JAVA

m2iformation.fr

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

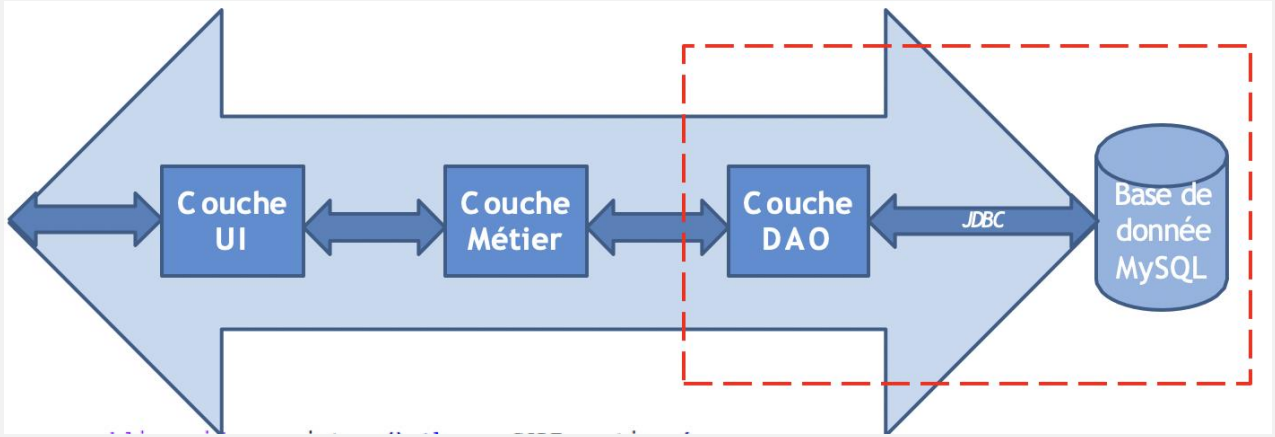
JPA

Java

Programme

- Historique
- ORM : Object Relational Mapping
- JPA : JAVA Persistence API
- Entity: Entités
- Autres annotations
- Contexte de persistance
- Opérations prises en charge par le gestionnaire d’entités
- Cycle de vie d’une instance d’entité
- Obtention d’une fabrique EntityManagerFactory
- Création du gestionnaire d’entité EntityManager
- Exemple d’insertion d’un livre
- Relations entre entités

Historique : Accès directement à la base de donnée grâce à l'API standard JDBC de Java




```
public void enregistrer() throws SQLException {
        //ouverture de la connexion jdbc
        Class.forName("com.mysql.jdbc.driver");
        String url = "jdbc:mysql://localhost/Bdbanque";
        Connection con = DriverManager.getConnection(url, "login", "password");

        //preparation  ou construction de la requête sql
        String insertStatement = "Insert into Client(Nom, Prenom,Nature) values (?, ?, ?)";
        PreparedStatement prepStm1 = con.prepareStatement(insertStatement);
        prepStm1.setString(1,txtNom.getText() );
        prepStm1.setString(2,txtPrenom.getText());
        prepStm1.setString(3,txtAge.getText() );
        //execution de la requête
        prepStm1.executeUpdate();
        prepStm1.close();

        //fermeture de la connexion jdbc
        con.close();
}
```

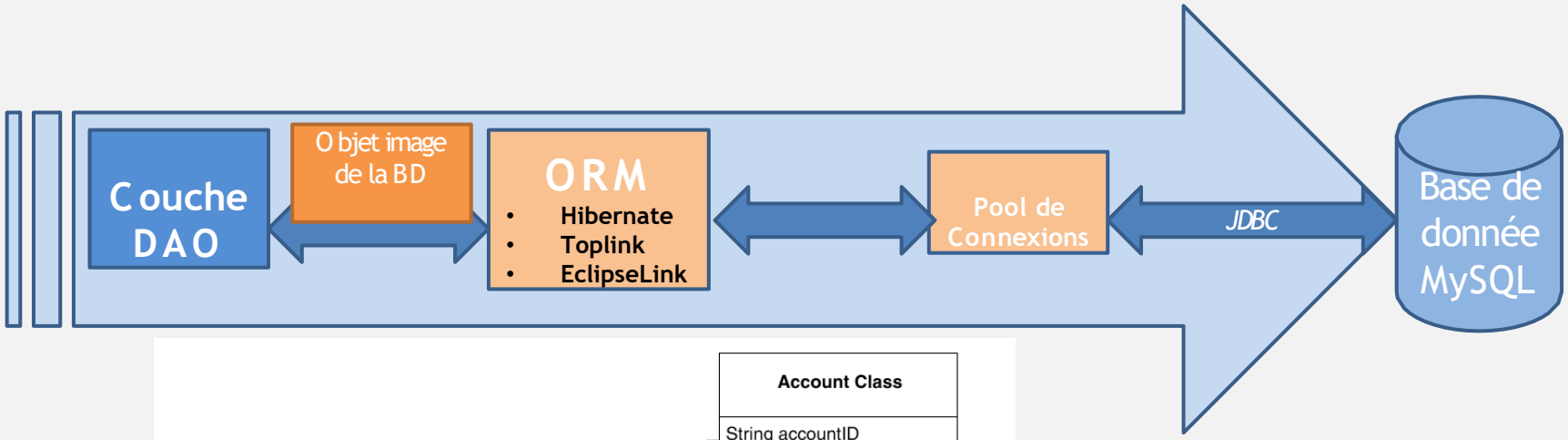
Problématiques de l'accès direct à la BD à l'aide de jdbc :

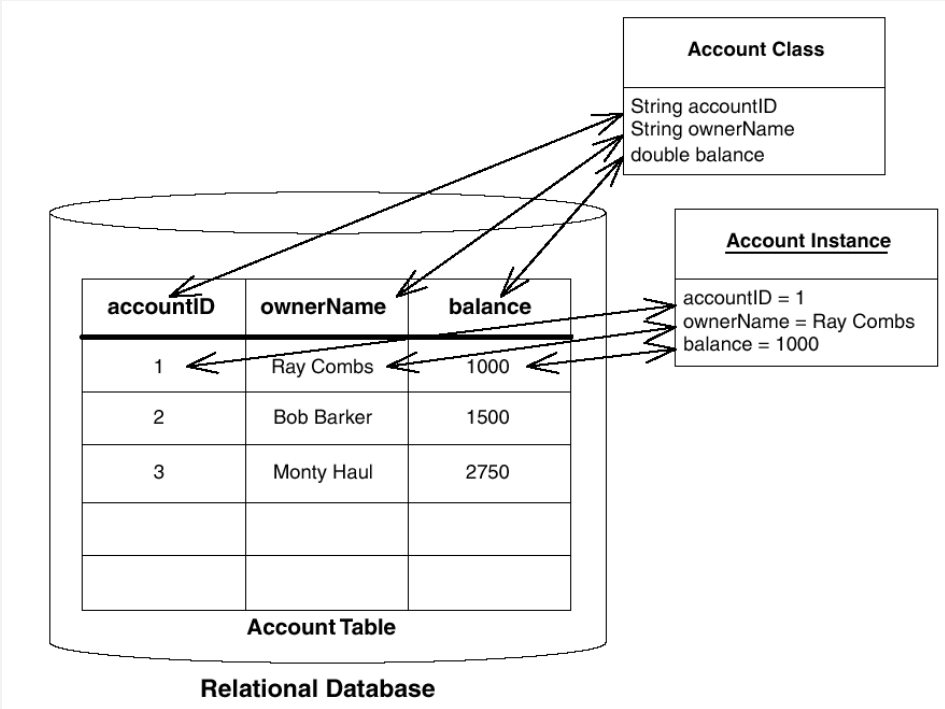
- Pour des raisons de performances, le coût d'ouverture / fermeture d'une connexion n'est pas négligeable,
  - Connection con = DriverManager.getConnection(url, "login", "password");
  - //les ordres SQL
  - con.close();
- L'utilisation du langage SQL rend la couche DAO difficilement maintenable,
  - String insertStatement = "Insert into Client(Nom, Prenom,Nature) values (?, ?, ?)"

