

Java Persistence Api (JPA)

JPA



- JPA est une API qui permet de sauvegarder un graphe d'objets Java dans une base de données relationnelle.
- JPA s'utilise dans les applications :
 - Java Standard
 - JEE (identique aux EJB Entities)
 - JPA a été implantée par différents providers :
 - JBoss Hibernate (le plus utilisé)
 - OpenJPA
 - IBM
 - ...
- JPA masque JDBC

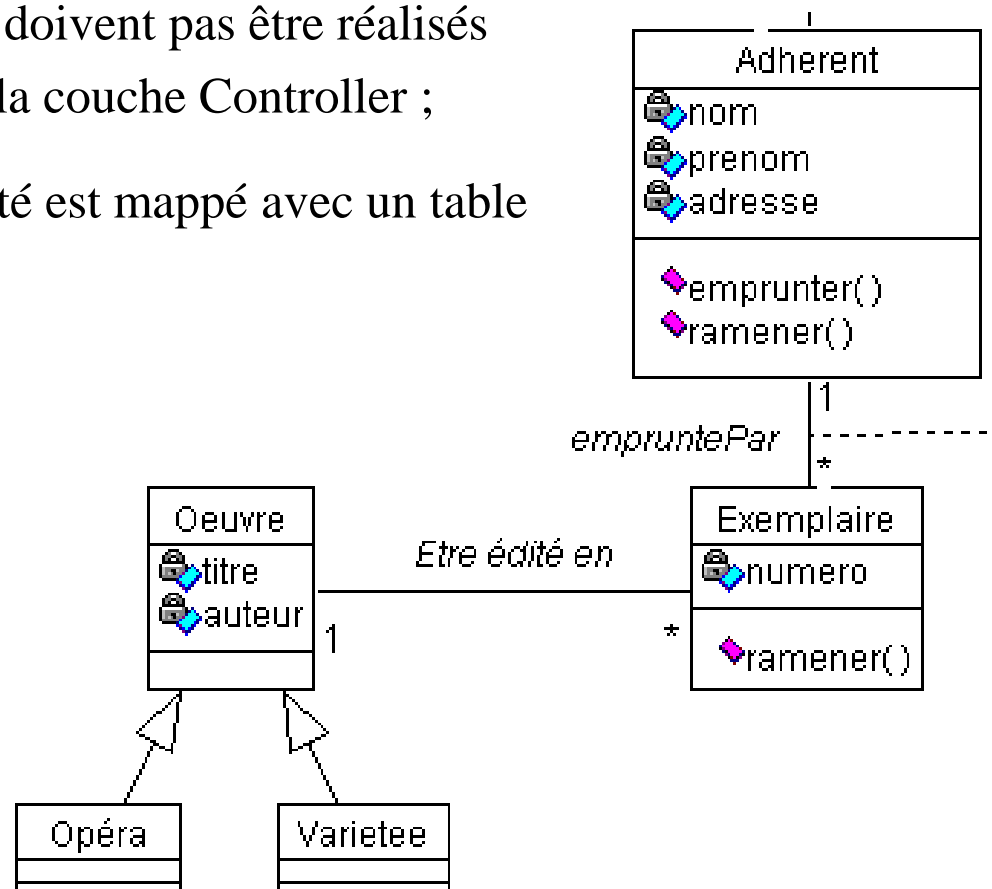
Les Entités

Les Entities

- Une *entity* est un objet métier, léger et persistant :
 - métier car élément de la couche Model (au sens Model View Controller du terme) ;
 - léger car les traitements lourds ne doivent pas être réalisés par des entités mais être fait dans la couche Controller ;
 - persistant car typiquement un entité est mappé avec un table d'une base de données.

Nom	Prénom	Adresse
Duval	Jacques	15 rue de la République
Lyon	Albert	24 rue de la République

Base de données
des adhérents



Les règles de programmation des entités

Les règles de programmation d'une Entity

- La classe qui définit une entity doit :
 - être annotée avec *javax.persistence.Entity* ;
 - ne doit pas être déclarée *final* ; pas de méthode ni de champs persistant *final* ;
 - a un constructeur sans argument ;
 - implémenter *java.io.Serializable* si les instances doivent être transmises à des objets détachés ;
 - les champs persistants ne doivent pas être déclarés *public* et les autres classes ne doivent pas y accéder directement (elles doivent passer par des méthodes) ;
 - l'héritage est permis (on peut même hériter de classes qui ne sont pas des entités).

