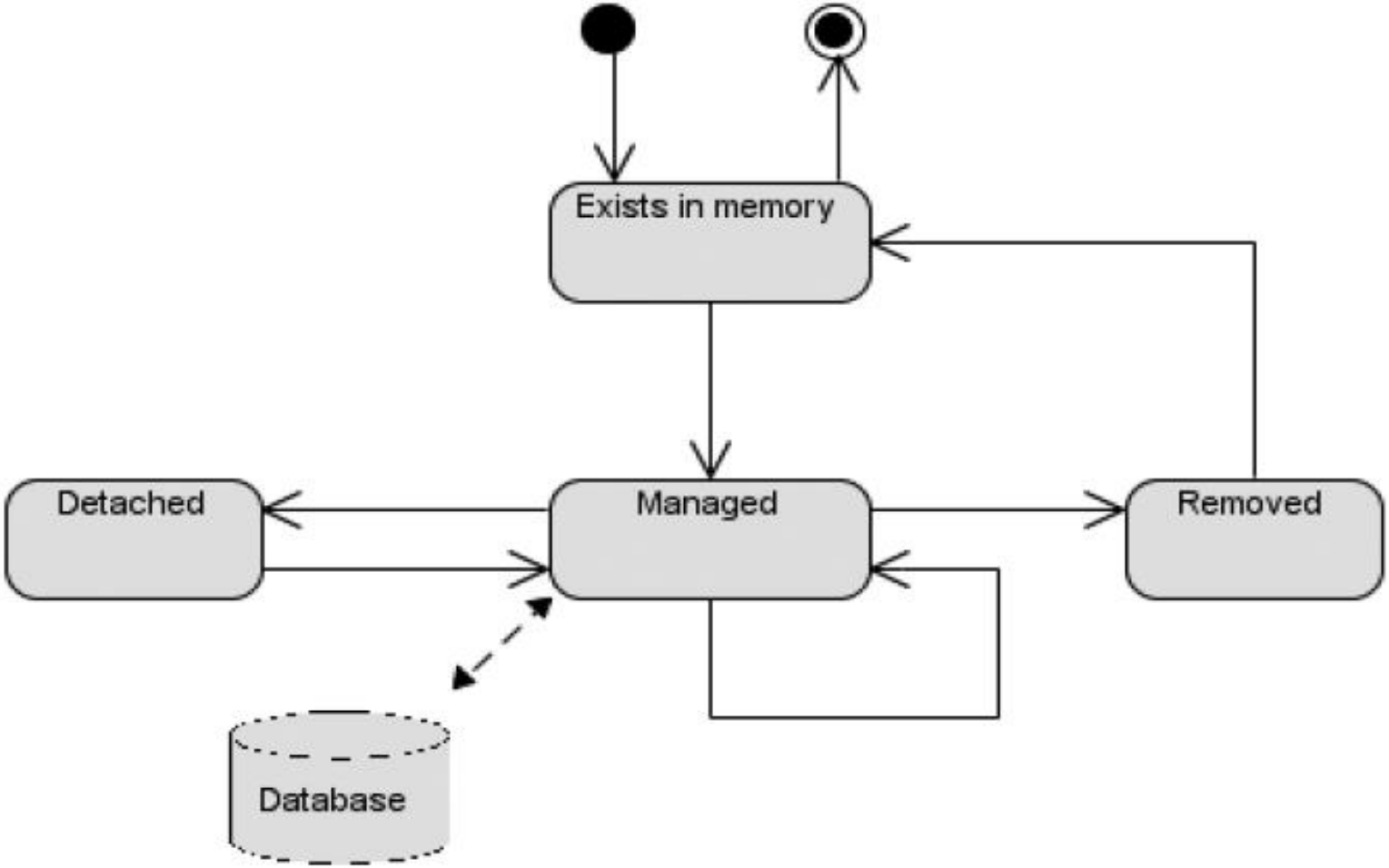
- L'instance peut être :
  - nouvelle (new) : elle est créée mais pas associée à un contexte de persistance
  - gérée par un gestionnaire de persistance ; elle a une identité dans la base de données (un objet peut devenir géré par la méthode **persist**, ou **merge** d'une entité détachée, ou si c'est une instance « récupérée » dans la base par une requête)

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 devenir détachée à la fin d'une transaction ou par un passage par valeur en paramètre d'une méthode distante)