Pour pallier à cette problématique et faciliter l'écriture de la couche DAO,

La communauté Java a donc fait naître des **Frameworks ORM**, tels que **Hibernate**, **Toplink**, **EclipseLink**

ORM : Object Relational Mapping

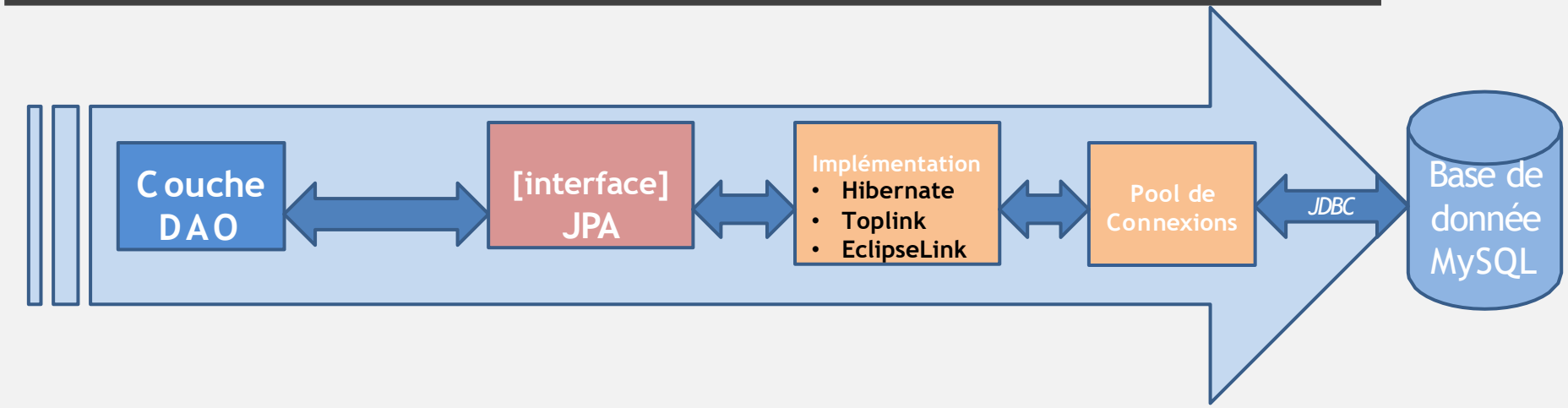




Devant le succès des Frameworks ORM,

Sun a décidé de standardiser une couche ORM via une spécification appelée **JPA** apparue en même temps que Java 5

JPA : Java Persistence Api



La spécification JPA est un ensemble d'interface du package javax.persistence

Exemple :

javax.persistence.Entity;
javax.persistence.EntityManagerFactory;
javax.persistence.EntityManager;
javax.persistence.EntityTransaction;

[EntityManager]
void **persist(**Entity entité**)**

JPA : Java Persistence Api

Ses principaux avantages sont les suivants :

1. JPA peut être utilisé par toutes les applications **Java, Java SE ou Java EE.**

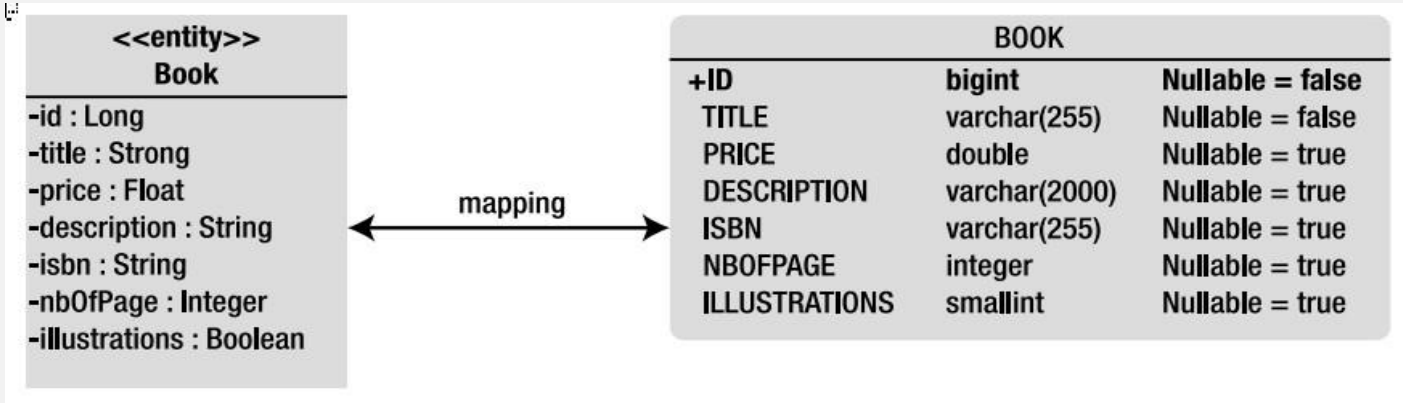
2. Mapping O /R (objet-relationnel) avec les tables de la BD , facilitée par les Annotations.

3. Un langage de requête objet standard **JPQL** pour la récupération des objets,

```
select p from Personne p order by p.nom asc
```

Entity: Entités

```
@Entity
@Table(name = "BOOK")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    @Column(name = "TITLE",nullable = false)
    private String title;
    private Float price;
    @Basic(fetch = FetchType.LAZY)
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations

  //Les Getters et les Setters
```



Contexte de persistance

Ensemble des instances d'entités gérées a un instant donné,

- **gérées par qui ?**
- Le gestionnaire d'entités : EntityManager
- Ce contexte peut donc être considéré comme **un cache de premier niveau** : c'est un espace réduit ou le gestionnaire stocke les entités avant d'écrire son contenu dans la base de données,

Les Entités

- **Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécification de JPA**
  - **Le développeur indique qu'une classe est une entité en lui associant l'annotation @Entity**
  - **Il faut importer javax.persistence.Entity dans les classes entités.**
- Les entités dans les spécifications de l'API Java Persistence permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables.
- Ce sont de simples POJO
  - Un POJO est une classe Java qui n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.

Classes entités : Entity Class

- **Une classe entité doit posséder un attribut qui représente la clé primaire dans la BD (@Id)**
  - **Elle doit avoir un constructeur sans paramètre protected ou public** sans argument et la classe du bean doit obligatoirement être marquée avec l'annotation **@javax.persistence.Entity**. Cette annotation possède un attribut optionnel nommé **name** qui permet de préciser le nom de l'entité dans les requêtes. Par défaut, ce nom est celui de la classe de l'entité.
  - **Elle ne doit pas être final**
  - **Aucune méthode ou champ persistant ne doit être final**
  - **Une entité peut être une classe abstraite mais elle ne peut pas être une interface**

Les annotations

Annotation	Rôle
@javax.persistence.Table	Préciser le nom de la table concernée par le mapping
@javax.persistence.Column	Associé à un getter, il permet d'associer un champ de la table à la propriété
@javax.persistence.Id	Associé à un getter, il permet d'associer un champ de la table à la propriété en tant que clé primaire
@javax.persistence.GeneratedValue	Demander la génération automatique de la clé primaire au besoin
@javax.persistence.Basic	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
@javax.persistence.Transient	Demander de ne pas tenir compte du champ lors du mapping

L'annotation @javax.persistence.Table

- Permet de lier l'entité à une table de la base de données. Par défaut, l'entité est liée à la table de la base de données correspondant au nom de la classe de l'entité. Si ce nom est différent alors l'utilisation de l'annotation @Table est obligatoire.