L'état persistant d'une Entity

- Les champs persistants peuvent être du types :
 - int, float, ... (types primitifs) ;
 - Integer, Float, ... (types de base objet) ;
 - java.lang.String ;
 - java.math.BigDecimal, java.math.BigInteger ;
 - java.util.Date, java.util.Calendar ;
 - java.sql.Date, java.sql.Time, java.sql.TimeStamp ;
 - types sérialisable définis par l'utilisateur ;
 - byte[], Byte[], char[], Character[] ;
 - types énumérés ;
 - autres Entity ou collections d'Entities ;
 - Classes incluses.
 - Collections de types précédents (Collection, Set, List, Map).

La déclaration des champs persistants

- Les propriétés persistantes peuvent être déduites des méthodes *get* et *set* => il faut se conformer aux règles des Java Beans :

```
@javax.persistence.Entity
```

```
public class Personne{
```

```
private String nom;
```

```
public String getNom(){
```

```
    return nom;
```

```
}
```

```
public void setNom( String nom ){
```

```
    this.nom = nom;
```

```
}
```

```
private String numSecuSociale;
```

```
@javax.persistence.Transient
```

```
public String getNumSecuSociale(){
```

```
    return numSecuSociale;
```

```
}
```

```
public void setNumSecuSociale( String numSecuSociale ){
```

```
    this.numSecuSociale = numSecuSociale;
```

```
}
```

```
}
```

- champs persistants.

- propriété persistante :

- String prenom.

- champs non persistant.



- Propriété non persistante.

Les clefs primaires des entités

Les clefs primaires

- chaque *entity* a une clef primaire (un identifiant unique) ;
- cette clef peut être définie par :
 - un champs unique ;
 - une composition de plusieurs champs.
- une clef peut être générée automatiquement ou calculée ;
- les éléments d'une clef peuvent être :
 - int, float, ... (types primitifs) ;
 - Integer, Float, ... (types de base objet) ;
 - java.lang.String ;
 - java.util.Date, java.sql.Date.

Exemple de la déclaration d'une clef primaire

```
@Entity   
public class Personne{  
  
    private String numSecuSociale;  
  
    @Id   
    public String getNumSecuSociale(){  
        return numSecuSociale;  
    }  
  
    public void setNumSecuSociale( String numSecuSociale ){  
        this.numSecuSociale = numSecuSociale;  
    }  
}
```

Exemple d'une clef générée automatiquement

```
@Entity
public class Article{

    private Long id;

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

    public void setId(Long id ){
        this.id = id;
    }

}
```

Exemple d'une clef composite

```
public class ClefEtudiant implements
Serializable{
private String nomId;
private String prenomId;

public String getNomId(){
    return nomId;
}
public void setNomId( String nomId ){
    this.nomId = nomId;
}
public String getPrenomId(){
    return prenomId;
}
public void setPrenomId( String prenomId ){
    this.prenomId = prenomId;
}
public int hashCode(){
    return ...
}
public boolean equals(Object otherOb) {
    ...
}
}
```

```
@IdClass(ClefEtudiant.class)
@Entity
public class Etudiant{

private String nomId;
@id
public String getNomId(){
    return nomId;
}
public void setNomId( String nomId ){
    this.nomId = nomId;
}

private String prenomId;
@id
public String getPrenomId(){
    return prenomId;
}
public void setPrenom( String prenomId ){
    this.prenomId = prenomId;
}
}
```

Les règles d'écriture d'une classe clef primaire


- La classe qui définit une clef primaire doit :
 - être déclarée *public* ;
 - avoir un constructeur sans argument ;
 - implémenter *hashCode* et *equals* ;
 - implémenter *java.io.Serializable*.

La gestion des associations

La gestion des associations

- Des entités peuvent être associées entre-elles pour former un diagramme de classes ;
- Les associations peuvent-être persistantes ;
- Les associations permises sont du type :
 - *un vers un* ;
 - *un vers plusieurs* ;
 - *plusieurs vers un* ;
 - *plusieurs vers plusieurs*.
- Les associations sont *unidirectionnelles* ou *bi-directionnelles*.