- 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 instance d'entité



Configuration

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

Nom de table

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

```
/**
 * Define an employee with an id and a name.
 */
@Entity
@Table(name = "EMPLOYEES")
public class Employee implements java.io.Serializable {
   …
}
```

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="AUTRENOM",
updatable=false, length=80)
public String getName() { ... }
```

Classes *Embeddable*

- Les entités persistantes ne sont pas les seules classes persistantes
- Il existe aussi des classes « insérées » ou « incorporées » (*embedded*) dont les données n'ont pas d'identité dans la BD mais sont insérées dans une des tables associées à une entité persistante
- Par exemple, une classe **Adresse** dont les valeurs sont insérées dans la table **Employe**

Exemple

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

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

Restrictions

- La version actuelle de JPA a plusieurs restrictions (peut-être enlevées dans une prochaine version) :
  - une entité ne peut posséder une collection d'objets insérés
  - un objet inséré ne peut référencer un autre objet inséré
  - ni avoir une association avec une entité
  - un objet inséré ne peut être référencé par plusieurs entités différentes

Classes insérées partagées

- Si une classe insérée est utilisée par 2 classes entités, il est possible que les noms des colonnes soient différentes dans les tables associées aux 2 entités
- En ce cas, un champ annoté par **@Embedded** peut être complété par une annotation **@AttributeOverride**, ou plusieurs de ces annotations insérées dans une annotation **@AttributeOverrides**

Exemple

```
@Entity
public class Employe {
  @Embedded
  @AttributeOverrides({
  @AttributeOverride(
  name="numero",
  column=@column(name="num")),
  @AttributeOverride(...)
  })
  private Adresse adresse;
  …
```

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**

Annotation pour énumération

- Une annotation spéciale n'est pas nécessaire si un attribut est 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 la forme d'une **String** qui représente le nom de la valeur de l'énumération, il faut utiliser l'annotation **@Enumerated**

```
@Enumerated(EnumType.STRING)

private TypeEmploye typeEmploye;
```

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**

- Les 2 types temporels du paquetage **java.util** (**Date** et **Calendar**) nécessitent l'annotation **@Temporal** pour indiquer s'il s'agit d'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)

```
@Temporal(TemporalType.DATE)

private Calendar dateEmb;
```

Tables multiples

- Il est possible de sauvegarder une entité sur plusieurs tables
- Voir **@SecondaryTable** dans la spécification JPA

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
- 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
  - classe qui enveloppe un type primitif
  - **java.lang.String**
  - **java.util.Date**
  - **java.sql.Date**

- Ne pas utiliser les types numériques non entiers

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

Types de génération

- **AUTO** : le SGBD choisit (séquence, table,…) ; c'est la valeur par défaut
- **SEQUENCE** : il utilise une séquence
- **IDENTITY** : il utilise une colonne de type IDENTITY
- **TABLE** : il utilise une table qui contient la prochaine valeur de l'identificateur
- 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

- Par exemple **@SequenceGenerator** permet de préciser la valeur initiale ou le nombre de clés récupérées à chaque appel de la séquence

- Voir la spécification de JPA pour plus de précisions

Exemple

```
@Id
@GeneratedValue(
  strategy = SEQUENCE,
  generator = "EMP_SEQ")
public long getId() {
  return id;
}
```

Clé composite

- Pas recommandé, mais une clé primaire peut être composée de plusieurs colonnes
- Peut arriver quand la BD existe déjà ou quand la classe correspond à une table association (association M:N)
- 2 possibilités :
  - **@IdClass**
  - **@EmbeddedId** et **@Embeddable**

@EmbeddedId

- **@EmbeddedId** correspond au cas où la classe entité comprend un seul attribut annoté **@EmbeddedId**

- La classe clé primaire est annotée par **@Embeddable**

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 Date dateNaissance;
  ...
}
```

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

Rq:

- **@IdClass** existe pour assurer une compatibilité avec la spécification EJB 2.0
- Il vaut mieux utiliser **@EmbeddedId**

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

Exemple

```
@ManyToOne
public Departement getDepartement() {
  …
}
```

Associations bidirectionnelles

- Le développeur est responsable de la gestion correcte des 2 bouts de l'association

- 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'association correspondante dans le bout propriétaire

Exemple

- Dans la classe **Employe** :

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

- Dans la classe **Departement** :

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

Persistance des objets associés

- Lorsqu'un objet o est rendu persistant (méthode **persist**), les objets référencés par o devraient être rendus eux-aussi persistants
- Ce concept s'appelle la « persistance par transitivité »
- Ca serait 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 en cas de persistance par transitivité

- Par exemple, que se passe-t-il si un objet supprimé est référencé par un autre objet ?

