Système d'Information à Objets : Java Persistence API 2nde partie

http://ocaron.polytech-lille.net

Informatique et Statistique 4ème année par apprentissage

Olivier Caron[1]

[1]École d'ingénieurs Polytech Lille
Université de Lille

27 février 2025

© Auteur inconnu

Il y a 10 sortes de personnes, ceux qui comprennent le binaire et ceux qui ne le comprennent pas.

Programmation d'entités JPA au sein de classes services

Programmation d'entités JPA au sein de classes services

Structuration logicielle :

Programmation d'entités JPA au sein de classes services

Structuration logicielle :

- Couche des objets métiers :
  obtenue grâce aux entités JPA

Programmation d'entités JPA au sein de classes services

Structuration logicielle :

- Couche des objets métiers : obtenue grâce aux entités JPA
- Couche des services métiers : obtenue par des classes qui exploitent un **gestionnaire d'entités** pour assurer la persistance des entités

Programmation d'entités JPA au sein de classes services

Structuration logicielle :

- Couche des objets métiers : obtenue grâce aux entités JPA
- Couche des services métiers : obtenue par des classes qui exploitent un **gestionnaire d'entités** pour assurer la persistance des entités

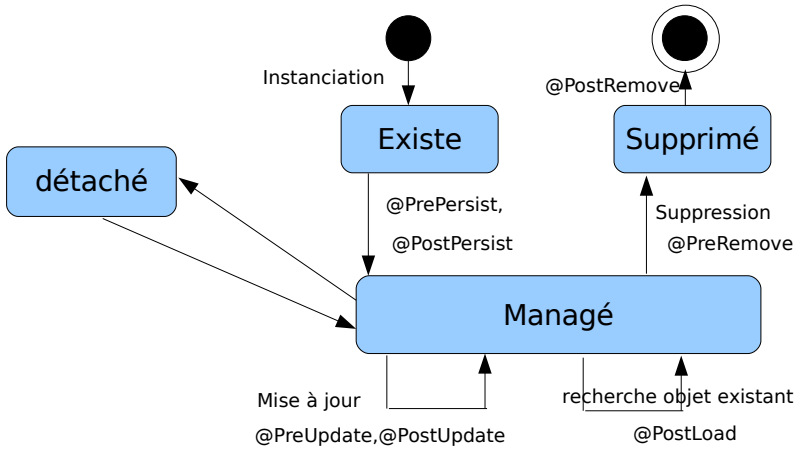
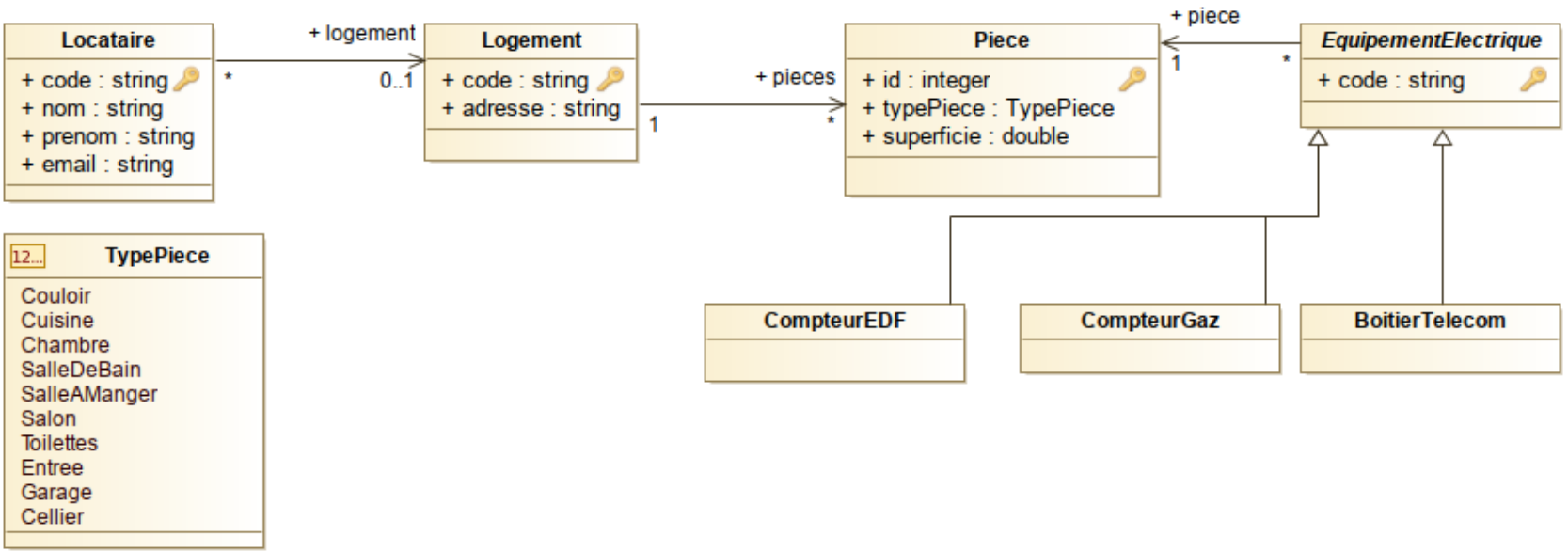


Figure – Cycle de vie des entités JPA

Exemple entités JPA



Gestion d'une agence immobilière

L'agence gère des locataires affectés à un logement. Les logements disposent de pièces. Des équipements nécessaires à la gestion (relevés de compteur) sont localisés dans les pièces des logements.

Exemple persistence.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
6
7 <persistence-unit name="agence-unit" transaction-type="RESOURCE_LOCAL">
8   <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9
10  <class>jpa.entites.Locataire</class>
11  <class>jpa.entites.Logement</class>
12  ...
13
14  <properties>
15    <property name="jakarta.persistence.jdbc.user" value="demo" />
16    <property name="jakarta.persistence.jdbc.password" value="postgres" />
17    <property name="jakarta.persistence.jdbc.driver"
18              value="org.postgresql.Driver" />
19    <property name="jakarta.persistence.jdbc.url"
20              value="jdbc:postgresql://localhost/agence-immo" />
21    <!-- Configuration [4] -->
22    <!-- <property name="eclipselink.logging.level" value="INFO" /> -->
23    <!-- EclipseLink should create the database schema automatically : -->
24    <property name="jakarta.persistence.schema-generation.database.action"
25              value="create"/>
26  </properties>
27 </persistence-unit>
28 </persistence>
```

Le gestionnaire d'entités : jakarta.persistence.EntityManager

- Assure le lien entre objets Java et données BD, initialisation :

```
1  private EntityManager em ;
2
3  public ServiceImmobilier() {
4    EntityManagerFactory emf ;
5    emf = Persistence.createEntityManagerFactory("agence-unit");
6    this.em = emf.createEntityManager();
7  }
```

Le gestionnaire d'entités : jakarta.persistence.EntityManager

- Assure le lien entre objets Java et données BD, initialisation :

