

Cours de JPA

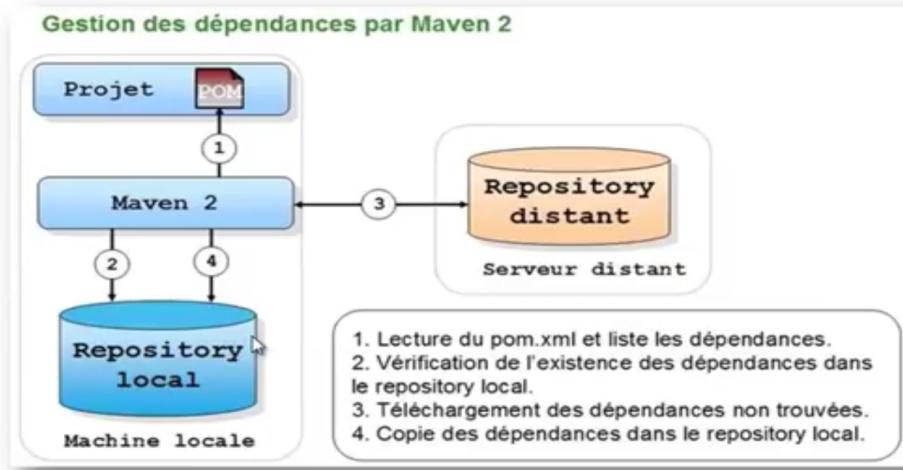
Maven



Maven

- **Maven**, géré par l'organisation *Apache Software Foundation.* (*Jakarta Project*), est un **outil pour la gestion et l'automatisation de production des projets logiciels Java en général et Java EE en particulier**.
- L'objectif recherché est de
 - produire un logiciel à partir de ses sources,
 - en optimisant les tâches réalisées à cette fin
 - et en garantissant le bon ordre de fabrication.
 - **Compiler, Tester, Contrôler, produire les packages livrables**
 - **Publier la documentation et les rapports sur la qualité**
- **Apports :**
 - Simplification du processus de construction d'une application
 - Fournit les bonnes pratiques de développement
 - Tend à uniformiser le processus de construction logiciel
 - Vérifier la qualité du code
 - Faciliter la maintenance d'un projet

