MODULE JPA

JPA avec Hibernate

1.

Introduction

Introduction JPA

- **JPA** : Java Persistence API
  - API de **persistance** en Java
    - ORM : Object-Relational Mapping (exemple : Hibernate)
  - **Principes** de base
    - Définition de la correspondance entre le structure des classes Java et le schéma relationnel de la Base de données
    - Manipulation directe des objets dans le code Java

Introduction JPA

- **JPA** : Java Persistence API
  - Le API s'occupe de la **transformation**
    - Plus besoin de requêtes SQL !
    - Langage de requêtage propre mis à disposition

2.

Problématique de la persistance

Problématique de la persistance

- **Limites** de JDBC
  - Nécessite l'utilisation de **requêtes SQL**
  - Représentation **différente** des données
    - Langage SQL pour le requêtage
    - Classes Java pour les entités

Problématique de la persistance

- **Limites** de JDBC
  - Ça marche, **mais…**
    - Beaucoup de code à produire
    - Si un grand nombre de références entre les classes, nécessité de charger beaucoup de choses (problématique des **ressources**)
    - Problème de **cohérence** entre les objets et la Base de données
      - C'est au développeur de gérer la cohérence entre le contenu des objets et la Base de données

Problématique de la persistance

- **De JDBC à JPA**
  - Avec JPA, on définit des **correspondances** entre des classes (POJO : Plain Old Java Object) et des tables
  - JPA gère la **cohérence** entre les objets et les données en Base de données
  - Beaucoup **moins de code** technique à produire !

Problématique de la persistance

- **Fonctionnement de JPA**
  - Manipulation **d'objets** métier Java **uniquement**
  - API proposant des fonctionnalités :
    - Récupération d'objets à partir des données de la Base
      - Langage propre : JPQL
    - Persistance des objets en base
      - Insertion, modification, suppression
      - Avec gestion des transactions

Problématique de la persistance

- **Fonctionnement de JPA**
  - Nécessite une **implémentation**
    - Hibernate, TopLink, EclipseLink, ...
    - **Attention** : éviter d'utiliser les fonctionnalités spécifiques d'une implémentation particulière pour ne pas être dépendant

3.

Configuration de JPA + Hibernate

Configuration JPA + hibernate

- Configuration **maven**

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.5.4.Final</version>
    </dependency>
</dependencies>
```

Configuration JPA + hibernate

- Configuration d'une **unité de persistance**
  - **Fichier XML** définissant la liste des classes correspondant à des entités, la connexion à la base de données, des paramètres, …
  - Dans l'application **Java**
    - Récupération d'un **entity manager** pour manipuler les entités
    - Les modifications se font via une **transaction**

Configuration JPA + hibernate

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
         version="2.2">
   <persistence-unit name="crm">
      <class>fr.m2i.crm.model.Customer</class>
      <class>fr.m2i.crm.model.Order</class>
      <properties>
         <!-- database connection -->
         <property name="hibernate.connection.driver_class" value="com.mysql.cj.jdbc.Driver" />
         <property name="hibernate.connection.url" value="jdbc:mysql://localhost/crm" />
         <property name="hibernate.connection.user" value="crm" />
         <property name="hibernate.connection.password" value="crm" />
         <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL57Dialect" />
         <property name="hibernate.show_sql" value="true" />
      </properties>
   </persistence-unit>
</persistence>
```

Configuration JPA + hibernate

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <persistence-unit ...>
      - Définit le nom de l'unité de persistance et le type de transaction utilisée

Configuration JPA + hibernate

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <class>
      - Définit qu'une classe Java sera une entité dont les instances seront persistantes en base de données

Configuration JPA + hibernate

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <properties>
      - Ensemble de propriétés de configuration
      - Par exemple, les paramètres de connexion à la base (URL, driver, utilisateur et mot de passe)

Configuration JPA + hibernate

- **Entity Manager**
  - Récupéré à partir de la fabrique de gestionnaire d'entité et via le nom donné à l'unité de persistance

```
private EntityManager em = null;
...
EntityManagerFactory emf = Persistence.createEntityManagerFactory("monPu");
em = emf.createEntityManager();
...
```

Modèle de persistance

Exercice : création SessionHelper

- Créer un package **helper**
- Créer un fichier **SessionHelper.java**
- Ajouter l'attribut **entityManager** de type **EntityManager**
- Créer une méthode **EntityManager getEntityManager()**, celle-ci devra :
  - Créer un entity manager et le stocker dans l'attribut **entityManager**
  - Uniquement si aucun entity manager n'a été créé
  - Elle retourne l'attribut **entityManager**

Configuration JPA + hibernate

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
         version="2.2">
   <persistence-unit name="crm">
            …
      <properties>
            …
            <property name="hibernate.hbm2ddl.auto" value="create" />
      </properties>
   </persistence-unit>
</persistence>
```

Configuration JPA + hibernate

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
        version="2.2">
   <persistence-unit name="crm">
          …
      <properties>
          …
          <property name="hibernate.hbm2ddl.auto" value="update" />
      </properties>
   </persistence-unit>
</persistence>
```

4.

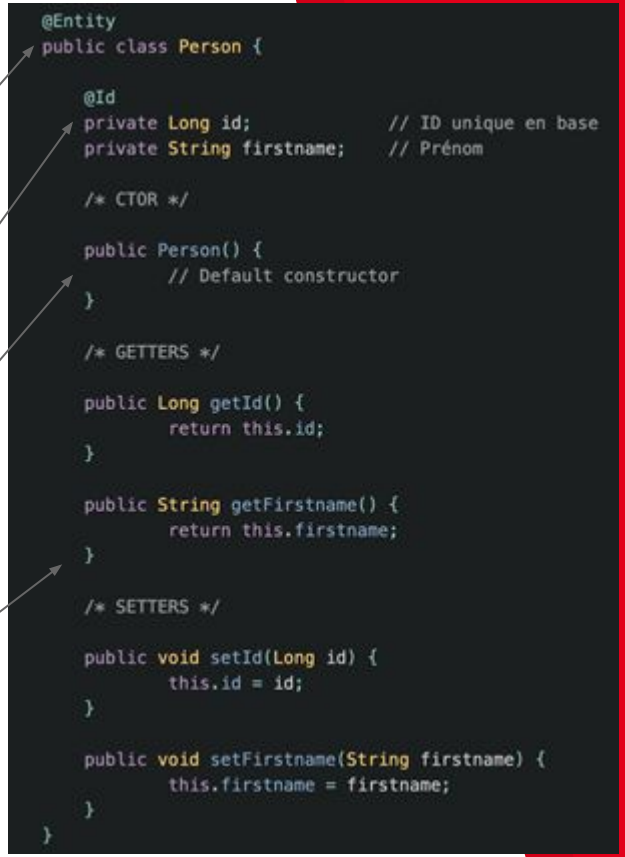
Modèle de persistance

Modèle de persistance

- **Mapping sur les entités Java**
  - Une **entité** = un objet **métier**
  - Une classe Java est **mappée** sur une table SQL
    - **Correspondance** entre les attributs de la classe et les colonnes de la table
    - Mapping réalisé avec des **annotations** dans la classe

Modèle de persistance

- **Mapping sur les entités Java**
  - Au minimum :
    - **@Entity** : défini comme entité
    - **@Id** : pour l'identifiant de l'entité
    - Un constructeur (pour instancier)
    - Un getter et un setter pour chaque attribut



```java
@Entity
public class Person {

    @Id
    private Long id;              // ID unique en base
    private String firstname;     // Prénom

    /* CTOR */

    public Person() {
            // Default constructor
    }

    /* GETTERS */

    public Long getId() {
            return this.id;
    }

    public String getFirstname() {
            return this.firstname;
    }

    /* SETTERS */

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

    public void setFirstname(String firstname) {
            this.firstname = firstname;
    }
}
```

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - Les annotations **@Entity** et **@Id** **ne suffisent pas** à faire comprendre à l'ORM comment manipuler cette entité
  - Il faut ajouter **@Table** pour décrire plus spécifiquement les détails à propos de la base de données (ex: nom, schéma)

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - **@GeneratedValue** est utilisé conjointement avec @Id pour les valeurs de clé générée :
    - **IDENTITY** : pour spécifier une colonne d'identité de la base de données
    - **AUTO** : pour choisir automatiquement une implémentation basée sur la base de données utilisée
    - **SEQUENCE** : pour utiliser une séquence (si la base de données la supporte)(Voir @SequenceGenerator)
    - **TABLE** : pour spécifier qu'une base de données utilisera une table et une colonne d'identité pour s'assurer son caractère unique (voir @TableGenerator)

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - **Exemple :**

```
@Entity
@Table(name = "person")
public class Person {

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

    private String firstname;
```

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - En l'absence d'information précise, Hibernate va assumer certaines configurations par défaut
    - L'utilisation de **@Column** permet de définir les noms de colonnes souhaités, ainsi que des informations à leur propos

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - L'utilisation de **@Column** permet de définir les noms de colonnes souhaités, ainsi que des informations à leur propos :
      - columnDefinition
      - insertable
      - length
      - name
      - nullable
      - precision
      - scale
      - table
      - unique
      - updatable

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
  - Sans les informations de l'annotation, Hibernate aurait considéré l'attribut « firstname » comme :
    - Étant mappé à une colonne nommée "firstname" (et pas "first_name")
    - Pouvant avoir une valeur à NULL
    - Ayant une longueur à 100 caractères

```java
@Entity
@Table(name = "person")
public class Person {

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

    @Column(
        name = "first_name",
        nullable = false,
        length = 100
    )
    private String firstname;
```

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - En dehors de « simplement décrire » la base de données dans le code, Hibernate possède aussi des annotations permettant d'écrire du code plus simple :
    - Exemple : l'utilisation de **@Enumerated** permet de dire à Hibernate de « mapper » des chaînes de caractères, des nombres en base vers des « **Enums** » Java (plus pratique)
    - Exemple : l'utilisation de **@Temporal** permet de dire à Hibernate de « mapper » des dates SQL (TIMESTAMP, DATETIME, etc.) vers des « **Date** » Java (plus pratique)

Modèle de persistance

Exercice : création de l'entité **Customer**

- Créer un package **model**
- Créer un fichier **Customer.java**
- L'entité **Customer** devra contenir les attributs suivant :

Field	Type	Length
id	INT	11
address	VARCHAR	255
city	VARCHAR	100
company_name	VARCHAR	100
country	VARCHAR	100
email	VARCHAR	100
first_name	VARCHAR	100
last_name	VARCHAR	100
phone	VARCHAR	20
zip_code	VARCHAR	12

Modèle de persistance

Exercice : création de l'entité **Customer**

- Créer un package **state**
- Créer un fichier **CustomerState.java**
- Il s'agit d'un **enum** qui contiendra les valeurs suivante : INACTIVE, ACTIVE
- Ajouter l'attribut **state** de type **CustomerState** dans l'entité **Customer**
- Utiliser l'annotation **@Enumerated** avec le paramètre EnumType.ORDINAL ou EnumType.STRING
- En base le champs **state** sera de type **INT**

5.

Manipulation des entités

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Récupération d'une instance d'une entité en précisant sa classe et son identifiant
    - <T> T find(Class<T> entityClass, Object id)
      - Customer customer = em.find(Customer.class, 3) ;

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Modification du contenu de la BDD en mode transactionnel

```
EntityTransaction trans = null;
try {
        trans = em.getTransaction();
        trans.begin();
                … ici les actions ...
        trans.commit();
} catch(Exception e) {
 if (trans != null) trans.rollback();
}
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Rendre persistant en BDD un objet qui devient géré par le gestionnaire d'entités
    - void persist(Object entity)

```
trans.begin();
Customer customer = new Customer(...);
em.persist(customer);
trans.commit();
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Récupérer une copie gérée par le gestionnaire d'entité de l'objet passé en paramètre
    - <T> T merge(T entity)

```
Customer newCustomer = em.merge(customer);
// Modifications effectuées sur l'objet retourné par le merge, pas l'initial
newCustomer.setLastname("Dupont");
trans.commit();
```