Attributs	Rôle
name	Nom de la table
catalog	Catalogue de la table
…	…

L'annotation @javax.persistence.Column

- Permet d'associer un membre de l'entité à une colonne de la table. Par défaut, les champs de l'entité sont liés aux champs de la table dont les noms correspondent. Si ces noms sont différents alors l'utilisation de l'annotation **@Column** est obligatoire.

Attributs	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi–table
unique	Indique si la colonne est unique
Nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update

L'annotation @Id

- **Il faut obligatoirement** définir une des propriétés de la classe avec l'annotation @Id pour la déclarer comme étant la clé primaire de la table.
- Cette annotation peut marquer soit **le champ** de la classe concernée soit le **getter** de la propriété.
- L'utilisation de l'un ou l'autre précise au gestionnaire s'il doit se baser sur les champs ou les getter pour déterminer les associations entre l'entité et les champs de la table. La clé primaire peut être constituée d'une seule propriété ou composées de plusieurs propriétés qui peuvent être de type primitif ou chaîne de caractères

EXEMPLE

```java
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {

    @Id  @GeneratedValue
    private int id;

    private String prenom;
    private String nom;
    private static final long serialVersionUID = 1L;

    public Personne() {
      super();
    }
    public int getId() {
      return this.id;
    }
    public void setId(int id) {
      this.id = id;
    }
    public String getPrenom() {
      return this.prenom;
    }
    public void setPrenom(String prenom) {
      this.prenom = prenom;
    }
    public String getNom() {
      return this.nom;
    }
    public void setNom(String nom) {
      this.nom = nom;
    }

}
```

Cas d’une Clé primaire composée

- Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes.
- L'API Java Persistence propose deux façons de gérer ce cas de figure :
  - L'annotation @javax.persistence.IdClass
  - L'annotation @javax.persistence.EmbeddedId

L'annotation @javax.persistence.IdClass

- L'annotation @IdClass s'utilise avec une classe qui va encapsuler les propriétés qui composent la clé primaire. Cette classe doit obligatoirement :
  - Être sérialisable
  - Posséder un constructeur sans arguments
  - Fournir une implémentation dédiée des méthodes **equals() et hashCode()**

Exemple: PersonnePK (1/3)

```java
public class PersonnePK implements java.io.Serializable {

    private static final long serialVersionUID = 1L;
    private String nom; private String prenom;

    public PersonnePK() { }

    public PersonnePK(String nom, String prenom) {
      this.nom = nom; this.prenom = prenom;
    }

    public String getNom() {
      return this.nom;
    }
    public void setNom(String nom) {
      this.nom = nom;
    }
    public String getPrenom() {
      return prenom;
    }
    public void setPrenom(String prenom) {
      this.nom = prenom;
    }
    public boolean equals(Object obj) {

      boolean resultat = false;
        if (obj == this) {
          resultat = true;
        }else {
          if (!(obj instanceof PersonnePK)) {
            resultat = false;
          }else {
            PersonnePK autre = (PersonnePK) obj;

            if (!nom.equals(autre.nom)) {
                resultat = false;
            }else {
                if (prenom != autre.prenom) {
                    resultat = false;
                }else {
                    resultat = true;
                }
            }
         }
      }
      return resultat;
    }

    public int hashCode() {
      return (nom + prenom).hashCode()
        ;
    }
}
```

Exemple: (2/3)

Il est nécessaire de définir la classe de la clé primaire dans le fichier de configuration persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.0" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
<persistence-unit name="MaBaseDeTestPU">
<provider>oracle.toplink.essentials.PersistenceProvider</provider>
<class>com.jmd.test.jpa.Personne</class>
<class>com.jmd.test.jpa.PersonnePK</class>
</persistence-unit>
</persistence>
```

Exemple (3/3)

- **Il faut utiliser l'annotation @IdClass sur la classe de l'entité.**
- Il est nécessaire de marquer chacune des propriétés de l'entité qui compose la clé primaire avec l'annotation @Id.
- **Ces propriétés doivent avoir le même nom dans l'entité et dans la classe qui encapsule la clé primaire**.

```java
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {

    private String prenom;
    private String nom;
    private int taille;
    private static final long serialVersionUID = 1L;