Maven et JUnit : Quelques Principes fondamentaux

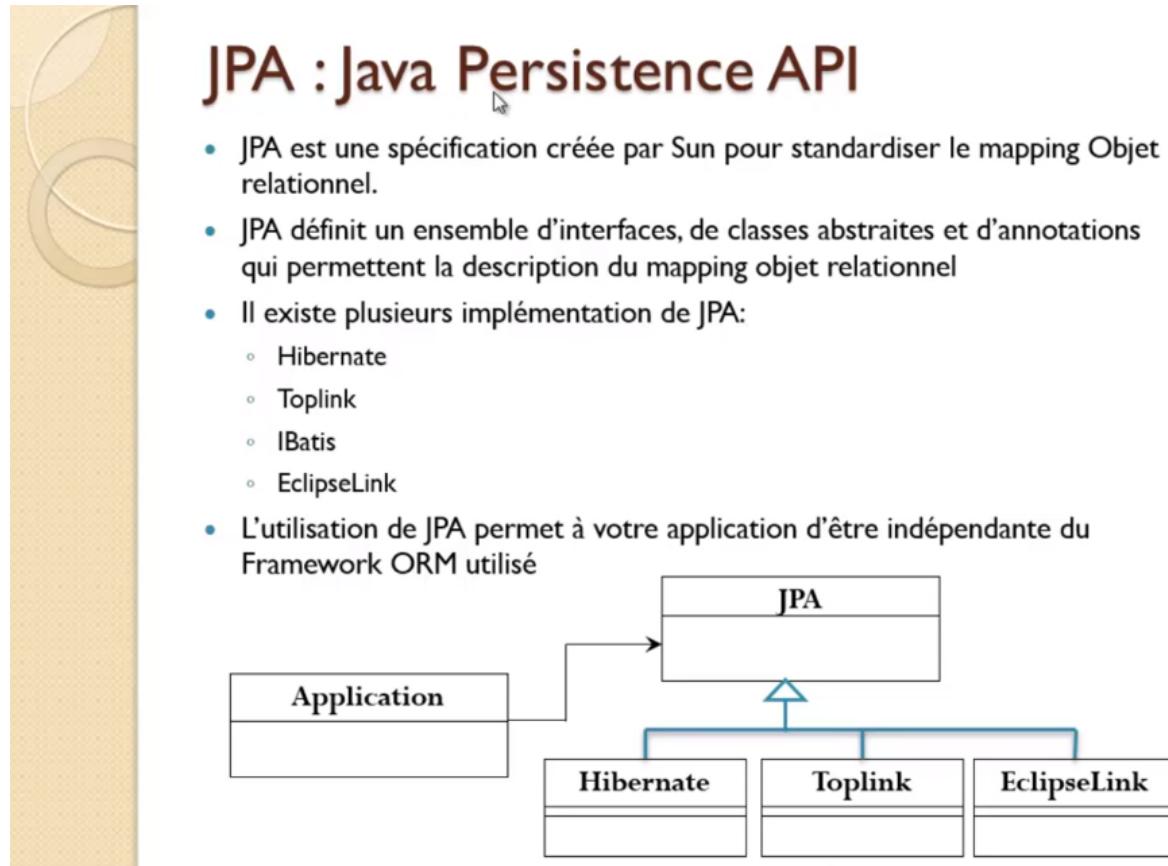
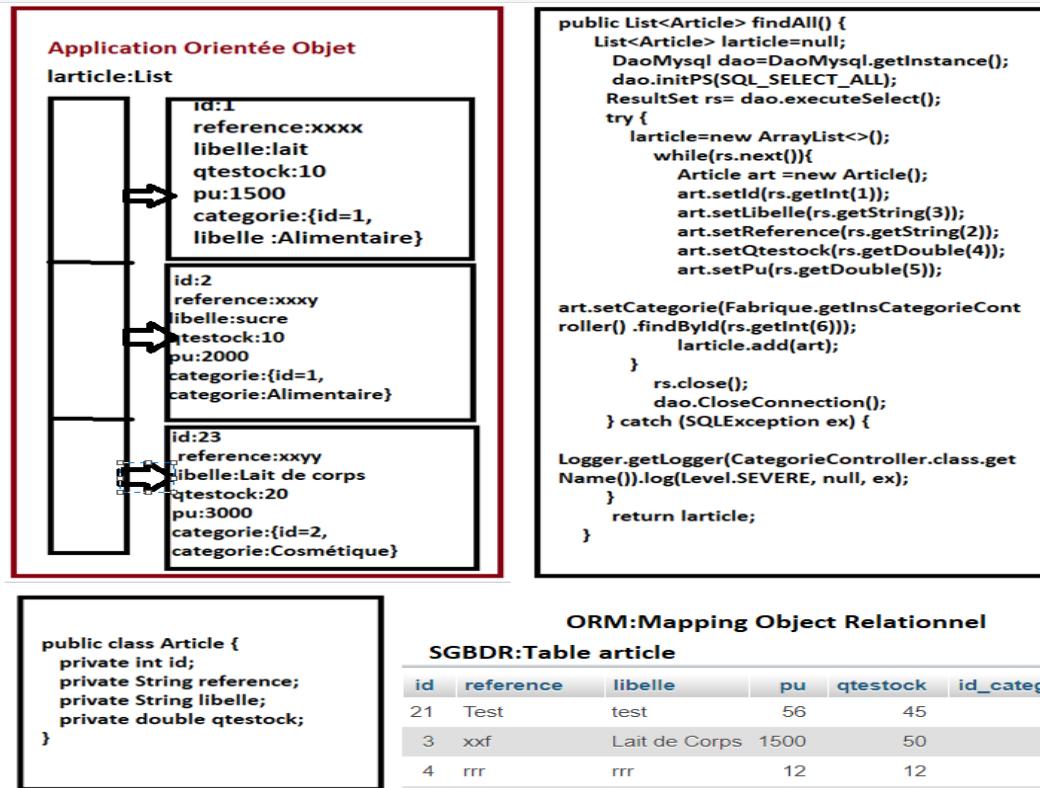


Notion ORM(Object Relational Mapping)

Introduction

- Travailler dans les deux univers que sont l'orienté objet et la base de données relationnelle peut être lourd et consommateur en temps dans le monde de l'entreprise d'aujourd'hui.
- EclipseLink est un outil de mapping objet/relationnel pour le monde Java.
- Le terme mapping objet/relationnel (ORM) décrit la technique consistant à faire le lien entre la représentation objet des données et sa représentation relationnelle basée sur un schéma SQL.

Exemple



NB:

Dans notre cours nous utiliserons implémentation avec
Hibernate

● **Hibernate**

Hibernate

- framework ORM (le premier) pour Java implémentant les spécifications de l'API JPA
- Open-source
- Créé par **JBoss** (Entreprise productrice de serveurs d'application JEE JBoss)
- Possédant une extension **NHibernate** pour la plateforme .NET (de **Microsoft**)
- Pouvant être utilisé dans tout type de projet Java (Web, Client lourd...)