Manipulation des entités

TP : création de **CustomerDAO**

- Créer un package **dao**
- Créer un fichier **CustomerDAO.java**
- Cette class utilisera le **SessionHelper**
- Implémenter les méthodes suivantes :
  - Customer findById(long id)
  - void create(Customer customer)
  - void update(Customer customer)

Manipulation des entités

- **JPQL et Query**
  - **JPQL** n'est pas du SQL
  - Langage **d'interrogation** centré sur les **objets Java**

```
private EntityManager em;
// ...

public List<Person> getAll() {
    Query query = em.createQuery("SELECT p FROM Person p");
    List<Person> results = query.getResultList();
}
```

Manipulation des entités

- **JPQL et Query**
  - **Query** avec paramètres
    - Liaison par **nom de paramètre** (“name parameter binding”)

Requête SELECT p FROM Person p WHERE p.name = :searched

query.setParameter(“searched”, “Harry”);
query.getSingleResult(); ou **query.getResultList();**

Manipulation des entités

- **JPQL et Query**
  - **Query** avec paramètres
    - Liaison par **position de paramètre** (“positionnal parameter binding”)

Requête SELECT p FROM Person WHERE p.name = ?1

query.setParameter(1, “Harry”);
query.getSingleResult(); ou **query.getResultList();**

Manipulation des entités

- **JPQL et Query**
  - **Native Query**
    - Pour **reprendre** la main sur le **SQL**

```
List<Object[]> persons = session
    .createNativeQuery("SELECT id, name FROM PERSON" )
    .list();

for(Object[] person : persons) {
        Number id = (Number) person[0];
        String name = (String) person[1];
}
```

```
List<Phone> phones = session.createNativeQuery(
        "SELECT id, phone_number, phone_type, person_id " +
        "FROM Phone" )
    .addEntity( Phone.class )
    .list();
```

Manipulation des entités

Exercice : évolution de CustomerDAO

- Ajouter la méthode suivante :
  - List<Customer> findAll()

Manipulation des entités

- Concepts **ORM** : relation des **entités**
  - Il existe 4 types de relation entre entités :
    - Relation « one to one » : **relation 1 -> 1**
    - Relation « one to many » : **relation 1 -> n**
    - Relation « many to one » : **relation n -> 1**
    - Relation « many to many » : **relation n -> n**

Manipulation des entités

- Concepts **ORM** : relation des **entités**
  - S'ajoutent à ces types de relation « des **configurations** »
    - Notion de **sens** :
      - Relation **unidirectionnelle**
      - Relation **bidirectionnelle**
    - Notion de **cascade** : que faire de B, lié à A,
      - Lors d'une **mise à jour** de A ?
      - Lors d'une **suppression** de A ?

Manipulation des entités

- Concepts **ORM** : relation des **entités**
  - Relation **1 -> 1** : annotation **@OneToOne**
    - Pour définir une relation **forte**, **bidirectionnelle** entre 2 entités
    - Exemple : « Un navire est gouverné par un capitaine »

Manipulation des entités

- Concepts **ORM** : relation des **entités**
  - Relation **1 -> N** : annotation **@OneToMany**
    - Pour définir une relation entre 1 entité et une liste d'entités
    - Exemple : « Une personne possède un ou plusieurs téléphones »

```
// Person.java

@OneToMany(mappedBy = "person")
private List<Phone> phones;
```