Le choix de JPA

- Par défaut, JPA n'effectue pas de persistance par transitivité

- Ce comportement permet plus de souplesse, et un meilleur contrôle de l'application sur ce qui est rendu persistant

- Pour que les objets associés à un objet persistant deviennent automatiquement persistants, il faut l'indiquer dans les informations de mapping de l'association (attribut **cascade**)

Attribut **cascade**

- Les annotations qui décrivent les associations entre objets peuvent avoir un attribut **cascade** pour indiquer que certaines opérations de GE doivent être appliquées aux objets associés

- Ces opérations sont **persist**, **remove**, **refresh** et **merge**

- Par défaut, aucune opération n'est appliquée transitivement

Exemples

```
@OneToMany(
  cascade=CascadeType.PERSIST)
@OneToMany(
  cascade={CascadeType.PERSIST,
  CascadeType.MERGE})
```

Association 1:1

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

```
@OneToOne
public Adresse getAdresse() {
…
}
```

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 »)

Exemple

```
class Employe {
...
   @ManyToOne
   public Departement getDepartement() {
      …
   }
}
```

```
class Departement {
   …
   @OneToMany(MappedBy="departement")
   public List<Employe> getEmployes() {
      …
   }
}
```

Association M:N (1)

- Annotation **@ManyToMany**
- Représentée par une table association
- 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 (2)

- 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**

@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

```
@ManyToMany
@JoinTable(
      name="EMP_PROJET"
      joinColumns=@JoinColumn(name="matr")
      inverseJoinColumns=
      @JoinColumn(name="codeProjet")
  )
public Collection<Projet> getProjets() {
```

(classe **Employe**)

```
@ManyToMany(mappedBy="projets")
public Collection<Employe> getEmps() {
```

(classe **Projet**)

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

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***)

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)
public Collection<Employe>
getEmployes()
```

Mode de récupération des attributs

- Le mode de récupération par défaut est le mode EAGER pour les attributs (ils sont chargés en même temps que l'entité)

- Si un attribut est d'un type de grande dimension (LOB), il peut aussi être marqué par **@Basic(fetch=FetchType.LAZY)** (à utiliser avec parcimonie)

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

Héritage

Stratégies

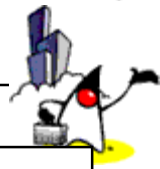
- A ce jour, les implémentations de JPA doivent obligatoirement offrir 2 stratégies pour la traduction de l'héritage :
  - une seule table pour une hiérarchie d'héritage (**SINGLE_TABLE**)
  - une table par classe ; les tables sont jointes pour reconstituer les données (**JOINED**)

- La stratégie « une table distincte par classe concrète » est seulement optionnelle (**TABLE_PER_CLASS**)

Une table par hiérarchie

- 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 (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

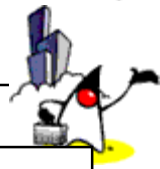
- Chaque classe est différenciée par une valeur de la colonne discriminatrice
- Cette valeur est passée en paramètre de l'annotation **@DiscriminatorValue**
- Par défaut cette valeur est le nom de la classe

Une table par classe

- Nécessite des jointures pour retrouver les propriétés d'une instance d'une classe

- Une colonne discriminatrice permet de simplifier certaines requêtes

Exemple



A mettre dans la classe racine de la hiérarchie

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

« Employe » par défaut

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

Une table par classe concrète

- Stratégie seulement optionnelle
- Pas recommandé car le polymorphisme est plus complexe à obtenir (voir premier cours)
- 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

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'entité (**persist** ou **merge**)
- Cette conception a été voulue par les concepteurs de l'API par souci de flexibilité et pour permettre un contrôle fin de l'application sur la persistance des entités

Contexte de persistance

- Un contexte de persistance est un ensemble d'entités qui vérifie la propriété suivante : il ne peut exister 2 entités différentes qui représentent des données identiques dans la base
- Un contexte de persistance est géré par un gestionnaire d'entités qui veille à ce que cette propriété soit respectée
- Un contexte de persistance ne peut appartenir qu'à une seule unité de persistance
- Une unité de persistance peut contenir plusieurs contextes de persistance

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 de la validation (*commit*) de la transaction

Classe **EntityManager -** GE

- En dehors d'un serveur d'application, 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 supprimé avec la méthode **close()** de la classe **EntityManager**

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`**`()`
- `void` **`clear`**`()`
- `boolean` **`contains`**`(Object` *`entité`*`)`
- `Query` **`createQuery`**`(String` *`requête`*`)`
- `Query` **`createNamedQuery`**`(String` *`nom`*`)`

Méthodes de **EntityManager**

- `Query` **`createNativeQuery`**`(String` *`requête`*`)`
- `Query` **`createNativeQuery`**`(String` *`requête`*`, Class` *`classeRésultat`*`)`
- `void` **`joinTransaction`**`()`
- `void` **`close`**`()`
- `boolean` **`isOpen`**`()`
- `EntityTransaction` **`getTransaction`**`()`

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

- 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

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 instruction ne sera nécessaire pour faire enregistrer dans la base de données les modifications effectuées sur l'entité par l'application ; en effet le GE conserve toutes les informations nécessaires sur les entités qu'il gère

remove

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

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

find

- La recherche est polymorphe : l'entité récupérée peut être de la classe passée en paramètre ou d'une sous-classe

- 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

En dehors d'une transaction

- Les méthodes suivantes (*read only*) peuvent être lancées en dehors d'une transaction :
- **find**, **getReference**, **refresh** et requêtes (**query**)
- Les méthodes **persist**, **remove**, **merge** peuvent être exécutées en dehors d'une transaction mais elles seront enregistrées pour un flush dès qu'une transaction est active
- Les méthodes **flush**, **lock** et modifications de masse (**executeUpdate**) ne peuvent être lancées en dehors d'une transaction

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
- Elle peut ainsi être transférée 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 par la méthode **merge()**

Requêtes - JPQL

Chercher par identité

- **find** et **getReference** (de **EntityManager**) permettent de retrouver une entité en donnant son identité dans la BD
- Elles prennent 2 paramètres de type
  - **Class<T>** pour indiquer le type de l'entité recherchée (le résultat renvoyé sera de cette classe ou d'une sous-classe)
  - **Object** pour indiquer la clé primaire

getReference

- **getReference** renvoie une référence vers une entité, sans que cette entité ne soit nécessairement initialisée

- Le plus souvent il vaut mieux utiliser **find**

Étapes pour récupérer des données

- Souvent il est nécessaire de rechercher des données sur des critères plus complexes que la simple identité

- Les étapes sont alors les suivantes :
  1. Décrire ce qui est recherché (langage JPQL)
  2. Créer une instance de type **Query**
  3. Initialiser la requête (paramètres, pagination)
  4. Lancer l'exécution de la requête

Langage JPQL

- Le langage JPQL (*Java Persistence Query Language*) permet de décrire ce que l'application recherche

- Il ressemble beaucoup à SQL

Type du résultat

- Un select peut renvoyer
  - une (ou plusieurs) expression « entité », par exemple un employé
  - une (ou plusieurs) expression « valeur », par exemple le nom et le salaire d'un employé

Exemples de requêtes

```
select e from Employe as e

select e.nom, e.salaire from Employe e

select e from Employe e where
  e.departement.nom = 'Direction'

select d.nom, avg(e.salaire) from
  Departement d join d.employes e
  group by d.nom having count(d.nom) >
  5
```

Nombre d'éléments du résultat

- Pour le cas où une seule valeur ou entité est renvoyée, le plus simple est d'utiliser la méthode **getSingleResult()** ; elle renvoie un **Object**

Nombre d'éléments du résultat

- Pour le cas où une plusieurs valeurs ou entités peuvent être renvoyées, il faut utiliser la méthode **getResultList()**
- Elle renvoie une liste « raw » (pas générique) des résultats, instance de **java.util.List**
- Un message d'avertissement sera affiché durant la compilation si le résultat est rangé dans une liste générique (**Liste<Employe>** par exemple)

Type du résultat

- Le type des éléments de la liste est **Object** si la clause select ne comporte qu'une seule expression, ou **Object[]** si elle comporte plusieurs expressions
- Exemple :

```
texte = "select e.nom, e.salaire from
  Employe as e";
query = em.createQuery(texte);
List<Object[]> liste =
  (List<Object[]>)query.getResultList();
for (Object[] info : liste) {
   System.out.println(info[0] + " gagne
     "   + info[1]);
}
```

Interface **Query**

- Représente une requête

- Une instance de **Query** est obtenue par les méthodes **createQuery**, **createNativeQuery** ou **createNamedQuery** de l'interface **EntityManager**

Méthodes de **Query** (1)