Hibernate

ORM : avantages

- Suppression d'une très grande partie du code nécessaire avec JDBC
 - Gain de temps
 - Simplification du code source
- Code portable indépendant vis-à-vis de SGBD

Remarque

Pour définir une entité, **Hibernate** utilise

- Soit un fichier de Mapping appelé `entity.hbm.xml`
- Soit des annotations JPA comme `@Entity`, `@Id...`

NB : Dans ce cours nous utiliserons Hibernate avec les annotations

- **Mise oeuvre JPA avec Implémentation hibernate**

Différentes étapes pour persister des données avec **Hibernate**

- Préparer une connexion
- Créer un projet (avec maven)
- Ajouter les dépendances pour *Hibernate* et *MySQL Connector* dans `pom.xml`
- Créer le fichier de configuration `hibernate.cfg.xml`
- Créer les entités
- Persister les données

NB : Dans ce projet nous utiliserons le fichier de configuration `persistence.xml` au lieu `hibernate.cfg.xml`

1) Préparer une connexion

Avant de créer une connexion

créer une base de données hibernate

2) Ajouter les Dépendances pour Hibernate et Mysql Connector dans le fichier `pom.xml` :

```
<dependencies>
  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.1.0</version>
  </dependency>
```

```

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.1.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.6.1.Final</version>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.30</version>

<scope>provided</scope>

</dependency>

</dependencies>

```

3) Créer le fichier de configuration `persistence.xml`

Dans le dossier `resources` , créer le dossier `META-INF`

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  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_1.xsd">
  <persistence-unit name="demonJpa" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>java.demon.jpa.entity.Personne</class>
    <class>java.demon.jpa.entity.Patient</class>
    <class>java.demon.jpa.entity.Medecin</class>
    <class>java.demon.jpa.entity.RV</class>
  <properties>

```

```
<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:8889/jpa_demon" />
<property name="javax.persistence.jdbc.user" value="root" />
<property name="javax.persistence.jdbc.password" value="root" />
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>

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

4) Création des entités

Mapping Objet Relationnel des entités

- Il existe deux moyens pour mapper les entités :
 - Créer des fichiers XML de mapping
 - Utiliser les Annotations JPA
- L'utilisation des annotations JPA laisse votre code indépendant de **EclipseLink**
- La création des fichiers XML de mapping a l'avantage de séparer le code java du mapping objet relationnel.
- Dans ce cours, nous allons utiliser les annotations JPA

Quelques annotations JPA de Mapping des Entités

- **@Table**
 - Préciser le nom de la table concernée par le mapping. Par défaut c'est le nom de la classe qui sera considérée
- **@Column**
 - Associer un champ de la colonne à la propriété. Par défaut c'est le nom de la propriété qui sera considérée.
- **@Id**
 - Associer un champ de la table à la propriété en tant que clé primaire
- **@GeneratedValue**
 - Demander la génération automatique de la clé primaire au besoin
- **@Basic**
 - Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
- **@Transient**
 - Demander de ne pas tenir compte du champ lors du mapping
- **@OneToMany, @ManyToOne**
 - Pour décrire une association de type un à plusieurs et plusieurs à un
- **@JoinColumn**
 - Pour décrire une clé étrangère dans une table
- **@ManyToMany**
 - Pour décrire une association plusieurs à plusieurs

Attributs de l'annotation @Table

Attribut	désignation
name	permet de définir le nom de la table s'il est différent de celui de l'entité
uniqueConstraints	permet de définir des contraintes d'unicité sur un ensemble de colonnes

```
@Table(  
    name="personne",  
    uniqueConstraints={  
        @UniqueConstraint(name="nom_prenom", columnNames  
            ={ "nom", "prenom" })  
    }  
)
```

Attributs de l'annotation @Column

Attribut	désignation
name	permet de définir le nom de la colonne s'il est différent de celui de l'attribut
length	permet de fixer la longueur d'une chaîne de caractères
unique	indique que la valeur d'un champ est unique
nullable	précise si un champ est null (ou non)
...	...

Attention

L'annotation @Transient

L'attribut annoté par @Transient n'aura pas de colonne associée dans la table correspondante à l'entité en base de données.

5) Exécution de l'application

```
#package views Main.java
```

```
EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("INVERSION_CONTROLE");
EntityManager em=emf.createEntityManager();
```

```
#Description du processus
```

- L'exécution de cette permet de:
 - Créer un de type EntityManagerFactory
 - EntityManagerFactory lit le fichier persistence.xml, ce qui entraînera la création du data source qui établira une connexion à la base de données
 - Lorsque les tables ne sont pas créer, EntityManagerFactory générer les tables relatives aux entités grâce à la propriété

```
<property name="hibernate.hbm2ddl.auto"
value="update"/>
```

a) Les Relations entre entity

Les relations JPA peuvent être *unidirectionnelles* ou *bidirectionnelles*. Cela signifie simplement que nous pouvons les

modéliser en tant qu'attribut sur exactement une des entités associées ou les deux.