    public Personne() {
      super();
    }

    @Id
    public String getPrenom() {
      return this.prenom;
    }
    public void setPrenom(String prenom) {
      this.prenom = prenom;
    }
    @Id
    public String getNom() {
      return this.nom;
    }
    public void setNom(String nom) {
      this.nom = nom;
    }
    public int getTaille() {
      return this.taille;
    }
    public void setTaille(int taille) {
      this.taille = taille;
    }
}
```

Classes entités : Embedded Class

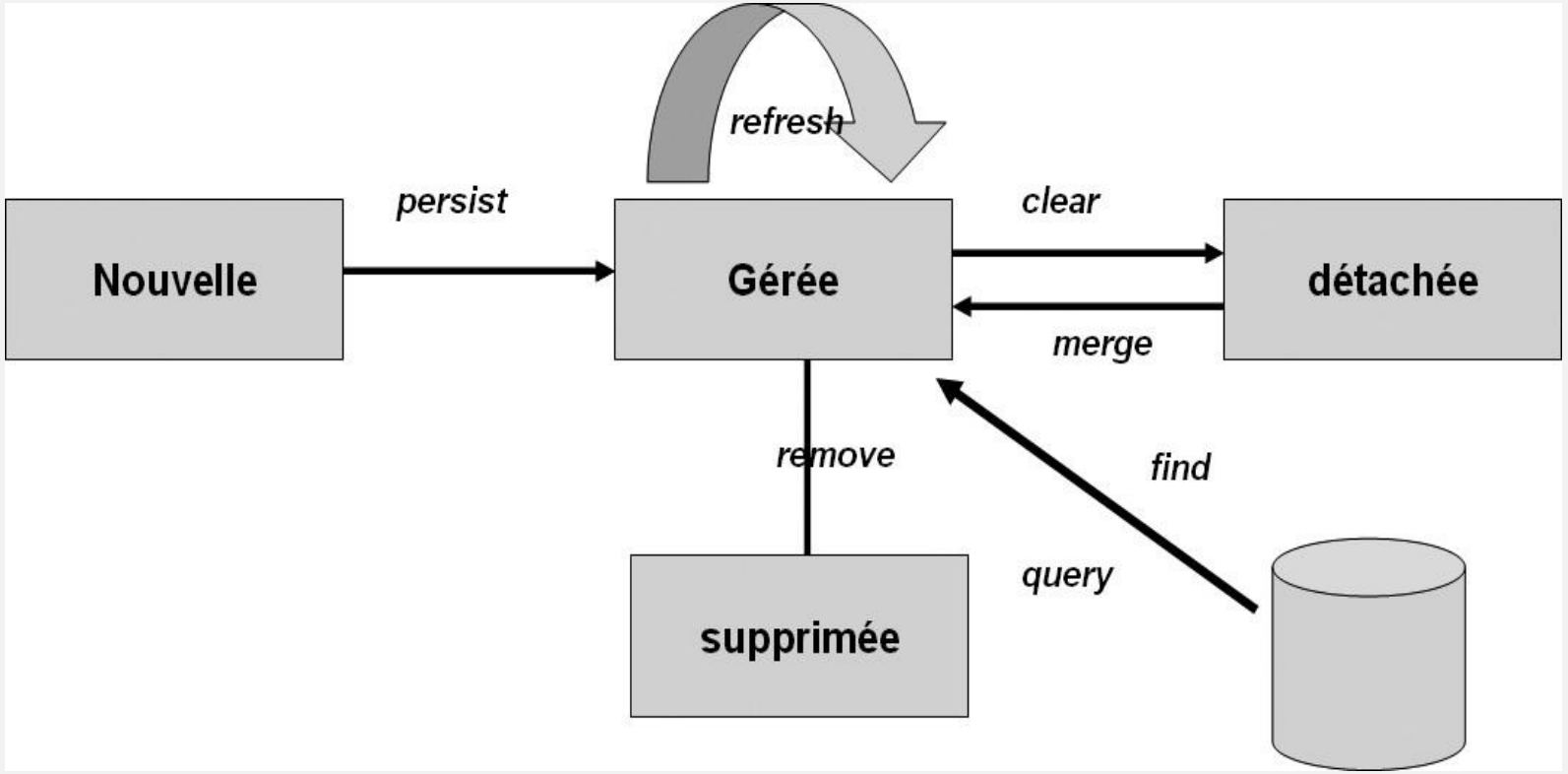
- **Il existe des classes incorporées (*embedded) dont les données n'ont* pas d'identité dans la BD mais sont insérées dans une des tables associées à une entité persistante**
  - **– Par exemple, une classe Adresse dont les valeurs sont insérées dans la table Employe**

Opérations prises en charge par le gestionnaire d'entité

Opération	Description
persist()	Insère l'état d'une entité dans la base de données. Cette nouvelle entité devient alors une entité gérée.
remove()	Supprime l'état de l'entité gérée et ses données correspondantes de la base.
refresh()	Synchronise l'état de l'entité à partir de la base, les données de la BD étant copiées dans l'entité.
merge()	Synchronise les états des entités « détachées » avec le PC. La méthode retourne une entité gérée qui a la même identité dans la base que l'entité passée en paramètre, bien que ce ne soit pas le même objet.
find()	Exécute une requête simple de recherche de clé.
CreateQuery()	Crée une instance de requête en utilisant le langage JPQL.
createNamedQuery()	Crée une instance de requête spécifique.
createNativeQuery()	Crée une instance de requête SQL.
contains()	Spécifie si l'entité est managée par le PC.
flush()	Toutes les modifications effectuées sur les entités du contexte de persistance gérées par le gestionnaire d'entités sont enregistrées dans la BD lors d'un flush du gestionnaire.

NB : Un flush() est automatiquement effectué au moins à chaque **commit de la transaction** en cours,

Cycle de vie d'une instance d'entité



Obtention d'une fabrique EntityManagerFactory

L' interface **EntityManagerFactory** permet d'obtenir une instance de l'objet EntityManager,

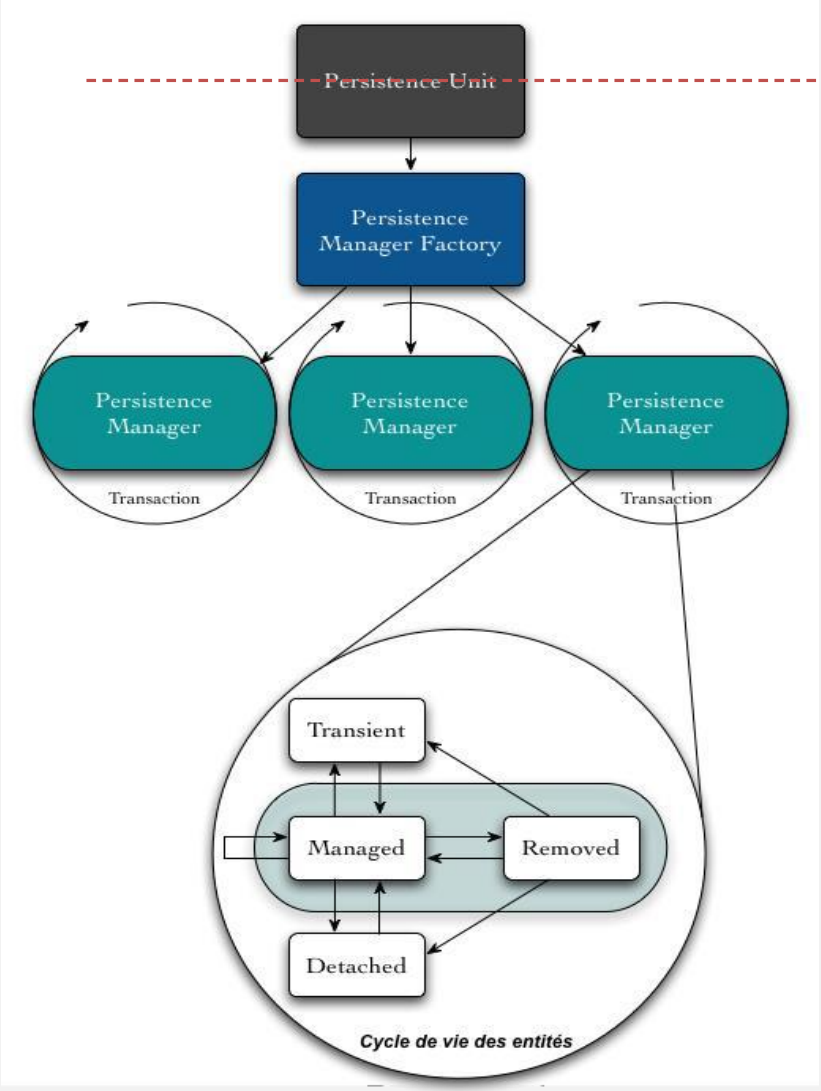
```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
```

« *jpa* » est le nom de l'unité de persistance, définie dans le fichier de configuration de la couche JPA *META-INF*/persistence.xml.

Création du gestionnaire d'entité EntityManager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");

EntityManager em = emf.createEntityManager() ;
```

Fonctionnement de JPA



• La **Persistence Unit** : organise les meta données qui définissent le mapping entre les entités et la base de donnée dans le fichier de configuration *METAINF/persistence.xml*

•La **Persistence Manager Factory** récupère ces metas données de la Persistence Unit et les interprètent pour créer des Persistence Manager (EntityManager)

•Le **Persistence Mangager** (EntiyManager) gère les échanges entre le code et la base de donnée, c'est à dire le cycle de vie des **entités**

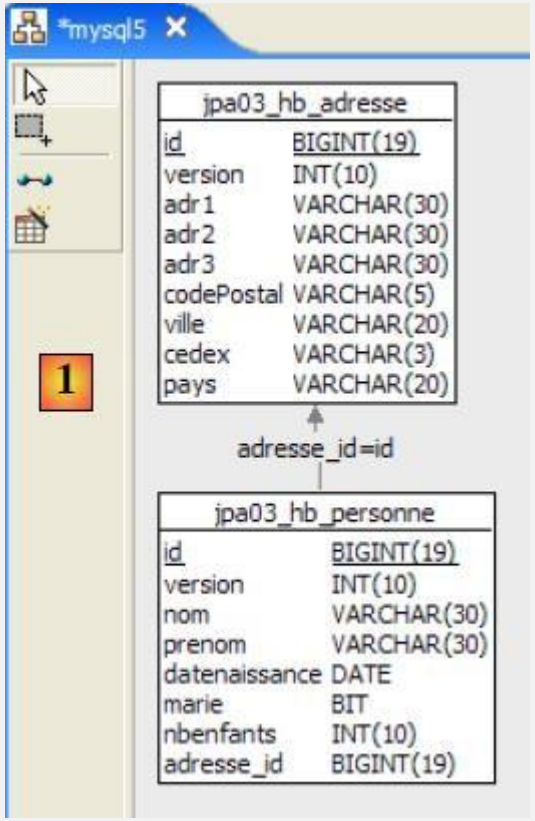
•Enfin, les opérations du EntityManager est englobé dans une **Transaction.**