Association bidirectionnelle de *un* vers *un*

```
public class Personne{  
  
    private Voiture voiture;  
  
    @OneToOne(mappedBy = "pilote")   
    public Voiture getVoiture(){  
        return voiture;  
    }  
  
    public void setVoiture(Voiture voiture ){  
        this.voiture = voiture;  
    }  
}
```

```
public class Voiture{  
  
    private Personne pilote;  
  
    @OneToOne  
    public Personne getPilote(){  
        return pilote;  
    }  
  
    public void setPilote( Personne pilote){  
        this.pilote = pilote;  
    }  
}
```

Association récursive de *un* vers *plusieurs*

```
public class Groupe{  
  
    private Collection<Groupe> sousGroupes;  
  
    @OneToMany  
    public Collection<Groupe> getSousGroupes(){  
        return sousGroupes;  
    }  
  
    public void setSousGroupes (Collection< Groupe > sousGroupes){  
        this.sousGroupes = sousGroupes;  
    }  
  
}
```

*



Association bidirectionnelle de *un* vers *plusieurs* avec *cascade*

- Pour supprimer des entités associées, on peut utiliser *cascade*.

*

```
public class Personne{  
  
    private Voiture voiture;  
  
    @ManyToMany  
    public Voiture getVoiture(){  
        return voiture;  
    }  
  
    public void setVoiture( Voiture voiture )  
        this.voiture = voiture;  
}  
}
```

```
public class Voiture{  
  
    private Collection<Personne> passagers = new  
        ArrayList<Personne>();  
  
    @OneToMany(cascade=CascadeType.ALL,  
        mappedBy="voiture")  
    public Collection<Personne> getPassagers(){  
        return passagers;  
    }  
  
    public void setPassagers(Collection<Personne> passagers){  
        this.passagers = passagers;  
    }  
  
    public void addPassager( Personne passager ){  
        this. getPassagers().add( passager );  
        passager.setVoiture(this);  
    }  
}
```

La gestion de l'héritage

La gestion de l'héritage

- Les entités supportent l'héritage et le polymorphisme ;
- les entités peuvent être *concrètes* ou *abstraites* ;
- une entité peut hériter d'une classe non entité ;
- une classe non entité peut hériter d'une classe entité.

Exemple de l'héritage d'une classe abstraite

```
@Entity  
public abstract class Personne{  
  
    @Id  
    protected String numSecuSociale;  
  
}
```



```
@Entity  
public class Employe extends Personne{  
  
    protected float salaire;  
  
}
```

Les MappedSuperClasses

- Les entités peuvent hériter de classes qui ont des états persistants et des informations de mapping mais qui ne sont pas des entités.
- Les *MappedSuperClasses* sont utiles pour mettre en commun des états ou des informations de mapping entre plusieurs classes ;
- Les *MappedSuperClasses* ne peuvent pas être utilisée par un EntityManager.

```
@MappedSuperclass  
public abstract class Personne{  
  
    @Id  
    protected String numSecuSociale;  
  
}
```



```
@Entity  
public class Employe extends Personne{  
  
    protected float salaire;  
  
}
```

Les stratégies de mapping de l'héritage

- Différentes stratégies peuvent être utilisées pour le mapping de l'héritage :
 - avoir seule table par hiérarchie de classe (choix sélectionnée par défaut) :

```
@Inheritance(strategy=SINGLE_TABLE)
```

- une table par classe entité concrète :

```
@Inheritance(strategy=TABLE_PER_CLASS)
```

- une stratégie de type “join” (où les propriétés spécifiques à une classe héritée sont mappées dans un table différente des propriétés communes de la classe de base) :

```
@Inheritance(strategy=JOINED)
```


La stratégie “une seule table par hiérarchie de classe”

- Avec la stratégie “une seule table par hiérarchie de classe”, une colonne ajoutée à la table sert de discriminateur pour sélectionner la classe utilisée ;
- le nom par défaut de cette colonne est *DTYPE* ;
- le types possibles sont *DiscriminatorType.STRING* (choix par défaut), *DiscriminatorType.CHAR*, *DiscriminatorType.INTEGER*.
- exemple où on choisit le nom de la colonne ainsi que son type :