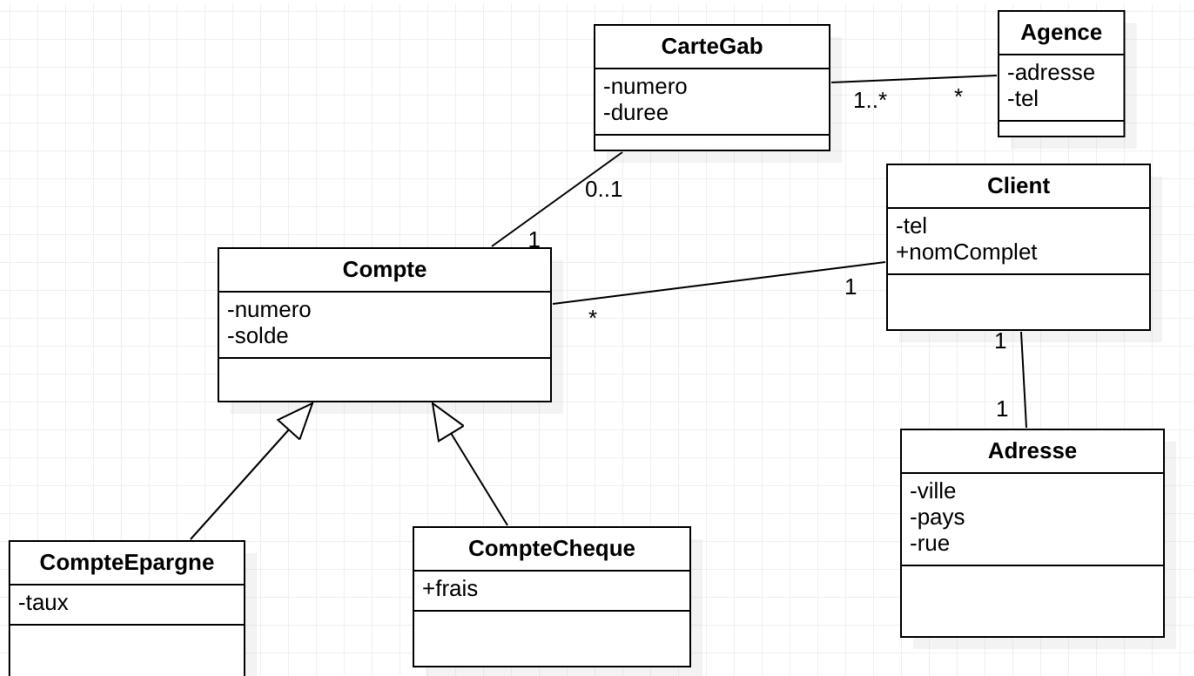
La définition de la direction de la relation entre les entités n'a aucun impact sur le mappage de la base de données. Il définit uniquement les directions dans lesquelles nous utilisons cette relation dans notre modèle de domaine.

Pour une relation bidirectionnelle, on définit généralement :

- le côté propriétaire
- l'inverse ou le côté référencement

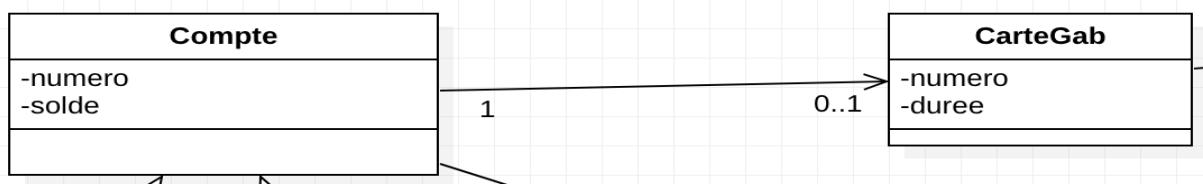
L' annotation `@JoinColumn` nous aide à spécifier la colonne que nous utiliserons pour joindre une association d'entités ou une collection d'éléments. D'autre part, l' attribut `mappedBy` est utilisé pour définir le côté référencement (**côté non propriétaire**) de la relation.

Soit le diagramme de classe ci dessous



b) Relations Unidirectionnelle

1) OneToOne entre Compte et CarteGab



Dans cette relation l'entité **Compte** est propriétaire et l'entité **CarteGab** est inverse

Description de la Relation

- Relation unidirectionnelle de Compte vers CarteGab
- Cascade :
 - La création d'un compte peut entraîner la création d'un carteGab (`CascadeType.PERSIST`)
 - La Suppression d'un compte entraîne pas la suppression d'un carteGab (`CascadeType.REMOVE`)
- JoinColumn est défini dans Compte car il est le propriétaire

```
#package models / CarteGab.java

@Entity
@Table(name = "carte_gab")
public class CarteGab {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(unique = true)
    private String numero;
    private int duree;

    //Constructeurs
    //Getters and Setters
}
```

```

#package models / Compte.java

#Définition de l'association OneToOne

@OneToOne(cascade =
{CascadeType.PERSIST,CascadeType.REMOVE})
@JoinColumn(name =
"numero_carte_gab",referencedColumnName =
"numero",nullable = false,unique = true)
private CarteGab carte;

```

NB : Les Cascades

Les relations d'entité dépendent souvent de l'existence d'une autre entité, par exemple la relation **Client** - **CarteGab**. Sans **Client**, l' entité **CarteGab** n'a aucune signification propre. Lorsque nous supprimons l' entité **Client**, notre entité **CarteGab** doit également être supprimée.

La cascade est le moyen d'y parvenir. Lorsque nous effectuons une action sur l'entité cible, la même action sera appliquée à l'entité associée.