```
1  private EntityManager em ;
2
3  public ServiceImmobilier() {
4    EntityManagerFactory emf ;
5    emf = Persistence.createEntityManagerFactory("agence-unit");
6    this.em = emf.createEntityManager();
7  }
```

- Ligne 5 : "**agence-unit**" conforme à la ligne 7 de `persistence.xml`.

Le gestionnaire d'entités : jakarta.persistence.EntityManager

- Assure le lien entre objets Java et données BD, initialisation :

```
1  private EntityManager em ;
2
3  public ServiceImmobilier() {
4    EntityManagerFactory emf ;
5    emf = Persistence.createEntityManagerFactory("agence-unit");
6    this.em = emf.createEntityManager();
7  }
```

- Ligne 5 : "**agence-unit**" conforme à la ligne 7 de `persistence.xml`.
- Le gestionnaire d'entités sera alors relié à la source de données spécifiée par les lignes 15-18 de `persistence.xml`.

Le gestionnaire d'entités : jakarta.persistence.EntityManager

- Assure le lien entre objets Java et données BD, initialisation :

```
1   private EntityManager em ;
2
3   public ServiceImmobilier() {
4     EntityManagerFactory emf ;
5     emf = Persistence.createEntityManagerFactory("agence-unit");
6     this.em = emf.createEntityManager();
7   }
```

- Ligne 5 : "**agence-unit**" conforme à la ligne 7 de `persistence.xml`.
- Le gestionnaire d'entités sera alors relié à la source de données spécifiée par les lignes 15-18 de `persistence.xml`.
- Seules les classes spécifiées par ce descripteur seront prises en compte (lignes 10-12 - `persistence.xml`)

Le gestionnaire d'entités : jakarta.persistence.EntityManager

- Assure le lien entre objets Java et données BD, initialisation :

```
1   private EntityManager em ;
2
3   public ServiceImmobilier() {
4     EntityManagerFactory emf ;
5     emf = Persistence.createEntityManagerFactory("agence-unit");
6     this.em = emf.createEntityManager();
7   }
```

- Ligne 5 : "**agence-unit**" conforme à la ligne 7 de `persistence.xml`.
- Le gestionnaire d'entités sera alors relié à la source de données spécifiée par les lignes 15-18 de `persistence.xml`.
- Seules les classes spécifiées par ce descripteur seront prises en compte (lignes 10-12 - `persistence.xml`)
- Création, suppression des tables associées selon le mode défini lignes 24-25 - `persistence.xml` (valeurs possibles : `none`, `create`, `drop-and-create`, `drop`)

Les transactions

- Le gestionnaire d'entités intègre un moniteur transactionnel.

Les transactions

- Le gestionnaire d'entités intègre un moniteur transactionnel.
- Requis pour toute opération de modification.

Les transactions

- Le gestionnaire d'entités intègre un moniteur transactionnel.
- Requis pour toute opération de modification.
- 3 méthodes de base : `begin`, `commit` et `rollback`

Les transactions

- Le gestionnaire d'entités intègre un moniteur transactionnel.
- Requis pour toute opération de modification.
- 3 méthodes de base : `begin`, `commit` et `rollback`
- Illustration :

```
1  try {
2    em.getTransaction().begin();
3    // Opérations à insérer ici
4    em.getTransaction().commit();
5  }
6  finally {
7    if (em.getTransaction().isActive())
8          em.getTransaction().rollback();
9  }
```

Les transactions

- Le gestionnaire d'entités intègre un moniteur transactionnel.
- Requis pour toute opération de modification.
- 3 méthodes de base : `begin`, `commit` et `rollback`
- Illustration :

```
1  try {
2    em.getTransaction().begin();
3    // Opérations à insérer ici
4    em.getTransaction().commit();
5  }
6  finally {
7    if (em.getTransaction().isActive())
8          em.getTransaction().rollback();
9  }
```

- Les modifications sont répercutées en base à la fin de la transaction réussie

Fonctionnalités de base du gestionnaire d'entités

Fonction	Description	équivalent BD
`find`	Recherche d'instance en base guidée par clé primaire	`select`
`persist`	Ajout d'une instance en base	`insert`
`merge`	Synchronisation objet Java→ objet BD	`insert` et `update`
`flush`	Écriture des modifications en base avant fin de transaction	`insert`,`update` et `delete`
`remove`	Suppression d'instance en base	`delete`
`query`	Actions sur groupe d'objets en base	`select`,`update` et `delete`

EntityManager : récupération d'un objet BD ➔objet Java

- Pré-requis : l'objet existe dans la base.

EntityManager : récupération d'un objet BD ➔objet Java

- Pré-requis : l'objet existe dans la base.
- Méthode `find`, deux paramètres requis :

EntityManager : récupération d'un objet BD ➔objet Java

- Pré-requis : l'objet existe dans la base.
- Méthode `find`, deux paramètres requis :
  1. le type de la classe `@Entity` recherchée

EntityManager : récupération d'un objet BD ➔objet Java

- Pré-requis : l'objet existe dans la base.
- Méthode `find`, deux paramètres requis :
  1. le type de la classe `@Entity` recherchée
  2. la valeur de la clé primaire (`@Id`)

EntityManager : récupération d'un objet BD ➔objet Java

- Pré-requis : l'objet existe dans la base.
- Méthode `find`, deux paramètres requis :
  1. le type de la classe `@Entity` recherchée
  2. la valeur de la clé primaire (`@Id`)
- l'instance récupérée se trouve à l'état **géré**

EntityManager : récupération d'un objet BD ➔objet Java

- Pré-requis : l'objet existe dans la base.
- Méthode `find`, deux paramètres requis :
  1. le type de la classe `@Entity` recherchée
  2. la valeur de la clé primaire (`@Id`)
- l'instance récupérée se trouve à l'état **géré**
- renvoie `null` si l'instance n'existe pas dans la base

```
Locataire loc ;
loc = (Locataire) em.find(Locataire.class,"loc227");
if (loc==null)
  System.err.println("locataire loc227 inconnu");
else loc.setEmail("jdupont@laposte.net");
// sera modifié en base car état géré
```

EntityManager : insertion d'un objet dans la base

- Pré-requis : l'objet n'existe **pas** dans la base.

```
Logement log = new Logement(); log.setCode("LD_221b_BS");
log.setAdresse("221b Baker Street, london");
em.persist(log);
```

EntityManager : insertion d'un objet dans la base

- Pré-requis : l'objet n'existe **pas** dans la base.

```
Logement log = new Logement(); log.setCode("LD_221b_BS");
log.setAdresse("221b Baker Street, london");
em.persist(log);
```

- Méthode `persist(Object instance)`

EntityManager : insertion d'un objet dans la base

- Pré-requis : l'objet n'existe **pas** dans la base.

```
Logement log = new Logement(); log.setCode("LD_221b_BS");
log.setAdresse("221b Baker Street, london");
em.persist(log);
```

- Méthode `persist(Object instance)`
- Exception `java.lang.IllegalArgumentException` si l'objet n'est pas une entité

état de l'instance avant	**après**
état nouveau	état géré
état géré	état géré (pas de changement)
état détaché/non géré	exception **lors** du commit : (jakarta.persistence.RollbackException)
état supprimé	état géré

EntityManager : suppression d'un objet entité

```
em.remove(log);
```

- Méthode `remove(Object instance)` :

état de l'instance avant	**après**
état nouveau	action ignorée
état géré	état supprimé
état détaché	une exception intervient : (java.lang.IllegalArgumentException)
état supprimé	action ignorée

EntityManager : fusion d'un objet entité

```
em.merge(log);
```

- Propagation de l'état détaché vers la base (pour les attributs autre que la clé primaire)

EntityManager : fusion d'un objet entité

```
em.merge(log);
```

- Propagation de l'état détaché vers la base (pour les attributs autre que la clé primaire)
- Méthode `merge(Object instance)` :

état de l'instance avant	**après**
état nouveau	état géré (équivalent `persist`)
état géré	action ignorée (déjà synchronisé)
état détaché	propagation base, état géré
état supprimé	action ignorée

EntityManager : Mise à jour des écritures en base

```
em.flush();
```

- Les opérations telles que `persist`, `remove`, `merge` ont pour effet de modifier la base mais ces écritures en base ne sont réalisées qu'à la fin de la transaction

EntityManager : Mise à jour des écritures en base

```
em.flush();
```

- Les opérations telles que `persist`, `remove`, `merge` ont pour effet de modifier la base mais ces écritures en base ne sont réalisées qu'à la fin de la transaction
- La commande `flush()` provoque ces écritures en base

EntityManager : Mise à jour des écritures en base

```
em.flush();
```

- Les opérations telles que `persist`, `remove`, `merge` ont pour effet de modifier la base mais ces écritures en base ne sont réalisées qu'à la fin de la transaction
- La commande `flush()` provoque ces écritures en base
- Peut donc déclencher des exceptions SQL (duplication de clé primaire, etc) qu'il est alors possible de capturer.

EntityManager : pour finir

```
if (em.contains(e1)) System.out.println("e1 is managed") ;
...
em.clear();
...
em.close();
```

- `contains` : indique si l'objet paramètre est dans l'état géré.

EntityManager : pour finir

```
if (em.contains(e1)) System.out.println("e1 is managed") ;
...
em.clear();
...
em.close();
```

- `contains` : indique si l'objet paramètre est dans l'état géré.
- `clear` : toutes les entités gérées passent à l'état détaché

EntityManager : pour finir

```
if (em.contains(e1)) System.out.println("e1 is managed") ;
...
em.clear();
...
em.close();
```

- `contains` : indique si l'objet paramètre est dans l'état géré.
- `clear` : toutes les entités gérées passent à l'état détaché
- `close` : fermeture de la connexion à la base (libération de la charge mémoire)

EntityManager : Requêtes JPQL

- Un langage de requêtes défini dans la norme : JPQL

EntityManager : Requêtes JPQL

- Un langage de requêtes défini dans la norme : JPQL
- JPQL se rapproche de plus en plus de SQL : introduction de group by, having, sous-requêtes, etc dans la dernière version.

EntityManager : Requêtes JPQL

- Un langage de requêtes défini dans la norme : JPQL
- JPQL se rapproche de plus en plus de SQL : introduction de group by, having, sous-requêtes, etc dans la dernière version.
- JPQL et SQL possèdent pratiquement la même syntaxe.

EntityManager : Requêtes JPQL

- Un langage de requêtes défini dans la norme : JPQL
- JPQL se rapproche de plus en plus de SQL : introduction de group by, having, sous-requêtes, etc dans la dernière version.
- JPQL et SQL possèdent pratiquement la même syntaxe.
- Les jointures JPQL sont conformes SQL-2 (left, right, outer join)

EntityManager : Requêtes JPQL

- Un langage de requêtes défini dans la norme : JPQL
- JPQL se rapproche de plus en plus de SQL : introduction de group by, having, sous-requêtes, etc dans la dernière version.
- JPQL et SQL possèdent pratiquement la même syntaxe.
- Les jointures JPQL sont conformes SQL-2 (left, right, outer join)
- Introduction des requêtes update et delete

EntityManager : Requêtes JPQL

- Un langage de requêtes défini dans la norme : JPQL
- JPQL se rapproche de plus en plus de SQL : introduction de group by, having, sous-requêtes, etc dans la dernière version.
- JPQL et SQL possèdent pratiquement la même syntaxe.
- Les jointures JPQL sont conformes SQL-2 (left, right, outer join)
- Introduction des requêtes update et delete
- **JPQL exprime des requêtes sur les objets Java pas sur les objets de la base**

EntityManager : programmation des requêtes

```
1 String reqJPQL = "select ..." ;      // définition de la requête
2 Query q = em.createQuery(reqJPQL) ; // création requête
3 q.setParameter("nomPar1", valeur) ; // optionel :
4 q.setParameter("nomPar2", valeur) ; // définir paramètres
5 Object result = q.getResultList();  // exécution et résultat
```

- Méthode : `createQuery(String requeteJPQL)`, fournit un objet de type query (ligne 2)

EntityManager : programmation des requêtes

```
1 String reqJPQL = "select ..." ;      // définition de la requête
2 Query q = em.createQuery(reqJPQL) ; // création requête
3 q.setParameter("nomPar1", valeur) ; // optionel :
4 q.setParameter("nomPar2", valeur) ; // définir paramètres
5 Object result = q.getResultList();  // exécution et résultat
```

- Méthode : `createQuery(String requeteJPQL)`, fournit un objet de type `query` (ligne 2)
- Possibilité de fournir des valeurs aux éventuels paramètres de la requête JPQL via la méthode `setParameter` (lignes 3-4)

EntityManager : programmation des requêtes

```
1 String reqJPQL = "select ..." ;      // définition de la requête
2 Query q = em.createQuery(reqJPQL) ; // création requête
3 q.setParameter("nomPar1", valeur) ; // optionel :
4 q.setParameter("nomPar2", valeur) ; // définir paramètres
5 Object result = q.getResultList();  // exécution et résultat
```

- Méthode : `createQuery(String requeteJPQL)`, fournit un objet de type `query` (ligne 2)
- Possibilité de fournir des valeurs aux éventuels paramètres de la requête JPQL via la méthode `setParameter` (lignes 3-4)
- Selon la requête, l'exécution fournit :

EntityManager : programmation des requêtes

```
1 String reqJPQL = "select ..." ;      // définition de la requête
2 Query q = em.createQuery(reqJPQL) ; // création requête
3 q.setParameter("nomPar1", valeur) ; // optionel :
4 q.setParameter("nomPar2", valeur) ; // définir paramètres
5 Object result = q.getResultList();  // exécution et résultat
```

- Méthode : `createQuery(String requeteJPQL)`, fournit un objet de type `query` (ligne 2)
- Possibilité de fournir des valeurs aux éventuels paramètres de la requête JPQL via la méthode `setParameter` (lignes 3-4)
- Selon la requête, l'exécution fournit :
  - une unique ligne résultat via la méthode `getSingleResult()`

EntityManager : programmation des requêtes

```
1 String reqJPQL = "select ..." ;      // définition de la requête
2 Query q = em.createQuery(reqJPQL) ; // création requête
3 q.setParameter("nomPar1", valeur) ; // optionel :
4 q.setParameter("nomPar2", valeur) ; // définir paramètres
5 Object result = q.getResultList();  // exécution et résultat
```

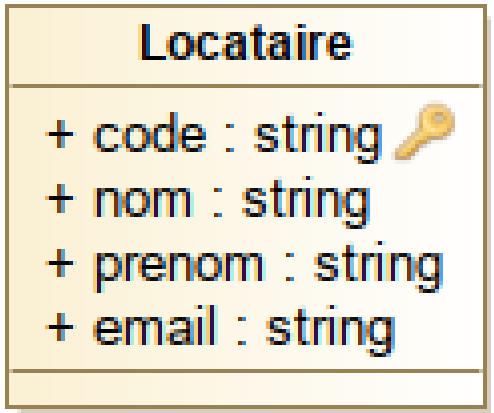
- Méthode : `createQuery(String requeteJPQL)`, fournit un objet de type `query` (ligne 2)
- Possibilité de fournir des valeurs aux éventuels paramètres de la requête JPQL via la méthode `setParameter` (lignes 3-4)
- Selon la requête, l'exécution fournit :
  - une unique ligne résultat via la méthode `getSingleResult()`
  - une collection de lignes résultats via la méthode `getResultList()` (ligne 5).

EntityManager : programmation des requêtes

```
1 String reqJPQL = "select ..." ;      // définition de la requête
2 Query q = em.createQuery(reqJPQL) ; // création requête
3 q.setParameter("nomPar1", valeur) ; // optionel :
4 q.setParameter("nomPar2", valeur) ; // définir paramètres
5 Object result = q.getResultList();  // exécution et résultat
```

- Méthode : `createQuery(String requeteJPQL)`, fournit un objet de type `query` (ligne 2)
- Possibilité de fournir des valeurs aux éventuels paramètres de la requête JPQL via la méthode `setParameter` (lignes 3-4)
- Selon la requête, l'exécution fournit :
  - une unique ligne résultat via la méthode `getSingleResult()`
  - une collection de lignes résultats via la méthode `getResultList()` (ligne 5).
- Variable `result` devra être convertie selon le bon type (ligne 5).

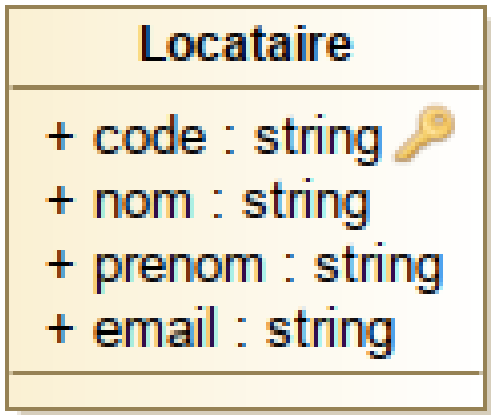
EntityManager : requête mono-table paramétrée (version 1)



- Ex : liste des locataires filtrés par leur nom.
- JPQL : langage de requêtes sur objets **Java** : classe `Locataire` et attribut `nom` (ligne 3).
- Cette requête retourne une collection de locataires, **cast explicite** (lignes 1 et 8)

```
1  @SuppressWarnings("unchecked")
2  public Collection<Locataire>
3          getLocatairesParNom(String debutMot) {
4    String reqJPQL="select loc from Locataire loc "+
5                   " where loc.nom like :pattern" ;
6    Query q=em.createQuery(reqJPQL) ;
7    q.setParameter("pattern",debutMot+"%");
8    return (Collection<Locataire>) q.getResultList();
9  }
```

EntityManager : requête mono-table paramétrée (version 2)

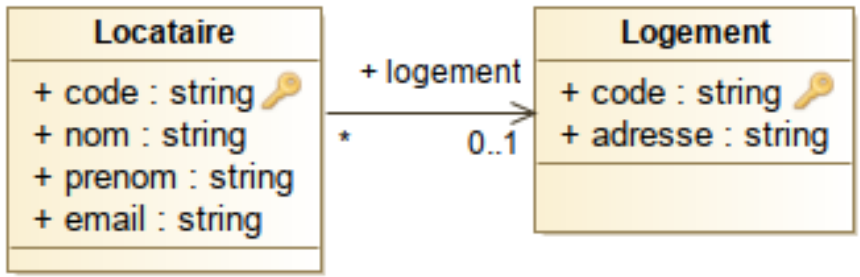


- La même requête que la diapositive précédente mais :
- Utilisation de `TypedQuery` et ajout paramètre à `createQuery` (ligne 4)
- Ne fonctionne que pour requêtes multi-tables avec jointures explicites.
- L'annotation `@SuppressWarnings` peut être retirée.

```
1  public Collection<Locataire>
2         getLocatairesParNomV2(String debutMot){
3   String reqJPQL="select loc from Locataire loc "+
4                  " where loc.nom like :pattern" ;
5   TypedQuery<Locataire> q=em.createQuery(reqJPQL,Locataire.class) ;
6   q.setParameter("pattern",debutMot+"%");
7   return (Collection<Locataire>) q.getResultList();
8  }
```

EntityManager : requête jointure - version 1

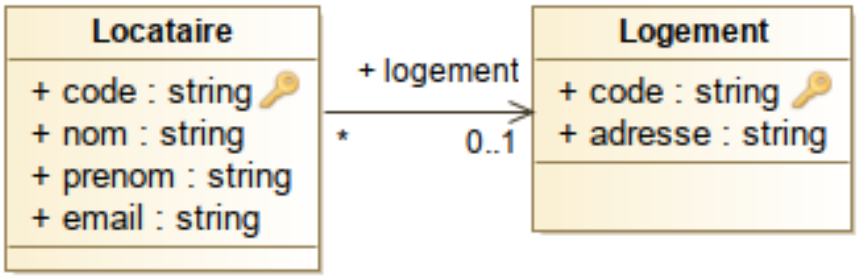
- Ex : liste des logements occupés.
- Sol 1 : navigation par rôle (jointure non explicite)
- Syntaxe JPQL (ex ligne 4) : `variable.nomRole`




```
1  @SuppressWarnings("unchecked")
2  @Override
3  public Collection<Logement> getLogementsOccupesV1() {
4   String reqJPQL="select loc.logement from Locataire loc" ;
5   Query q=em.createQuery(reqJPQL);
6   return (Collection<Logement>)q.getResultList();
7  }
```

EntityManager : requête jointure - version 2

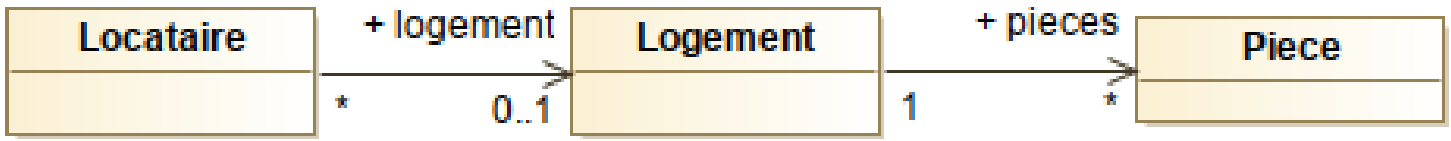
- Ex : liste des logements occupés.
- Sol 2 : opérateur `join`
- Syntaxe JPQL (ex ligne 4) :
  `var1 join var1.role var2`
- Jointure interne, externe,...
- Jointure explicite : `TypedQuery` possible




```
1  public Collection<Logement> getLogementsOccupesV2() {
2   String reqJPQL="select logementsOccupes from Locataire "+
3                  "loc join loc.logement logementsOccupes"   ;
4   TypedQuery<Logement> q=em.createQuery(reqJPQL, Logement.class);
5   return (Collection<Logement>)q.getResultList();
6  }
```

EntityManager : requête multi jointures - version 1

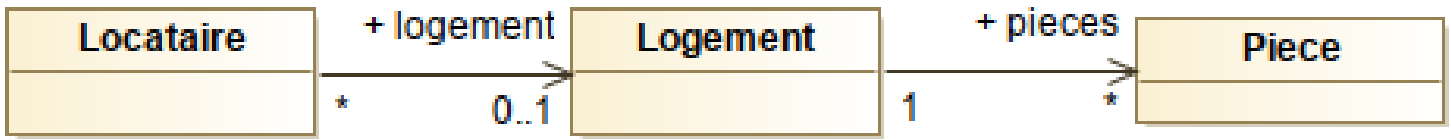
- Ex : liste des pièces des logements occupés.
- Sol 1 : navigation par rôles




```
1  @SuppressWarnings("unchecked")
2  public Collection<Piece> getPiecesLogementsOccupesV1() {
3    String reqJPQL="select loc.logement.pieces from Locataire loc" ;
4    Query q=em.createQuery(reqJPQL);
5    return (Collection<Piece>) q.getResultList();
6  }
```

EntityManager : requête multi jointures - version 2

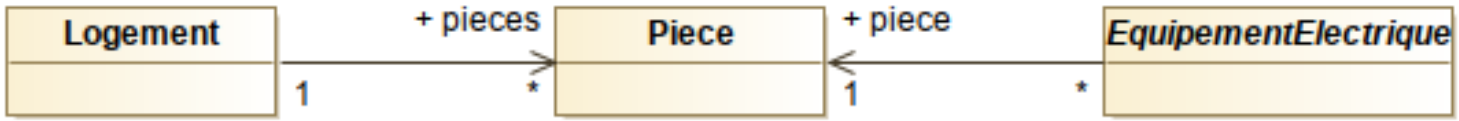
- Ex : liste des pièces des logements occupés.
- Sol 2 : opérateur `join`




```
1  @SuppressWarnings("unchecked")
2  public Collection<Piece> getPiecesLogementsOccupesV2() {
3    String reqJPQL="select pieces from Locataire loc "+
4                   "join loc.logement logementsOccupes "+
5                   "join logementsOccupes.pieces pieces";
6    TypedQuery<Piece> q=em.createQuery(reqJPQL,Piece.class);
7    return (Collection<Piece>) q.getResultList();
8  }
```

EntityManager : requête jointure - navigation inverse

- Association non navigable (ex : `Piece` vers `EquipementElectrique`) :



- Unique solution : opérateur `join` et clause `on` :

```
var1 join Entite2 var2 on var2.role=var1
```

```
1  public Collection<EquipementElectrique>
2    getEquipementsDuLogement(String codeLogement)
3      throws LogementInconnuException {
4    Logement log= (Logement) em.find(Logement.class, codeLogement);
5    if (log==null) throw new LogementInconnuException() ;
6    String reqJPQL="select equip from Logement lo join lo.pieces p "+
7      "join EquipementElectrique equip on equip.piece=p where lo.code=:code";
8    TypedQuery<EquipementElectrique> q ;
9    q=em.createQuery(reqJPQL,EquipementElectrique.class) ;
10   q.setParameter("code", codeLogement);
11   return (Collection<EquipementElectrique>) q.getResultList();
12 }
```

EntityManager : requêtes JPQL avec fonctions d'agrégation

- Possibilité d'utiliser les fonctions d'agrégation de SQL
- Récupération d'un seule ligne ou valeur via la méthode `getSingleResult()`
- Prendre en compte `NoResultException` (erreur Runtime)
- Un exemple :

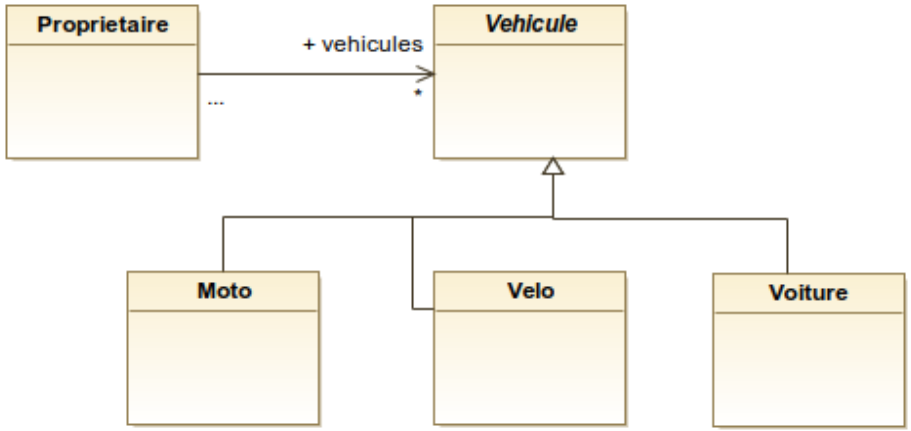
```
1  public double getSuperficieLogementsOccupes()
2    throws NoResultException {
3    String reqJPQL="select sum(p.superficie) "+
4    "from Locataire loc join loc.logement lo join lo.pieces p";
5    Query q=em.createQuery(reqJPQL);
6    return (Double) q.getSingleResult();
7  }
```

EntityManager : multi valeurs

- Possibilité de fixer une liste d'expressions dans la clause `select`
- Récupération des données via `List<Object []>`
- Exemple du nombre de pièces par logement :

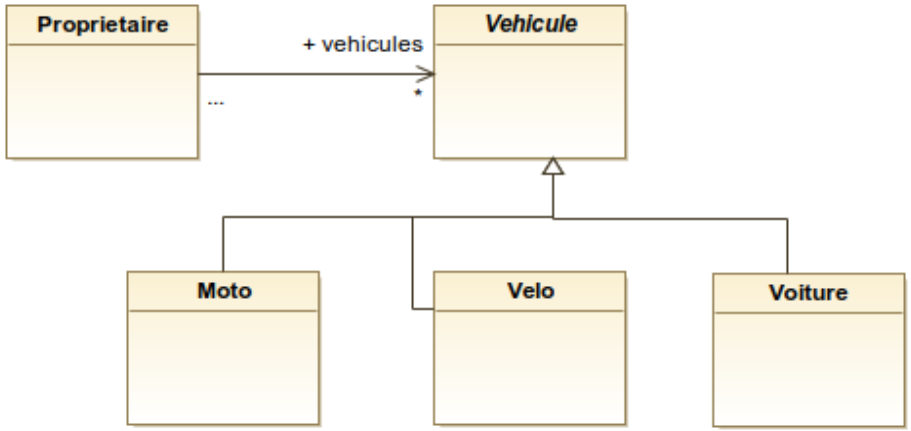
```
String reqJPQL="select lo, count(*) as nombre "+
                "from Logement lo join lo.pieces p group by lo";
Query q=em.createQuery(reqJPQL) ;
List<Object[]> resultat= (List<Object []>) q.getResultList();
System.out.println("Les logements et leur pièces:");
for (Object ligne[] : resultat) {
  Logement lg= (Logement) ligne[0] ;
  long nb = (Long) ligne[1];
  System.out.println("Code "+lg.getCode()+": "+nb+" pièce(s)");
}
```

EntityManager : requête avec gestion héritage



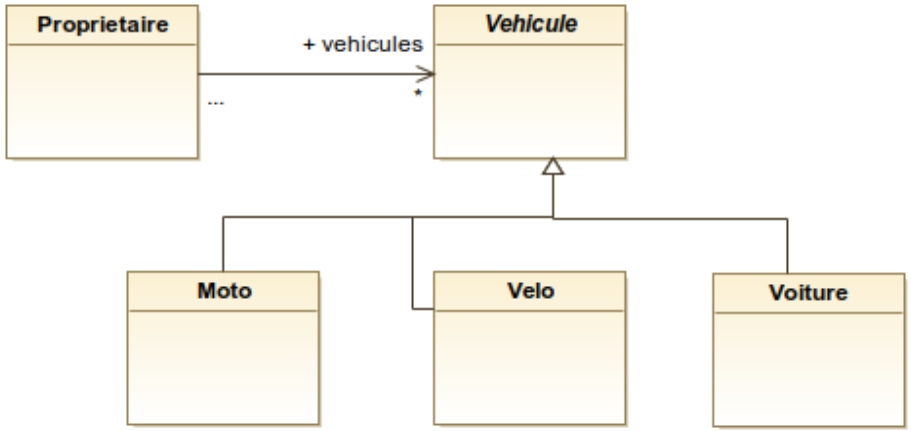
- Obtenir une hiérarchie de classes :
  `select v from Vehicule v`
  On obtient toutes les instances des 3-sous-classes

EntityManager : requête avec gestion héritage



- Obtenir une hiérarchie de classes :
  `select v from Vehicule v`
  On obtient toutes les instances des 3-sous-classes
- Obtenir une sous-classe particulière :
  `select m from Moto m`

EntityManager : requête avec gestion héritage



- Obtenir une hiérarchie de classes :
  `select v from Vehicule v`
  On obtient toutes les instances des 3-sous-classes
- Obtenir une sous-classe particulière :
  `select m from Moto m`
- Obtenir une partie de la hiérarchie de classes :
  `select m from Owner a join a.vehicules m where TYPE(m)=Moto`
  On obtient les seules instances de `Moto`

EntityManager : requêtes JPQL de modification

- Évite le chargement d'objets Java

EntityManager : requêtes JPQL de modification

- Évite le chargement d'objets Java
- Méthode `executeUpdate()`

EntityManager : requêtes JPQL de modification

- Évite le chargement d'objets Java
- Méthode `executeUpdate()`
- Retourne le nombre de ligne(s) concernées par l'opération

EntityManager : requêtes JPQL de modification

- Évite le chargement d'objets Java
- Méthode `executeUpdate()`
- Retourne le nombre de ligne(s) concernées par l'opération
- Transaction obligatoire, peut déclencher des exceptions si contrainte d'intégrité non respectée

```
public int supprimerLocataires()
  throws jakarta.persistence.PersistenceException {
  try {
    em.getTransaction().begin();
    String requete="delete from Locataire" ;
    int nbDel = em.createQuery(requete).executeUpdate() ;
    em.getTransaction().commit();
    return nbDel;
  } finally {
    if (em.getTransaction().isActive())
      em.getTransaction().rollback();
  }
}
```

Programmer efficacement avec JPA

- La force de JPA est de proposer une programmation orienté-objet pour gérer la persistance de données.

Programmer efficacement avec JPA

- La force de JPA est de proposer une programmation orienté-objet pour gérer la persistance de données.
- JPQL permet d'être efficace pour gérer une collection d'objets $\longrightarrow$ évite de charger inutilement des objets Java en mémoire.

Programmer efficacement avec JPA

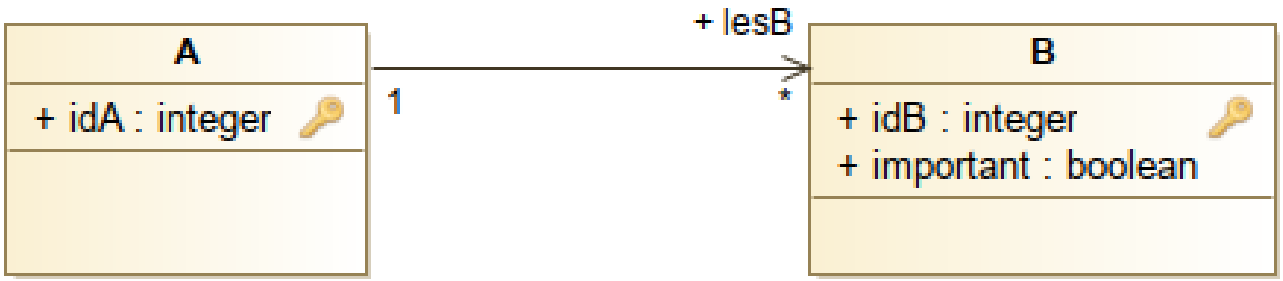
- La force de JPA est de proposer une programmation orienté-objet pour gérer la persistance de données.
- JPQL permet d'être efficace pour gérer une collection d'objets $\longrightarrow$ évite de charger inutilement des objets Java en mémoire.
- Par défaut, JPA propose un mécanisme qui ne charge que les seuls objets dont on a besoin (mécanisme "Lazy"), ce mécanisme est configurable.

Programmer efficacement avec JPA

- La force de JPA est de proposer une programmation orienté-objet pour gérer la persistance de données.
- JPQL permet d'être efficace pour gérer une collection d'objets $\longrightarrow$ évite de charger inutilement des objets Java en mémoire.
- Par défaut, JPA propose un mécanisme qui ne charge que les seuls objets dont on a besoin (mécanisme "Lazy"), ce mécanisme est configurable.
- Sujet à erreurs dans le cas d'applications réparties (objets gérés versus objets détachés)

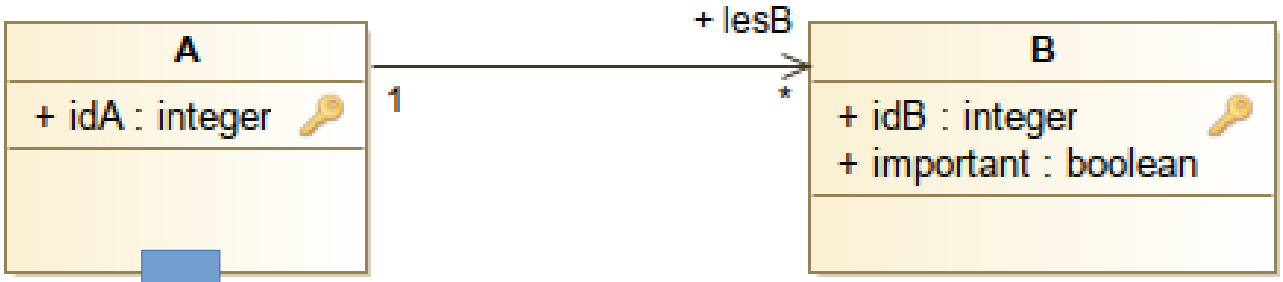
Entités gérées vs détachées (1/4)

Partie entités :



Entités gérées vs détachées (1/4)

Partie entités :




```
@Entity
public class A implements java.io.Serializable {
  @Id private int idA ;
  private @OneToMany Set<B> lesB ;

  public A() {}
   ...
```

Entités gérées vs détachées (2/4)

Partie sessions (chargement des instances **si nécessaire**)

```
@jakarta.ejb.Remote
public interface IGestion {
  /** these methods return null if not exists **/
  public A getAIf_lesB_are_important(int idA)  ;
  public A getA(int idA)  ;
}
```

Entités gérées vs détachées (2/4)

Partie sessions (chargement des instances **si nécessaire**)

```java
@jakarta.ejb.Remote
public interface IGestion {
  /** these methods return null if not exists **/
  public A getAIf_lesB_are_important(int idA)  ;
  public A getA(int idA)  ;
}
```

```java
@Override
public A getAIf_lesB_are_important(int idA) {
  A a = this.getA(idA) ;
  if (a != null) {
    for (B unB : a.getLesB())
      if (! unB.isImportant()) return null ;
    return a ;
  }
  return null ;
}
@Override public A getA(int idA) {
  return (A) em.find(A.class,idA) ;
}
```

Entités gérées vs détachées (2/4)

Partie sessions (chargement des instances **si nécessaire**)

```java
@jakarta.ejb.Remote
public interface IGestion {
  /** these methods return null if not exists **/
  public A getAIf_lesB_are_important(int idA)  ;
  public A getA(int idA)  ;
}
```

```java
@Override
public A getAIf_lesB_are_important(int idA) {
  A a = this.getA(idA) ;
  if (a != null) {
    for (B unB : a.getLesB())
      if (! unB.isImportant()) return null ;
    return a ;
  }
  return null ;
}
@Override public A getA(int idA) {
  return (A) em.find(A.class,idA) ;
}
```

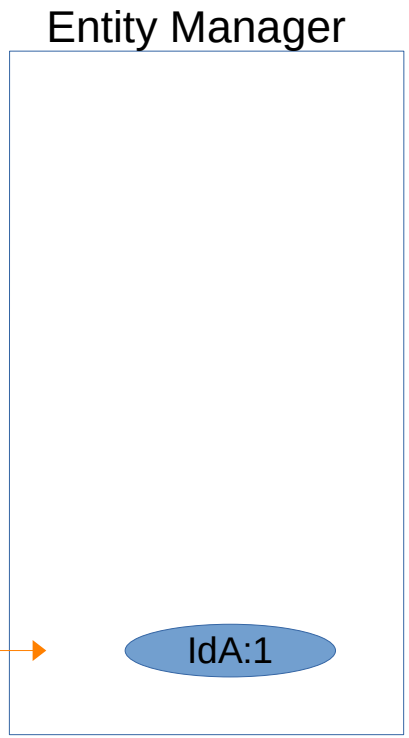
Entity Manager

Entités gérées vs détachées (2/4)

Partie sessions (chargement des instances **si nécessaire**)

```java
@jakarta.ejb.Remote
public interface IGestion {
  /** these methods return null if not exists **/
  public A getAIf_lesB_are_important(int idA)  ;
  public A getA(int idA)  ;
}
```

```java
@Override
public A getAIf_lesB_are_important(int idA) {
  A a = this.getA(idA) ;
  if (a != null) {
    for (B unB : a.getLesB())
      if (! unB.isImportant()) return null ;
    return a ;
  }
  return null ;
}
@Override public A getA(int idA) {
  return (A) em.find(A.class,idA) ;
}
```

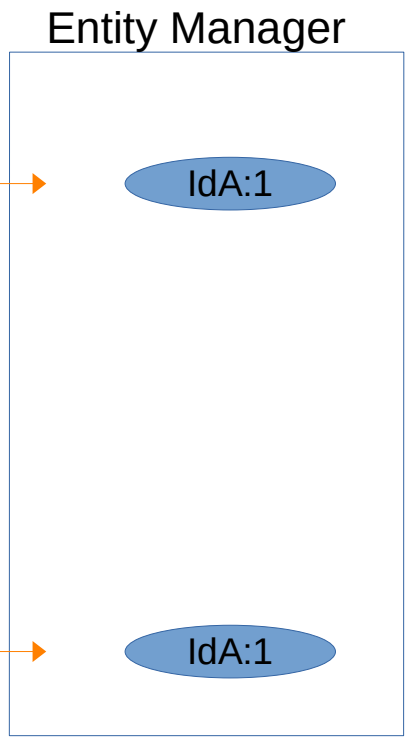


Entités gérées vs détachées (2/4)

Partie sessions (chargement des instances **si nécessaire**)