Relations

- Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations.
- Ces relations sont transposées dans les liaisons que peuvent les différentes entités correspondantes.
- **4 types de relations à définir entre les entités de la JPA:**
  - 1−1 (One to One)
  - 1−n (Many to One)
  - 1−n (One to Many)
  - n−n (Many to Many)

Relations entre entités

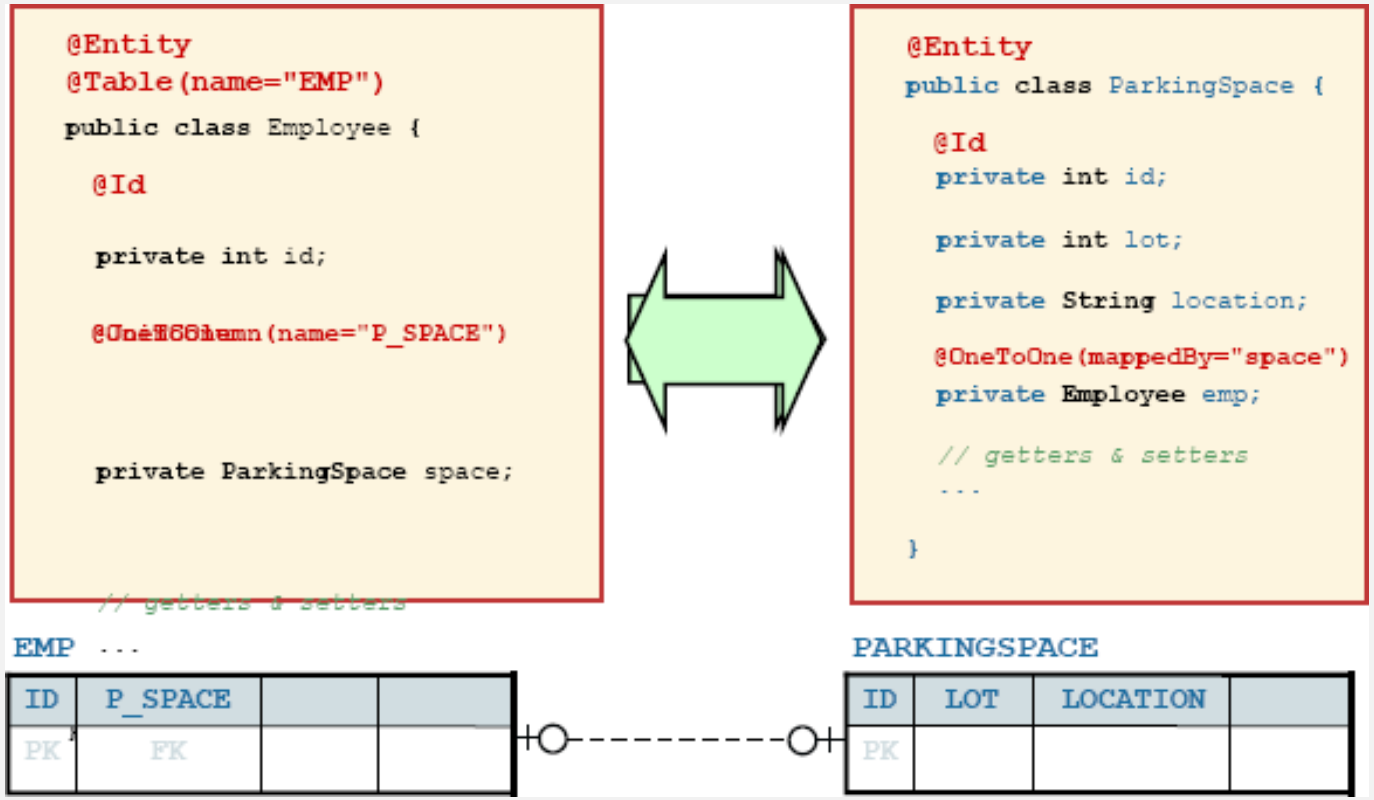
Relation un-à-un




```java
@Entity
@Table(name = "jpa03_hb_personne")
public class Personne {
...
    @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
    /*@OneToOne(cascade = {CascadeType.MERGE,    CascadeType.PERSIST,
      CascadeType.REFRESH, CascadeType.REMOVE}, fetch=FetchType.LAZY) */
    @JoinColumn(name = "adresse_id", unique = true, nullable = false)
    private Adresse adresse;
  ...

}
```

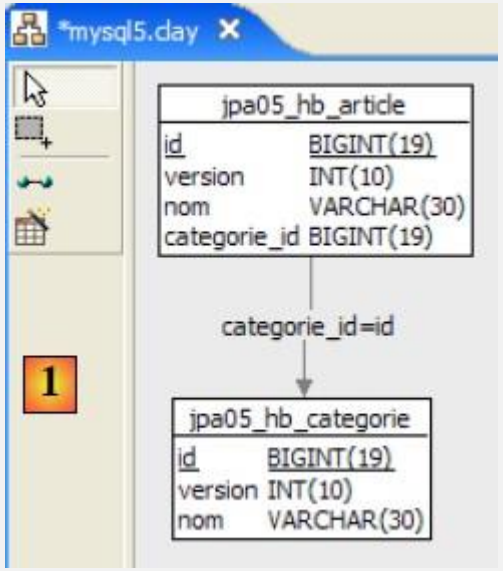
```java
@Entity
@Table(name = "jpa03_hb_adresse")
public class Adresse{
    ...
    /*n'est pas obligatoire*/
    @OneToOne(mappedBy = "adresse", fetch=FetchType.EAGER)
    private Personne personne;
        ...

}
```

