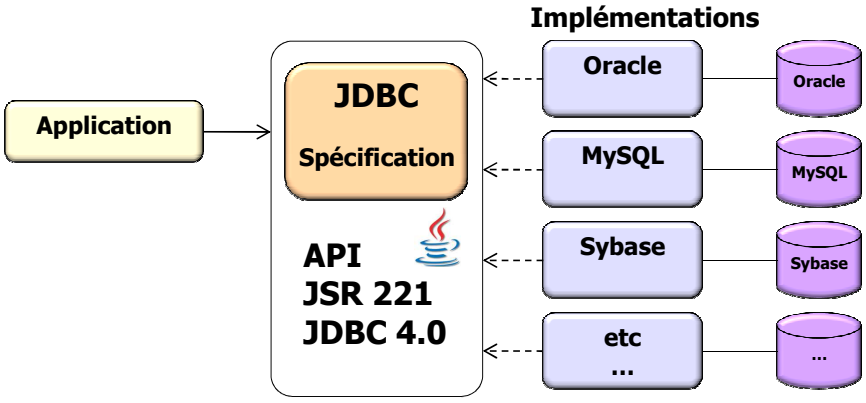
JPA
Java Persistence API

AJC INGÉNIERIE

Introduction

De JDBC à JPA

- Pendant longtemps la plate-forme Java n'a disposé que d'une API de bas niveau pour gérer l'accès aux bases de données relationnelles :
  JDBC (Java DataBase Connectivity)



3

De JDBC à JPA

- **JDBC**
  - Orienté base de données relationnelle
  - Permet d'utiliser des requêtes SQL pour consulter et mettre à jour des « records » dans des « tables »
  - API de « bas niveau » (gestion des connexions, des requêtes SQL, des « ResultSet », etc…)
- Des frameworks de plus haut niveau sont apparus : **Hibernate**, **TopLink**, **iBatis**, …
- Une nouvelle spécification a été ajoutée dans la plate-forme Java : **JDO** (+/- échec)
  - JDO 1.0 – JSR 12 (2002)
  - JDO 2.0 – JSR 243 (fin 2005)

4

De JDBC à JPA

- Sous l'influence des frameworks de type « **ORM** » (Object Relational Mapping) et pour répondre aux nombreuses critiques concernant les EJB, une nouvelle spécification est proposée et adoptée :
  JPA (Java Persistence API)

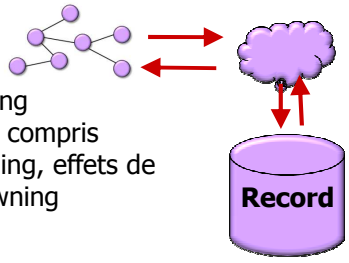
- **JPA 1.0**
  Java EE 5 (JSR 220 / EJB 3.0) – Mai 2006

- **JPA 2.0**
  Java EE 6 (JSR 317) – Déc. 2009

5

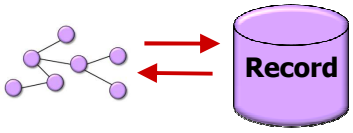
Différents niveaux d'API de persistance

ORM : JPA (ou Hibernate, TopLink, ...)
Haut niveau, « Orienté Objet »
Tente de cacher la complexité du mapping
NB : concepts sous-jacents souvent mal compris par les développeurs ("lazy/eager" loading, effets de cache, "attached/detached" entities, "owning side"/"inverse side" links, ...)



Autres API
Niveau intermédiaire

JDBC API
Bas niveau, « Orienté Record & SQL »
Adresse directement la structure de la base en SQL



6

Exemple JDBC

```
Connection conn = null;
try {
  conn = getConnection() ;
  PreparedStatement ps = conn.prepareStatement(
                         "SELECT .. FROM EMPLOYEE WHERE ..." );
  ResultSet rs= ps.executeQuery();
  while (rs.next()) {
     employee.setId( rs.getInt(1) );
     employee.setName ( rs.getString(2) );
  }
  rs.close();
} catch (SQLException e) {
  // ...
} finally {
  if ( conn != null ) {
    try {
      conn.close();
    } catch (Exception ex) {
      // ...
    }
  }
}
```

SLQ natif

Mapping manuel

Gestion de la connexion

7

Exemple ORM (JPA)

- Liste d'employés :

```
static String displayAllQuery = "Select emp from Employee emp" ;
TypedQuery  e = em.createQuery(displayAllQuery, Employee.class);
List <Employee> employees=e.getResultList();
for ( Employee emp : employees ) {
  // ...
}
```

- Un employé :

```
Employee e = (Employee) em.find(Employee.class, id);
```

Moins de code

Les connexions sont masquées

Mapping automatique sur la classe Employee

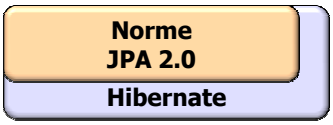
8

Pourquoi JPA ?

- Pourquoi utiliser JPA plutôt que TopLink ou Hibernate ou autre ?
  - JPA fait partie de la plate-forme **Java EE**
  - JPA est **normalisé**
    - JPA 1.0 **JSR 220** / EJB 3 (Java EE 5)
    - JPA 2.0 **JSR 317** (Java EE 6)

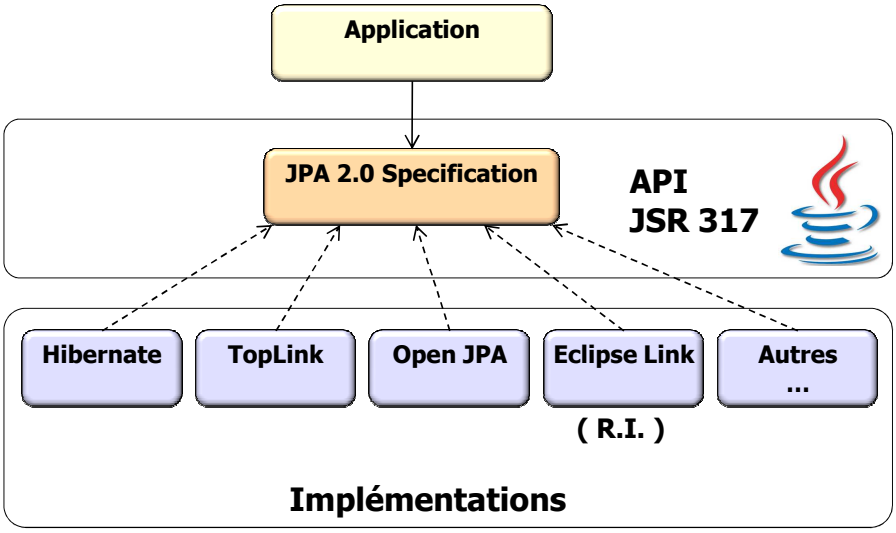
- Cependant, certains frameworks de persistance peuvent offrir des possibilités supplémentaires

Exemple :



9

Spécification et implémentations



10

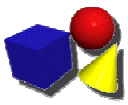
Mapping Objet / Relationnel

Problématique du mapping « O / R »

- Problème :

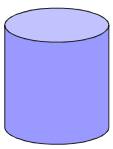
« **Modèle Objet** » ≠ « **Modèle Relationnel** »

Graphe d' objets



?