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
```

Explication

- cascade : ici on cascade les deux opérations PERSIST et REMOVE qu'on peut faire de l'entité propriétaire à l'entité inverse
- On peut cascader d'autres opérations telles que DETACH, MERGE, et REFRESH...
- on peut cascader toutes les opérations avec ALL

Remarque

CascadeType.ALL propage toutes les opérations, y compris celles spécifiques à Hibernate, d'un parent à une entité enfant.

2) ManyToOne entre Compte et Client



Dans cette relation l'entité **Compte** est propriétaire et l'entité **Client** est inverse

Description de la Relation

- Relation unidirectionnelle de Compte vers Client
- Cascade :
 - La création d'un compte peut entraîner la création d'un client (`CascadeType.PERSIST`)
 - La suppression d'un compte n'entraîne pas la suppression d'un client
- JoinColumn est défini dans Compte car il est le propriétaire

```
#package models / Client.java
```

```
@Entity
@Table(name = "clients")
public class Client {
    @javax.persistence.Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false)
    private Long Id;
    private String telephone;

    @Column(name = "nom_complet")
    private String nomComplet;
    //Constructeurs
    //Getters and Setters
}
```

```
#package models / Compte.java
```

```
#Définition de l'association ManyToOne

@ManyToOne(cascade = {CascadeType.PERSIST})
@JoinColumn(name = "client_id", referencedColumnName =
"id", nullable = false, unique = true)
private Client client;
```

3) OneToMany entre Client et Compte



Dans cette relation l'entité **Compte est propriétaire** et l'entité **Client est inverse**

Description de la Relation

- Relation unidirectionnelle de Compte vers Client
- Pas de Cascade dans Client
- Pas JoinColumn dans Client
- `mappedBy("client")` pour référencer l'objet dans l'entité propriétaire(Compte)

```
#package models / Client.java

#Définition de l'association OneToMany

@OneToMany(mappedBy = "client")
private List<Compte> comptes=new ArrayList<>();
```

#Ensuite

- Il faut Générer les Getters et les Setters de la Liste de comptes
- Il faut générer la méthode `add()` et `remove()` qui permettent d'ajouter ou de supprimer un compte pour un objet de Client (utiliser Generate Delegate Methods)

Ensuite

- il faut générer le getter et le setter d'adresses
- il faut aussi générer la méthode add et remove qui permettent d'ajouter ou de supprimer une adresse pour un objet personne (Dans Source, choisir Generate Delegate Methods).
- Renommer add en addAdresse et remove en removeAdresse

NB :

L'attribut **fetch** avec les valeurs de l'énumération **FetchType(EAGER, LAZY)** définit si Hibernate charge les données avec empressement(**EAGER**) ou paresseusement(**LAZY**) .

Exemple : Dans notre Relation Compte->Client,

Dans l'entité inverse Client

- **FetchType.EAGER** : Le chargement d'un objet Client entraînera le chargement immédiat des objets comptes associés
- **FetchType.LAZY (par défaut)** : Le chargement d'un objet Client n'entraînera pas forcément le chargement d'objets comptes associés. Le chargement d'objets comptes ne se fera que lorsqu'on appellera la méthode **client.getComptes()** .

Remarque : l'attribut fetch peut être aussi défini dans l'entité propriétaire

#Définition du fetch dans l'entité Client

```
@OneToOne(  
    mappedBy = "client", fetch = FetchType.LAZY)  
    private List<Compte> comptes=new ArrayList<>();
```

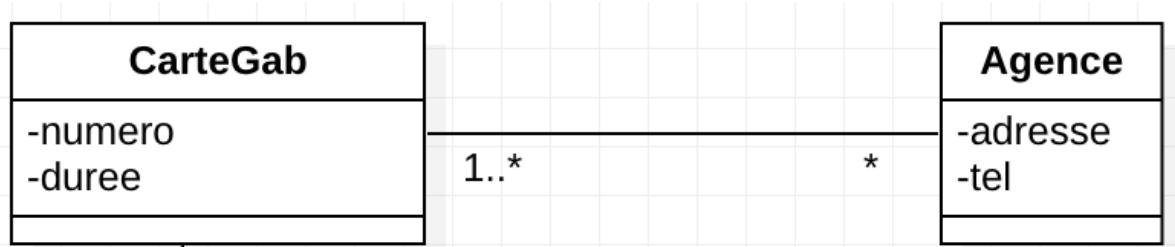
Attention :

- Possible d'avoir des tuples dupliqués avec le chargement **Eager**
- Pour Corriger ce problème il faudra ajouter l'annotation

```
@Fetch(FetchMode.Select)
```

4) ManyToMany

(a) Sans Classe Association



Dans cette relation l'entité **CarteGab** est propriétaire et l'entité **Agence** est inverse

Description de la Relation

- Relation unidirectionnelle de **CarteGab** vers **Agence**
- Pas de Cascade dans **CarteGab**
- JoinTable dans **CarteGab**
- Pas de mappedBy
- fetch dans **CarteGab**