```
List getResultList()
Object getSingleResult()
int executeUpdate()
Query setMaxResults(int nbResultats)
Query setFirstResult(int
  positionDepart)
Query setFlushMode(FlushModeType
  modeFlush)
```

Méthodes de **Query** (2)

```
Query setParameter(String nom,
  Object valeur)

Query setParameter(String nom, Date
  valeur, TemporalType typeTemporel)

Query setParameter(String nom,
  Calendar valeur, TemporalType
  typeTemporel)
```

Types temporels

- On a vu que les 2 types java temporels du paquetage **java.util** (**Date** et **Calendar**) nécessitent une annotation **@Temporal**

- Ils nécessitent aussi un paramètre supplémentaire pour la méthode **setParameter**

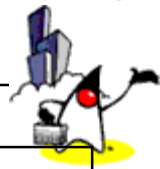
Exemple

```
@Temporal(TemporalType.DATE)
private Calendar dateEmb;
em.createQuery(
"select e from employe e"
+ " where e.dateEmb between ?1 and ?2")
.setParameter(1, debut,
  TemporalType.DATE)
.setParameter(2, fin,
  TemporalType.DATE)
.getResultList();
```

Paramètres des requêtes

- Un paramètre peut être désigné par son numéro (***?n***) ou par son nom (***:nom***)
- Les valeurs des paramètres sont données par les méthodes **setParameter**
- Les paramètres sont numérotés à partir de 1
- Un paramètre peut être utilisé plus d'une fois dans une requête
- L'usage des paramètres nommés est recommandé (plus lisible)

Exemples



```
Query query = em.createQuery( "select
  e from Employe as e where e.nom =
  ?1");
query.setParameter(1, "Dupond");
```

```
Query query = em.createQuery( "select
  e from Employe as e where e.nom =
  :nom");
query.setParameter("nom", "Dupond");
```

Plan

- Présentation générale
- Notion d'entités persistantes
- Compléments sur les entités : identité, associations, héritage
- Gestionnaire de persistance
- Langage d'interrogation
  - JPQL
- Opération de modification en volume
- Optimisation
- Exceptions
- Callback
- Configuration de l'unité de persistance
- Outils
- Transaction
- Entités détachées
- Concurrence

JPQL in a nutshell

Clauses d'un select

- **select** : type des objets ou valeurs renvoyées
- **from** : où les données sont récupérées
- **where** : sélectionne les données
- **group by** : regroupe des données
- **having** : sélectionne les groupes (ne peut exister sans clause group by)
- **order by** : ordonne les données

- Les mots-clés **select**, **from**, **distinct**, **join**,… sont insensibles à la casse

Polymorphisme dans les requêtes

- Toutes les requêtes sont polymorphes : un nom de classe dans la clause **from** désigne cette classe et toutes les sous-classes
- Exemple :

```
select count(a) from Article as a
```

- compte le nombre d'instances de la classe **Article** et de tous les sous-classes de **Article**

distinct

- Dans une clause select, indique que les valeurs dupliquées sont éliminées (la requête ne garde qu'une seule des valeurs égales)

- Exemple :

```
select distinct e.departement
  from Employe e
```

Clauses **where** et **having**

- Ces clauses peuvent comporter les mots-clés suivants :
  - **[NOT] LIKE**, **[NOT] BETWEEN**, **[NOT] IN**
  - **AND**, **OR**, **NOT**
  - **[NOT] EXISTS**
  - **ALL**, **SOME/ANY**
  - **IS [NOT] EMPTY**, **[NOT] MEMBER OF**

Exemple

```
select d.nom, avg(e.salaire)
from Departement d join
  d.employes e
group by d.nom
having count(d.nom) > 3
```

having

- Restriction : la condition doit porter sur l'expression de regroupement ou sur une fonction de regroupement portant sur l'expression de regroupement
- Par exemple, la requête suivante provoque une exception :

```
select d.nom, avg(e.salaire)
from Departement d join d.employes e
group by d.nom
having avg(e.salaire) > 1000
```

Sous-requête (1)

- Les clauses **where** et **having** peuvent contenir des sous-requêtes
- Exemple :