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATEUR_PERSONNE"
discriminatorType=DiscriminatorType.INTEGER)
public class Personne{
...
}
```

Comparaison des stratégies

- la stratégie “une seule table par hiérarchie de classe” est implémentée par toutes les solutions offrant le service de Java Persistence :
 - bon support du polymorphisme ;
 - mais les colonnes correspondant aux états des sous-classes doivent pouvoir être “null”.
- la stratégie “une table par classe concrète” n'est pas toujours implémentée :
 - elle ne gère pas pleinement le polymorphisme ;
- la stratégie de type “join” n'est pas toujours implémentée :
 - bon support du polymorphisme ;
 - mais les opérations de jointure consomment du temps.

La gestion des entités

Le manager des entités

- Chaque entité existe dans un espace appelé un *PersistenceContext* ;
- le *PersistenceContext* gère le cycle de vie des entités qui lui sont associées (création, persistance, recherche et destruction d'entités) ;
- par programmation, le cycle de vie est géré par un *EntityManager* ;
- un *EntityManager* peut être géré :
 - par le conteneur des entités (si les entités s'exécutent dans un serveur d'applications) ;
 - par l'application.

Les *EntityManager* gérés par un conteneur

- Si des entités s'exécutent dans un serveur d'applications, elles sont gérées par un conteneur ;
- Le contexte de persistance est injecté automatiquement dans les composants qui l'utilisent => inutile de transmettre un *EntityManager* au composant dans un constructeur par exemple ;
- Pour obtenir un *EntityManager* géré par un conteneur :

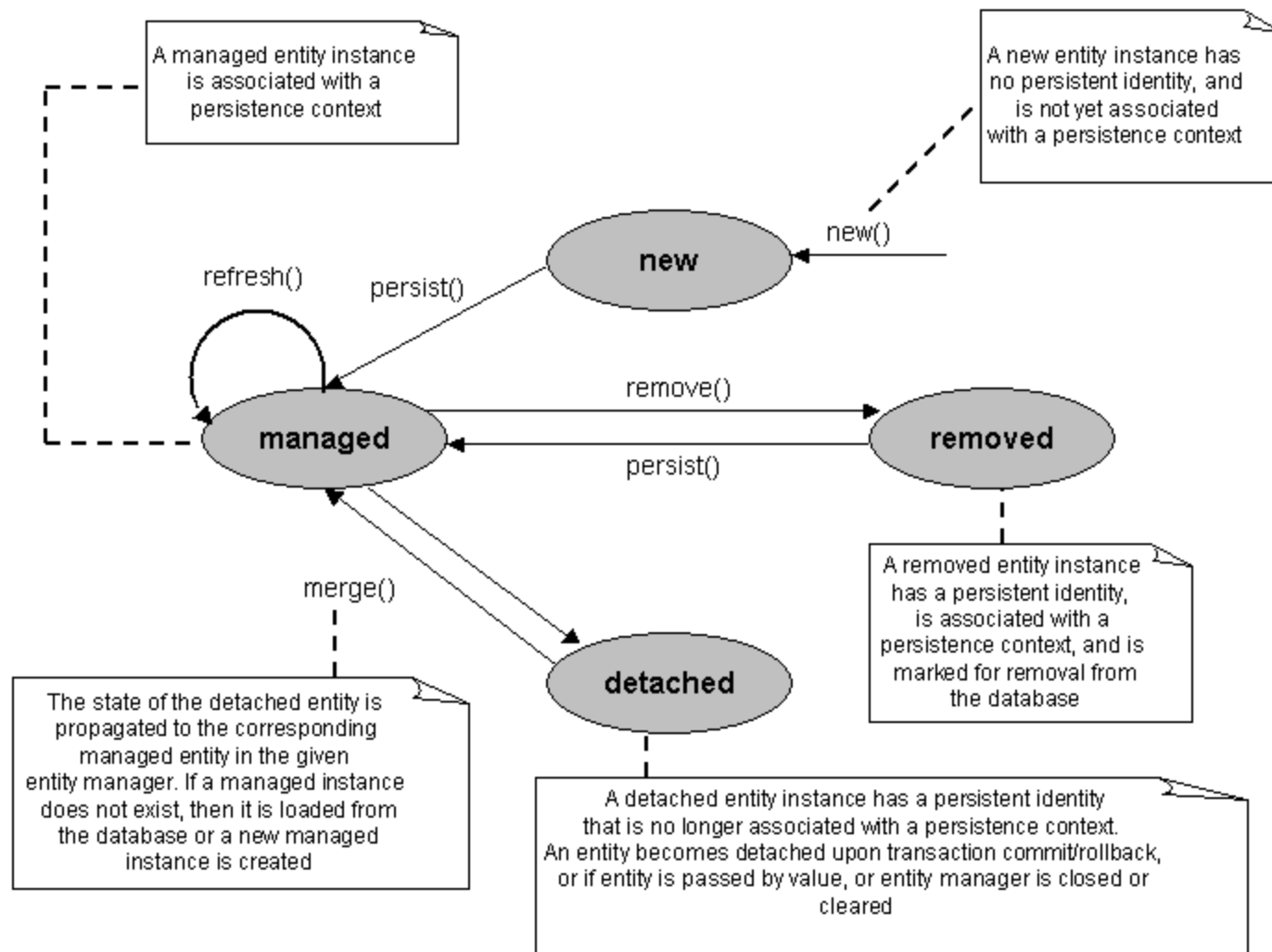
```
@PersistenceContext  
EntityManager em;
```

Les EntityManager gérés par une application

- Une application qui souhaite gérer des *EntityManager* doit :
- les créer et les détruire explicitement ;
- les transmettre aux composants qui les utilisent ;
- chaque EntityManager crée un nouveau contexte de persistance isolé des autres ;
- pour obtenir un *EntityManager* géré par une application :