Base de données Relationnelle

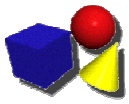


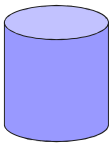
. Instances de classes
. Références (graphe)
. « clé primaire » pas obligatoire
. Héritage

. Records dans des tables
. Relations (FK → PK)

12

Mapping simple





1 classe **1 table**

13

Exemple de mapping

Exemple de mapping avec « Telosys Tools Eclipse Plugin » :

Mapping table - object | Foreign keys | Generation

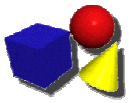
Database Name	Database Type	JDBC Type	Java Name	Java Type
ID	INTEGER	4 : integer	id	int
PUBLISHER_ID	INTEGER	4 : integer	publisherId	int
AUTHOR_ID	INTEGER	4 : integer	authorId	int
ISBN	VARCHAR(13)	12 : varchar	isbn	String
TITLE	VARCHAR(160)	12 : varchar	title	String
PRICE	DECIMAL	3 : decimal	price	BigDecimal
QUANTITY	INTEGER	4 : integer	quantity	Integer
DISCOUNT	INTEGER	4 : integer	discount	Integer
AVAILABILITY	SMALLINT	5 : smallint	availability	Short
BEST_SELLER	SMALLINT	5 : smallint	bestSeller	Short

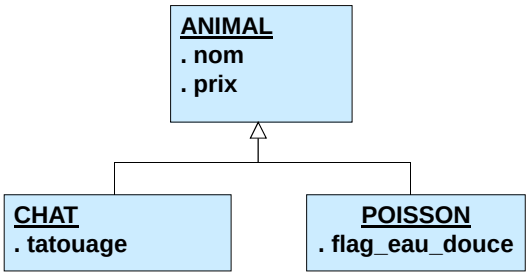
Base de données **Objet (Java)**

http://marketplace.eclipse.org/content/telosys-tools

14

Le cas de l'héritage



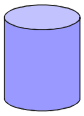


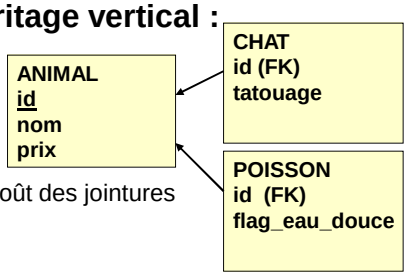
- Cas classique d'héritage : 3 possibilités :
  - Héritage vertical => 3 tables
  - Héritage horizontal => 2 tables
  - Filtrage par type => 1 seule table

15

Le cas de l'héritage

Héritage vertical :

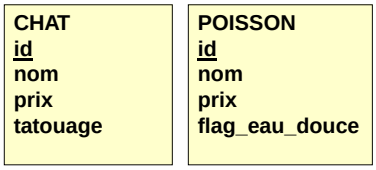




⚠ Coût des jointures

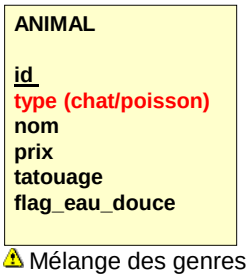
Remarques :
. On choisit en fonction des contraintes (performances, normes,…)
. Souvent le modèle physique (relationnel) existe déjà !

Héritage horizontal :



⚠ Pb d'unicité de l'id
Réplication des attributs (nom, prix,…)

Filtrage par type :



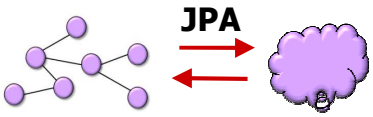
⚠ Mélange des genres

16

Les grands principes de JPA

Objectif : rester au niveau objet

- JPA donne l'impression au développeur de travailler avec une base de données orientée objet (« object database »)
- JPA masque toute la « plomberie » relationnelle
- Les connexions à la base de données ne sont pas visibles (objets JDBC « Connection »)
- Dans les cas usuels le développeur n'utilise jamais le SQL (l'utilisation de requêtes SQL natives reste cependant possible pour les cas particuliers)

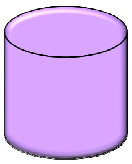


SQL

18

Mapping par annotations

```
27 @Table(name="BOOK", schema="ROOT" )
28 public class Book implements Serializable
29 {
30     private static final long serialVersionUID = 1L;
31
32     @Id
33     @GeneratedValue(strategy=GenerationType.IDENTITY)
34     @Column(name="ID", nullable=false)
35     private int        id          ;
36
37     @Column(name="PUBLISHER_ID", nullable=false)
38     private int        publisherId ;
39
40     @Column(name="AUTHOR_ID", nullable=false)
41     private int        authorId    ;
42
43     @Column(name="ISBN", nullable=false, length=13)
44     private String     isbn        ;
45
46     @Column(name="TITLE", length=160)
47     private String     title       ;
48
49     @Column(name="PRICE")
50     private BigDecimal price       ;
51
52     @Column(name="QUANTITY")
53     private Integer    quantity    ;
54
55     @Column(name="DISCOUNT")
56     private Integer    discount    ;
57
58     @Column(name="AVAILABILITY")
59     private Short      availability ;
```



BOOK
ID
TITLE
PRICE

un mapping par fichier XML est aussi possible (orm.xml)

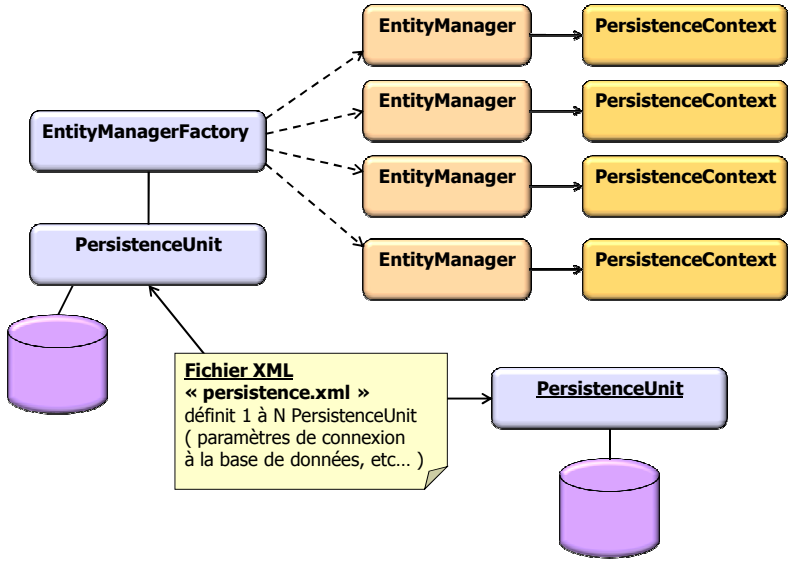
19

Quelques objets pour tout gérer

- Chaque base de données est une « **Persistence Unit** » décrite dans un fichier de configuration XML
- A chaque « Persistence Unit » correspond un « **EntityManagerFactory** »
- Les opérations de persistance reposent sur un objet central : « **EntityManager** »
  qui est fourni par l' « EntityManagerFactory » de la base de données à adresser

20

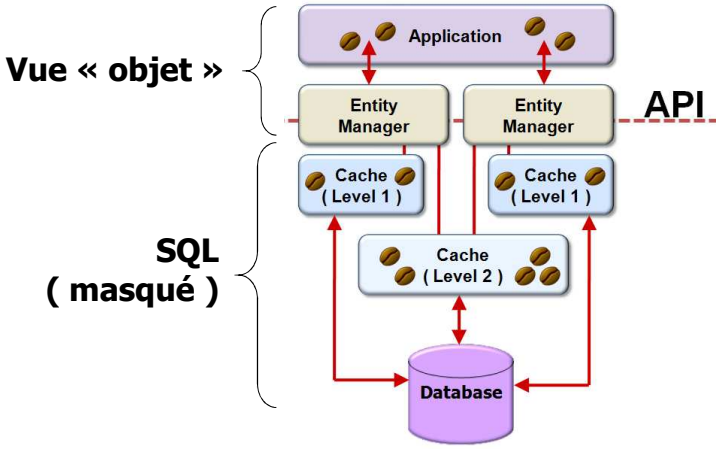
Les principaux objets



21

JPA : une affaire de caches

- « **EntityManager** » fournit des méthodes de gestion d'objets stockés dans des **caches**, ce sont des mécanismes internes qui gèrent les requêtes SQL



22

Le fichier « persistence.xml »

- Situé dans « **META-INF** »

```
<persistence version="2.0" xmlns=".. ..">
 <persistence-unit      name="jpa-tests"
                  transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

  <class>org.demo.entities.Badge</class>
  <class>org.demo.entities.Author</class>

  <properties>
    <property name="javax.persistence.jdbc.driver"
              value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="javax.persistence.jdbc.url«
              value="jdbc:derby://localhost:1527/bookstore"/>
    <property name="javax.persistence.jdbc.user"     value="root"/>
    <property name="javax.persistence.jdbc.password" value="admin"/>
  </properties>

 </persistence-unit>
</persistence>
```

Nom

Implémentation JPA

Classes des entités gérées

Config connexion JDBC

23

Démarrage

- Début et fin en Java SE :

```
public static void main(String[] args)
{
   EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("jpa-tests");

   // Récupération d'une instance de "EntityManager"
   EntityManager em = emf.createEntityManager();

   // Utilisation de l' "EntityManager" ...

   // Fermeture de l' "EntityManager"
   em.close();

   emf.close();
}
```

persistence.xml

Nom

24

Le mapping O/R avec JPA

JPA – Mapping O/R

Le mapping JPA repose sur des **annotations** définies dans le package « javax.persistence »

(cf JavaDoc)

Annotation Types Summary			
Access	Id	PersistenceContext
AssociationOverride	IdClass	PersistenceContexts
AssociationOverrides	Inheritance	PersistenceProperty
AttributeOverride	JoinColumn	PersistenceUnit
AttributeOverrides	JoinColumns	PersistenceUnits
Basic	JoinTable	PostLoad
Cacheable	Lob	PostPersist
CollectionTable	ManyToMany	PostRemove
Column	ManyToOne	PostUpdate
ColumnResult	MapKey	PrePersist
DiscriminatorColumn	MapKeyClass	PreRemove
DiscriminatorValue	MapKeyColumn	PreUpdate
ElementCollection	MapKeyEnumerated	PrimaryKeyJoinColumn
Embeddable	MapKeyJoinColumn	PrimaryKeyJoinColumns
Embedded	MapKeyJoinColumns	QueryHint
EmbeddedId	MapKeyTemporal	SecondaryTable
Entity	MappedSuperclass	SecondaryTables
EntityListeners	MapsId	SequenceGenerator
EntityResult	NamedNativeQueries	SqlResultSetMapping
Enumerated	NamedNativeQuery	SqlResultSetMappings
ExcludeDefaultListeners	NamedQueries	Table
ExcludeSuperclassListeners	NamedQuery	TableGenerator
FieldResult	OneToMany	Temporal
GeneratedValue	OneToOne	Transient
	OrderBy	UniqueConstraint
	OrderColumn	Version

26

JPA – Mapping O/R

- Mapping de l'entité (classe Java)
  - Association Classe → Table

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }
```

- Mapping des champs (attributs Java)

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private int id;
    @Column(name="NAME")
    private String name;
    @Column(name="SAL")
    private long salary;
    ...
```

Les annotations peuvent être placées au niveau des champs ou au niveau des accesseurs (setXxxx)

27

JPA – Mapping O/R

- ID (Clé Primaire) : **@Id**

```
@Id
@Column(name="EMP_ID")
private int id;
```

- Types BLOB/CLOB : **@Lob**

```
@Basic(fetch=FetchType.LAZY)
@Lob
@Column(name="PIC")
private byte[] picture;
```

- Les types de dates **@Temporal**

```
@Temporal(TemporalType.DATE)
@Column(name="START_DATE")
private Date startDate;
```

.DATE
.TIME
.TIMESTAMP

28

JPA – Mapping O/R

- Génération d'id

AUTO : c'est le provider JPA qui décide

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private int id;
```

TABLE : id stocké dans une table

```
@Id
@TableGenerator(name="EmpGen", table="x",
      pkColumnName="x" , valueColumnName="x" )
@GeneratedValue( generator = "EmpGen" )
private int id;
```

SEQUENCE: id géré par une séquence

```
@Id
@SequenceGenerator(name="EmpGen", sequenceName="SEQ1")
@GeneratedValue( generator = "EmpGen" )
private int id;
```

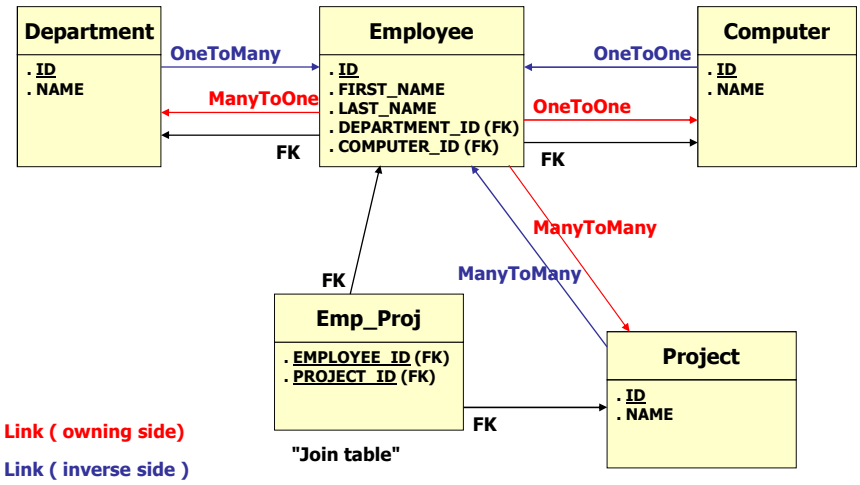
IDENTITY : id généré par une colonne auto-incrémentée

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

29

JPA – Mapping O/R – Links

- Les types de liens JPA



30

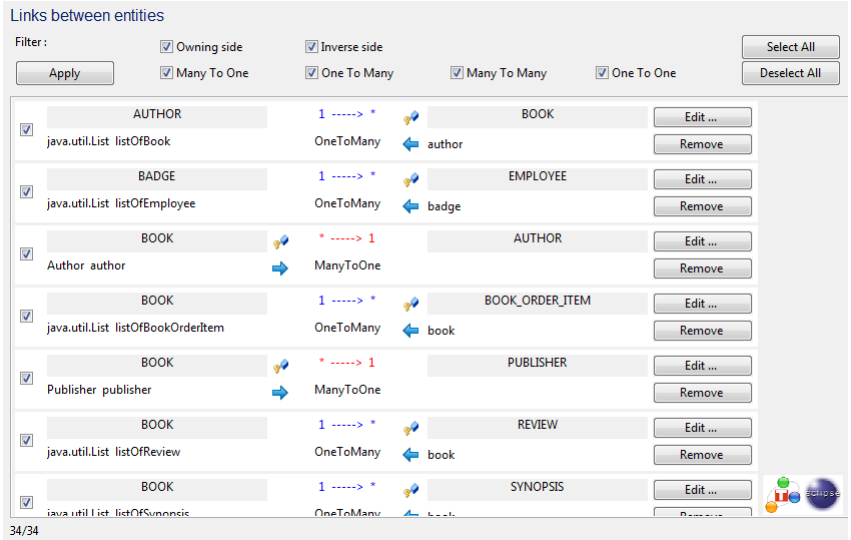
JPA – Mapping O/R – Links

- Une relation entre 2 entités repose sur **2 liens** (un lien pour chaque sens)
- Chaque lien à un **sens** et une **cardinalité** :
  One To One, **One To Many,**
  Many To One, **Many To Many**
- **Owning Side**
  - Le côté « propriétaire » du lien
  - Celui qui possède la Foreign Key
- **Inverse Side**
  - Le côté « référencé » par le « propriétaire »
- « *The owner has the power* »

31

JPA – Mapping O/R – Links

- Exemple



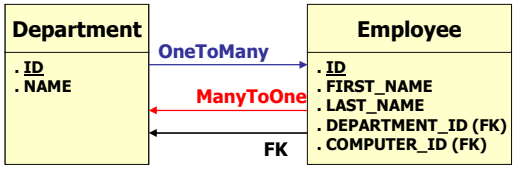
32

JPA – Mapping O/R – Links

- Exemple : **Many To One (Owning Side)**

```
@Entity
public class Employee {
    @Id
    private int id;

    @ManyToOne
    @JoinColumn(name="DEPARTMENT_ID")
    private Department department;
    // ...
}
```

ONE : simple référence sur une entité



33

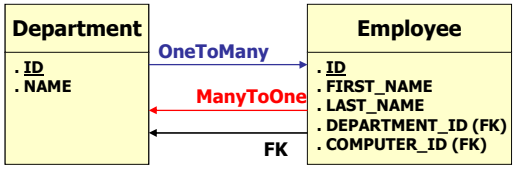
JPA – Mapping O/R – Links

- Exemple : **One To Many (Inverse Side)**

```
@Entity
public class Department {
    @Id
    private int id;

    private String name;

    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
}
```

Attribut qui pointe sur cette entité dans l'Owning Side

MANY : Collection



34

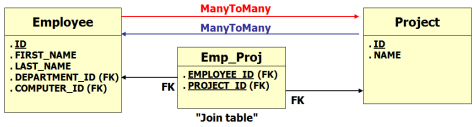
JPA – Mapping O/R – Links

- Exemple : **Many To Many**

```
@Entity
public class Employee {
    @Id
    private int id;
    //...

    @ManyToMany
    @JoinTable( name="EMP_PROJ",
       joinColumns=@JoinColumn(name="EMPLOYEE_ID"),
       inverseJoinColumns=@JoinColumn(name="PROJECT_ID")  )
    private Collection<Project> projects;
    // ...
}
```

Owning Side (choix arbitraire)

Inverse Side



35

JPA – Mapping O/R – Loading

- Lorsqu'une entité est chargée dans le « PersistenceContext », ses liens peuvent être
  → immédiatement chargés : « **Eager Loading** »
  → chargés plus tard, uniquement quand l'application va les utiliser : « **Lazy Loading** »

```
@Entity
public class Employee {
    @Id
    private int id;

    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    // ...
}
```

36

JPA – Mapping O/R – Autres exemples

- Lien basé sur **une seule colonne**

```
@ManyToOne
@JoinColumn (name="ADDR_ID", referencedColumnName="ID")
public Address getAddress() { return address; }
```

- Lien basé sur **deux colonnes**

```
@ManyToOne
@JoinColumns ( {
   @JoinColumn (name="ADDR_ID", referencedColumnName="ID") ,
   @JoinColumn (name="ADDR_ZIP", referencedColumnName="ZIP")
} )
public Address getAddress() { return address; }
```

37

JPA – Mapping O/R – Autres exemples

- « join table » avec clé composite

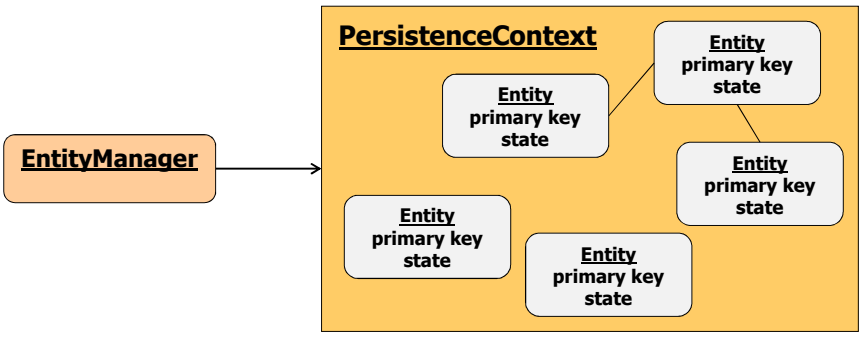
```
@ManyToMany
@JoinTable(
 name="EMP_PROJECT",
 joinColumns = {
  @JoinColumn (name="EMP_COUNTRY", referencedColumnName="COUNTRY"),
  @JoinColumn (name="EMP_ID", referencedColumnName="EMP_ID") },
 inverseJoinColumns = @JoinColumn(name="PROJECT_ID")
)
private Collection<Project> projects;
```

38

Architecture de JPA

Le contexte de persistance

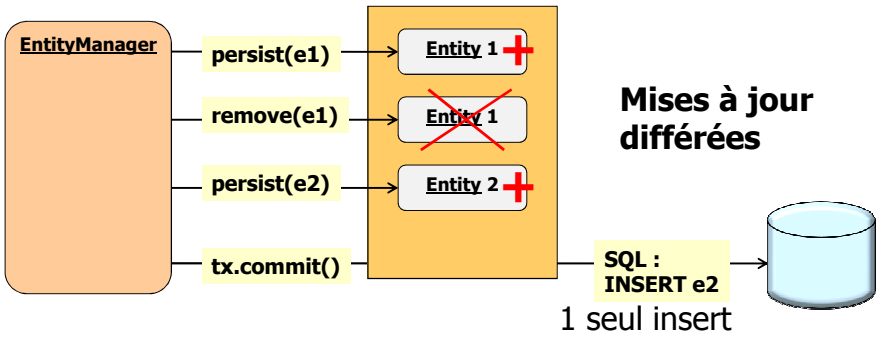
- Le « **PersistenceContext** » est un espace de stockage en mémoire qui contient des « entités »
- Chaque entité a un état et est identifiée par sa clé primaire (il ne peut pas y avoir 2 instances d'une même classe avec la même clé primaire)



40

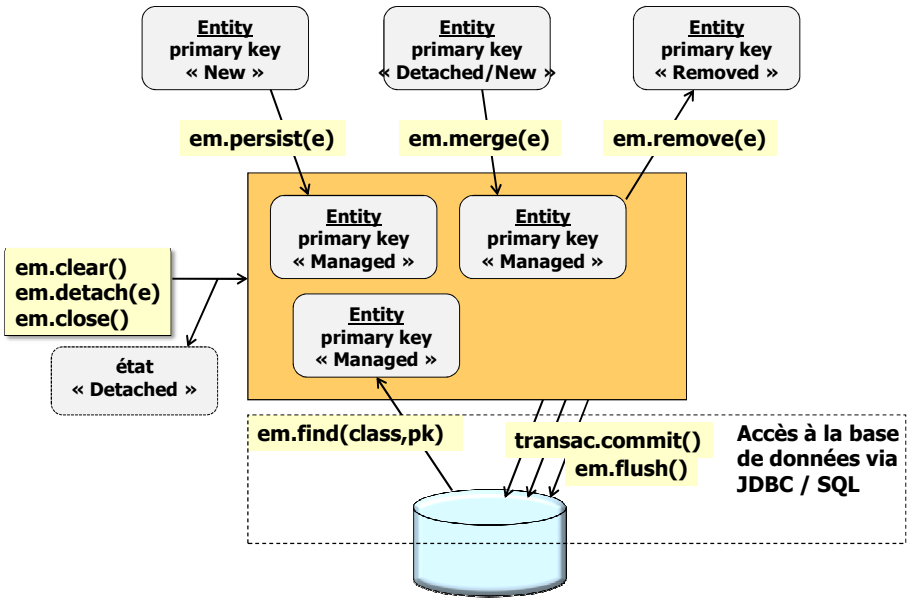
Le contexte de persistance

- L' « **EntityManager** » gère l'état des instances dont il a la charge (objets « Managed » dans le « PersistenceContext »)
- Il décide quand et comment répercuter les mises à jour dans la base données



41

Gestion des entités



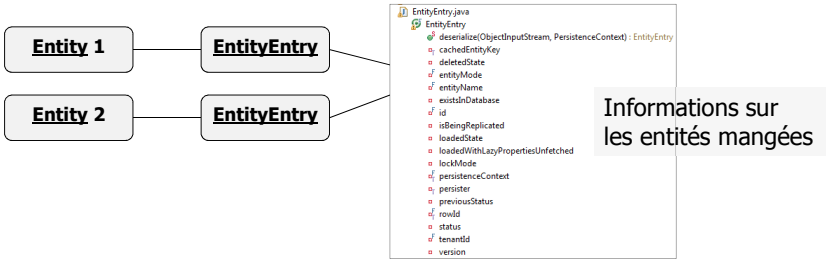
42

Etat des entités

- Pour suivre les évolutions des entités gérées il faut avoir **un état pour chaque instance**
- A chaque instance sont donc associées des informations complémentaires qui permettront de suivre les évolutions de l'instance

Exemple : Hibernate