```
select e from Employe e
where e.salaire >= (
select e2.salaire from Employe e2
where e2.departement = 10)
```

Sous-requête (2)

- {**ALL** | **ANY** | **SOME**} (*sous-requête*) fonctionne comme dans SQL

- Exemple :

```
select e from Employe e
where e.salaire >= ALL (
select e2.salaire from Employe e2
where e2.departement =
e.departement)
```

Sous-requête synchronisée

- Une sous-requête peut être synchronisée avec une requête englobante

- Exemple :

```
select e from Employe e
where e.salaire >= ALL (
select e2.salaire from Employe e2
where e2.departement =
e.departement)
```

Sous-requête - **exists**

- **[not]exists** fonctionne comme avec SQL
- Exemple :

```
select e from Employe e
where exists (
select ee from Employe ee
where ee = e.epouse)
```

Jointure

- Une jointure permet de combiner plusieurs entités dans un select
- Il existe plusieurs types de jointures :
  - jointure interne (jointure standard **join**)
  - jointure externe (**outer join**)
  - jointure avec récupération de données en mémoire (**join fetch**)
- Il est aussi possible d'utiliser plusieurs entités dans une requête grâce à la navigation

Fonctions

- Fonctions de chaînes de caractères : *concat*, *substring*, *trim*, *lower*, *upper*, *length*, *locate* (localiser une sous-chaîne dans une autre)
- Fonctions arithmétiques : *abs*, *sqrt*, *mod*, *size* (d'une collection)
- Fonctions de date : *current_date*, *current_time*, *current_timestamp*
- Fonctions de regroupement : *count*, *max*, *min*, *avg*

Travail avec les collections

- Une expression chemin d'un select peut désigner une collection
- Exemples :

```
departement.employes
facture.lignes
```

- La fonction **size** donne la taille de la collection
- La condition « **is [not] empty** » est vraie si la collection est [n'est pas] vide
- La condition « **[not] member of** » indique si une entité appartient à une collection

Enchaînement des méthodes

- Les méthodes **setParameter**, **setMaxResults** renvoient le **Query** modifié
- On peut donc les enchaîner
- Exemple :

```
em.createQuery(texteQuery)
    .setParameter(nomParam,valeurParam)
    .setMaxResults(30)
    .getResultList();
```

Pagination du résultat

- **Query setMaxResults(int n)** : indique le nombre maximum de résultats à retrouver

- **Query setFirstResult(int n)** : indique la position du 1er résultat à retrouver (numéroté à partir de 0)

Opération de modification en volume

Utilité

- Pour les performances il est parfois mauvais de charger toutes les données à modifier dans des instances d'entités
- En ce cas, EJBQL permet de lancer un ordre *update* ou *delete* qui modifie les données de la base directement, sans créer les entités correspondantes

Cas d'utilisation

- Si on veut augmenter de 5% les 1000 employés de l'entreprise il serait mauvais de récupérer dans 1000 instances les données de chaque employés, de modifier le salaire de chacun, puis de sauvegarder les données
- Un simple ordre SQL sera énormément plus performant