Relationships: One to One



Relations entre entités

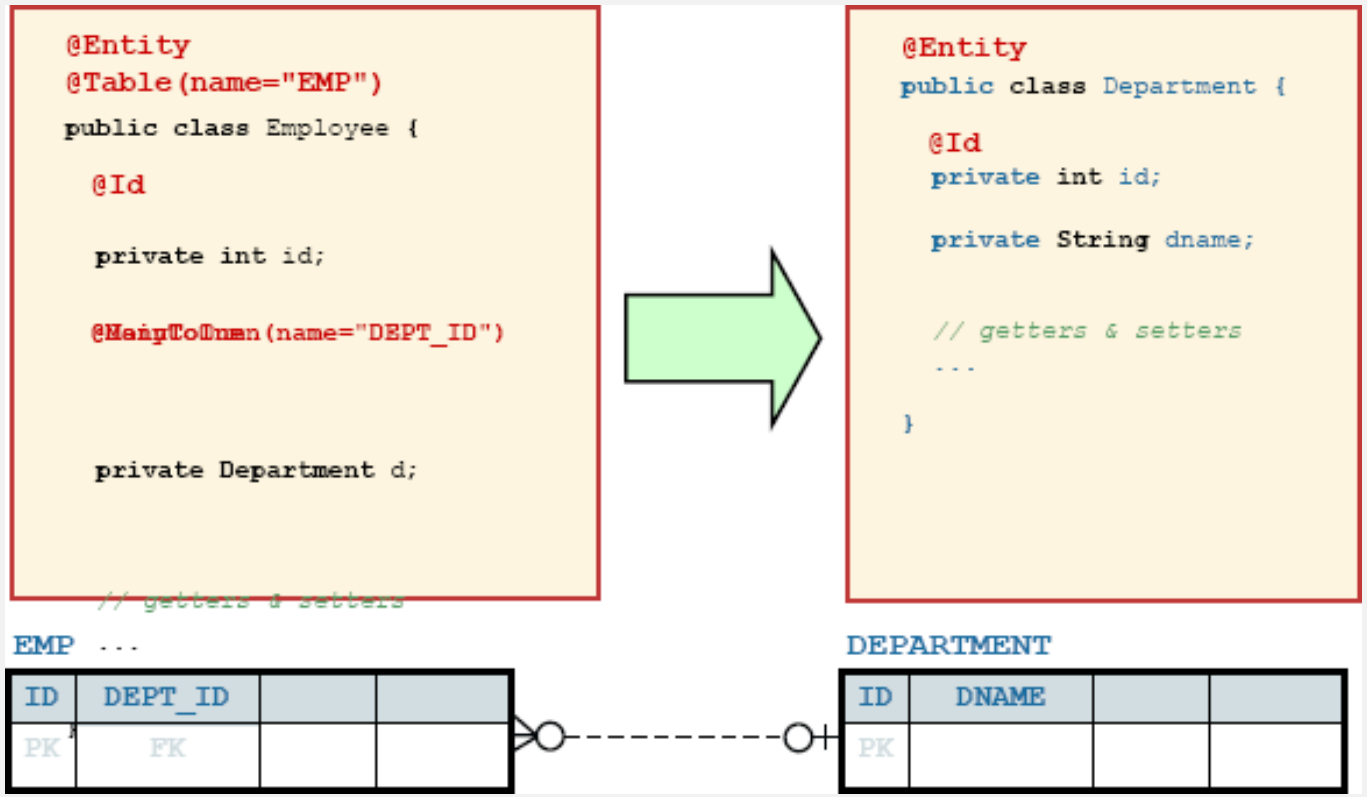
Relation un-à-plusieurs et plusieurs-à-un




```java
@Entity
@Table(name="jpa05_hb_article")
public class Article{
...
// relation principale Article (many) -> Category (one) implémentée par une clé
// étrangère (categorie_id) dans Article
// 1 Article a nécessairement 1 Categorie (nullable=false)

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name = "categorie_id", nullable = false)
    private Categorie categorie;
    ...
}
```

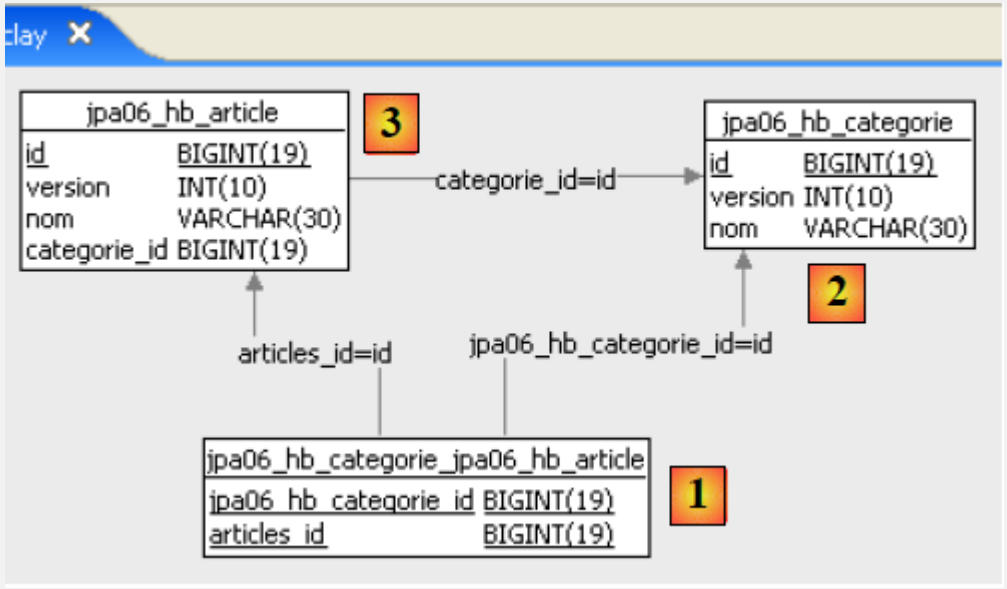
```java
@Entity
@Table(name="jpa05_hb_categorie")
public class Categorie{
    ...
// relation inverse Categorie (one) -> Article (many) de la relation Article (many)— > Categorie
(one)
      @OneToMany(mappedBy = "categorie", cascade = { CascadeType.ALL })
      private Set<Article> articles = new HashSet<Article>();
        ...

}
```

Relationship: Many to One



Relations entre entités

Relation un-à-plusieurs et plusieurs-à-un




```java
@Entity  @Table(name="jpa06_hb_categorie")
public class Categorie{
...
// relation OneToMany non inverse (absence de mappedby) Categorie (one) -> Article (many)
// implémentée par une table de jointure Categorie_Article pour qu'à partir d'une catégorie
// on puisse atteindre plusieurs articles

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private Set<Article> articles = new HashSet<Article>();
      ...

}
```

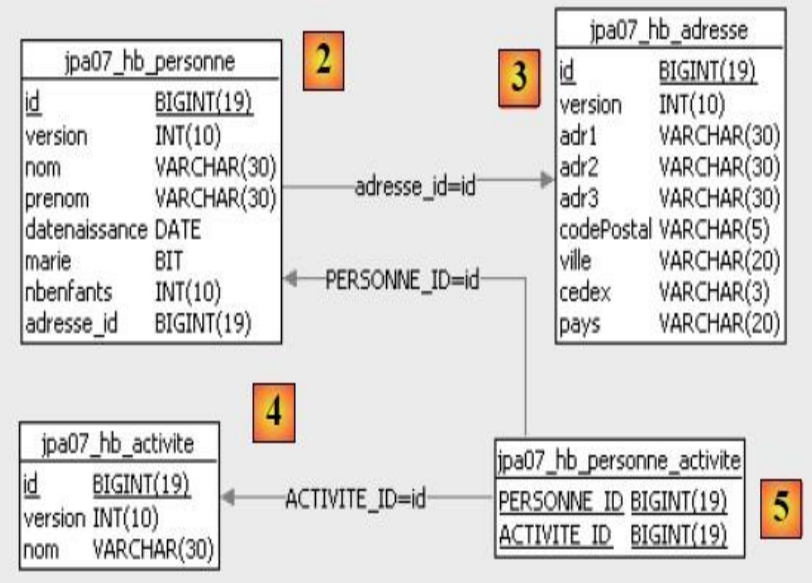
Relationship: One to Many

```java
@Entity
@Table(name="EMP")
public class Employee {

  @Id

  private int id;
  @ManyToOne
  @JoinColumn(name="DEPT_ID")


  private Department d;


  // getters & setters
```

```java
@Entity
public class Department {

  @Id
  private int id;

  private String dname;

  @OneToMany(mappedBy="d")
  private Collection<Employee> emps;

  // getters & setters
  ...

}
```