```
#package models / CarteGab.java

#Définition de l'association ManyToMany

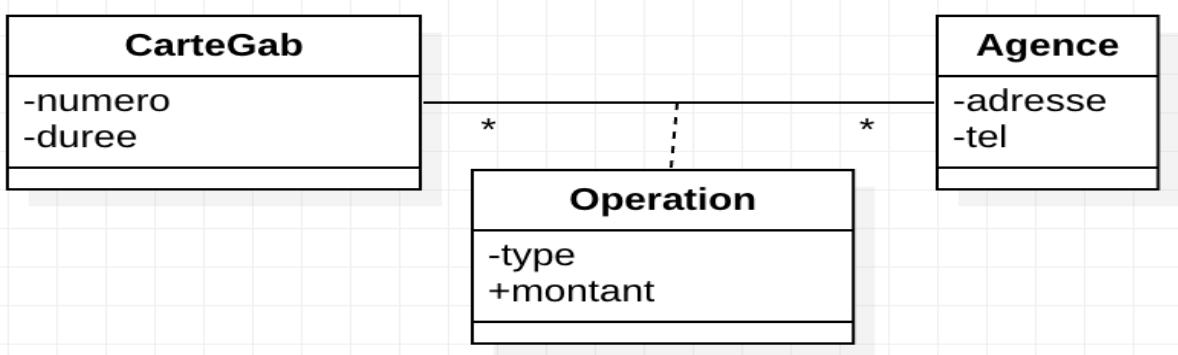
@OneToMany(mappedBy = "client")

@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(
    name = "carte_agence",
    joinColumns = @JoinColumn(name = "agence_id"),
    inverseJoinColumns = @JoinColumn(name =
"carte_id")
)
private List<Agence> agences;
```

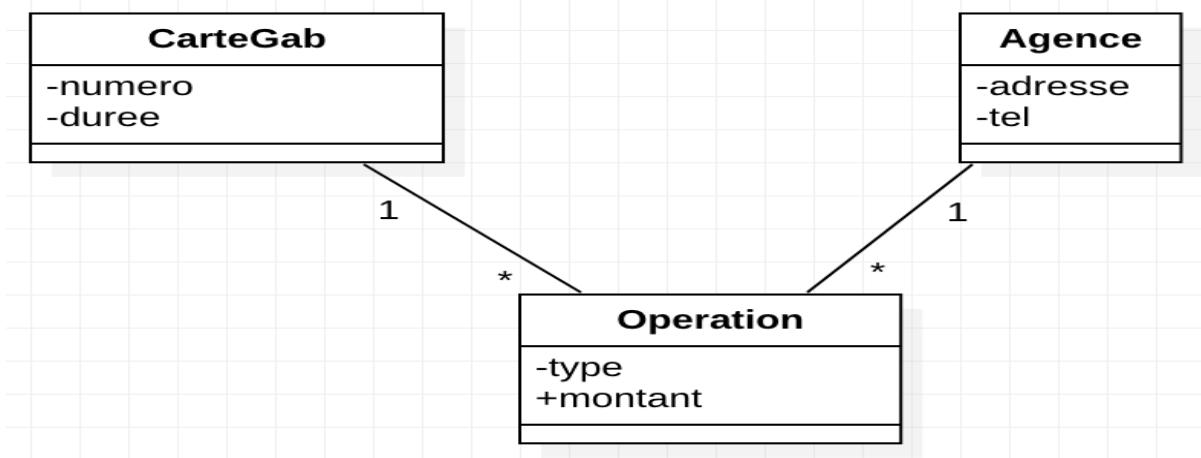
NB: Dans une Relation ManyToMany, une troisième table sera créée en base de Donnée (**Table Association**), pour configurer cette Table on utilisera :

- **@JoinTable** avec les attributs :
 - **name** qui définit le nom de la table association
 - **joinColumns** qui définit les colonnes de jointures avec **@JoinColumn**(attribut de la classe propriétaire) et **inverseJoinColumns** (attribut de la classe inverse)

- **Avec Classe Association**



Lors qu'on a une relation **ManyToMany** avec une classe **Association** on le traduit en **deux relations OneToMany ,ManyToOne**



ii) Relations Bidirectionnelles

Pour définir une relation bidirectionnelle entre deux entités

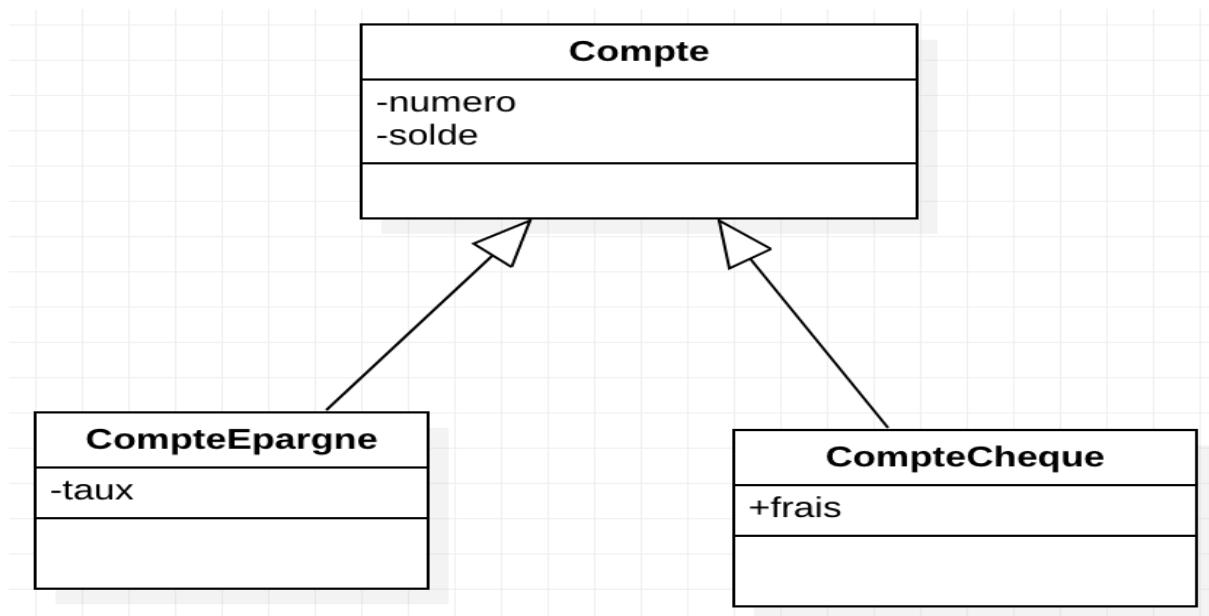
- si dans l'entité propriétaire la relation définie est `OneToMany`, alors dans l'entité inverse la relation sera `ManyToOne`, et inversement.
- si dans l'entité propriétaire la relation définie est `OneToOne`, alors dans l'entité inverse la relation sera aussi `OneToOne`.
- si dans l'entité propriétaire la relation définie est `ManyToMany`, alors dans l'entité inverse la relation sera aussi `ManyToMany`.

Reprendre toutes les relations en des relations bidirectionnelles.

iii) Relation Heritage

Trois possibilités avec l'héritage

- SINGLE_TABLE
- TABLE_PER_CLASS
- JOINED



- Un Classe Mère Compte
- Deux Classes Filles **Compte Épargne et Compte Chèque**

Transformation

Pour indiquer comment transformer les classes mère et filles en tables

Il faut utiliser l'annotation @Inheritance

- **SINGLE_TABLE** :toutes les Classes sont transformées en une seule table

Tout dans une seule table

Dans la classe mère on ajoute

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

Exemple

Pour distinguer un **Compte** ,d'un **CompteEpargne**, et d'un **CompteCheque**

- Dans la classe Mère Compte
 - `@DiscriminatorColumn(name="TYPE_COMPTE")`
 - `@DiscriminatorValue(value="COMPTE")`
- Dans la classe Fille CompteEpargne
 - `@DiscriminatorValue(value="EPARGNE")`
- Dans la classe Fille CompteCheque
 - `@DiscriminatorValue(value="CHEQUE")`

- Du côté de la Base de Donnée
 - Les 3 entités sont transformées en une table,la table compte
 - Les colonnes des classes filles sont nullables
 - La Colonne `TYPE_COMPTE` n'est créé pas dans la table compte