```
update Employe as e
set e.salaire = salaire * 1.05
```

Exemple

```
em.getTransaction().begin();
String ordre = "update Employe e "
  + " set e.salaire = e.salaire *
  1.05";
int nbEntiteModifiees =
  em.executeUpdate(ordre);
em.getTransaction().commit();
```

Remarques

- Les modifications doivent être faites dans une transaction
- Exécuter ces opérations au début d'une transaction avant la récupération dans la base d'entités touchées par l'opération
- En effet, les entités en mémoire ne sont pas modifiées par l'opération et elles ne correspondront alors donc plus aux nouvelles valeurs modifiées dans la base de données

Exceptions

Exceptions non contrôlées

- JPA n'utilise que des exceptions non contrôlées (descendantes de **`RuntimeException`**)
- JPA utilise les 2 exceptions **`IllegalArgumentException`** et **`IllegalStateException`** du paquetage **`java.lang`**
- Sinon, toutes les autres exceptions sont dans le paquetage **`javax.persistence`** et héritent de **`PersistenceException`**

Types d'exception

- **EntityNotFoundException**
- **EntityExistsException**
- **NonUniqueResultException**
- **NoResultException**
- **TransactionRequiredException**
- **RollbackException**
- **OptimisticLockException**

Callback

Méthodes « callback »

- Des méthodes peuvent être annotées pour indiquer qu'elles seront appelées par le fournisseur de persistance quand une entité passera dans une nouvelle étape de son cycle de vie
- Ces méthodes peuvent appartenir à une classe entité (**entity** ou classe mère « **mapped** ») ou à une classe « écouteur »

Types de méthode

- **@PrePersist** : quand persist (ou merge) s'est terminé avec succès
- **@PostPersist** : après l'insertion dans la BD
- **@PreRemove** : quand remove est appelé
- **@PostRemove** : après suppression dans la BD
- **@PreUpdate** : avant modification dans la BD
- **@PostUpdate** : après modification dans la BD
- **@PostLoad** : après la lecture des données de la BD pour construire une entité

Concurrence

Concurrence « BD »

- Le fournisseur de persistance peut apporter automatiquement une aide pour éviter les problèmes d’accès concurrents aux données de la BD, pour les entités gérées par un GE

Exemple de problème de concurrence « BD »

- Une entité est récupérée depuis la BD (par un **find** par exemple)
- L'entité est ensuite modifiée par l'application puis la transaction est validée
- Si une autre transaction a modifié entretemps les données de la BD correspondant à l'entité, la transaction doit être invalidée

Entité « versionnée »

- C’est une entité qui possède un attribut qui possède l’annotation **@version** (ou le tag correspondant dans les fichiers XML)

@Version

- Annote un attribut dont la valeur représentera un numéro de version (incrémenté à chaque modification) pour l'entité et sera utilisée pour savoir si l'état d'une entité a été modifiée entre 2 moments différents (stratégie optimiste)
- L'attribut doit être de type **int**, **Integer**, **short**, **Short**, **long**, **Long** ou **java.sql.TimeStamp** (si possible, éviter ce dernier type)
- L'application ne doit jamais modifier un tel attribut (modifié par JPA)
- Exemple

```
@Version
private int version;
```

lock(*A*, *mode*)

- Sert à protéger une entité contre les accès concurrents pour les cas où la protection offerte par défaut par le fournisseur ne suffit pas

- Cette entité doit être versionnée (sinon, le traitement n'est pas portable)

Modes de blocage

- L'énumération **LockModeType** (paquetage **javax.persistence**) définit 2 modes
- **READ** : les autres transactions peuvent lire l'entité mais ne peuvent pas la modifier
- **WRITE** : comme **READ**, mais en plus l'attribut de version est incrémenté, même si l'entité n'a pas été modifiée

Prévenir les injections SQL

Injection SQL

- Due aux requêtes dynamiques

```
// UNSAFE !!! DON'T DO THIS !!!
String sql = "select "
  + "customer_id,acc_number,branch_id,balance "
  + "from Accounts where customer_id = '"
  + customerId
  + "'";
Connection c = dataSource.getConnection();
ResultSet rs = c.createStatement().executeQuery(sql);
// ...
```

- customerId = **abc' or '1'='1**

```
select customer_id, acc_number,branch_id, balance
  from Accounts where customerId = 'abc' or '1' = '1'
```

Injection SQL JPA (1)

```
public List<AccountDTO> unsafeJpaFindAccountsByCustomerId(String customerId) {
    String jql = "from Account where customerId = '" + customerId + "'";
    TypedQuery<Account> q = em.createQuery(jql, Account.class);
    return q.getResultList()
      .stream()
      .map(this::toAccountDTO)
      .collect(Collectors.toList());
}
```

- Nous utilisons des données non validées pour créer une requête JPA

Injection SQL JPA (2)

- Utiliser une requête JPA paramétrée

```
String jql = "from Account where customerId = :customerId";
TypedQuery<Account> q = em.createQuery(jql, Account.class)
  .setParameter("customerId", customerId);
// Execute query and return mapped results (omitted)
```

- Une erreur/exception sera levée avec un customerId = **abc' or '1'='1**

Injection SQL JPA (3)

```
String sql = "select "
  + "customer_id,acc_number,branch_id,balance from Accounts"
  + "where customer_id = ? ";
if (VALID_COLUMNS_FOR_ORDER_BY.contains(orderBy)) {
    sql = sql + " order by " + orderBy;
} else {
    throw new IllegalArgumentException("Nice try!");
}
```

Pour aller plus loin:

https://www.google.com/search?client=firefox-b-d&q=**jpa+sql+injection**

Outils

- JPA dans eclipse
  - Le projet dali
    - https://www.eclipse.org/webtools/dali/

Outils

Merci pour votre attention