Manipulation des entités

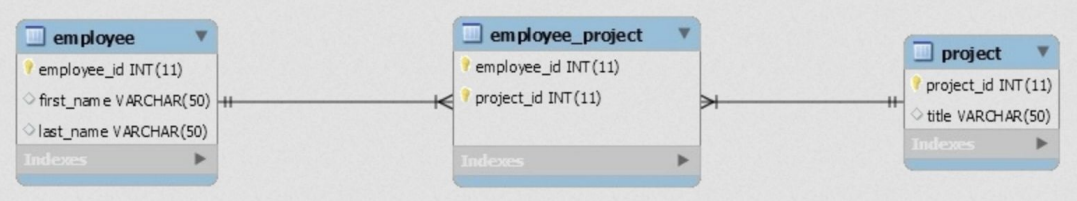
- Concepts **ORM** : relation des **entités**
  - Relation **N -> 1** : annotation **@ManyToOne**
    - Pour définir une relation **contraire** à @OneToMany (plusieurs entités liées à une autre)
    - Exemple : « Un ou plusieurs téléphones peuvent être détenus par une personne »

```
// Phone.java

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "person_id")
private Person person;
```

Manipulation des entités

- Concepts **ORM** : relation des **entités**
  - Relation **N -> N** : annotation **@ManyToMany**
    - Pour définir des relations entres entités dont on ne peut faire les liens qu'à travers des **tables de jointure**
    - Exemple : « Un ou plusieurs employés travaillent sur un ou plusieurs projets »



Manipulation des entités

- Concepts **ORM** : relation des **entités**
  - Relation **N -> N** : annotation **@ManyToMany**

```
// Employee.java

@ManyToMany(cascade = { CascadeType.ALL })
@JoinTable(
    name = "Employee_Project",
    joinColumns        = { @JoinColumn(name = "employee_id") },
    inverseJoinColumns = { @JoinColumn(name = "project_id") }
)
List<Project> projects;
```

```
// Project.java

@ManyToMany(mappedBy = "projects")
private List<Employee> employees;
```

Manipulation des entités

```java
// Phone.java

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "person_id")
private Person person;
```

- Concepts **ORM** : mode de récupération d'**entités** liées entre elles
  - Il existe **2 modes** de récupération d'entités
    - **LAZY** : interroge la base de données seulement quand la propriété est appelée (exemple: appel à un getter)
    - **EAGER** : interroge la base de données dès que l'objet original est créé

Manipulation des entités

- Concepts **ORM** : types d'effet « **cascade** » entre entités
  - Les relations entre entités **dépendent** souvent de l'existence d'une autre
  - Exemple : relation **Personne** <-> **Adresse**
    - Sans la personne, **l'adresse n'aurait pas de signification métier**
    - En **supprimant** la personne, on souhaiterait que son adresse soit **supprimée** aussi