```
#package models / Compte.java

@Entity
@Table(name = "comptes")

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_COMPTE")
@DiscriminatorValue(value="COMPTE")

public class Compte {
```

```

#package models / CompteEpargne.java

@Entity
@Table(name = "comptes_epargne")
@DiscriminatorValue(value="EPARGNE")
public class CompteEpargne extends Compte{
    private double taux;
    #Getters and Setters

}

#package models / CompteCheque.java

@Entity
@Table(name = "comptes_chaque")
@DiscriminatorValue(value="CHEQUE")
public class CompteEpargne extends Compte{
    private double frais;
    #Getters and Setters

}

```

- **TABLE_PER_CLASSE**

Exemple avec une table pour chaque entité

- @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS) :
Chaque entité sera transformée en table.

- **Du côté de la Base de Donnée**

- Chaque entité est transformée en une table
- Les colonnes communes sont dupliquées
- La Colonne `TYPE_COMPTE` n'est créé pas dans la table compte

- JOINED

- `@Inheritance(strategy=InheritanceType.JOINED)`
- Du côté de la Base de Donnée
 - Chaque entité est transformée en une table
 - Les colonnes communes ne sont pas dupliquées
- La Colonne `TYPE_COMPTE` est créé dans la table compte

6) Manipulation des Entités

Gestion des entités par EntityManager

- EntityManager est une interface définie dans JPA.
- Chaque framework ORM possède sa propre implémentation de cette interface.
- EntityManager définit les méthodes qui permettent de gérer le cycle de vie de la persistance des Entity.
 - La méthode `persist()` permet rendre une nouvelle instance d'un EJB Entity persistante. Ce qui permet de sauvegarder son état dans la base de données
 - La méthode `find()` permet de charger une entité sachant sa clé primaire.
 - La méthode `createQuery()` permet de créer une requête EJBQL qui permet de charger une liste d'entités selon des critères.
 - La méthode `remove()` permet de programmer une entité pour la suppression.
 - La méthode `merge()` permet de rendre une entité détachée persistante.