```
Map<Object,EntityEntry> entityEntries;
```



43

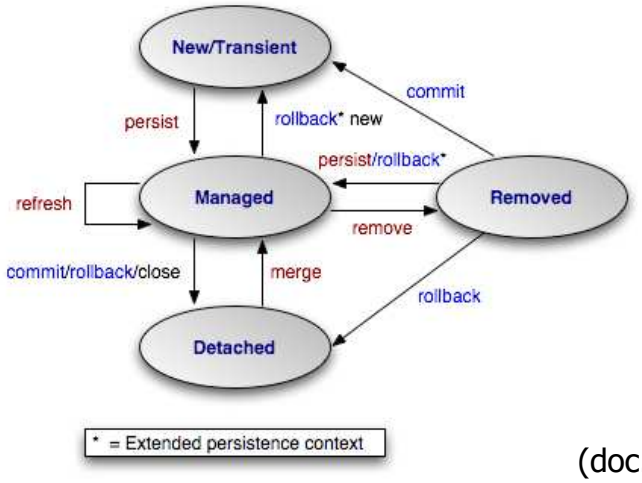
Différents états d'une entité

- Chaque entité a un état qui peut être :
  - **New (ou Transient)** : non géré
  - **Managed** : géré
  - **Removed** : supprimé (suppression logique)
  - **Detached** : détaché (qui n'est plus géré)
- Cet état évolue en fonction des appels aux méthodes de l' « EntityManager »

44

Différents états d'une entité

- Evolution de l'état :

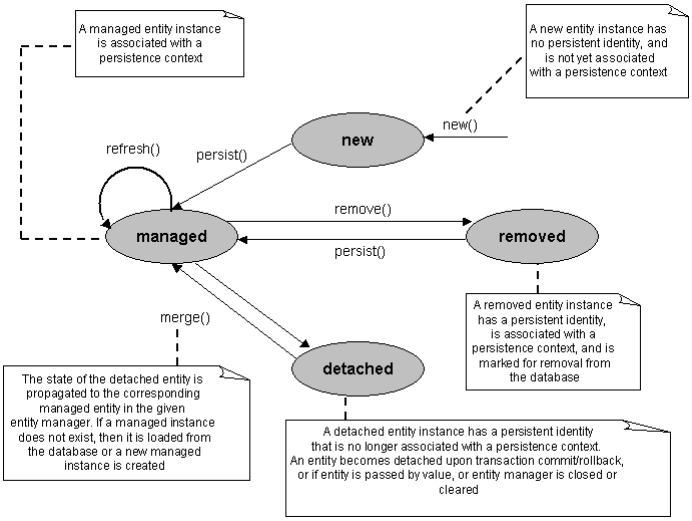


(doc. Open JPA)

http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/jpa_overview_em_lifecycle.html

45

Différents états d'une entité



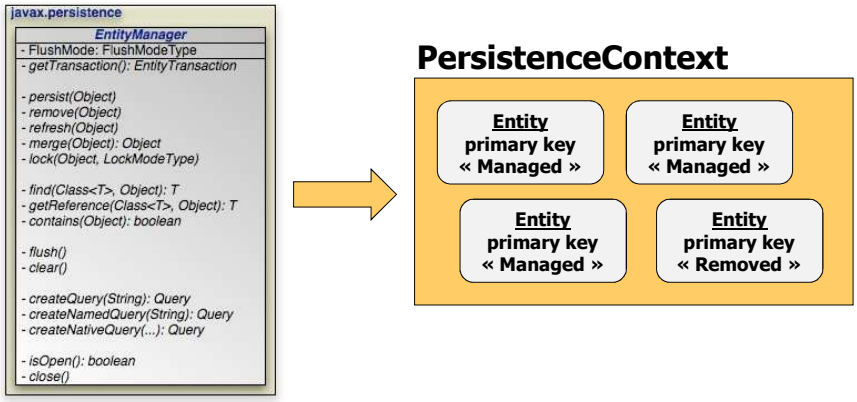
http://java.boot.by/scbcd5-guide/ch06.html

46

Entity Manager

EntityManager

- C'est l' **EntityManager** qui gère toutes les opérations de persistance des entités



48

EntityManager : les principales méthodes

- **persist(entity)** → ajout d'une nouvelle entité
- **merge(entity)** → mise à jour d'un entité (ajout si inex.)
- **remove(entity)** → suppression d'une entité
- **find(type, key)** → recherche d'une entité par son id
- getReference(type,key) → idem mais renvoie un « proxy »
- refresh(entity) → rafraichissement (DB → entité)
- lock(entity,mode) → verrouillage
- contains(entity) → entité présente dans le contexte ?
- flush() → force la mise à jour en base
- clear() → vide le PersistenceContext
- **getTransaction()** → récupère la transaction courante
- close() → fin d'utilisation (ne commit pas)

49

EntityManager

- Un paramètre de type « entité » est attendu par la plupart des méthodes (persist, merge, remove, …)

  => ce paramètre doit être une instance d'une classe annotée « **@Entity** » (avec le mapping champs Java ←→ colonnes de la table)

  NB : cette classe doit être déclarée dans le fichier « **persistence.xml** »

  Si l'entité passée en paramètre ne satisfait pas ces conditions → « **IllegalArgumentException** »

50

Entity Manager
Les opérations de base

Notion de « CRUD »

- **CRUD :**
  - **Create**
  - **Retrieve**
  - **Update**
  - **Delete**
- **CRUD en SQL :**
  - **C :** **Insert** into … values
  - **R :** **Select** … from … where
  - **U :** **Update** … set … where
  - **D :** **Delete** from … where
- Avec JPA c'est différent …

52

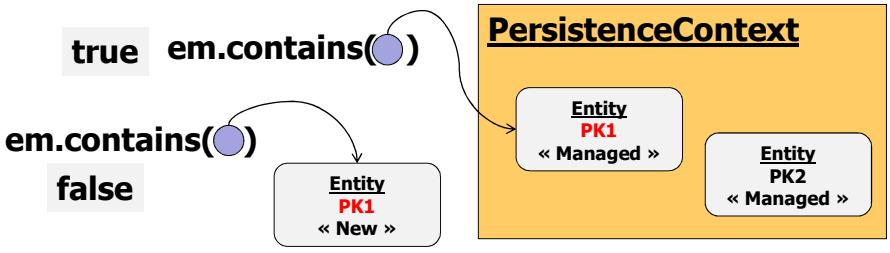
Fonctionnement de JPA

- Les méthodes de mise à jour persist(), merge() et remove() **ne réalisent pas d'actions immédiates dans la base de données**
- Ces mises à jour sont réalisées dans le « **PersistenceContext** » (en mémoire)
- C'est l' EntityManager qui décide quand et comment répercuter ces mises à jour dans la base de données, en fonction du « FlushModeType » (AUTO ou COMMIT)
- Il n'y a pas de correspondance directe entre une méthode JPA et un ordre SQL

53

EntityManager : contains

- **public boolean contains(Object entity);**
- Renvoie true si le PersistenceContexte contient la référence sur cette entité (même instance)
- Le test est fait sur l'instance et non sur la clé primaire.
  => Attention aux entités stockées par copie (notamment par « merge »).



54

EntityManager : find (Retrieve)

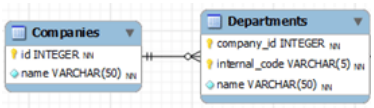
- **public <T> T find (Class<T> cl, Object primKey);**
- Recherche une entité par sa clé primaire et la charge dans le PersistenceContext (si trouvée)
- Un « **find** » se traduit par un « **SELECT** » SQL
- L'entité chargée est créée par l'EntityManager, son état est « Managed »
- Exemple :

```
System.out.println("find...");
Badge badge = em.find(Badge.class, 305);

if ( badge != null ) {
    System.out.println("Found : " + badge );
}
else {
    System.out.println("Not found");
}
```

55

EntityManager : find / PK Composite

- Si la clé primaire est composée de plusieurs champs (plusieurs colonnes dans la base), il faut créer une **classe dédiée** qui va contenir les différentes informations qui composent la clé
- Exemple




```
public class DepartmentPK implements Serializable {
    private static final long serialVersionUID = 1L;

    private Integer companyId;

    private String internalCode;
```

```
DepartmentPK pk = new DepartmentPK(2, "ABC");
System.out.println("find...");
Department dep = em.find(Department.class, pk);
```

56

EntityManager : persist (Create)

- **public void persist(Object entity);**
- Doit être utilisé dans une transaction active
  (sinon TransactionRequiredException)
- Comportement
  - Changement d'état → « **Managed** »
    - Si l'état est « **New** » : passe à « **Managed** »
    - Si l'état est « **Managed** » : ignoré
    - Si l'état est « **Removed** » : passe à « **Managed** »
    - Si l'état est « **Detached** » : **IllegalArgumentException**
  - Application en cascade sur toutes les relations de l'entité (liens) ayant une annotation
    cascade = { **CascadeType.PERSIST**, … }
    (même s'il n'y a pas eu de changement d'état,
    cas « Managed » → « Managed »)

57

EntityManager : persist

- Exemple :

```
em.getTransaction().begin();

System.out.println("persist...");
em.persist(badge301);
System.out.println("contains : " + em.contains(badge301) );

System.out.println("remove...");
em.remove(badge301);
System.out.println("contains : " + em.contains(badge301) );

System.out.println("persist...");
em.persist(badge302);
System.out.println("contains : " + em.contains(badge302) );

em.getTransaction().commit();
```

true

false

true

→ **SQL : Insert**

Un seul Insert pour « badge 302 »

58

EntityManager : persist (instances)

- Gestion des instances par « persist »
  - Le PersistenceContext contient une **référence** sur l' instance qui lui a été passée en paramètre.
  - Toute modification ultérieure de cette instance sera donc implicitement prise en compte par le PersistenceContext et répercutée dans la base de données lors de l'INSERT.
  - L'appel a « **em.contains(e)** » renvoie true si e est une référence à l'entité passée à « persist »
  - Si l'instance est déjà présente dans le contexte : pas d'erreur
  - Si une autre instance avec la même clé primaire est déjà présente dans le context
    => PersistanceException (NonUniqueObjectException)

59

EntityManager : persist (instances)

- Exemple :

```
em.getTransaction().begin();

Badge badge = new Badge();
badge.setBadgeNumber(305);
badge.setAuthorizationLevel((short) 1305 );

System.out.println("persist...");
em.persist(badge);

badge.setAuthorizationLevel((short) 2000 );

System.out.println("commit..." );

em.getTransaction().commit();
```

persist

puis modification

→ **SQL : Insert**

Résultat :

BADGE_NUMBER	AUTHORIZATION_LEVEL
305	2000

La modification faite après « persist » est prise en compte

60

EntityManager : remove (Delete)

- **public void remove(Object entity);**
- Doit être utilisé dans une transaction active
- Comportement
  - Changement d'état → « **Removed** »
    - Si l'état est « **New** » : ignoré
    - Si l'état est « **Managed** » : passe à « **Removed** »
    - Si l'état est « **Removed** » : ignoré (état inchangé)
    - Si l'état est « **Detached** » : **IllegalArgumentException**
  - Application en cascade sur toutes les relations de l'entité (liens) ayant une annotation
    cascade = { **CascadeType.REMOVE**, … }

61

EntityManager : remove

- On ne peut faire un « remove » que sur une entité « managed » => il faut d'abord la charger dans le « PersistenceContext »
- Exemple :

```
Badge badge = em.find(Badge.class, id);
```
→ **SQL : Select**

```
if ( badge != null ) {
    System.out.println("Found");

    em.getTransaction().begin();

    em.remove(badge);
    em.getTransaction().commit();

    System.out.println("Removed");
}
else {
    System.out.println("Not found");
}
```

→ **remove**

→ **SQL : Delete**

62

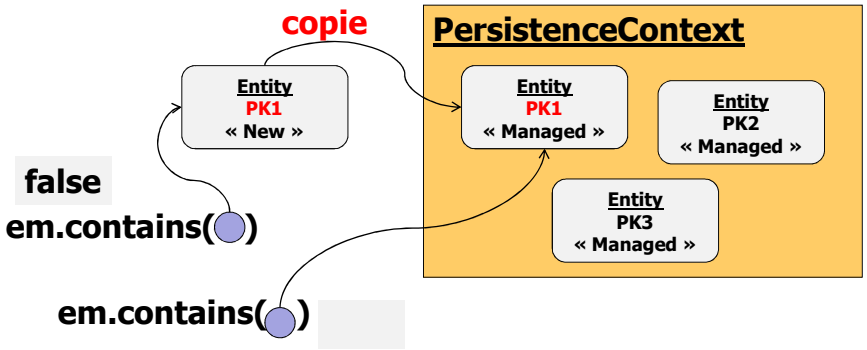
EntityManager : merge (Create/Update)

- **public void merge(Object entity);**
- Fonctionnement par copie
- Doit être utilisé dans une transaction active
- Comportement
  - Changement d'état → « **Managed** »
    - Si l'état est « **New** » : création d'une nouvelle entité et copie
    - Si l'état est « **Detached** » : copie dans l'entité existante
    - Si l'état est « **Managed** » : ignoré
    - Si l'état est « **Removed** » : **IllegalArgumentException**
  - Application en cascade sur toutes les relations de l'entité (liens) ayant une annotation
    cascade = { **CascadeType.MERGE**, … }

63

EntityManager : merge

- « Merge » = « **Fusion** » de 2 entités
  C'est-à-dire **recopie** d'une instance dans une autre
- L'Entity Manager a besoin d'une **entité cible**
  => si elle n'est pas déjà dans le PersistenceContext
  une instance est créée



64

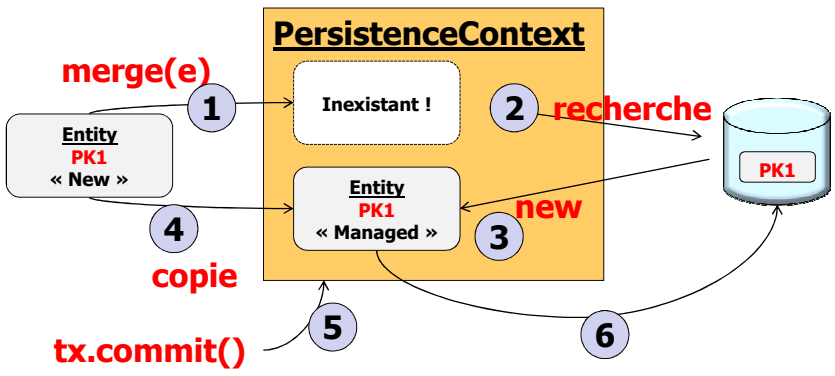
EntityManager : merge (instances)

- Gestion des instances par « merge »
  - L'entité passée en paramètre est recopiée dans une autre instance présente dans le contexte de persistance.
  - Si l'entité n'existe pas dans le contexte :
    elle est chargée à partir de la base de données
    ou une nouvelle instance est créée
  - Il y a donc 2 instances distinctes
  - « em.contains(e) » renvoie donc **false**
    (« e » fait référence à une entité différente de celle stockée dans le contexte)
  - Les modifications ultérieures sont sans effet sur l'entité du contexte de persistance.

65

EntityManager : merge (instances)

- Exemple :



NB : Attention à travailler sur la bonne instance (« managed » ou non)

66

EntityManager : merge (instances)

- Exemples :

```
em.merge(badge);

boolean b = em.contains(badge); // FALSE
```

Sans récupération de l'entité renvoyée
pas de visibilité sur l'instance « Managed »
Inutile de modifier l'instance « badge »
pour une modification en base car ce n'est pas celle qui est gérée dans le PersistenceContext

```
Badge managedBadge = em.merge(badge);

boolean b1 = em.contains(badge); // FALSE
boolean b2 = em.contains(managedBadge); // TRUE

managedBadge.setAuthorizationLevel((short)999);
```

Avec récupération de l'entité renvoyée
possibilité de travailler sur l'instance « Managed »

Après « commit » on aura 999 dans la base de données

67

EntityManager : refresh

- **public void refresh (entity);**
- S'assure que les informations de l'entité sont bien synchronisées avec la base de données
- Comportement
  - Pas de changement d'état (reste à « **Managed** »)
    - Si l'état est « **New** » : ignoré
    - Si l'état est « **Managed** » : rafraichi à partir de la base
    - Si l'état est « **Removed** » : ignoré
    - Si l'état est « **Detached** » : **IllegalArgumentException**
  - Application en cascade sur toutes les relations de l'entité (liens) ayant une annotation
    cascade = { **CascadeType.REFRESH**, … }

68

EntityManager : refresh

- Exemple

```
System.out.println("find...");
Badge badge = em.find(Badge.class, 305);

System.out.println("Badge after find : " + badge );

badge.setAuthorizationLevel((short) 2 ) ;
System.out.println("Badge after change : " + badge );

System.out.println("refresh...");
em.refresh(badge);

System.out.println("Badge after refresh : " + badge );
```

→ **SQL : Select**

→ **SQL : Select**

« refresh » recopie les données de la base dans l'instance et donc annule toutes les modifications non commitées

69

EntityManager : et l'Update ?

- **Il n'y a pas d' Update explicite en JPA**
- Le commit suffit pour répercuter tous les changements du « PersistenceContext » dans la base de données
- Exemple :

```
System.out.println("find...");
Badge badge = em.find(Badge.class, 305);
if ( badge != null ) {
    System.out.println("Found : " + badge );

    em.getTransaction().begin();

    badge.setAuthorizationLevel((short) 555 );
    System.out.println("Updated in memory : " + badge );

    em.getTransaction().commit();

    System.out.println("Commited");
}
else {
    System.out.println("Not found");
}
```

70

EntityManager : clear / detach / flush

- Méthode **clear()**
  - Pour détacher toutes les entités gérées par le contexte
  - Toutes les modifications apportées aux entités du contexte sont perdues
- Méthode **detach(entity)** (JPA 2)
  - Idem pour une entité spécifique
- Méthode **flush()**
  - Permet de forcer les mises à jour dans la base de données
  - N'est pas un commit() : ne fait que forcer l'exécution des requêtes SQL en attente
  - Comportement variable selon les implémentations de JPA

L'utilisation de ces méthodes est généralement déconseillée

71

Gestion des transaction

Rappel sur les transactions

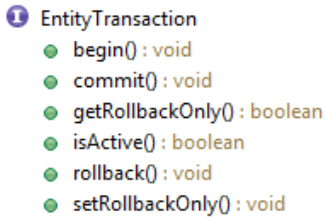
- Une transaction est un regroupement d'instructions SQL effectuant un traitement fonctionnel atomique
  Exemple : opération débit/crédit
- Une transaction doit être **A**tomique, **C**ohérente, **I**solée et **D**urable (**ACID**)
  - **Atomique** : indivisible, tout ou rien
  - **Cohérente** : le contenu final de la base de données doit être cohérent
  - **Isolée** : quand 2 transactions sont exécutées simultanément elles ne doivent pas interférer
  - **Durable** : le résultat final d'une transaction validé (état cohérent) est conservé définitivement, même en cas d'incident

Atomicity, Consistency, Isolation, Durability

73

L'objet Transaction

- Dans JPA la gestion des transactions de fait via l'interface « **EntityTransaction** »



- Une instance de transaction est récupérée à partir de l' EntityManager

```
EntityTransaction transaction = em.getTransaction();
```

74

Délimiter une transaction

- **Début**
  - em.getTransaction().**begin**();
- **Fin**
  - em.getTransaction().**commit**();
  - em.getTransaction().**rollback**();

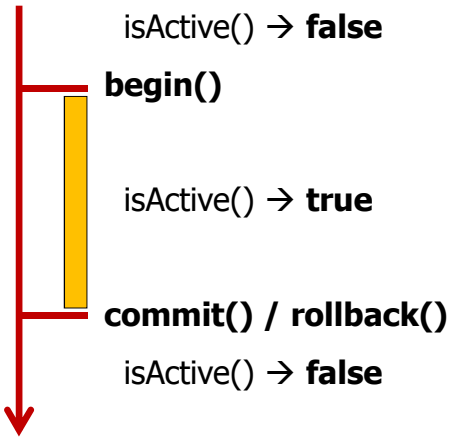
- Exemple :

```
em.getTransaction().begin();

em.remove(badge);

em.getTransaction().commit();
```

75

Etat d'une transaction

- Comment savoir si une transaction est déjà active ?
  - em.getTransaction().**isActive**();



76

Cas d'erreur

- Si la transaction est déjà active
  - em.getTransaction().**begin**();
  - → **java.lang.IllegalStateException**
- Si la transaction n'est pas active
  - em.getTransaction().**commit**(); (ou **rollback**)
  - → **java.lang.IllegalStateException**

77

Les requêtes JPQL, SQL et Criteria API

Requêtes : les différents langages/API

- Un des principaux objectifs de JPA est de ne plus utiliser SQL pour les accès à la base de données
- JPA fournit donc un langage de requêtes indépendant du SQL (parfois spécifique) de la base de données :
  Java Persistence Query Language (**JP QL**)
- JPA fournit également une API Java permettant de construire des requêtes dynamiquement par des appels de méthodes Java : la « **Criteria API** »
- Enfin, des requêtes exprimées en SQL natif sont parfois nécessaires : JPA permet d'appeler des **requêtes écrites en SQL**

79

Requêtes / JP QL

- **JP QL** est une révision (extension et amélioration) de **EJB QL** (langage de requête pour les EJB)
- La syntaxe reste très proche du SQL
  (SELECT, FROM, WHERE, ...)
- Principale différence :
  en JP QL on ne fait pas un SELECT sur une TABLE, mais sur une CLASSE JAVA (un type d'entité)
- Exemples :

```
SELECT e.name FROM Employee e
```

Employee.java
Employee.class

```
SELECT e FROM Employee e
WHERE  e.department.name = 'AB'
AND    e.address.state IN ( 'NY', 'CA' )
```

80

Requêtes / JP QL – Paramètres

- Les paramètres des requêtes JP QL peuvent être représentés par un nom symbolique précédé de '**:**'
  :nom

```
SELECT b FROM Badge b
WHERE b.badgeNumber >= :min
AND   b.badgeNumber <= :max
```

- Ou par un numéro précédé de '**?**'
  ?indice (numéros quelconques : ?11 ?32 ...)

```
SELECT b FROM Badge b
WHERE b.badgeNumber >= ?1
AND   b.badgeNumber <= ?2
```

81

Requêtes / JP QL – Utilisation en Java

- Requête JP QL sans paramètre

```
final String QUERY = "SELECT b.badgeNumber FROM Badge b " ;

Query query = em.createQuery( QUERY ) ;

//--- Execute query
System.out.println("execute query ...");
List<Integer> list = query.getResultList() ;

System.out.println("Number of badges : " + list.size() );
for ( Integer i : list ) {
   System.out.println(" . badge number : " + i );
}
```

0 .. N instances de type **Integer**

```
final String QUERY = "SELECT b FROM Badge b" ;
...
List<Badge> list = query.getResultList() ;
..
```

0 .. N instances de type **Badge**

instances « managed » dans le « PersistenceContext »

82

Requêtes / JP QL – Utilisation en Java

- Requête JP QL avec paramètres

```
final String QUERY = "SELECT b FROM Badge b "
 + "WHERE b.badgeNumber >= :min AND b.badgeNumber <= :max  " ;
Query query = em.createQuery( QUERY ) ;

//---- Set parameters
query.setParameter("min", 100);
query.setParameter("max", 310);

//--- Execute query
List<Badge> list = query.getResultList();
```

Paramètres nommés

```
final String QUERY = "SELECT b FROM Badge b "
 + "WHERE b.badgeNumber >= ?1 AND b.badgeNumber <= ?2" ;
Query query = em.createQuery( QUERY ) ;

//---- Set parameters
query.setParameter( 1, 100);
query.setParameter( 2, 310);
...
```

Paramètres numérotés

83

Requêtes / JP QL – Update & Delete

- JP QL permet de faire des **mise à jour** (utile notamment pour appliquer une mise à jour sur plusieurs « records » en une seule requête)

Bulk Update

```
String QUERY = "UPDATE Badge b SET b.authorizationLevel = 123 "
             + " WHERE b.badgeNumber > 300 " ;
```

Bulk Delete

```
String QUERY = "DELETE Badge b WHERE b.badgeNumber > 400" ;
```

```
em.getTransaction().begin();
int r = query.executeUpdate();  // retour : Nb de records affectés
em.getTransaction().commit();
```

Update & Delete uniquement (pas d'insert)
Utilisable sur un seul type d'entité à la fois

Les entités stockées dans le « PersistenceContext » ne sont pas mises à jour => risque d'incohérence

A exécuter dans une transaction dédiée ou au début d'une transaction

84

Requêtes nommées

- Définition d'une requête nommée :

Définition par annotations dans une entité

```
@Entity(name = "EMPLOYEE")
@NamedQueries({
 @NamedQuery( name  = "Employee.findAll",
              query = "SELECT EMP FROM EMPLOYEE AS EMP" )
})
public class Employee implements Serializable {
  @Id
  @Column(name = "EMP_ID", nullable = false)
  private Long id;
  @Column(name = "EMP_NAME", nullable = false, length = 100)
  private String name;
  // .. ..
}
```

- Utilisation :

```
public List findAllEmployees(){
  Query query = entityManager.createNamedQuery( "Employee.findAll" );
  List employees = query.getResultList();
  return (employees);
}
```

85

Requêtes / SQL natif

- Bien que JP QL soit le langage recommandé pour exprimer des requêtes, dans certains cas des requêtes écrites en SQL natif (avec éventuellement des spécificités liées à la base de données) peuvent s'avérer nécessaires
- Les requêtes SQL sont gérées par l'EntityManager ce qui permet de profiter du mapping des entités : le « ResultSet » de la requête peut renvoyer des entités JPA (@Entity)

86

Requêtes / SQL natif - Mapping

Mapping sur **Badge** doit être une entité connue de JPA (@Entity)

```
final String QUERY = "SELECT * FROM BADGE" ;

Query query = em.createNativeQuery( QUERY, Badge.class) ;

List<Badge> list = query.getResultList();
```

instances « managed » dans le « PersistenceContext »

1 seule colonne (int)

```
final String QUERY = "SELECT BADGE_NUMBER FROM BADGE" ;

Query query = em.createNativeQuery( QUERY ) ;

List<Integer> list = query.getResultList();
```

Rien

Type compatible avec le ResultSet

87

Requêtes / SQL natif - Mapping

1 seule colonne (int)

Mapping sur une entité (@Entity)

```
final String QUERY = "SELECT BADGE_NUMBER FROM BADGE" ;

Query query = em.createNativeQuery( QUERY, Badge.class) ;

List<Badge> list = query.getResultList();
```

instances contenant toutes les informations
de l'entité (pas uniquement la colonne sélectionnée)
Les entités ne sont pas partiellement valorisées

88

Requêtes / SQL natif - Mapping

- Pour récupérer les valeurs de plusieurs colonnes ne correspondant à aucune entité
  => définir un « **Resultset Mapping** »
  (par annotations dans une des entités)

Définition

```
@Entity
@Table(name = "EMP")
@SqlResultSetMappings( {
  @SqlResultSetMapping(
    name = "ProfessorAndManager",
    columns = { @ColumnResult(name = "EMP_NAME" ),
                @ColumnResult(name = "MANAGER_NAME" ) })
})
public class Professor { ... }
```

Utilisation

```
final String QUERY =
  "SELECT e.name AS emp_name, m.name AS manager_name "
+ "FROM emp e, emp m " + "WHERE e.manager_id = m.emp_id" ;
Query query = em.createNativeQuery( QUERY, "ProfessorAndManager" );
```

89

Requêtes / SQL natif - Paramètres

- Les paramètres d'une requête SQL native sont notés '**?**' et sont valorisés par leur indice
  (comme en JDBC)

```
final String QUERY = "SELECT * FROM BADGE "
" WHERE BADGE_NUMBER >= ? AND BADGE_NUMBER <= ? " ;

Query query = em.createNativeQuery( QUERY, Badge.class) ;

query.setParameter(1, 300);
query.setParameter(2, 310);

//--- Execute query
List<Badge> list = query.getResultList();
```

90

Requêtes / SQL natif – Mises à jour

- Utilisation possible de : **Update**, **Delete**, **Insert**

```
final String QUERY = "DELETE FROM BADGE WHERE BADGE_NUMBER > 400" ;
```

```
final String QUERY =
     "INSERT INTO BADGE (BADGE_NUMBER, AUTHORIZATION_LEVEL) "
   + "VALUES (801,3 ) " ;
```

```
Query query = em.createNativeQuery( QUERY ) ;

em.getTransaction().begin();

//--- Execute query
int r = query.executeUpdate();

em.getTransaction().commit();
```

Utilisation généralement déconseillée car ce type de requête peut provoquer des écarts entre les entités stockées dans le « PersistenceContext » et la base de données
=> à utiliser avec précaution

91

Requêtes / Criteria API

- **Criteria API** est une alternative à JP QL
- Cette API permet de construire des **critères de requêtes** à l'aide d'objets Java
- Les objectifs de Criteria API :
  - s'affranchir des chaines de caractères qui définissent les requêtes en JP QL (pas de contrôle à la compilation Java, erreurs de syntaxe décelées à l'exécution)
  - apporter un contrôle de type (« type-safe »)
- Criteria API repose essentiellement sur 2 objets :
  - **CriteriaBuilder**
  - **CriteriaQuery**

92

Requêtes / Criteria API - Exemples

- Avec **JP QL**

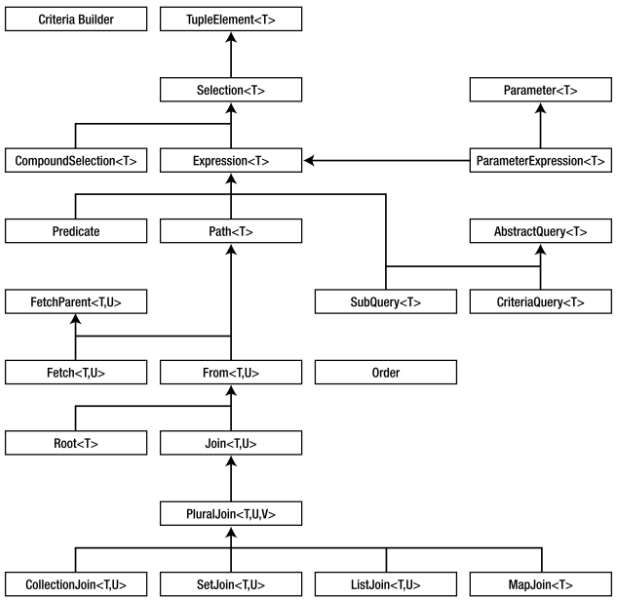
```
String QUERY = "SELECT b FROM Badge b where badgeNumber = 305" ;

Query query = em.createQuery( QUERY ) ;

List<Badge> list = query.getResultList();
```

- Avec **Criteria API**

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Badge> c = cb.createQuery(Badge.class);

//--- Criteria definition
Root<Badge> badge = c.from(Badge.class);
c.select(badge)
 .where( cb.equal( badge.get("badgeNumber"), 305 ) );

Query query = em.createQuery( c ) ;

List<Badge> list = query.getResultList();
```

93

Requêtes / Criteria API



Criteria API Interfaces

Très riche

94

Requêtes / Criteria API - Exemples

```
String param = "Archive";
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Long> query = cb.createQuery(Long.class);

Root<Music> music = query.from(Music.class);
query.select(cb.count(music));
query.where(cb.equal(music.<String>get("artisteName"),
            cb.parameter(String.class,"artisteNameParam")));

TypedQuery<Long> tq = em.createQuery(query);
tq.setParameter("artisteNameParam", param);
```

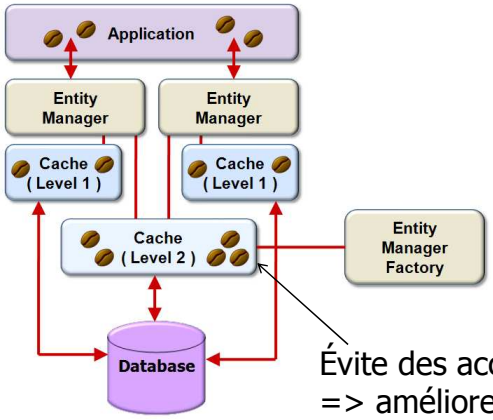
```
String param = "Arc%";
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<String> query = cb.createQuery(String.class);

Root<Music> music = query.from(Music.class);
query.select(music.<String>get("artisteName"));
query.distinct(true);
query.where(cb.like(music.<String>get("artisteName"),
            cb.parameter(String.class, "param")));
TypedQuery<String> tq = em.createQuery(query);
tq.setParameter("param", param);
```

95

Cache de « 2ème niveau »

Le cache de 2ème niveau

- « **Shared Cache** » ou « **Second Level Cache** »
- Cache géré au niveau « EntityManagerFactory »
- Cache partagé par tous les « EntityManager »



Évite des accès à la base
=> améliore les performances

97

Gestion du cache (4 méthodes)

Vérifier si une entité est dans le cache

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
if ( cache.contains(Person.class, personPK) ) {
  // the data is cached
} else {
  // the data is NOT cached
}
```

Retirer une/des entité(s) du cache

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;

cache.evict(Person.class, personPK); // Une instance (Prim. Key )

cache.evict(Person.class); // Toutes les instances d'une classe

cache.evictAll(); // Toutes les instances
```

98

Configuration du cache

- Les différents modes :

ALL	**All entity data** is stored in the second-level cache for this persistence unit.
NONE	**No data** is cached in the persistence unit. The persistence provider must not cache any data.
ENABLE_SELECTIVE	Enable caching for entities that have been **explicitly** set with the **@Cacheable** annotation.
DISABLE_SELECTIVE	Enable caching for all entities except those that have been **explicitly** set with the **@Cacheable(false)** annotation.
UNSPECIFIED	The caching behavior for the persistence unit is **undefined**. The persistence provider's default caching behavior will be used.

persistence.xml

```
<persistence-unit name="examplePU" transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <jta-data-source>jdbc/__default</jta-data-source>
  <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

Dans le code

```
Properties prop = new Properties();
prop.add("javax.persistence.sharedCache.mode", "ENABLE_SELECTIVE" ));
EntityManagerFactor emf =
    Persistence.createEntityManagerFactory( "examplePU", prop );
```

99

FIN

AJC INGÉNIERIE