Manipulation des entités

- Concepts **ORM** : types d'effet « **cascade** » entre entités

```java
@Entity
public class Person {

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

    private String name;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Address> addresses;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private int zipCode;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

Manipulation des entités

- Ne pas partir tête baissée pour **écrire les entités**
- Attention à positionner le “fetch type” en LAZY sur les collections d’objets qui ne sont **pas nécessaires** au premier abord
  - Syndrome de « ramener la terre entière »
- Penser aux effets « **cascade** » entre entités
  - Souhaite-t-on garder une carte d’identité en base si l’on a supprimé la personne qui la possédait ?

Modèle de persistance

Exercice : création de l'entité **Order**

- Dans le package **model**
- Créer un fichier **Order.java**
- **Attention** le nom de la table en base sera **orders**
- L'entité **Order** devra contenir les attributs suivant :

Field	Type	Length
id	INT	11
designation	VARCHAR	100
nb_days	INT	11
total_exclude_taxe	DOUBLE	
total_with_taxe	DOUBLE	
type_presta	VARCHAR	100
unit_price	DOUBLE	

Modèle de persistance

Exercice : création de l'entité **Order**

- Dans le package **state**
- Créer un fichier **OrderState.java**
- Il s'agit d'un **enum** qui contiendra les valeurs suivante : CANCELED, OPTION, CONFIRMED
- Ajouter l'attribut **state** de type **OrderState** dans l'entité **Order**
- Utiliser l'annotation **@Enumerated** avec le paramètre EnumType.ORDINAL ou EnumType.STRING
- En base le champs **state** sera de type **INT**

Modèle de persistance

Exercice : création de l'entité **Order**

- Ajouter un attribut de type **Customer**
- Le nom en base est **customer_id** -> utiliser **@JoinColumn**
- Cet attribut permet de définir la relation avec l'entity **Customer**
- Un **Order** est lié à un **Customer** existant dans la table **customer** via le **customer_id**
- Plusieurs **Order** différents peuvent être lié à un même **Customer**
- Choisir la bonne relation parmit **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**

id	designation	nb_days	total_exclude_taxe	total_with_taxe	type_presta	unit_price	custo...	state
3	React Techlead	20	18000	21600	Coaching	900	1	2
4	Nest.js Techlead	50	40000	48000	Coaching	800	1	1

Modèle de persistance

Exercice : création de l'entité **Order**

- Définir la cascade en tant que **Cascade.MERGE**
- Définir fetch en tant que **FetchType.EAGER**

Manipulation des entités

- **Etat d'un objet persistant**
  - Plusieurs états pour l'instance d'une classe entité
    - **Persistant** : l'entité a une correspondance de contenu en BDD
      - **Gérée** : état synchronisé par le gestionnaire d'entité avec le contenu en BDD
      - **Détachée** : état non géré, les modifications ne sont plus synchronisées avec la BDD
    - **Transient** : objet java classique avec existence uniquement en mémoire de la JVM
      - Cas de l'instanciation d'un objet

Manipulation des entités

- **Etat d'un objet persistant**
  - Plusieurs états pour l'instance d'une classe entité
    - **Supprimé** : instance persistante dont on a supprimé le contenu associé en BDD
      - L'objet existe toujours en mémoire de la JVM mais n'a plus de correspondance en base

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Détacher un objet du gestionnaire d'entité (les modifications sur l'objet ne sont alors plus reportées sur la BDD)
    - void detach(Object entity)

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Détacher tous les objets du gestionnaire d'entité
    - void clear()

Manipulation des entités

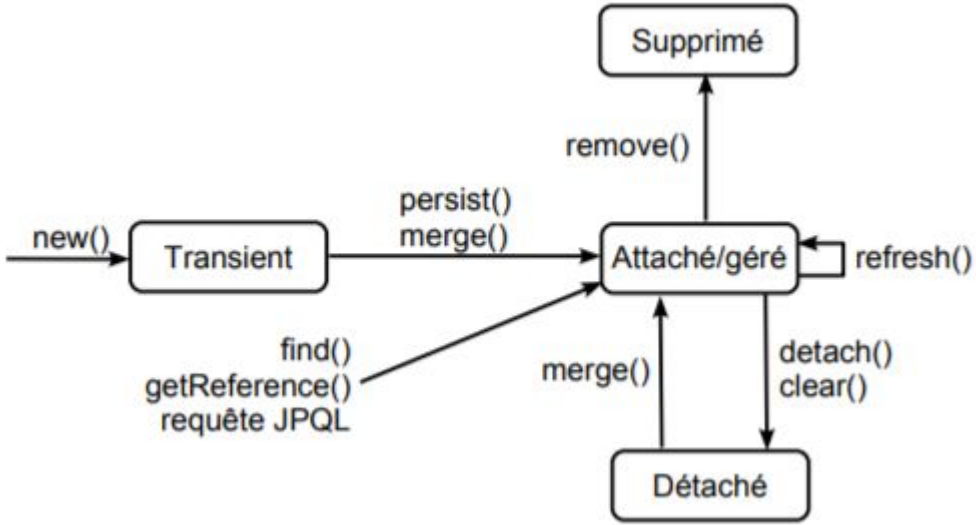
- **Opérations sur les instances d'entité**
  - Supprimer un objet : effacer ses données en base
    - void remove(Object entity)

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Remettre à jour le contenu de l'objet par rapport au contenu en base
    - void refresh(Object entity)

Manipulation des entités

- **Cycle de vie d'un objet persistant**



Manipulation des entités

TP : création de **OrderDAO**

- Dans le package **dao**
- Créer un fichier **OrderDAO.java**
- Cette class utilisera le **SessionHelper**
- Implémenter les méthodes suivantes :
  - List<Order> findAll();
  - Order findById(long id);
  - void create(Customer customer);
  - void update(Customer customer);
  - void delete(long id);