```java
@jakarta.ejb.Remote
public interface IGestion {
  /** these methods return null if not exists **/
  public A getAIf_lesB_are_important(int idA)  ;
  public A getA(int idA)  ;
}
```

```java
@Override
public A getAIf_lesB_are_important(int idA) {
  A a = this.getA(idA) ;
  if (a != null) {
    for (B unB : a.getLesB())
      if (! unB.isImportant()) return null ;
    return a ;
  }
  return null ;
}
@Override public A getA(int idA) {
  return (A) em.find(A.class,idA) ;
}
```

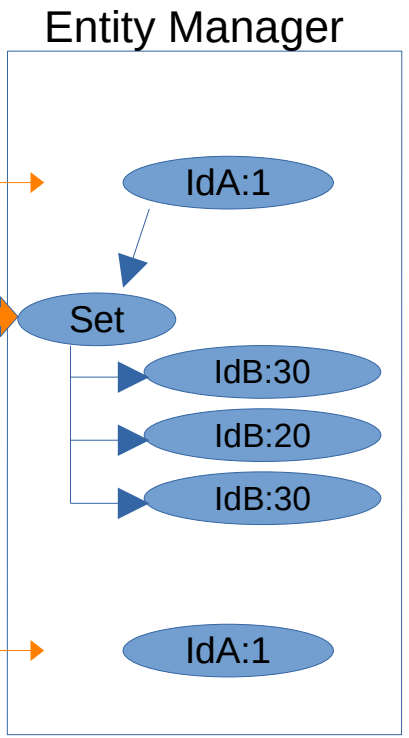


Entités gérées vs détachées (2/4)

Partie sessions (chargement des instances **si nécessaire**)

```java
@jakarta.ejb.Remote
public interface IGestion {
  /** these methods return null if not exists **/
  public A getAIf_lesB_are_important(int idA)  ;
  public A getA(int idA)  ;
}
```

```java
@Override
public A getAIf_lesB_are_important(int idA) {
  A a = this.getA(idA) ;
  if (a != null) {
    for (B unB : a.getLesB())
      if (! unB.isImportant()) return null ;
    return a ;
  }
  return null ;
}
@Override public A getA(int idA) {
  return (A) em.find(A.class,idA) ;
}
```



Entités gérées vs détachées (3/4)

Partie cliente distante : les entités propagées par le réseau sont **détachées**

```
InitialContext ctx = new InitialContext();
System.out.println("Accès au service distant") ;
Object obj = ctx.lookup("ejb:demo/demoSessions//Gestion"+
                        "!ejb.sessions.IGestion");

IGestion service = (IGestion) obj ;
A unA = service.getAIf_lesB_are_important(1) ;
if (unA !=null) {
  System.out.println ("1. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("1. unB.idB="+unB.getIdB()) ;
}
unA = service.getA(1) ;
if (unA !=null) {
  System.out.println ("2. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("2. unB.idB="+unB.getIdB()) ;
}
```

Entités gérées vs détachées (3/4)

Partie cliente distante : les entités propagées par le réseau sont **détachées**

```
InitialContext ctx = new InitialContext();
System.out.println("Accès au service distant") ;
Object obj = ctx.lookup("ejb:demo/demoSessions//Gestion"+
                        "!ejb.sessions.IGestion");

IGestion service = (IGestion) obj ;
A unA = service.getAIf_lesB_are_important(1) ;
if (unA !=null) {
  System.out.println ("1. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("1. unB.idB="+unB.getIdB()) ;
}
unA = service.getA(1) ;
if (unA !=null) {
  System.out.println ("2. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("2. unB.idB="+unB.getIdB()) ;
}
```

```
1. unA.idA=1
1. unB.idB=30
1. unB.idB=20
1. unB.idB=40
```

Entités gérées vs détachées (3/4)

Partie cliente distante : les entités propagées par le réseau sont **détachées**

```
InitialContext ctx = new InitialContext();
System.out.println("Accès au service distant") ;
Object obj = ctx.lookup("ejb:demo/demoSessions//Gestion"+
                        "!ejb.sessions.IGestion");

IGestion service = (IGestion) obj ;
A unA = service.getAIf_lesB_are_important(1) ;
if (unA !=null) {
  System.out.println ("1. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("1. unB.idB="+unB.getIdB()) ;
}
unA = service.getA(1) ;
if (unA !=null) {
  System.out.println ("2. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("2. unB.idB="+unB.getIdB()) ;
}
```

```
1. unA.idA=1
1. unB.idB=30
1. unB.idB=20
1. unB.idB=40
```

```
2. unA.idA=1
```

Entités gérées vs détachées (3/4)

Partie cliente distante : les entités propagées par le réseau sont **détachées**

```
InitialContext ctx = new InitialContext();
System.out.println("Accès au service distant") ;
Object obj = ctx.lookup("ejb:demo/demoSessions//Gestion"+
                        "!ejb.sessions.IGestion");

IGestion service = (IGestion) obj ;
A unA = service.getAIf_lesB_are_important(1) ;
if (unA !=null) {
  System.out.println ("1. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("1. unB.idB="+unB.getIdB()) ;
}
unA = service.getA(1) ;
if (unA !=null) {
  System.out.println ("2. unA.idA="+unA.getIdA()) ;
  for (B unB : unA.getLesB())
    System.out.println ("2. unB.idB="+unB.getIdB()) ;
}
```

```
1. unA.idA=1
1. unB.idB=30
1. unB.idB=20
1. unB.idB=40
```

```
2. unA.idA=1
```

```
Exception in thread "main" org.hibernate.LazyInitializationException: ...
```

Entités gérées vs détachées (4/4)

Niveau Serveur - état géré le mode par défaut d'obtention des instances Java d'entités en mémoire est le mode `LAZY`("paresseux" eq. "load on demand").

Niveau prog. client - état détaché risque de `java.lang.NullPointerException` ou de `org.hibernate.LazyInitialisationException` si les instances téléchargées sont incomplètes.

- Deux solutions :
  1. Forcer le chargement des objets avant l'envoi des données (méthode `getAIf_lesB_are_important`)
  2. Fixer le mode d'acquisition des données en mode `EAGER` :

```
// fichier A.java
...
private @OneToMany(fetch=FetchType.EAGER) Set<B> lesB ;
```

Conclusion JPQL

- Ce cours est une **introduction** à JPQL
- Pouvoir d'expression des requêtes (sous-requêtes, clauses exists, all, any, group by, having, ...)
- Permet d'obtenir une sélection d'objets Java sans charger au préalable en mémoire tous les éléments nécessaires de la requête.
- Le modèle UML-JPA est le guide parfait pour l'élaboration des requêtes JPQL (nom des classes, attributs et rôles)
- Références :

  `https://docs.oracle.com/javaee/7/tutorial/`

  `https://thorben-janssen.com/jpql/`

Le mot de la fin

© Auteur inconnu

Vous ne pouvez pas comprendre la récursivité sans avoir d'abord compris la récursivité.