```
#package views / Main.java
```

a) Les Méthodes de EntityManager

i) La méthode persist() :

Cette opération a pour effet de rendre une entité persistante. Si cette entité est déjà persistante, alors cette opération n'a aucun effet. Rendre une entité persistante consiste à l'écrire en base sur le prochain commit de la transaction dans laquelle on se trouve.

o createQuery()

- Pour sélectionner des données à partir de la base de données, on peut créer un objet Query en utilisant la méthode createQuery() de entityManager.
- La requête est spécifiée en utilisant le langage de requêtes JPA appelé HQL ou JPA QL.
- HQL ressemble à SQL, sauf que au lieu de des tables et des relations, entre les tables, on utilise les classes et les relations entre les classes.
- En fait, avec JPA, quant on fait la programmation orientée objet, on n'est pas sensé connaître la structure de la base de données, mais plutôt on connaît le diagramme de classes des différentes entités.
- C'est Hibernate qui va traduire le HQL en SQL. Ceci peut garantir à notre application de fonctionner correctement quelque soit le type de SGBD utilisé

Exemples Application

1. Entities

```
@Entity  
  
@Table(name = "personnes")  
  
@Inheritance(strategy = InheritanceType.JOINED)  
  
@DiscriminatorColumn(name = "type")  
  
@Data  
  
@AllArgsConstructor  
  
@Getter  
  
@Setter  
  
public class Personne {  
  
    @Id  
  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
  
    protected Long id;  
  
    protected String nomComplet;  
  
  
    public Personne() {  
    }  
  
    public Personne(String nomComplet) {  
        this.nomComplet = nomComplet;  
    }  
}
```

```
@Entity  
  
@DiscriminatorValue(value = "MED")  
  
@Table(name = "medecins")  
  
@AllArgsConstructor  
  
@Getter
```

```
@Setter  
  
public class Medecin extends Personne{  
  
    private String grade;  
  
    @OneToMany(mappedBy = "medecin")  
  
    List<RV> rvs;  
  
    public Medecin() {  
  
    }  
  
    public Medecin(String nomComplet, String grade) {  
  
        super(nomComplet);  
  
        this.grade = grade;  
  
    }  
  
}
```

```
@Entity  
  
@DiscriminatorValue(value = "PAT")  
  
@Table(name = "patients")  
  
@AllArgsConstructor  
  
@NoArgsConstructor  
  
@Getter  
  
@Setter  
  
public class Patient extends Personne {  
  
    private String antecedents;  
  
    @OneToMany(mappedBy = "patient")  
  
    List<RV> rvs;  
  
}
```

```

@Entity
@Table(name = "rvs")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class RV {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date dateRv;
    private String heureDb;
    private String heureFin;
    @ManyToOne()
    private Patient patient;
    @ManyToOne()
    private Medecin medecin;
}

```

2. Entitymanager

```

EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("demonJpa");
EntityManager em=emf.createEntityManager();
EntityTransaction transaction= em.getTransaction();

```

3. Les requêtes MAJ

a. Persist

```
transaction.begin();

try {

Medecin medecin=new Medecin("Ophtalmo","Birane Baila Wane");

em.persist(medecin);

transaction.commit();

} catch (Exception e) {

transaction.rollback();

}
```

b. Merge

```
transaction.begin();

try {

Medecin medecin=em.find(Medecin.class,1L);

medecin.setNomComplet("dddd");

em.merge(medecin);

transaction.commit();

} catch (Exception e) {

transaction.rollback();

}
```

4. CreateQuery

```
List<Medecin> medecins= em.createQuery("select m from Medecin m",Medecin.class)
.getResultList();

List<String> medecinsName= em.createQuery("select m.nomComplet from Medecin
m",String.class)

.getResultList();

Medecin medecin=em.createQuery("select m from Medecin m where m.nomComplet like :x"
,Medecin.class)

.setParameter("x","ddddd")

.getSingleResult();
```

5. NamedQuery

```
@NamedQuery(name="findMedecinByNom", query="select i from Medecin i where
i.nomComplet = :nom")

public class Medecin extends Personne{}
```

```
medecin= em.createNamedQuery("findMedecinByNom", Medecin.class)

.setParameter("nom", "ddddd")

.getSingleResult();
```

6. NamedQueries

```
@NamedQueries({

@NamedQuery(name="findPatientByNom", query="select i from Patient i where
i.nomComplet = :nom") ,

@NamedQuery(name="deletePatientByNom", query="delete from Patient i where
i.nomComplet = :nom") ,

@NamedQuery(name="deleteAllPatient", query="delete from Patient i")

})

public class Patient extends Personne {}
```

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

em.createNamedQuery("deletePatientByNom")
.setParameter("nom", "Seckou Diallo")
.executeUpdate();

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