Relations entre entités

Relation plusieurs-à-plusieurs



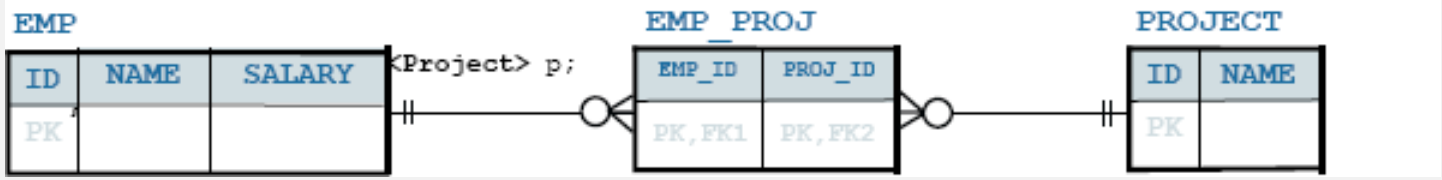

```java
@Entity
@Table(name = "jpa07_hb_personne")
public class Personne {
...
    // relation Personne (many) -> Activite (many) via une table de jointure  personne_activite

    @ManyToMany(cascade={CascadeType.PERSIST})
    @JoinTable(name="jpa07_hb_personne_activite", joinColumns = @JoinColumn(name = "PERSONNE_ID"),
    inverseJoinColumns = @JoinColumn(name = "ACTIVITE_ID"))
    private Set<Activite> activites = new HashSet<Activite>();
    ...

}
```

```java
@Entity
@Table(name = "jpa07_hb_activite")
public class Activite {
...
// relation inverse Activite -> Personne  @ManyToMany(mappedBy = "activites")
    private Set<Personne> personnes = new HashSet<Personne>();
    ...

}
```

Relationships: Many to Many

```java
@Entity
@Table(name="EMP")
public class Employee {

  @Id

  private int id;

  @JoinTable(name="EMP_PROJ",
      joinColumns=
      @JoinColumn(name="EMP_ID"),
      inverseJoinColumns=
@ManyToMany(name="PROJ_ID"))
```

```java
@Entity
public class Project {

  @Id
  private int id;

  private String name;

  @ManyToMany(mappedBy="p")
  private Collection<Employee> e;

  // getters & setters
  ...

}
```



EntityManager

- **Classe javax.persistence.EntityManager**
  - **Le gestionnaire d'entités est l'interlocuteur principal pour le développeur.**
- Les interactions entre la base de données et les beans entité sont assurées par un objet de type javax.persistence.EntityManager : il permet de lire et rechercher des données mais aussi de les mettre à jour (ajout, modification, suppression). L'EntityManager est donc au coeur de toutes les actions de persistance.
  - **Il fournit les méthodes pour gérer les entités :**
    - les rendre persistantes,
    - les supprimer de la base de données,
    - retrouver leurs valeurs dans la base,
    - etc.

Cycle de vie d'un EntityManager

- **La méthode createEntityManager() de la classe EntityManagerFactory créé un Entity Manager**
  - **L'Entity Manager est supprimé avec la méthode close() de la classe EntityManager, il ne sera plus possible de l'utiliser ensuite.**

Fabrique de l'EntityManager

- La classe Persistence permet d'obtenir une fabrique de gestionnaire d'entités par la méthode **`createEntityManagerFactory`**
- **2 variantes surchargées de cette méthode :**
  - 1 seul paramètre qui donne le nom de l'unité de persistance (définie dans le fichier **persistence.xml**)
  - Un 2ème paramètre de type Map qui contient des valeurs qui vont écraser les propriétés par défaut contenues dans persistence.xml

Méthodes de EntityManager

- void flush()
  - Toutes les modifications effectuées sur les entités du contexte de persistance gérées par l'EntityManager sont enregistrées dans la BD
  - void persist(Object *entité)*
    - Une entité nouvelle devient une entité gérée, l'état de l'entité sera sauvegardé dans la BD au prochain *flush ou commit.*
  - void remove(Object *entité)*
    - Une entité gérée devient supprimée, les données correspondantes seront supprimées de la BD

Méthodes de EntityManager

- void lock(Object *entité, LockModeType lockMode)*
  - Le fournisseur de persistance gère les accès concurrents aux données de la BD représentées par les entités avec une stratégie optimiste, lock permet de modifier la manière de gérer les accès concurrents à une entité
- void **refresh(Object *entité)***
  - L'EntityManager peut synchroniser avec la BD une entité qu'il gère en rafraichissant son état en mémoire avec les données actuellement dans la BD.
  - Utiliser cette méthode pour s'assurer que l'entité a les mêmes données que la BD.
- <T> T find(Class<T> *classeEntité,Object cléPrimaire)*
  - La recherche est polymorphe : l'entité récupérée peut être de la classe passée en paramètre ou d'une sous-classe (renvoie **null si aucune entité n'a** l'identificateur passé en paramètre)

Utilisation de EntityManager pour la création d'une occurrence

- Pour insérer une nouvelle entité dans la base de données, il faut :
  - Instancier une occurrence de la classe de l'entité
  - Initialiser les propriétés de l'entité
  - Définir les relations de l'entité avec d'autres entités, et au besoin
  - Utiliser la méthode persist() de l'EntityManager en passant en paramètre l'entité

EXEMPLE

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA {

    public static void main(String[]argv) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transac = em.getTransaction();
        transac.begin();
        Personne nouvellePersonne = new Personne();
        nouvellePersonne.setId(4);
        nouvellePersonne.setNom("nom4");
        nouvellePersonne.setPrenom("prenom4");
        em.persist(nouvellePersonne);
        transac.commit();
        em.close();
        emf.close();
    }
}
```

Utilisation de EntityManager pour rechercher des occurrences

- Pour effectuer des recherches de données, l'EntityManager propose deux mécanismes :
  - La recherche à partir de la clé primaire
  - La recherche à partir d'une requête utilisant une syntaxe dédiée
- Pour la recherche par clé primaire, la classe EntityManager possède les méthodes `find()` et `getReference()` qui attendent toutes les deux en paramètres un objet de type Class représentant la classe de l'entité et un objet qui contient la valeur de la clé primaire.

Avec Find()

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class TestJPA5 {

    public static void main(String[] argv) {

    EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();  Personne personne = em.find(Personne.class, 4);

      if (personne != null) {
            System.out.println("Personne.nom=" + personne.getNom());
      }

    em.close();
    emf.close();
    }
}
```

Avec getReference()

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityNotFoundException;
import javax.persistence.Persistence;

public class TestJPA6 {

    public static void main(String[] argv) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
            try {
                Personne personne = em.getReference(Personne.class, 5);
                System.out.println("Personne.nom=" + personne.getNom());
            } catch (EntityNotFoundException e) {
                System.out.println("personne non trouvée");
            }
        em.close();
        emf.close();
    }
}
```

Queries

- **Il est possible de rechercher des données sur des critères plus complexes que la simple identité. Les étapes sont alors les suivantes :**
  1. Décrire ce qui est recherché (langage JPQL)
  2. Créer une instance de type **Query**
  3. Initialiser la requête (paramètres, pagination)
  4. Lancer l'exécution de la requête
- **JPA introduit le JPQL (*Java Persistence Query Language),* qui est, tout comme le EJBQL ou encore le HQL, un langage de requête du modèle objet, basé sur SQL.**

Interface Query

- **Représente une requête**
  - **Une instance de Query (d'une classe implémentant Query) est obtenue** des méthodes dédiées de la classe EntityManager **:**
    - **createQuery (),**
    - **createNativeQuery () ou**
    - **createNamedQuery ()**

L'objet Query

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA7 {

    public static void main(String[] argv) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
          EntityManager em = emf.createEntityManager();
        Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
        Personne personne = (Personne) query.getSingleResult();

        if (personne == null) {
            System.out.println("Personne non trouvée");
        } else {
            System.out.println("Personne.nom=" + personne.getNom());
        }
        em.close();
        emf.close();
    }
}
```

