

JPA (*Java Persistence API*)

Université de Nice - Sophia Antipolis
Version 1.6 – 20/11/07
Richard Grin

Plan de cette partie

- ❑ Présentation de JPA
- ❑ Entités persistantes
- ❑ Gestionnaire de persistance
- ❑ Compléments sur les entités : identité, associations, héritage
- ❑ Langage d'interrogation
- ❑ Modifications en volume
- ❑ Exceptions

R. Grin

JPA

page 2

Plan de cette partie

- ❑ Transaction
- ❑ Concurrency
- ❑ Entités détachées
- ❑ Fichiers de configuration XML
- ❑ JPA et DAOs
- ❑ Optimisation
- ❑ Callbacks

R. Grin

JPA

page 3

Présentation de JPA

R. Grin

JPA

page 4

EJB 3.0

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

R. Grin

JPA

page 5

JPA

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

R. Grin

JPA

page 6

Solution ORM

- ❑ Transparente : les classes des entités persistantes sont indifférentes au mécanisme de persistance
- ❑ Automatique : des appels simples de haut niveau pour gérer la persistance, tels que `persist(objet)` pour rendre un objet persistant ; pas d'appel de bas niveau comme avec JDBC

JPA

- ❑ JPA est 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'applications
- ❑ 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 un autre support complémentaire est en préparation

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
- ❑ GlassFish utilise *TopLink essentials* comme fournisseur de persistance pour JPA (produit Oracle)
- ❑ D'autres implémentations : TopLink, Hibernate Entity Manager, 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 (idem pour toutes les annotations)

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 – les champs

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

R. Grin

JPA

page 13

Les constructeurs

```
/**
 * Constructeur sans paramètre
 * obligatoire.
 */
public Departement() { }
public Departement(String nom,
                    String lieu) {
    this.nom = nom;
    this.lieu = lieu;
}
```

R. Grin

JPA

page 14

Exemple d'entité – l'identificateur

```
@Id
@GeneratedValue
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
```

R. Grin

JPA

page 15

Exemple d'entité – une propriété

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

R. Grin

JPA

page 16

Exemple d'entité – une association

```
@OneToMany(mappedBy="dept")
public Collection<Employee> getEmployes() {
    return employees;
}
public void setEmployes(Collection<Employee>
emps) {
    this.employees = emps;
}
}
```

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

R. Grin

JPA

page 17

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

R. Grin

JPA

page 18

Configuration de la connexion

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

R. Grin

JPA

page 19

Exemple (début)

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

R. Grin

JPA

page 20

Exemple (fin)

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

R. Grin

JPA

page 21

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.

R. Grin

JPA

page 22

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...

...au moment du
commit

R. Grin

JPA

page 23

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();
```

R. Grin

JPA

page 24

- ❑ Remarque : le code précédent 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

R. Grin

JPA

page 25

Contexte de persistance

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

R. Grin

JPA

page 26

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 (voir support complémentaire)

R. Grin

JPA

page 27

Entités

R. Grin

JPA

page 28

Caractéristiques

- ❑ Seules les entités peuvent être
 - renvoyées par une requête (**Query**)
 - passées en paramètre d'une méthode d'un **EntityManager** ou d'un **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 *Embedded* étudiées plus loin)

R. Grin

JPA

page 29

Conditions pour les classes entités

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

R. Grin

JPA

page 30

Conditions pour les classes entités

- ❑ Une classe entité ne doit pas être une classe interne
- ❑ Une entité peut être une classe abstraite mais elle ne peut être une interface

R. Grin

JPA

page 31

Convention de nommage *JavaBean*

- ❑ Un JavaBean possède des propriétés
- ❑ Une propriété est représentée par 2 accesseurs (« *getter* » et « *setter* ») qui doivent suivre la convention de nommage suivante :
si **prop** est le nom de la propriété, le *getter* doit être **getProp** (ou **isProp** si la propriété est de type **boolean**) et le *setter* **setProp**
- ❑ Souvent une propriété correspond à une variable d'instance

R. Grin

JPA

page 32

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 (associées aux variables d'instance ou aux *getter*)

R. Grin

JPA

page 33

Accès par propriété

- ❑ Les accesseurs (*setter* et *getter*) doivent être **protected** ou **public**
- ❑ Ils 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 invalidée ; les exceptions contrôlées sont enveloppées par une **PersistenceException** (non contrôlée, sous **RuntimeException**)

R. Grin

JPA

page 34

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

R. Grin

JPA

page 35

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
- ❑ Rappel : le choix est déterminé par l'emplacement des annotations ; elles sont associées soit aux accesseurs, soit aux variables d'instance ; ne pas mélanger les 2 !

R. Grin

JPA

page 36

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

R. Grin

JPA

page 37

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)

R. Grin

JPA

page 38

Cycle de vie d'une instance d'entité

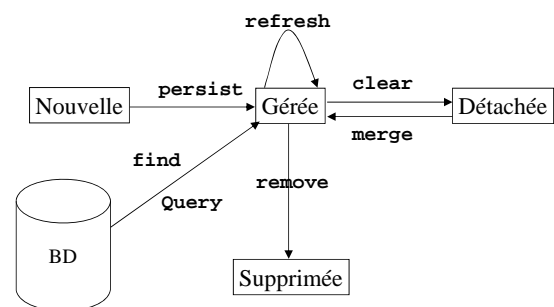
- détachée : elle a une identité dans la base mais elle n'est plus associée à un contexte de persistance (une entité peut, par exemple, devenir détachée si le contexte de persistance est vidé ou si elle est envoyée dans une autre JVM par RMI)
- supprimée : elle a une identité dans la base ; elle est 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**)

R. Grin

JPA

page 39

Cycle de vie d'une entité



R. Grin

JPA

page 40

Les 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 **Département** sont enregistrées dans la table **Département** (ou **DEPARTEMENT**) dont les colonnes se nomment **id**, **nom**, **lieu**

R. Grin

JPA

page 41

Configuration « par exception »

- ❑ 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 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 la classe

R. Grin

JPA

page 42

Nom de table

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

- ❑ Exemple :

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

R. Grin

JPA

page 43

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="AUTRENO",
        updatable=false, length=80)
public String getTruc() { ... }
```

R. Grin

JPA

page 44

Classe *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
- ❑ Elles peuvent être annotées comme les entités (avec `@Column` par exemple)
- ❑ Par exemple, une classe **Adresse** dont les valeurs sont insérées dans la table **Employe**

R. Grin

JPA

page 45

Classe *Embeddable*

- ❑ Comme les entités, ces classes doivent avoir un constructeur sans paramètre
- ❑ Les types permis pour leurs attributs sont les mêmes que les types permis pour les attributs des entités

R. Grin

JPA

page 46

Exemple

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

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

Rappel : à mettre sur `getAdresse()`
si accès par propriété

R. Grin

page 47

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

R. Grin

JPA

page 48

Classes insérées partagées

- ❑ Une classe entité peut référencer plusieurs instances d'une même classe insérée
- ❑ 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

R. Grin

JPA

page 49

@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

R. Grin

JPA

page 50

Exemple

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

R. Grin

JPA

page 51

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**

R. Grin

JPA

page 52

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 forme d'une **string** qui représente le nom de la valeur de l'énumération, il faut utiliser l'annotation **@Enumerated**

R. Grin

JPA

page 53

Exemple

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

R. Grin

JPA

page 54

Types temporels

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

R. Grin

JPA

page 55

Annotation pour les types temporels

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

R. Grin

JPA

page 56

Exemple

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

R. Grin

JPA

page 57

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

R. Grin

JPA

page 58

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
- ❑ Voir la spécification JPA pour plus de détails

R. Grin

JPA

page 59

Exemple

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

R. Grin

JPA

page 60

Types persistants

- ❑ Les champs des entités peuvent être d'à peu près n'importe quel type :
- ❑ Types primitifs
- ❑ Types String, BigInteger, BigDecimal, classes enveloppes de type primitifs, Date (des paquetages util et sql), Calendar, Time, Timestamp, et plus généralement Serializable
- ❑ Enumérations
- ❑ Entités et collections d'entités, classes « Embeddable »

R. Grin

JPA

page 61

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

R. Grin

JPA

page 62

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 sur la persistance des entités

R. Grin

JPA

page 63

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

R. Grin

JPA

page 64

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 suite du cours

R. Grin

JPA

page 65

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

R. Grin

JPA

page 66

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)

R. Grin

JPA

page 67

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 dans la base, les données récupérées ne tiennent pas compte de ces modifications

R. Grin

JPA

page 68

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 est au contraire malvenu, il est possible de récupérer des modifications effectuées en parallèle sur les données d'une entité en utilisant la méthode **refresh** de **EntityManager**

R. Grin

JPA

page 69

Interface **EntityManager**

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

R. Grin

JPA

page 70

Types de GE

- ❑ GE géré par le container (uniquement disponible dans un serveur d'applications ; pas étudié dans ce cours) ; le contexte de persistance peut être limité à 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

R. Grin

JPA

page 71

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** crée un GE
- ❑ Le GE est supprimé avec la méthode **close()** de la classe **EntityManager** ; il ne sera plus possible de l'utiliser ensuite

R. Grin

JPA

page 72

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**)
 - Un 2^{ème} paramètre de type **Map** qui contient des valeurs qui vont écraser les propriétés par défaut contenues dans **persistence.xml**

R. Grin

JPA

page 73

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

R. Grin

JPA

page 74

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 **Persistence.createEntityManagerFactory** ne se terminera pas correctement et lancera une exception

R. Grin

JPA

page 75

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)**

R. Grin

JPA

page 76

Méthodes de **EntityManager**

- ❑ **void lock(Object entité, LockModeType lockMode)**
- ❑ **void refresh(Object entité)**
- ❑ **void clear()**
- ❑ **boolean contains(Object entité)**
- ❑ **Query createQuery(String requête)**
- ❑ **Query createNamedQuery(String nom)**

R. Grin

JPA

page 77

Méthodes de **EntityManager**

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

R. Grin

JPA

page 78

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)

R. Grin

JPA

page 79

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

R. Grin

JPA

page 80

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** sera levée et la transaction est invalidée (*rollback*)
- ❑ Sinon, si Y est détachée et X possède l'association, Y est *flushée* ; si Y est la propriétaire, le comportement est indéfini

R. Grin

JPA

page 81

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**)

R. Grin

JPA

page 82

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 au moment du commit 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

R. Grin

JPA

page 83

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**
- ❑ Si A est supprimée (a été passée en paramètre à **remove**), elle devient gérée
- ❑ Si A est détachée, une **IllegalArgumentException** est lancée

R. Grin

JPA

page 84

remove

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

R. Grin

JPA

page 85

remove(A)

- ❑ Si A est gérée, elle devient « supprimée » (les données correspondantes de la base seront supprimées de la base au moment du flush du contexte de persistance)
- ❑ Ignorée si A est nouvelle ou supprimée
- ❑ Si A est détachée, une **IllegalArgumentException** est lancée
- ❑ Ne peut être utilisé que dans le contexte d'une transaction

R. Grin

JPA

page 86

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

R. Grin

JPA

page 87

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

R. Grin

JPA

page 88

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**,...)

R. Grin

JPA

page 89

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

R. Grin

JPA

page 90

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

R. Grin

JPA

page 91

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**

R. Grin

JPA

page 92

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 créée 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**

R. Grin

JPA

page 93

merge (A)

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

R. Grin

JPA

page 94

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**, l'application devra donc, sauf cas exceptionnel, ne plus utiliser l'objet A ; on pourra avoir ce type de code :
a = em.merge(a);
l'objet anciennement pointé par **a** ne sera plus référencé

R. Grin

JPA

page 95

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 **persist**, **remove**, **merge** peuvent être exécutées en dehors d'une transaction ; les modifications qu'elles ont provoquées seront enregistrées par 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

R. Grin

JPA

page 96

En dehors d'une transaction (2)

- ❑ En fait, certains SGBD se mettent 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 n'effectuer les modifications sur les entités gérées que dans le contexte d'une transaction, ou au moins de tester le comportement du SGBD

R. Grin

JPA

page 97

Transaction non terminée

- ❑ Il ne faut jamais oublier de terminer une transaction par `commit()` ou `rollback()` car le résultat dépend du fournisseur de persistance et du SGBD

R. Grin

JPA

page 98

Identité des entités

R. Grin

JPA

page 99

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)

R. Grin

JPA

page 100

Annotation

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

R. Grin

JPA

page 101

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

R. Grin

JPA

page 102

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**)

R. Grin

JPA

page 103

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**

R. Grin

JPA

page 104

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

R. Grin

JPA

page 105

Exemple

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

R. Grin

JPA

page 106

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'incrément de la séquence ; on aura alors souvent ce type d'annotation :

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

R. Grin

JPA

page 107

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 TopLink et Oracle (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

R. Grin

JPA

page 108

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**

R. Grin

JPA

page 109

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**

R. Grin

JPA

page 110

@EmbeddedId

- ❑ **@EmbeddedId** correspond au cas où la classe entité comprend un seul attribut annoté **@EmbeddedId**
- ❑ La classe clé primaire est annoté par **@Embeddable**
- ❑ 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

R. Grin

JPA

page 111

Exemple avec @EmbeddedId

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

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

R. Grin

JPA

page 112

@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

R. Grin

JPA

page 113

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;
    ...
}
```

R. Grin

JPA

page 114

Quelle solution choisir ?

- ❑ **@IdClass** existe pour assurer une compatibilité avec la spécification EJB 2.0
- ❑ Quand c'est possible il vaut mieux utiliser **@EmbeddedId**
- ❑ **@IdClass** a aussi son utilité, par exemple, pour les associations M:N si on veut que la clé primaire de la table association soit composée des clés étrangères vers les tables qui sont associées (voir section sur les associations)

R. Grin

JPA

page 115

Associations

R. Grin

JPA

page 116

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

R. Grin

JPA

page 117

Exemple

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

R. Grin

JPA

page 118

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>**

R. Grin

JPA

page 119

Types à utiliser

- ❑ Le plus souvent **Collection** sera utilisé
- ❑ **Set** peut être utile pour éliminer les doublons
- ❑ Les types concrets, tels que **HashSet** ou **ArrayList**, ne peuvent être utilisés que pour des entités « nouvelles » ; dès que l'entité est gérée, les types interfaces doivent être utilisés (ce qui permet au fournisseur de persistance d'utiliser son propre type concret)
- ❑ **List** peut être utilisé pour conserver un ordre mais nécessite quelques précautions

R. Grin

JPA

page 120

Ordre dans les collections

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

R. Grin

JPA

page 121

@OrderBy

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

R. Grin

JPA

page 122

Exemples

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

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

R. Grin

JPA

page 123

Associations bidirectionnelles

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

R. Grin

JPA

page 124

Bout propriétaire

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

R. Grin

JPA

page 125

Exemple

- ❑ Dans la classe **Employe** :

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

- ❑ Dans la classe **Departement** :

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

R. Grin

JPA

page 126

Méthode de gestion de l'association

- ❑ Pour faciliter la gestion des 2 bouts d'une association le code peut comporter une méthode qui effectue tout le travail
- ❑ En particulier, dans les associations 1-N, le bout « 1 » peut comporter ce genre de méthode (dans la classe **Département** d'une association département-employé) :

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

R. Grin

JPA

page 127

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>

R. Grin

JPA

page 128

Exemple

- ❑ Pour l'association qui détermine le département d'un employé
- ❑ Par défaut, la colonne clé étrangère placée dans la table **EMPLOYE** s'appellera **Département_ID**
- ❑ Pour changer ce nom (dans la classe **Employe**) :

```
@ManyToOne  
@JoinColumn(name="DEPT_ID")  
public Departement getDepartement() {
```

R. Grin

JPA

page 129

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)

R. Grin

JPA

page 130

Exemple

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

R. Grin

JPA

page 131

Persistence 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

R. Grin

JPA

page 132

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 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 réursive, ce qui peut nuire grandement aux performances

R. Grin

JPA

page 133

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

R. Grin

JPA

page 134

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 persistants, une `IllegalStateException` va être lancée au moment du commit, et la transaction va être invalidée (rollback)

R. Grin

JPA

page 135

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

R. Grin

JPA

page 136

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` et `MERGE` ; `ALL` correspond à toutes ces opérations
- ❑ Par défaut, aucune opération n'est appliquée transitivement

R. Grin

JPA

page 137

Exemples

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

R. Grin

JPA

page 138

Association 1:1

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

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

R. Grin

JPA

page 139

Association 1:1 sur les clés

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

R. Grin

JPA

page 140

Exemple

```
@OneToOne
@PrimaryKeyJoinColumn
private Employee employee
```

R. Grin

JPA

page 141

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 »)

R. Grin

JPA

page 142

Exemple

```
❑ class Employee {
    ...
    @ManyToOne
    public Departement getDepartement() {
        ...
    }
}

❑ class Departement {
    ...
    @OneToMany(mappedBy="departement")
    public List<Employee> getEmployes() {
        ...
    }
}
```

R. Grin

JPA

page 143

Cas particulier

- ❑ Une association unidirectionnelle 1:N est traduite pas une table association
- ❑ Si on traduisait l'association par une clé étrangère dans la table du côté « N », ça poserait un problème pour rendre une instance du côté « N » persistante : que mettre dans la clé primaire ? Avec la table association, c'est seulement quand l'instance du côté « 1 » sera rendu persistant qu'une ligne sera ajoutée dans la table association

R. Grin

JPA

page 144

1:N unidirectionnelle

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

R. Grin

JPA

page 145

Association M:N

- ❑ Annotation **@ManyToMany**
- ❑ Représentée par une table association

R. Grin

JPA

page 146

Association M:N (1)

- ❑ 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

R. Grin

JPA

page 147

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**

R. Grin

JPA

page 148

@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

R. Grin

JPA

page 149

Exemple (classe **Employe**)

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

R. Grin

JPA

page 150

Exemple (classe **Projet**)

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

R. Grin

JPA

page 151

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

R. Grin

JPA

page 152

Classe association pour une association M:N

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

R. Grin

JPA

page 153

Exemple - 1 identificateur

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

R. Grin

JPA

page 154

Exemple - 1 identificateur

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

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

R. Grin

JPA

page 155

Exemple - 2 identificateurs (1)

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

R. Grin

JPA

page 156

Exemple - 2 identificateurs (2)

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

R. Grin

JPA

page 157

Exemple - 2 identificateurs

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

R. Grin

JPA

page 158

Exemple - 2 identificateurs

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

R. Grin

JPA

page 159

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

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

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

R. Grin

JPA

page 160

Classe **Participation** (2 choix pour Id)

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

R. Grin

JPA

page 161

Classe **Participation** (Id)

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

R. Grin

JPA

page 162

Classe **Participation** (champs)

```
private Employe employe;  
private long employeId;  
private Projet projet;  
private long projetId;  
private String fonction;  
public Participation() { }
```

R. Grin

JPA

page 163

Participation (constructeurs)

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

R. Grin

JPA

page 164

Établir une association

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

R. Grin

JPA

page 165

Participation (constructeurs)

```
public Participation() { } // Pour JPA  
public Participation(Employe employe,  
                    Projet projet,  
                    String fonction) {  
    this.employe = employe;  
    this.employeId = employe.getId();  
    this.projet = projet;  
    this.projetId = projet.getId();  
    employe.getParticipations().add(this);  
    projet.getParticipations().add(this);  
    this.fonction = fonction;  
}
```

R. Grin

JPA

page 166

Participation (associations)

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

R. Grin

JPA

page 167

Classe **ParticipationId**

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

R. Grin

JPA

page 168

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

R. Grin

JPA

page 169

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

R. Grin

JPA

page 170

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)

R. Grin

JPA

page 171

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)

R. Grin

JPA

page 172

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

R. Grin

JPA

page 173

Mode de récupération des attributs

- ❑ Les attributs aussi peuvent être récupérés en mode « retardé »
- ❑ 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

R. Grin

JPA

page 174

Map pour association

- ❑ A la place d'une collection il est possible d'utiliser une *map*
- ❑ En ce cas il faut annoter l'association avec **@MapKey** qui précise la clé de la *map* qui doit être un des attributs de l'entité contenue dans la *map*
- ❑ La classe de la clé doit avoir sa méthode **hashCode** compatible avec sa méthode **equals** et chaque clé doit être unique parmi toutes les autres clés de la *map*

R. Grin

JPA

page 175

Map pour association

- ❑ Si l'association est traduite par une *map* mais n'a pas d'annotation **@KeyMap**, la clé sera considérée être l'identificateur de l'entité

R. Grin

JPA

page 176

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="nom")  
    public Map<String, Employe> getEmployes() {  
        ...  
    }  
}
```

R. Grin

JPA

page 177

Héritage

R. Grin

JPA

page 178

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**)

R. Grin

JPA

page 179

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

R. Grin

JPA

page 180

Exemple

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

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

A mettre dans la
classe racine de
la hiérarchie

« Employe » par défaut

R. Grin

JPA

page 181

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

R. Grin

JPA

page 182

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**)

R. Grin

JPA

page 183

Colonne discriminatrice (2)

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

R. Grin

JPA

page 184

Exemple

```
@Entity
@Inheritance
@DiscriminatorColumn(
    name="TRUC",
    discriminatorType="STRING",
    length=5)
public class Machin {
    ...
}
```

R. Grin

JPA

page 185

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

R. Grin

JPA

page 186

Une table par classe

- ❑ 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
- ❑ Une colonne discriminatrice est ajoutée dans la table qui correspond à la classe racine de la hiérarchie d'héritage
- ❑ Cette colonne permet de simplifier certaines requêtes ; par exemple, pour retrouver les noms de tous les employés (classe `Personne` à la racine de la hiérarchie d'héritage)

R. Grin

JPA

page 187

Exemple

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

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

R. Grin

JPA

page 188

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)
- ❑ 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

R. Grin

JPA

page 189

Exemple

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

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

R. Grin

JPA

page 190

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

R. Grin

JPA

page 191

Compléments

- ❑ L'annotation `@Entity` ne s'hérite pas
- ❑ Les sous-classes entités d'une entité doivent être annotées 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

R. Grin

JPA

page 192

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**
- ❑ L'état de cette classe mère sera rendu persistant avec l'état de la classe entité fille, dans la même table que cette classe entité

R. Grin

JPA

page 193

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

R. Grin

JPA

page 194

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**

R. Grin

JPA

page 195

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;
    ...
}
```

R. Grin

JPA

page 196

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

R. Grin

JPA

page 197

Requêtes - JPQL

R. Grin

JPA

page 198

- ❑ Cette section concerne les entités (et valeurs) retrouvées en interrogeant la base de données

R. Grin

JPA

page 199

Rappel important

- ❑ Toutes les entités retrouvées par **find**, **getReference** ou un query sont automatiquement gérées par le gestionnaire d'entités
- ❑ Les modifications apportées à ces entités sont donc enregistrées au prochain commit (mais les nouveaux objets associés à l'entité retrouvée ne sont pas automatiquement persistants même s'il y a une cascade sur persist ; ça ne marche que pour un appel explicite de la méthode **persist**)

R. Grin

JPA

page 200

Chercher par identité

- ❑ **find** et **getReference** (de **EntityManager**) permettent de retrouver une entité en donnant son identificateur dans la BD
- ❑ `<T> T find(Class<T> classeEntité, Object cléPrimaire)`
- ❑ `<T> T get(Class<T> classeEntité, Object cléPrimaire)`
- ❑ Le résultat sera **null** si aucune entité n'a cet identificateur dans la base de donnée

R. Grin

JPA

page 201

Exemple

```
Departement dept =
    em.find(Departement.class, 10);
```

R. Grin

JPA

page 202

getReference

- ❑ **getReference** renvoie une référence vers une entité, sans que cette entité ne soit nécessairement initialisée
- ❑ Le plus souvent il vaut mieux utiliser **find**

R. Grin

JPA

page 203

getReference

- ❑ **getReference** peut être (rarement) utilisée pour améliorer les performances quand une entité non initialisée peut être suffisante, sans que l'entité entière soit retrouvée dans la base de données
- ❑ Par exemple, pour indiquer une association dont le but est unique (OneToOne ou ManyToOne) :

```
Departement dept =
    em.getReference(Departement.class, 10);
Employe emp.setDepartement(dept);
```

R. Grin

JPA

page 204

Étapes pour récupérer des données

- ❑ Il est possible 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

R. Grin

JPA

page 205

Langage JPQL

- ❑ Le langage JPQL (*Java Persistence Query Language*) permet de décrire ce que l'application recherche
- ❑ Il ressemble beaucoup à SQL

R. Grin

JPA

page 206

Requêtes sur les entités « objet »

- ❑ Les requêtes JPQL travaillent avec le modèle objet et pas avec le modèle relationnel
- ❑ Les identificateurs désignent les entités et leurs propriétés et pas les tables et leurs colonnes
- ❑ Les seules classes qui peuvent être explicitement désignées dans une requête (clause `from`) sont les entités

R. Grin

JPA

page 207

Désigner les entités

- ❑ Les entités sont désignées par leur nom
- ❑ Le nom d'une entité est donné par l'attribut `name` de `@Entity` ; par défaut c'est le nom terminal (sans le nom du paquetage) de la classe

R. Grin

JPA

page 208

Alias

- ❑ Le texte des requêtes utilise beaucoup les alias de classe
- ❑ Les attributs des classes doivent être préfixés par les alias
- ❑ Une erreur fréquente du débutant est d'oublier les alias en préfixe

R. Grin

JPA

page 209

Exemples de requêtes JPQL

- ❑ `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.employees e group by d.nom having count(d.nom) > 5`

R. Grin

JPA

page 210

Type du résultat (1)

- ❑ L'expression de la clause select peut être
 - une (ou plusieurs) expression « entité », par exemple un employé (**e** par exemple)
 - une (ou plusieurs) expression « valeur », par exemple le nom et le salaire d'un employé (**e.nom** par exemple)
- ❑ Si la requête renvoie des entités, elles sont automatiquement gérées par le GE (toute modification sera répercutée dans la base)

R. Grin

JPA

page 211

Type du résultat (2)

- ❑ L'expression ne peut être une collection (**d.employes** par exemple), bien que TopLink le permette !

R. Grin

JPA

page 212

Obtenir le résultat de la requête

- ❑ Pour le cas où une seule valeur ou entité est renvoyée, le plus simple est d'utiliser la méthode `getSingleResult()` ; elle renvoie un **Object**
- ❑ La méthode lance des exceptions s'il n'y a pas exactement une entité qui correspond à la requête :
 - **EntityNotFoundException**
 - **NonUniqueResultException**

R. Grin

JPA

page 213

Obtenir le résultat de la requête

- ❑ 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**, éventuellement vide si le résultat est vide
- ❑ Un message d'avertissement sera affiché durant la compilation si le résultat est rangé dans une liste générique (**List<Employe>** par exemple)

R. Grin

JPA

page 214

Type d'un élément du résultat

- ❑ Le type d'un élément de la liste (ou de l'unique valeur ou entité renvoyée) est
 - **Object** si la clause select ne comporte qu'une seule expression
 - **Object[]** si elle comporte plusieurs expressions

R. Grin

JPA

page 215

Exemple 1

```
String s = "select e from Employe as e";
Query query = em.createQuery(s);
List<Employe> listeEmployes =
    (List<Employe>)query.getResultList();
```

R. Grin

JPA

page 216

Exemple 2

```
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]);
}
```

R. Grin

JPA

page 217

Interface Query

- Représente une requête
- Une instance de **Query** (d'une classe implémentant **Query**) est obtenue par les méthodes **createQuery**, **createNativeQuery** ou **createNamedQuery** de l'interface **EntityManager**

R. Grin

JPA

page 218

Méthodes de Query (1)

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

R. Grin

JPA

page 219

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)**

R. Grin

JPA

page 220

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**

R. Grin

JPA

page 221

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();
```

R. Grin

JPA

page 222

Types de requête

- ❑ Requêtes dynamiques dont le texte est donnée en paramètre de `createQuery`
- ❑ Requêtes natives particulières à un SGBD (pas portables) ; requête SQL (pas JPQL) avec tables et colonnes (pas classes et attributs)
- ❑ Requêtes nommées dont le texte est donnée dans une annotation de l'entité concernée et dont le nom est passé en paramètre de `createNamedQuery` ; une requête nommée peut être dynamique ou native

R. Grin

JPA

page 223

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)

R. Grin

JPA

page 224

Requête nommée (1)

- ❑ Seules les entités peuvent contenir des définitions de requêtes nommées
- ❑ Une requête nommée peut être mise dans n'importe quelle entité, mais on choisira le plus souvent l'entité qui correspond à ce qui est renvoyé par la requête
- ❑ Le nom de la requête nommée doit être unique parmi toutes les entités de l'unité de persistance ; on pourra, par exemple, préfixer le nom par le nom de l'entité : `Employe.findAll`

R. Grin

JPA

page 225

Requête nommée (2)

- ❑ Les requêtes nommées peuvent être analysée et précompilées par le fournisseur de persistance au démarrage de l'application, ce qui peut améliorer les performances

R. Grin

JPA

page 226

Exemple de requête nommée

```
@Entity
@NamedQuery (
    name = "findNomsEmployes",
    query = "select e.nom from Employe as e
            where upper(e.departement.nom) = :nomDept"
)
public class Employe extends Personne {
    ...
}
```

```
Query q =
    em.createNamedQuery("findNomsEmployes");
```

R. Grin

JPA

page 227

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");`

R. Grin

JPA

page 228

Plusieurs requêtes nommées

- ❑ Si une classe a plusieurs requêtes nommées, il faut les regrouper dans une annotation

@NamedQueries :

```
@NamedQueries({
    @NamedQuery(...),
    @NamedQuery(...),
    ...
})
```

R. Grin

JPA

page 229

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
- ❑ Pour une requête il est possible d'éviter ce flush avec la méthode **setFlushMode** :
query.setFlushMode(FlushMode.COMMIT);
Dans ce mode, seul un commit provoquera un flush

R. Grin

JPA

page 230

- ❑ Les transparents suivants étudient en détails le langage JPQL

R. Grin

JPA

page 231

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

R. Grin

JPA

page 232

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

R. Grin

JPA

page 233

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**

R. Grin

JPA

page 234

Expression de chemin

- ❑ Les requêtes peuvent contenir des expressions de chemin pour naviguer entre les entités en suivant les associations déclarées dans le modèle objet (les annotations `@OneToOne`, `@OneToMany`, ...)
- ❑ La notation « pointée » est utilisée

R. Grin

JPA

page 235

Règle pour les expressions de chemin

- ❑ Une navigation peut être chaînée à une navigation précédente à la condition que la navigation précédente ne donne qu'une seule entité (`OneToOne` ou `ManyToOne`)
- ❑ Dans le cas où une navigation aboutit à plusieurs entités, il est possible d'utiliser la clause `join` étudiée plus loin pour obtenir ces entités

R. Grin

JPA

page 236

Exemples

- ❑ Si `e` est un alias pour `Employe`,
 - « `e.departement` » désigne le département d'un employé
 - « `e.projets` » désigne la collection de projets auxquels participe un employé
- ❑

```
select e.nom
from Employe as e
where e.departement.nom = 'Qualité'
```
- ❑ `e.projets.nom` n'est pas autorisé car `e.projets` est une collection (voir clause `join`)

R. Grin

JPA

page 237

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
```

R. Grin

JPA

page 238

new

- ❑ Il est possible de renvoyer des instances d'une classe dont le constructeur prend en paramètre des informations récupérées dans la base de données
- ❑ La classe doit être désignée par son nom complet (avec le nom du paquetage)
- ❑ Exemple :

```
select new p1.p2.Classe(e.nom, e.salaire)
from Employe e
```

R. Grin

JPA

page 239

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`

R. Grin

JPA

page 240

Exemple

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

R. Grin

JPA

page 241

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.employees e
group by d.nom
having avg(e.salaire) > 1000
```

R. Grin

JPA

page 242

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)
```

R. Grin

JPA

page 243

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)
```

R. Grin

JPA

page 244

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)
```

R. Grin

JPA

page 245

Sous-requête - **exists**

- ❑ **[not]exists** fonctionne comme avec SQL

- ❑ Exemple :

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

R. Grin

JPA

page 246

Exemple

```
select e.nom,  
       e.departement.nom,  
       e.superieur.departement.nom  
from Employe e
```

R. Grin

JPA

page 247

Contre-exemple

- ❑ « `d.employees.nom` » est interdit car `d.employees` est une collection
- ❑ Pour avoir les noms des employés d'un département, il faut utiliser une jointure

R. Grin

JPA

page 248

Jointure

- ❑ Une jointure permet de combiner plusieurs entités dans un select
- ❑ Rappel : il est possible d'utiliser plusieurs entités dans une requête grâce à la navigation
- ❑ Une jointure est le plus souvent utilisée pour résoudre les cas (interdit par JPA) où
 - l'expression du select serait une collection
 - la navigation partirait d'une collection

R. Grin

JPA

page 249

Types de jointures

- ❑ Il existe plusieurs types de jointures :
 - jointure interne (jointure standard `join`)
 - jointure externe (`outer join`)
 - jointure avec récupération de données en mémoire (`join fetch`)
 - jointure « à la SQL » dans un where pour joindre suivant des champs qui ne correspondent pas à une association (`where e1.f1 = e2.f2`)

R. Grin

JPA

page 250

Exemple

- ❑ Si on veut tous les employés d'un département, la requête suivante n'est pas permise par la spécification JPA (bien que TopLink l'autorise) :

```
select d.employees  
from Departement d  
where d.nom = 'Direction'
```
- ❑ Une jointure est nécessaire :

```
select e  
from Departement d join d.employees e  
where d.nom = 'Direction'
```

R. Grin

JPA

page 251

Autres exemples

- ❑

```
select e.nom  
from Departement d  
    join d.employees e  
where d.nom = 'Direction'
```
- ❑

```
select e.nom, parts.projet.nom  
from Employe e  
    join e.participations parts
```
- ❑

```
select e.nom, d.nom  
from Employe e, Departement d  
where d = e.departement
```
- ❑

```
select e, p  
from Employe e  
    join e.participations parts  
    join parts.projet p
```

R. Grin

JPA

page 252

Jointure externe

- ❑

```
select e, d
from Employe e left join e.departement d
ramènera aussi les employés qui ne sont pas
associés à un département
```

R. Grin

JPA

page 253

join fetch

- ❑ Permet d'éviter le problème des « N + 1 selects »
- ❑ L'entité placée à droite de **join fetch** sera créée en mémoire en même temps que l'entité de la clause select
- ❑ Le select SQL généré sera une jointure externe qui récupérera les données de toutes les entités associées en même temps que les données des entités principales de la requête

R. Grin

JPA

page 254

Exemple

- ❑

```
select e
from Employe e join fetch e.departement
```
- ❑ Cette requête récupérera tous les employés mais, en plus, l'appel de la méthode **getDepartement()** ne provoquera aucune interrogation de la base de données puisque le « **join fetch** » aura déjà chargé tous les départements des employés
- ❑ L'exemple suivant précharge les collections de participations aux projets

R. Grin

JPA

page 255

Exemple

```
String texteQuery =
    "select e "
    + " from Employe as e "
    + "      join fetch e.participations";
Query query = em.createQuery(texteQuery);
listeEmployes =
    (List<Employe>)query.getResultList();
```

R. Grin

JPA

page 256

Doublons possibles avec join fetch

- ❑ La requête SQL lancée par un **join fetch** fait une jointure pour récupérer les entités préchargées
- ❑ Ensuite, les entités préchargées sont enlevées des lignes du résultat pour qu'elles n'apparaissent pas dans le résultat du query
- ❑ Ce traitement, imposé par la spécification de JPA, peut occasionner des doublons dans le résultat si le select renvoie des valeurs
- ❑ Pour les éviter, il faut ajouter l'opérateur **DISTINCT** dans le texte de la requête ou placer le résultat dans une collection de type **set**

R. Grin

JPA

page 257

- ❑ Il existe aussi des variantes « **outer join** » de **join fetch** pour récupérer dans le select des entités non jointes à une autre entité

R. Grin

JPA

page 258

Produit cartésien (1)

- ❑ Le préchargement par join fetch de plusieurs collections d'une même entité peut occasionner un phénomène nuisible aux performances
- ❑ En effet, le select SQL généré par le fournisseur de persistance peut récupérer un produit cartésien des éléments des collections
- ❑ Le fournisseur s'arrange pour ne garder que les informations nécessaires mais le select peut renvoyer un très grand nombre de lignes qui vont transiter par le réseau

R. Grin

JPA

page 259

Produit cartésien (2)

- ❑ Dans le cas où les collections contiennent de nombreux éléments il faut donc vérifier avec les logs du fournisseur si le select généré renvoie effectivement un trop grand nombre de lignes et changer de stratégie de récupération si c'est le cas (récupérer séparément les informations sur les collections)
- ❑ En effet, si 2 collections contiennent 20 éléments, et si 1000 entités principales sont renvoyées le select renverra 400.000 lignes ! (au lieu de 40.000 si on ramène les informations avec 2 join fetch séparés)

R. Grin

JPA

page 260

Exemple

- ❑ Une entité principale ep avec 2 associations de l'entité principale avec d'autres entités e2 et e3
- ❑ Pour récupérer en mémoire les entités principales et associées, le select généré risque d'être du type suivant (consulter les logs du fournisseur de persistance) :

```
select ep.*, e2.*, e3.*
from EP ep
  left outer join E2 e2 on ep.id_e2 =
e2.id
  left outer join E3 e3 on ep.id_e3 =
e3.id
```

R. Grin

JPA

page 261

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**
- ❑ Spécification JPA pour plus d'informations,

R. Grin

JPA

page 262

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

R. Grin

JPA

page 263

Exemples

- ❑

```
select d
from Departement
where e.employes is empty
```
- ❑

```
select e
from Employe e
where :projet member of
e.participations.projet
```

R. Grin

JPA

page 264

Parcours d'une collection

- ❑ La requête suivante ne donnera pas tous les produits liés à une facture ; elle provoquera une exception à l'exécution

```
select distinct f.lignes.produit  
from Facture as f
```

- ❑ En effet, il est interdit de composer une expression de chemin en partant d'une expression qui désigne une collection (**f.lignes**)

R. Grin

JPA

page 265

Parcours d'une collection

- ❑ Le plus simple est d'utiliser un **join** dans le **from** avec un alias (« l » dans l'exemple ci-dessous) pour désigner un élément qui parcourt la collection :

```
select distinct l.produit  
from Facture as f join f.lignes as l
```

- ❑ L'autre solution, « in », une ancienne syntaxe héritée d'EJB 2, n'est pas recommandée

R. Grin

JPA

page 266

Pagination du résultat

- ❑ **Query setMaxResults(int n)** : indique le nombre maximum de résultats à retrouver
- ❑ **Query setFirstResult(int n)** : indique la position du 1^{er} résultat à retrouver (numéroté à partir de 0)

R. Grin

JPA

page 267

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();
```

R. Grin

JPA

page 268

Opérations de modification en volume

R. Grin

JPA

page 269

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 modifier les données de la base directement, sans créer les entités correspondantes

R. Grin

JPA

page 270

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

```
update employe
set salaire = salaire * 1.05
sera énormément plus performant
```

Exemple

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

Syntaxe

- ❑ Les ordres de modification en volume référencent les classes et les propriétés des classes mais ils ne créent aucune entité en mémoire et ils ne mettent pas à jour les entités déjà présentes en mémoire
- ❑ `update Entite as alias`
`set alias.prop1 = val1, alias.prop2 = val2,...`
`where condition`
- ❑ La condition peut être aussi complexe que la condition d'un select

Remarques

- ❑ Les modifications doivent être faites dans une transaction
- ❑ Le plus souvent il faut isoler le lancement de ces opérations dans une transaction à part, ou au moins 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

- ❑ `NonUniqueResultException`
- ❑ `NoResultException`
- ❑ `EntityNotFoundException`
- ❑ `EntityExistsException`
- ❑ `TransactionRequiredException`
- ❑ `RollbackException`
- ❑ `OptimisticLockException`

R. Grin

JPA

page 277

Exception et rollback

- ❑ Toute les exceptions de type `PersistenceException` provoquent un rollback, sauf `NonUniqueResultException` et `NoResultException`
- ❑ Rappel : si l'accès se fait par propriétés, une `RuntimeException` levée par une méthode d'accès à une propriété provoque un rollback de la transaction en cours

R. Grin

JPA

page 278

Transaction

R. Grin

JPA

page 279

2 types de transactions

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

R. Grin

JPA

page 280

Transactions dans Java EE

- ❑ Elles sont étudiées dans la dernière section de ce support de cours

R. Grin

JPA

page 281

Transactions dans Java SE (sans serveur d'applications)

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

R. Grin

JPA

page 282

EntityTransaction

- ❑ En dehors d'un serveur d'applications, une application doit utiliser l'interface `javax.persistence.EntityTransaction` pour travailler avec des transactions locales à une ressource
- ❑ Une instance de `EntityTransaction` peut s'obtenir par la méthode `getTransaction()` de `EntityManager`

R. Grin

JPA

page 283

EntityTransaction

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

R. Grin

JPA

page 284

Exemple

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

R. Grin

JPA

page 285

Rollback (1)

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

R. Grin

JPA

page 286

Rollback (2)

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

R. Grin

JPA

page 287

Transaction et contexte de persistance

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

R. Grin

JPA

page 288

Synchronisation d'un GE avec une transaction

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

R. Grin

JPA

page 289

Modifications et commit

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

R. Grin

JPA

page 290

Concurrence

R. Grin

JPA

page 291

GE et threads

- ❑ Une fabrique de GE peut être utilisée sans problème par plusieurs threads
- ❑ Mais un GE ne doit être utilisé concurremment que par un seul thread

R. Grin

JPA

page 292

Entités et threads

- ❑ Les entités ne sont pas prévues pour être utilisées par plusieurs threads en même temps
- ❑ Si ça doit être le cas, l'application doit prendre toutes ses précautions pour éviter les problèmes
- ❑ C'est aussi le rôle de l'application d'empêcher qu'une entité ne soit gérée par plusieurs GE en même temps
- ❑ Le rôle du fournisseur de persistance n'intervient pas pour ces cas

R. Grin

JPA

page 293

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

R. Grin

JPA

page 294

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 entre-temps les données de la BD correspondant à l'entité, la transaction doit être invalidée

R. Grin

JPA

page 295

Gestion de la concurrence

- ❑ Par défaut, le fournisseur de persistance gère les problèmes d'accès concurrents aux entités gérées par un GE avec une stratégie optimiste

R. Grin

JPA

page 296

@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)

R. Grin

JPA

page 297

Exemple

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

R. Grin

JPA

page 298

Entité « versionnée »

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

R. Grin

JPA

page 299

lock(entite, 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) et déjà gérée

R. Grin

JPA

page 300

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 (utilité expliquée dans les transparents suivants)

R. Grin

JPA

page 301

READ

- ❑ Empêche les lectures non répétables sur les données de la BD associées à l'entité
- ❑ Sans ce blocage, on n'a pas l'assurance d'avoir une lecture répétable avec des query qui renvoient des d'attributs d'entité (par exemple `employe.nom`)
- ❑ Le fournisseur de persistance peut choisir une stratégie pessimiste (du type « select for update ») ou optimiste

R. Grin

JPA

page 302

READ

- ❑ Le « blocage » réel peut n'être effectué qu'au moment du flush ou du commit, quelle que soit l'emplacement du `read` dans le code Java
- ❑ S'il y a un problème, une exception `OptimisticLockException` peut être lancée

R. Grin

JPA

page 303

Une situation à éviter

- ❑ Un bilan qui calcule le total des salaires des employés pour chaque département doit interdire qu'un employé ne change de département pendant le traitement, en passant du département 10 au département 20
- ❑ Sinon, cet employé sera compté 2 fois : après avoir fait le calcul du total du département 10, on va faire une requête pour récupérer les employés du département 20 et retrouver l'employé déjà décompté dans le département 10

R. Grin

JPA

page 304

Utilité des blocages READ

- ❑ Pour éviter ce problème, les employés peuvent être bloqués en mode **READ**, au fur et à mesure de leur prise en compte pour le calcul du total des salaires de leur département
- ❑ Le blocage peut être pessimiste (select for update sur les lignes bloquées) ou optimiste ; dans ce dernier cas le traitement recevra une exception si l'optimisme était injustifié

R. Grin

JPA

page 305

WRITE

- ❑ Force une incrémentation de la version
- ❑ Pour certains traitements il est intéressant de faire considérer que l'entité a été modifiée même si aucun de ses attributs n'a été modifié
- ❑ Dans l'exemple qui suit le blocage **WRITE** évite en quelque sorte les problèmes de lignes fantômes avec les associations qui concernent une entité

R. Grin

JPA

page 306

WRITE

- ❑ En effet, sans le blocage « WRITE », le fournisseur de persistance incrémente automatiquement le numéro de version si une entité est modifiée mais pas si les liens des associations qui partent de l'entité sont modifiées (voir cependant la remarque qui suit l'exemple donné dans les transparents suivants)

R. Grin

JPA

page 307

Utilité des blocages WRITE (1)

- ❑ Une entête de facture contient un champ dans lequel est stocké le total de la facture
- ❑ Supposons que le calcul de ce total est effectué par un processus à part du processus de l'ajout des lignes de la facture
- ❑ Lorsqu'une ligne de facture est ajoutée à une facture, l'en-tête de la facture n'est pas modifiée donc le fournisseur de persistance n'incrémente pas le numéro de version de l'en-tête

R. Grin

JPA

page 308

Utilité des blocages WRITE (2)

- ❑ Une ligne peut être ajoutée à une facture sans que le traitement qui calcule le total ne s'en rende compte ; il y aura inconsistance entre ce total et les lignes de la facture
- ❑ Si l'en-tête est bloquée en écriture pendant l'ajout d'une ligne de facture, le traitement qui effectue le calcul du total s'apercevra que la facture a changé pendant le calcul puisque le numéro de version aura été incrémenté et il n'y aura pas inconsistance

R. Grin

JPA

page 309

Remarque

- ❑ La spécification JPA ne dit pas avec précision quand le numéro de version est incrémenté
- ❑ Certains certains fournisseurs de persistance comme Hibernate incrémentent le numéro de version quand les collections d'une entité sont modifiées
- ❑ Dans ce cas le blocage Write sera tout de même utile si, par exemple, un des éléments de la collection est modifié, par exemple le prix d'un produit compris dans la facture

R. Grin

JPA

page 310

Niveau d'isolation

- ❑ Le fournisseur suppose que le SGBD gère la base de données avec un niveau d'isolation « READ COMMITTED » (voir cours sur les BD)

R. Grin

JPA

page 311

Entité détachée

R. Grin

JPA

page 312

- ❑ Les entités détachées sont surtout utiles dans les applications multi-tiers (avec Java EE et un serveur d'applications)
- ❑ Cette section survole les possibilités offertes par les entités détachées
- ❑ Les subtilités des entités détachées sont exposées dans le support complémentaire « JPA et serveurs d'applications »

R. Grin

JPA

page 313

Cas d'utilisation

- ❑ Une application multi-tiers
- ❑ Le serveur d'application récupère des données dans la BD
- ❑ Ces données sont passées à la couche client, montrées à l'utilisateur qui peut les modifier
- ❑ Les modifications sont repassées au serveur et enregistrées dans la BD
- ❑ Les entités détachées facilitent l'implémentation d'un tel cas d'utilisation

R. Grin

JPA

page 314

Utilisation des entités détachées

- ❑ Une fois détachées les entités peuvent être passées à la couche cliente
- ❑ La couche cliente peut modifier les entités détachées
- ❑ Ces entités peuvent ensuite être rattachées à un GE et les modifications effectuées dans la couche cliente peuvent être alors enregistrées dans la base de données lors d'un flush

R. Grin

JPA

page 315

Rattachement

- ❑ La méthode **merge** de **EntityManager** permet d'obtenir une entité gérée à partir d'une entité détachée
- ❑ Sa signature :

```
<T> T merge(T entité)
```
- ❑ Attention, l'entité passée en paramètre n'est pas rattachée ; c'est l'entité renvoyée par la méthode **merge** qui est rattachée ; cette entité a le même état et la même clé primaire que l'entité passée en paramètre

R. Grin

JPA

page 316

- ❑ Si l'entité passée en paramètre de **merge** est déjà gérée par le GE, elle est renvoyée par la méthode **merge**

R. Grin

JPA

page 317

État d'une entité détachée

- ❑ L'état d'une entité détachée peut ne pas être entièrement disponible

R. Grin

JPA

page 318

État d'une entité

- ❑ Pour des raisons de performances, l'état d'une entité gérée par un GE peut ne avoir été complètement récupéré dans la BD (récupération retardé ; *lazy*)
- ❑ Le reste de l'état ne sera récupéré, avec l'aide du GE, que lorsque l'entité en aura vraiment besoin
- ❑ Si l'entité est détachée alors qu'une partie de son état n'a pas encore été récupérée, la partie manquante de l'entité détachée ne sera pas disponible

R. Grin

JPA

page 319

Attribut disponible

- ❑ Un attribut persistant d'une entité est immédiatement disponible dans les 2 cas suivants :
 - l'attribut a déjà été utilisé
 - l'attribut n'a pas été marqué par **fetch=LAZY** (par défaut, les valeurs des attributs sont chargées en mémoire)

R. Grin

JPA

page 320

Association d'une entité détachée

- ❑ Si une association d'un objet détaché a le mode de récupération LAZY, il est possible que l'association ne puisse être récupérée (dépend des circonstances et des fournisseurs de JPA)

R. Grin

JPA

page 321

Association « lazy » disponible

- ❑ Si une association d'un objet détaché a le mode de récupération LAZY, il est possible de naviguer à travers cette association si
 - l'application a déjà effectué cette navigation
 - ou si l'association a été chargée par un **join fetch** lors d'une requête

R. Grin

JPA

page 322

Rendre accessible l'état d'une entité détachée

- ❑ 2 solutions pour accéder à tout l'état d'une entité détachée
- ❑ L'application récupère tout l'état de l'entité gérée avant le détachement
- ❑ L'application rattache l'entité par la méthode **merge** pour récupérer l'état manquant

R. Grin

JPA

page 323

Récupérer une association avant le détachement

- ❑ Si le but est une entité (OneToOne ou ManyToOne), il suffit de lire une des propriétés de l'entité
- ❑ Si le but est une collection (OneToMany ou ManyToMany), l'appel de la méthode **size()** de la collection suffit le plus souvent

R. Grin

JPA

page 324

Entité détachée et concurrence

- ❑ Avant que les modifications sur l'entité détachée ne soient enregistrées dans la BD, l'entité doit être rattachée à un contexte de persistance (celui d'origine ou un autre)
- ❑ A ce moment, ou au moment du commit qui enregistre vraiment les modifications, le contexte de persistance vérifie qu'il n'y a pas de conflit de concurrence

R. Grin

JPA

page 325

Conflit de concurrence

- ❑ Si les données de la BD associée à un objet détaché ont été modifiées depuis le détachement de l'objet, **merge** lance une exception, ou le commit échouera
- ❑ Tout se passe donc comme si un conflit de concurrence avait été détecté (avec une stratégie optimiste)

R. Grin

JPA

page 326

Détachement automatique

- ❑ Dans Java SE, quelques situations provoquent un détachement automatique des entités gérées :
 - Après un rollback
 - Après un clear du GE
 - Après la fermeture du GE

R. Grin

JPA

page 327

Configuration d'une unité de persistance

R. Grin

JPA

page 328

Fichier persistence.xml

- ❑ Les informations doivent être données dans un fichier **persistence.xml**
- ❑ Dans un environnement non géré, ce fichier doit se situer dans le répertoire META-INF d'un des répertoires du *classpath*
- ❑ Dans un serveur d'applications, le répertoire META-INF peut se trouver à divers endroits : dans WEB-INF/classes, un fichier jar de WEB-INF/lib, un fichier jar de la racine d'un fichier EAR,... (voir spécification pour la liste complète)

R. Grin

JPA

page 329

Configuration d'une unité de persistance

- ❑ Dans un environnement non géré par un serveur d'applications, il est nécessaire de donner les informations pour la connexion à la base de données

R. Grin

JPA

page 330

Configuration d'une unité de persistance

- ❑ Le fichier `persistence.xml` donne d'autres informations, comme les noms des classes entités, la configuration de l'environnement de mis au point (*logging*, affichage des ordres SQL lancés dans le SGBD,...) ou des propriétés particulières au driver du fournisseur de persistance

R. Grin

JPA

page 331

`persistence.xml`

```
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  version="1.0" >
  <persistence-unit name="Employes"
    transaction-type="RESOURCE_LOCAL">
    <class>p1.Employe</class>
    <class>p1.Dept</class>
    <properties>
      . . . <!-- Voir transparent suivant-->
    </properties>
  </persistence-unit>
</persistence>
```

R. Grin

JPA

page 332

Section `properties`

- ❑ La section `properties` dépend du fournisseur des fonctionnalités décrites dans la spécification JPA
- ❑ Elle contient les informations pour la connexion mais aussi d'autres informations pour le logging ou la création automatique des tables si elle n'existent pas déjà ; il faut consulter la documentation du fournisseur

R. Grin

JPA

page 333

Exemple de section `properties`

```
<properties>
  <property
    name="toplink.jdbc.driver"
    value="oracle.jdbc.OracleDriver"/>
  <property
    name="toplink.jdbc.url"
    value="jdbc:oracle:thin:@...:INFO "/>
  <property name="toplink.jdbc.user"
    value="toto"/>
  <property name="toplink.jdbc.password"
    value="mdp"/>
</properties>
```

R. Grin

JPA

page 334

Source de données

- ❑ Lorsque JPA est utilisé avec un serveur d'applications, une source de données fournie par le serveur d'applications est le plus souvent utilisée pour obtenir les connexions (voir `DataSource` dans le cours sur JDBC)
- ❑ On pourra alors trouver :

```
<persistence-unit name="Employe">
  <jta-data-source>jdbc/EmployeDS</jta-data-source>
</persistence-unit name=Employe>
```

R. Grin

JPA

page 335

Configuration dynamique

- ❑ Les informations pour la connexion à la base peuvent n'être connues qu'à l'exécution et pas au moment de l'écriture du code
- ❑ En ce cas, il est possible de ne donner les informations que lors de la création de la fabrique par la classe `Persistence`
- ❑ Il suffit de passer à la méthode `createEntityManagerFactory` une map qui contient les propriétés qui écrasent les valeurs contenues dans `persistence.xml`

R. Grin

JPA

page 336

Exemple

```
// Saisie du nom et du mot de passe
. . .
// Configuration de la connexion
Map props = new HashMap();
props.put("toplink.jdbc.user", nom);
props.put("toplink.jdbc.password", mdp);
EntityManagerFactory emf =
    Persistence
        .createEntityManagerFactory(
            "employes", props);
```

R. Grin

JPA

page 337

Les ajouts de TopLink

- ❑ TopLink a des extensions par rapport à la norme JPA
- ❑ Par exemple,
 - *logging*
 - génération automatique de tables
 - ...

R. Grin

JPA

page 338

Génération automatique des tables

- ❑ La propriété **toplink.ddl-generation** permet de créer automatiquement les tables au moment de la création de la fabrique de gestionnaires d'entités
- ❑ Cette propriété peut avoir 3 valeurs :
 - **none** : aucune création automatique
 - **create-tables** : les tables sont créées si elles n'existent pas déjà
 - **drop-and-create-tables** : si une table existe déjà, elle est d'abord supprimée avant d'être recréée (utile pour les tests)

R. Grin

JPA

page 339

Exemple

```
<properties>
. . .
<property name="toplink.ddl-generation"
    value="drop-and-create-tables"/>
</properties>
```

R. Grin

JPA

page 340

- ❑ **toplink.ddl-generation.output-mode** indique ce qui sera fait avec les fichiers DDL
- ❑ Valeurs possibles :
 - **sql-script** génère les fichiers mais ne les exécute pas
 - **database** exécute les ordres DDL mais ne génère pas les fichiers
 - **both** génère les fichiers et les exécute (valeur par défaut)

R. Grin

JPA

page 341

- ❑ Propriété **toplink.application-location** indique le nom du répertoire qui contiendra les fichiers contenant les ordres DDL de création et de suppression des tables (le répertoire courant par défaut)

R. Grin

JPA

page 342

- ❑ `toplink.create-ddl-jdbc-file-name` indique le nom du fichier qui contiendra les ordres de création des tables ; par défaut, `createDDL.jdbc`
- ❑ `toplink.drop-ddl-jdbc-file-name` idem pour la suppression des tables ; par défaut, `dropDDL.jdbc`

logging (1)

- ❑ La propriété `toplink.logging.level` peut avoir les valeurs
 - **OFF** : aucune information
 - **SEVERE** : uniquement les erreurs
 - **WARNING** : les avertissements (et les erreurs)
 - **INFO** (valeur par défaut) : assez peu d'information en plus
 - **CONFIG** : donne des informations au moment du déploiement sur la configuration

logging (2)

- **FINE** : donne des informations sur les ordres SQL utile pendant les tests et la mise au point
- **FINER** : encore plus d'informations, par exemple sur les transactions
- **FINEST** : encore plus d'informations, par exemple sur l'utilisation des séquences

Fichiers XML

Placement des fichiers XML

- ❑ Par défaut les fichiers XML contenant les méta données sont placées dans le fichier **META-INF/orm.xml**, sous un répertoire du *classpath*
- ❑ Il est possible d'indiquer d'autres emplacements avec le tag `<mapping-file>` dans le fichier **persistence.xml** qui définit l'unité de persistance

Exemple

```
<persistence-unit name="xxxx">
  ...
  <mapping-file>META-INF/queries.xml
</mapping-file>
  <mapping-file>META-INF/entities.xml
</mapping-file>
</persistence-unit>
```

Méta-données par défaut

- ❑ L'élément `<persistence-unit-defaults>` contient des valeurs par défaut qui s'appliquent à toutes les entités de l'unité de persistance
- ❑ Les sous-éléments peuvent être : `<schema>` (donner le nom du schéma relationnel), `<catalog>` (idem `<schema>` pour les SGBD qui supportent les catalogues), `<access>` (accès pour toutes les classes non annotées), `<cascade-persist>` (pour imposer la persistance par transitivité) et `<entity-listeners>` (écouteurs par défaut)

R. Grin

JPA

page 349

Fichiers XML – annotations

- ❑ Avantages des annotations :
 - méta données proches du code
 - pas besoin de donner le contexte comme dans les fichiers XML
 - plus simple
- ❑ Inconvénients :
 - changement des méta données nécessite une recompilation
- ❑ Les informations données dans les fichiers XML l'emportent sur les annotations

R. Grin

JPA

page 350

Fichiers XML – annotations

- ❑ Les annotations sont le plus souvent utilisées
- ❑ Les fichiers de configuration XML peuvent être préférés lorsque l'information
 - est dépendante de l'environnement d'exécution ; par exemple pour les noms de tables ou de colonnes
 - concerne plusieurs classes ; par exemple `@TableGenerator` qui donne le nom de la table qui génère automatiquement les clés d'identification des entités

R. Grin

JPA

page 351

Exemple de fichier orm.xml (en-tête)

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings
xmlns="http://java.sun.com/xml/ns/persistence
ce.orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema
-instance"
xsi:schemaLocation =
"http://java.sun.com/xml/ns/persistence.xml
http://java.sun.com/xml/ns/persistence/orm_
1_0.xsd" version="1.0">
```

R. Grin

JPA

page 352

Exemple de fichier orm.xml (fin)

```
<entity class="jpa.Employe">
  <table name="EMP" />
  <named-query name="findEmpByName"
    <query>select e from Employe e where
e.nom like :nomEmploye</query>
  </named-query>
</entity>
</entity-mappings>
```

R. Grin

JPA

page 353

JPA et DAO

R. Grin

JPA

page 354

Utilité des DAO ?

- ❑ La possibilité de détacher et rattacher une entité rend souvent les DTO inutiles (voir cours sur DAO et modèles de conception associés)
- ❑ Puisque JPA augmente fortement la portabilité du code chargé de la persistance, est-ce que les DAOs sont encore nécessaires ?

R. Grin

JPA

page 355

Les DAOs sont encore utiles

- ❑ Pour les petites applications, avec peu de fonctionnalités « métier », l'utilisation directe de JPA depuis le code métier, sans utilisation de DAOs, simplifie le code lié à la persistance ; les DAO ne sont alors pas vraiment utiles
- ❑ Pour les applications plus complexes, les DAOs peuvent apporter une meilleure abstraction et une meilleure factorisation du code lié à la persistance ; de plus des détails techniques liés à JPA peuvent être cachés

R. Grin

JPA

page 356

Exemple de factorisation du code

- ❑ Un DAO peut contenir des méthodes spécialement adaptées au « métier »
- ❑ Par exemple, une méthode qui renvoie la liste des candidats qui ont obtenu leur examen avec une certaine mention, passée en paramètre
- ❑ Cette méthode peut être appelée de plusieurs endroits du code « métier »

R. Grin

JPA

page 357

Variantes des DAOs

- ❑ Parmi toutes les variantes des méthodes des DAOs (voir cours sur les DAOs), celles qui prennent en paramètre ou qui renvoient des objets (des entités JPA dans ce cas) sont le plus souvent choisies
- ❑ En effet, les entités détachées peuvent jouer le rôle des DTOs qui sont donc inutiles, et les variantes qui utilisent plusieurs paramètres dont les valeurs correspondent aux propriétés des objets compliquent inutilement le code

R. Grin

JPA

page 358

DAOs génériques

- ❑ La généricité permet d'écrire une classe mère générique de tous les DAOs, ce qui allège d'autant les classes filles représentant chacune des classes DAOs
- ❑ Le type des entités gérées et le type de l'identificateur sont les paramètres de type de ce DAO générique
- ❑ Les transparents suivants sont un exemple d'implémentation que vous pouvez modifier à votre convenance

R. Grin

JPA

page 359

Interface DAO générique

```
public interface DaoGenerique<T,ID extends
Serializable> {
    T findById(ID id);
    List<T> findAll();
    void create(T objet);
    void delete(T entite);
    T update(T entite);
}
```

R. Grin

JPA

page 360

Interface DAO pour une entité

```
public interface StyloDao
    extends DaoGenerique<Stylo, Long> { }
```

On peut ajouter des méthodes adaptées au « métier », comme cette méthode qui renvoie toutes les marques de stylos en vente :

```
public interface StyloDao
    extends DaoGenerique<Stylo, Long> {
    List<String> findMarques();
}
```

R. Grin

JPA

page 361

Classe DAO générique (1)

```
public abstract class
    DaoGeneriqueJpa<T, ID extends Serializable>
    implements DaoGenerique<T, ID> {
    private Class<T> classeEntite;
    private EntityManager em;
    public DaoGeneriqueJpa() {
        // classe concrète de T passée à find
        this.classeEntite = (Class<T>)
            ((ParameterizedType)getClass().
                .getGenericSuperclass())
                .getActualTypeArguments()[0];
    }
}
```

R. Grin

JPA

page 362

Classe DAO générique (2)

```
public void insert(T objet) {
    em.persist(objet);
}
public void delete(T objet) {
    em.remove(objet);
}
public T findById(ID id) {
    return em.find(classeEntite, id);
}
public T update(T entite) {
    return em.merge(entite);
}
```

R. Grin

JPA

page 363

Classe DAO générique (3)

```
public void
    setEntityManager(EntityManager em) {
    this.em = em;
}
protected EntityManager
    getEntityManager() {
    return this.em;
}
}
```

R. Grin

JPA

page 364

DAO pour une entité

```
public class StyloDaoJpa extends
    DaoGeneriqueJpa<Stylo, Long>
    implements StyloDao {
    public List<Stylo> findAll() {
        Query query = getEntityManager()
            .createNamedQuery("Stylo.findAll");
        return
            (List<Stylo>)query.getResultList();
    }
}
```

R. Grin

JPA

page 365

Cache des données

R. Grin

JPA

page 366

Utilité

- ❑ Tous les fournisseurs de persistance utilisent un cache de second niveau (en plus du contexte de persistance) pour éviter des accès aux bases de données, et donc améliorer les performances
- ❑ La spécification JPA ne donne aucune indication sur les caches de 2^{ème} niveau ; il faut consulter la documentation du fournisseur de persistance pour connaître les possibilités

R. Grin

JPA

page 367

Utilisation des caches

- ❑ Il n'est pas toujours facile d'utiliser correctement les caches de 2^{ème} niveau
- ❑ Le plus souvent il est possible d'indiquer quelles classes utiliseront ces caches, et sur quel mode
- ❑ Par exemple, il peut être très intéressant d'utiliser un cache « *read-only* » pour des classes dont les données ne changent que très rarement comme les noms des pays

R. Grin

JPA

page 368

Problèmes éventuels

- ❑ Parfois ces caches posent des problèmes si la base de données est utilisée en parallèle par d'autres applications
- ❑ Par exemple, si des entités liées à une entité sont supprimées de la base en dehors de l'application, le cache peut penser modifier une ligne avec un UPDATE au lieu d'en insérer une nouvelle avec un INSERT

R. Grin

JPA

page 369

Solutions (1)

- ❑ L'utilisation de la méthode **refresh** de **EntityManager** permet de récupérer dans la base de données des données « fraîches », sans passer par le cache
- ❑ Un blocage pessimiste peut parfois être la solution si on souhaite interdire la modification de lignes par une autre application (implémentation dépendante du fournisseur de persistance)

R. Grin

JPA

page 370

Solutions (2)

- ❑ Parfois la seule solution est de vider le cache ou d'indiquer au fournisseur de ne pas utiliser le cache pour effectuer certaines opérations (implémentation dépendante du fournisseur de persistance)

R. Grin

JPA

page 371

Optimisations

R. Grin

JPA

page 372

- ❑ Les performances d'une application peuvent être grandement améliorées par
 - un choix adapté du mode de récupération des entités associées
 - l'utilisation d'opérations de modifications en volume, sans création d'entités
 - une bonne utilisation du cache de 2^{ème} niveau

R. Grin

JPA

page 373

Méthode *callback* et *listener*

R. Grin

JPA

page 374

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 « **mappedSuperclass** » ou à une classe « écouteur » (*listener*)
- ❑ Une méthode peut être annotée par plusieurs de ces annotations

R. Grin

JPA

page 375

Annotations

- ❑ **@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

R. Grin

JPA

page 376

Callback dans fichier XML

- ❑ Les méthodes callback peuvent aussi être indiquées dans un fichier XML
- ❑ Un fichier XML peut aussi indiquer des *listeners* par défaut qui seront appelées pour toutes les entités, dans le sous-élément **<entity-listeners>** de l'élément **<persistence-unit-defaults>**

R. Grin

JPA

page 377

Exemples d'utilisation

- ❑ Un trigger de la base de données peut donner les valeurs de l'utilisateur qui a modifié pour la dernière fois une entité, avec la date de cette modification
- ❑ Pour remplacer un tel trigger, il est possible d'ajouter une méthode annotée par **@PrePersist** qui remplit ces valeurs dans les entités
- ❑ Une méthode callback peut aussi initialiser des attributs non persistants d'une entité

R. Grin

JPA

page 378

Callback dans une entité

- ❑ Ces méthodes ne doivent aucune paramètre et le type retour doit être **void**
- ❑ Elles ne doivent pas avoir de clause **throws**
- ❑ Si elles renvoient une exception non contrôlée, les éventuelles méthodes callbacks suivantes ne sont pas appelées et la transaction est marquée pour un *rollback* (elle ne pourra pas être validée)
- ❑ Une seule méthode callback d'un certain type (par exemple **PrePersist**) par entité

R. Grin

JPA

page 379

Callback dans un écouteur

- ❑ La signature doit avoir un paramètre compatible avec le type de l'entité gérée pour qu'il puisse contenir l'instance de l'entité
- ❑ « Compatible » signifie que le type doit être un « surtype » de la classe entité : la classe de l'entité, ou une classe mère, ou une interface implémentée par la classe de l'entité
- ❑ Une classe *listener* ne doit contenir aucune variable d'instance (elle doit être « sans état ») et doit avoir un constructeur sans paramètre

R. Grin

JPA

page 380

Attacher un écouteur

- ❑ L'annotation **@EntityListeners** permet d'attacher un ou plusieurs écouteurs à une classe entité
- ❑ Exemple :

```
@Entity
@EntityListeners({C1.class, C2.class})
public class Entite {
```

R. Grin

JPA

page 381

Ordre d'appel

- ❑ Lorsque un événement du cycle de vie survient, les différentes méthodes sont appelées dans cet ordre :
 1. Méthodes des listeners par défaut
 2. Méthodes des listeners
 3. Méthode de l'entité en cause

R. Grin

JPA

page 382

Bibliographie

R. Grin

JPA

page 383

Sites Web (1)

- ❑ Spécification officielle de JPA :
<http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html> (cliquer sur le 1^{er} bouton « download the Specification for evaluation or to build an application that uses the Specification »)
- ❑ Toplink :
<http://www.oracle.com/technology/products/ias/toplink/jpa/index.html>

R. Grin

JPA

page 384

Sites Web (2)

- ❑ Hibernate :
http://www.hibernate.org/hib_docs/entitymanager/reference/en/html/
- ❑ Kodo :
http://e-docs.bea.com/kodo/docs40/full/html/ejb3_overview.html

R. Grin

JPA

page 385

Livres

- ❑ Pro EJB 3 – JPA
de Mike Keith et Merrick Schincariol
Edition Apress
(en anglais)
- ❑ Java Persistence with Hibernate
de Christian Bauer et Gavin King
Edition Manning
(en anglais)

R. Grin

JPA

page 386