```
@PersistenceUnit  
EntityManagerFactory emf;  
EntityManager em =  
emf.createEntityManager();
```

Le cycle de vie des entities



Creer en rendre des entités persistantes

- Une nouvelle entité n'est pas associée à un contexte de persistance, elle n'a pas d'identité de persistance :

```
Personne personne = new Personne();
```

- il faut appeler la méthode *persist* pour associer une entité à un contexte de persistance et lui donner un identité :

```
@PersistenceContext  
EntityManager em;  
  
...  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
Personne personne = new Personne();  
em.persist( personne );  
tx.commit();  

```

- la persistance se propage à toute les entités associées qui ont leur paramètre cascade à PERSIST ou ALL :

```
@OneToMany(cascade=PERSIST, mappedBy="voiture")  
public Collection<Personne> getPassagers(){  
    return passagers;  
}
```


Les objets attachés à une session

- attention : les objets persistants restent synchronisés avec la base de données tant que la session n'a pas été vidée (avec un clear).

```
MonEntity monEntity = new MonEntity();  
monEntity.setI(10);  
  
EntityTransaction tx = entityManager.getTransaction();  
tx.begin();  
entityManager.persist(monEntity);  
tx.commit();  
  
tx.begin();  
monEntity.setI(11);  
tx.commit();
```

Un update est fait dans la base => i = 11 !

Synchroniser, rechercher et détruire des entités

- Synchroniser une entité avec le base de données : la synchronisation est automatique et se produit quand la transaction associée à l'entité est commise ; on peut forcer la synchronisation avec :

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

- rechercher un entité :

```
@PersistenceContext  
EntityManager em;  
  
long id = ...  
Personne personne = em.find(Personne.class, id );
```



- détruire une entité :

```
@PersistenceContext  
EntityManager em;  
  
long id = ...  
Personne personne = em.find(Personne.class, id );  
  
em.remove( personne );
```

Le Java Persistence Query Language

- Il est possible d'utiliser des requêtes pour rechercher des entités répondant à des critères précis ;
- les requêtes peuvent être créées dynamiquement au moment de l'exécution :

```
@PersistenceContext  
EntityManager em;
```

```
List liste = em.createQuery( "SELECT p FROM Personne p WHERE p.name LIKE :nomPersonne" )  
.setParameter( "nomPersonne", ... )  
.getResultList();
```



```
for (Iterator it = liste.iterator(); it.hasNext();){  
    Personne personne = (Personne) it.next();  
    ...  
}
```

Le Java Persistence Query Language

- Les requêtes peuvent être statiques :

```
@Entity
@NamedQuery(name = "findAllPersonnes", query = "SELECT p FROM Personne p")
public class Personne{
    ...
}
```

```
@PersistenceContext
EntityManager em;

...
List personnes = em.createNamedQuery("findAllPersonnes").getResultList();
for (Iterator it = personnes.iterator(); it.hasNext();){
    Personne personne = (Personne) it.next();
    ...
}
```