Query+la méthode getResultList()

```java
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA8 {

    public static void main(String[] argv) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createQuery("select p.nom from Personne p where p.id > 2");
        List noms = query.getResultList();
        for (Object nom : noms) {
            System.out.println("nom = "+nom);
        }
        em.close();
        emf.close();
    }
}
```

Les paramètres nommés+ setParameter()

```java
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA9 {

    public static void main(String[] argv) {

      EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
      EntityManager em = emf.createEntityManager();
      Query query = em.createQuery("select p.nom from Personne p where p.id > :id");

      query.setParameter("id", 1);

      List noms = query.getResultList();

      for (Object nom : noms) {
        System.out.println("nom = "+nom);
      }

     em.close();

     emf.close();
   }
}
```

Modifier une occurrence avec le EntityManager

- Pour modifier une entité existante dans la base de données, il faut :
  - Obtenir une instance de l'entité à modifier (par recherche sur la clé primaire ou l'exécution d'une requête)
  - Modifier les propriétés de l'entité
  - Selon le mode de synchronisation des données de l'EntityManager, il peut être nécessaire d'appeler la méthode `flush()` explicitement

Exemple

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA10 {

  public static void main(String[] argv) {

    EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();
    EntityTransaction transac = em.getTransaction();

    transac.begin();
    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
        if (personne == null) {
          System.out.println("Personne non trouvée");
        }else {
          System.out.println("Personne.prenom=" + personne.getPrenom());
          personne.setPrenom("prenom2 modifié");

        em.flush();
    personne = (Personne) query.getSingleResult();
    System.out.println("Personne.prenom=" + personne.getPrenom());
    }

    transac.commit();
}
```

Supprimer une occurrence avec EntityManager

- Pour supprimer une entité existante dans la base de données:
  - Obtenir une instance de l'entité à supprimer (par recherche sur la clé primaire ou l'exécution d'une requête)
  - Appeler la méthode remove() de l'EntityManager en lui passant en paramètre l'instance de l'entité

Exemple

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA12 {

    public static void main(String[] argv) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();  EntityTransaction transac = em.getTransaction();
        transac.begin();

        Personne personne = em.find(Personne.class, 4);

        if (personne == null) {
            System.out.println("Personne non trouvée");
        } else {
            em.remove(personne);
        }
        transac.commit();
        em.close();
        emf.close();
    }
}
```

Rafraîchir les données d'une occurrence

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA13 {

    public static void main(String[] argv) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transac = em.getTransaction();
        transac.begin();
        Personne personne = em.find(Personne.class, 4);

        if (personne == null) {
            System.out.println("Personne non trouvée");
        } else {
            em.refresh(personne);
        }
        transac.commit();
        em.close();
        emf.close();
    }
}
```

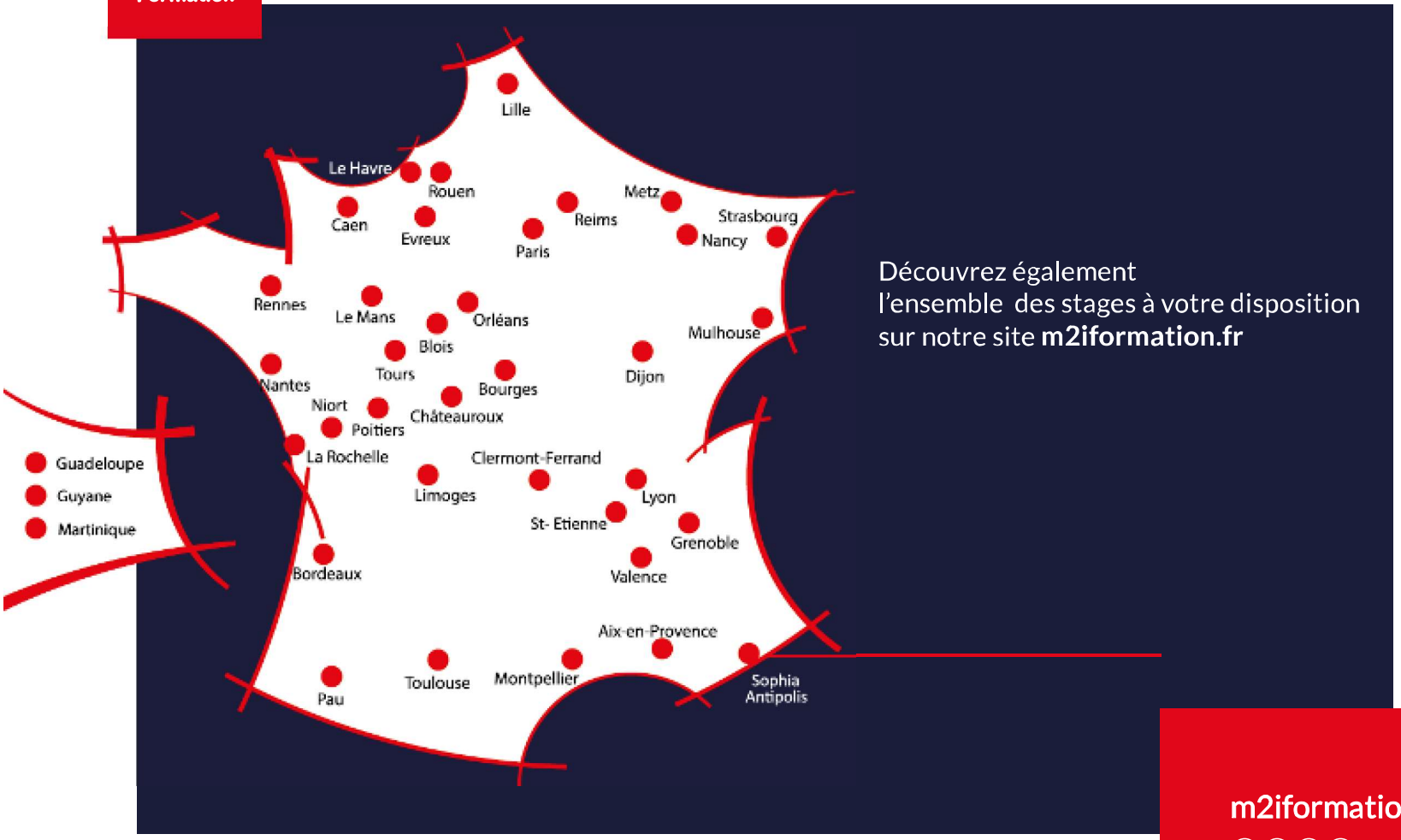
Queries: NamedQueries

- **Peut être mise dans n'importe quelle entité, mais on choisira le plus souvent l'entité qui correspond à ce qui est renvoyé par la requête**
- **On peut sauvegarder des gabarits de requête dans nos entités. Ceci permet :**
  - La réutilisation de la requête
  - D'externaliser les requête du code.

Queries: NativeQueries

- **Une façon de faire des requête en SQL natif.**
  - Sert principalement à avoir plus de contrôle sur les requêtes à la base de donnée

```
public class MyService {

    public void myMethod() {
    …
    List results = em.createNativeQuery("SELECT * FROM MyPojo", MyPojo.class).getResultList();
    …
    }
}
```



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr