



# JAVA

---

[m2information.fr](http://m2information.fr)





# HIBERNATE

La gestion de persistance

---

UTOPIOS



HIBERNATE

# INTRODUCTION

---

- La quasi-totalité des applications (développées en approche orienté objet) utilisent des système de gestion de base de données relationnel pour gérer et stocker leur données.
- La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :
  - le modèle objet propose plus de fonctionnalités : héritage, polymorphisme, ...
  - les relations entre les entités des deux modèles sont différentes.
  - les objets ne possèdent pas d'identifiant en standard (hormis son adresse mémoire qui varie d'une exécution à l'autre).
  - Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique

- La première approche pour faire une correspondance entre ces deux modèles a été d'utiliser l'API JDBC fournie en standard avec le JDK.
- Cependant, cette approche possède plusieurs inconvénients majeurs:
  - nécessite l'écriture de nombreuses lignes de codes, souvent répétitives.
  - le mapping entre les tables et les objets est un travail de bas niveau.
- Tous ces facteurs réduisent la productivité mais aussi les possibilités d'évolutions et de maintenance.
- De plus, une grande partie de ce travail peut être automatisé.



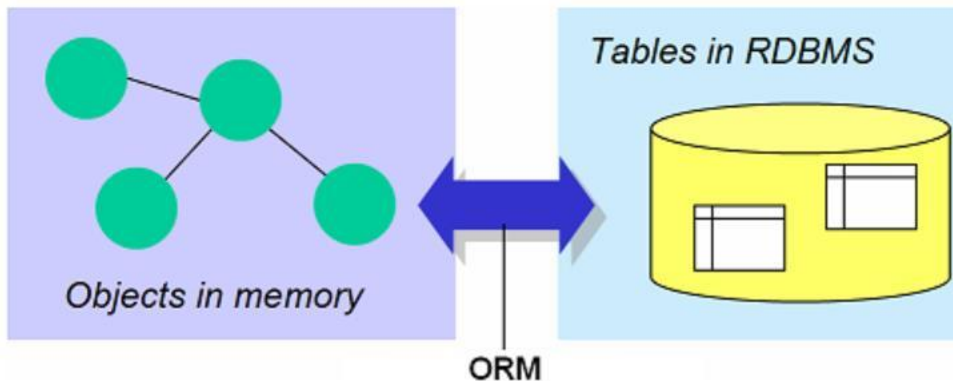
HIBERNATE

# PERSISTANCE DES DONNÉES & ORM

---

## La persistance des données

- La persistance des données est la notion qui traite de la sauvegarde de données contenue dans des objets sur un support informatique.
- Le mapping Objet/Relationnel (Object-Relational Mapping : ORM) consiste à réaliser la correspondance entre une application, modélisé en conception orientée objet, et une base de données relationnelle.
- L'ORM a pour but d'établir la correspondance entre : **une table de la base de données et une classe du modèle objet**



## Les fonctionnalités ORM

- Un outil de ORM doit proposer un certain nombre de fonctionnalités parmi lesquelles :
  - Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités.
  - Proposer une interface qui permette de facilement mettre en œuvre des actions de type CRUD.
  - Éventuellement permettre l'héritage des mappings.
  - Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée.
  - Supporter différentes formes d'identifiants générés automatiquement par les bases de données (identity, sequence, ...).





## Les fonctionnalités ORM

- Autres fonctionnalités proposées :
  - Support des transactions.
  - Gestion des accès concurrents (verrou, dead lock, ...).
  - Fonctionnalités pour améliorer les performances (cache, lazy loading, ...).
- Les solutions de mapping sont riches en fonctionnalités ce qui peut rendre leur mise en œuvre plus ou moins complexe.
- Les solutions de ORM permettent de réduire la quantité de code à produire mais impliquent une partie configuration (généralement sous la forme d'un ou plusieurs fichiers XML ou d'annotations).



## Java Persistence API (JPA)

- il s'agit d'un standard faisant partie intégrante de la plate-forme Java EE (JSR 220).
- une spécification qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données, ou autrement formulé la gestion de la persistance.
- les **interfaces** de JPA décrivent comment respecter le **standard**, mais il faut utiliser une solution/Framework qui fait l'**implémentation** de JPA.
- JPA repose sur des entités qui sont de simples **POJOs** annotés et sur un gestionnaire de ces entités (EntityManager) qui propose des fonctionnalités pour les manipuler (**CRUD**). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.
- JPA est définie dans le package ***javax.persistence***

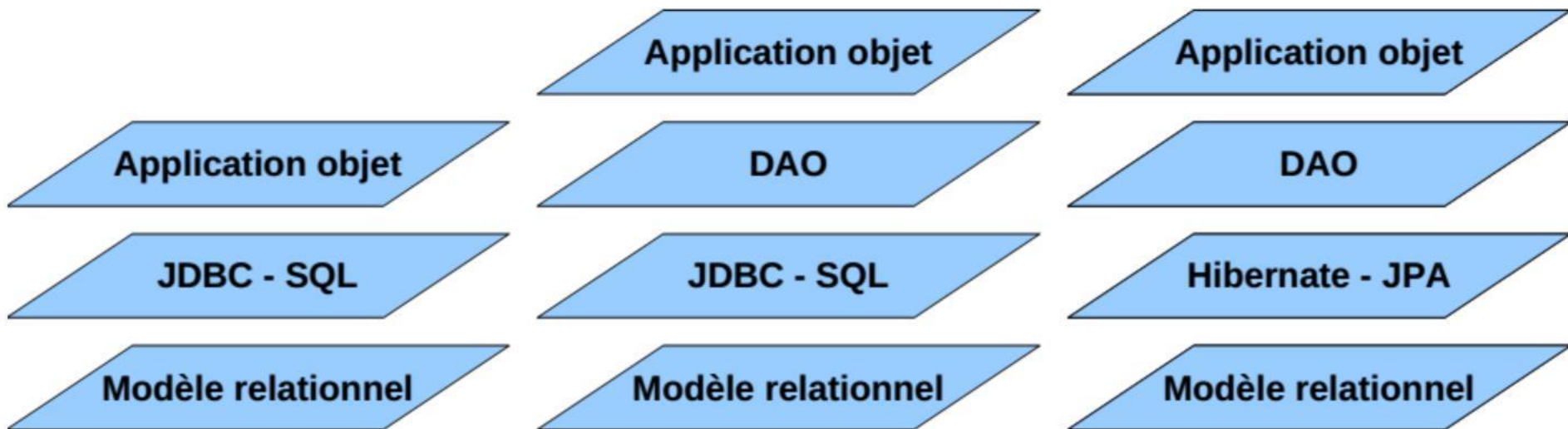
## Les Frameworks ORM

- Différentes solutions sont apparues :
  - **TopLink** est un framework d'ORM pour le développement Java. **TopLink Essentials** est la version open source du produit d'Oracle. Au mois de mars 2008, Sun Microsystems fait succéder EclipseLink à TopLink comme implémentation de référence de la JPA 2.0.
  - **EclipseLink** est un framework open source de ORM pour les développeurs Java. Il fournit une plateforme puissante et flexible permettant de stocker des objets Java dans une base de données relationnelle et/ou de les convertir en documents XML. Il est développé par l'organisation **Fondation Eclipse**
  - **Hibernate** est un framework open source gérant la persistance des objets en base de données relationnelle, développé par l'entreprise **JBoss**.
  - **Apache OpenJPA** est une implémentation open source de la JPA. Il s'agit d'une solution d'ORM pour le langage Java simplifiant le stockage des objets dans une base de données relationnelle. Il est distribué sous licence **Apache**.
- la plupart de ces solutions sortent du cadre défini par JPA, et proposent des fonctionnalités annexes qui leur sont propres.



- Modèles d'architecturaux:

1. JDBC sans DAO
2. JDBC avec DAO
3. ORM avec DAO



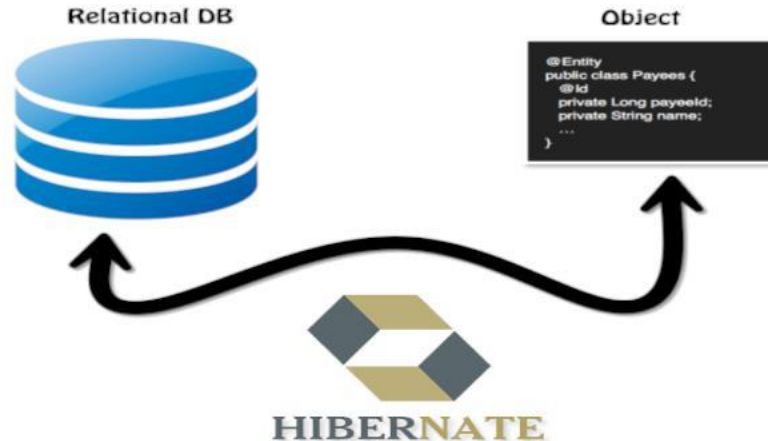


HIBERNATE

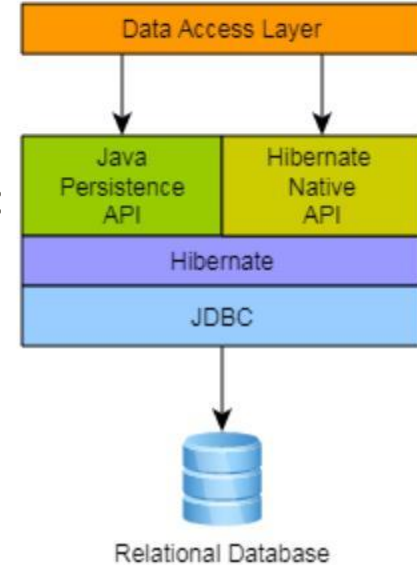
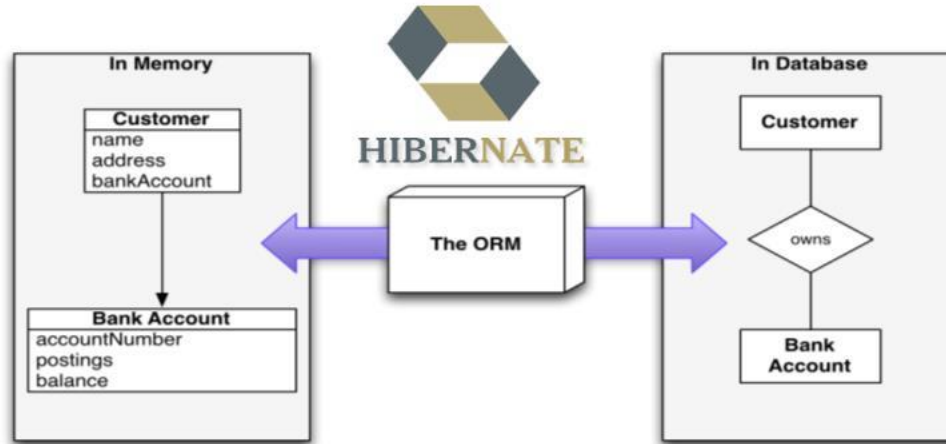
# PRÉSENTATION D'HIBERNATE

---

- Hibernate est une solution open source , écrit en java, de type qui permet de faciliter le développement de la couche persistance d'une application.
- Hibernate facilite la persistance et la recherche de données dans une base de données en réalisant lui-même la création des objets et les traitements de remplissage de ceux-ci en accédant à la base de données.
- Hibernate a été écrit sous la responsabilité de Gavin King qui fait partie de l'équipe JBoss.



- Hibernate propose son propre langage d'interrogation HQL et a largement inspiré les concepteurs de l'API JPA.
- Hibernate est très populaire notamment à cause de ses bonnes performances et de son ouverture à de nombreuses bases de données.
- Les bases de données supportées sont les principales du marché : HSQL Database Engine, DB2/NT, MySQL, PostgreSQL, FrontBase, Oracle, Microsoft SQL Server Database, Sybase SQL Server, Informix Dynamic Server, etc.




## Installation d'Hibernate

### 1 - Bibliothèque Hibernate

la version 5.2

Nom

-  antlr-2.7.7.jar
-  classmate-1.3.0.jar
-  dom4j-1.6.1.jar
-  hibernate-commons-annotations-5.0.1.Final.jar
-  hibernate-core-5.2.8.Final.jar
-  hibernate-jpa-2.1-api-1.0.0.Final.jar
-  jandex-2.0.3.Final.jar
-  javassist-3.20.0-GA.jar
-  jboss-logging-3.3.0.Final.jar
-  jboss-transaction-api\_1.2\_spec-1.0.1.Final.jar

### 2 – Drivers JDBC

Pour la communication avec la BD



mysql-connector-java-5.1.44-bin.jar

### 3 – Plugins Hibernate

Pour l'édition des fichiers de mapping



Hibernate Tools (Indigo, Helios, etc.)



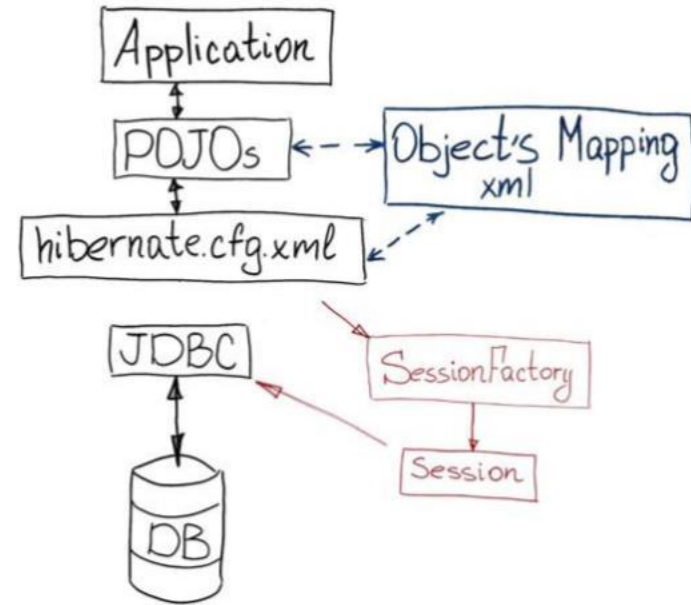


HIBERNATE

# CONFIGURATION D'HIBERNATE

---

- Hibernate a besoin de plusieurs éléments pour fonctionner:
  1. une classe de type **javabeen** qui encapsule les données d'une occurrence d'une table
  2. Un fichier de la configuration pour configurer les informations de la BD et les informations du mapping (**hibernate.properties** ou **hibernate.cfg.xml**)
  3. un fichier de correspondance qui configure la correspondance entre la classe et la table (**.hbm.xml**) ou bien l'utilisation **des annotations**.
- Une fois ces éléments correctement définis, il est possible d'utiliser Hibernate dans le code des traitements à réaliser.



### 3 - Création de fichier de Configuration

- Pour exécuter Hibernate, il faut lui fournir un certain nombre de propriétés concernant sa configuration pour qu'il puisse se connecter à la base de données.
- Ces propriétés peuvent être fournies sous plusieurs formes :
  - un fichier de configuration nommé ***hibernate.properties***
    - *Ce fichier contient des paires clé=valeur pour chaque propriété définie.*
  - Ou un fichier de configuration au format XML nommé ***hibernate.cfg.xml***
    - *Les propriétés sont alors définies par un tag <property>.*

### 3 - Création de fichier de Configuration

- Les principales propriétés pour configurer la connexion JDBC sont :
  - `hibernate.connection.driver_class` : nom pleinement qualifié de classe du pilote JDBC
  - `hibernate.connection.url` : URL JDBC désignant la base de données
  - `hibernate.connection.username` : nom de l'utilisateur pour la connexion
  - `hibernate.connection.password` : mot de passe de l'utilisateur
  - `hibernate.connection.pool_size` : nombre maximum de connexions dans le pool
- Les principales autres propriétés sont :
  - `hibernate.dialect` : nom de la classe pleinement qualifiée qui assure le dialogue avec la base de données
  - `hibernate.show_sql` : booléen qui précise si les requêtes SQL générées par Hibernate sont affichées dans la console
  - `hibernate.hbm2ddl.auto` : Exporte automatiquement le schéma DDL vers la base de données lorsque la SessionFactory est créée:
    - `validate`: valider la structure du schéma sans faire de modification dans la base de données
    - `update`: mettre à jour le schéma
    - `create`: créer le schéma en supprimant celui existant
    - `create-drop`: créer le schéma et le supprimer lorsque la sessionFactory est fermée

### 3 - Création de fichier de Configuration

#### *hibernate.properties*

##### **#MySQL Properties**

**hibernate.connection.driver\_class** = com.mysql.jdbc.Driver

**hibernate.connection.url** = jdbc:mysql://localhost:3306/base

**hibernate.connection.username** = myuser

**hibernate.connection.password** = mypassword

**hibernate.dialect** = org.hibernate.dialect.MySQL5Dialect

#### *hibernate.cfg.xml*

<?xml version= 1.0 encoding= utf-8 ?>

<hibernate-configuration>

<session-factory>

<property name= connection.url >jdbc:mysql://localhost:3306/base</property>

<property name= connection.driver\_class >com.mysql.jdbc.Driver</property>

<property name= connection.username >myuser</property>

<property name= connection.password >mypassword</property>

<property name= dialect >org.hibernate.dialect.MySQL5Dialect</property>

</session-factory>

</hibernate-configuration>

### 3 - Création de fichier de Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/hibernate_db</property>
    <property name="hibernate.connection.password">admin</property>
    <property name="hibernate.connection.username">admin</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">validate</property>
    <mapping class="ma.boukouchi.tp.entity.Personne"/>
    <!-- <mapping resource="ma/boukouchi/tp/entity/Personne.hbm.xml"/> -->
  </session-factory>
</hibernate-configuration>
```

## 1 - Création d'une classe encapsulant les données

- Cette classe doit respecter le standard des **javabeans** notamment encapsuler les propriétés dans des champs **private** avec des getters et setters et avoir un constructeur par défaut.
- On supposera qu'il existe une table Personne dans une base de données de type MySQL.

```
package entity;

public class Personne {
    private int id;
    private String nom;
    private String prenom;
    private int age;
    public Personne() {}
    // les getters et les setters
}
```

Table Personne			
	Name	Type	
1	<u>Id Person</u>	Int(11)	Primary Key
2	person_nom	varchar(45)	
3	person_prenom	varchar(45)	
4	person_age	Int(11)	

```
CREATE TABLE personne ( id_person INT NOT NULL,
person_nom VARCHAR(45) NULL,
person_prenom VARCHAR(45) NULL,
person_age INT NULL,
PRIMARY KEY (id_person));
```

## 2 - Création de mapping O/R

- Pour assurer le mapping, Hibernate a besoin d'un fichier de correspondance (mapping file) au format XML qui va contenir des informations sur la correspondance entre la classe définie et la table de la base de données.
- Il faut définir un fichier de mapping par classe, nommé du nom de la classe suivi par **.hbm.xml**.
- Ce fichier doit être situé dans le même répertoire que la classe correspondante ou dans la même archive pour les applications packagées.
  - Différents éléments sont précisés dans ce document XML :
    - la classe qui va encapsuler les données,
      - l'identifiant dans la base de données et son mode de génération,
      - le mapping entre les propriétés de classe et les champs de la base de données,
      - les relations,
      - etc.



## 2 - Création de mapping O/R

### Deux formes de mapping

#### personne.hbm.xml

##### Mapping par fichier XML

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<!-- Generated 24 d c. 2018 00:19:34 by Hibernate Tools 3.5.0.Final -->
<hibernate-mapping>
  <class name="ma.boukouchi.tp.entity.Personne" table="PERSONNE">
    <id name="id" type="int">
      <column name="id_Person" />
      <generator class="assigned" />
    </id>
    <property name="nom" type="java.lang.String">
      <column name="person_nom" />
    </property>
    <property name="prenom" type="java.lang.String">
      <column name="person_prenom" />
    </property>
    <property name="age" type="int">
      <column name="person_age" />
    </property>
  </class>
</hibernate-mapping>
```

##### Mapping par annotations Java

```
import javax.persistence.*;

@Entity
public class Personne {
    @Id
    @Column(name="id_person")
    private int id;
    @Column(name="person_nom")
    private String nom;
    @Column(name="person_prenom")
    private String prenom;
    @Column(name="person_age")
    private int age;

    // constructeurs
    public Personne() { }

    // les getters et les setters
```



HIBERNATE

# OBJECT/RELATIONAL MAPPING @NNOTATIONS

## @Entity & @Table

@Entity

```
public class Livre{  
    private String titre;  
    // getters & setters  
}
```

@Entity

@Table(name="LIVRE")

```
public class Livre{  
    private String titre;  
    // getters & setters  
}
```

- **@Entity** déclare la classe comme un entity bean.
  - **@Entity(access=AccessType.FIELD)** permet d'accéder directement aux champs à rendre persistant. (par défaut)
  - **@Entity(access=AccessType.PROPERTY)** oblige le fournisseur à utiliser les accesseurs (les getters).
- **@Table** définit le nom de la table, du catalogue, du schéma et des contraintes d'unicité.
  - Si aucune @Table n'est définie les valeurs par défaut sont utilisées (le nom de la classe de l'entité).

## @Entity & @Table

### @Entity

```
public class Livre{  
    @Column(name="titre")  
    private String titre;  
    @Column  
    private String edition;  
    @Transient  
    private double prix;  
    // getters & setters  
}
```

- **@Column** indique le nom de la colonne dans la table. Par défaut, toutes les propriétés non-statiques d'une classe-entité sont considérées comme devant être stockées dans la base.
- Voici les principaux attributs pour **@Column**.
  - **name** indique le nom de la colonne dans la table;
  - **length** indique la taille maximale de la valeur de la propriété;
    - **nullable** (avec les valeurs false ou true) indique si la colonne accepte ou non des valeurs à NULL ;
    - **unique** indique que la valeur de la colonne est unique.
- **@Transient** indique que la propriété n'est pas persistante dans la base.

## @Id & @GeneratedValue

```
@Entity
public class Livre{
    @Id
    private int id;
    // getters & setters
}
```

```
@Entity
public class Livre{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    // getters & setters
}
```

- **@Id** déclare la propriété identifiant de cet entity bean.
- **@GeneratedValue** définit la stratégie de génération automatique de l'identifiant:
  - **Strategy = GenerationType.AUTO** : Hibernate produit lui-même la valeur des identifiants grâce à une table *hibernate\_sequence*. Le générateur AUTO est le type préféré pour les applications portables.
  - **Strategy = GenerationType.IDENTITY** : Hibernate s'appuie alors sur le mécanisme propre au SGBD pour la production de l'identifiant. (Dans le cas de MySQL, c'est l'option AUTO-INCREMENT, dans le cas de postgres ou Oracle, c'est une séquence).
  - **Strategy = GenerationType.TABLE**: Hibernate utilise une table dédiée qui stocke les clés des tables générées, l'utilisation de cette stratégie nécessite l'utilisation de l'annotation **@TableGenerator**.
  - **Strategy = GenerationType.SEQUENCE**: Hibernate utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle. L'utilisation de la stratégie « SEQUENCE » nécessite l'utilisation de l'annotation **@SequenceGenerator**.

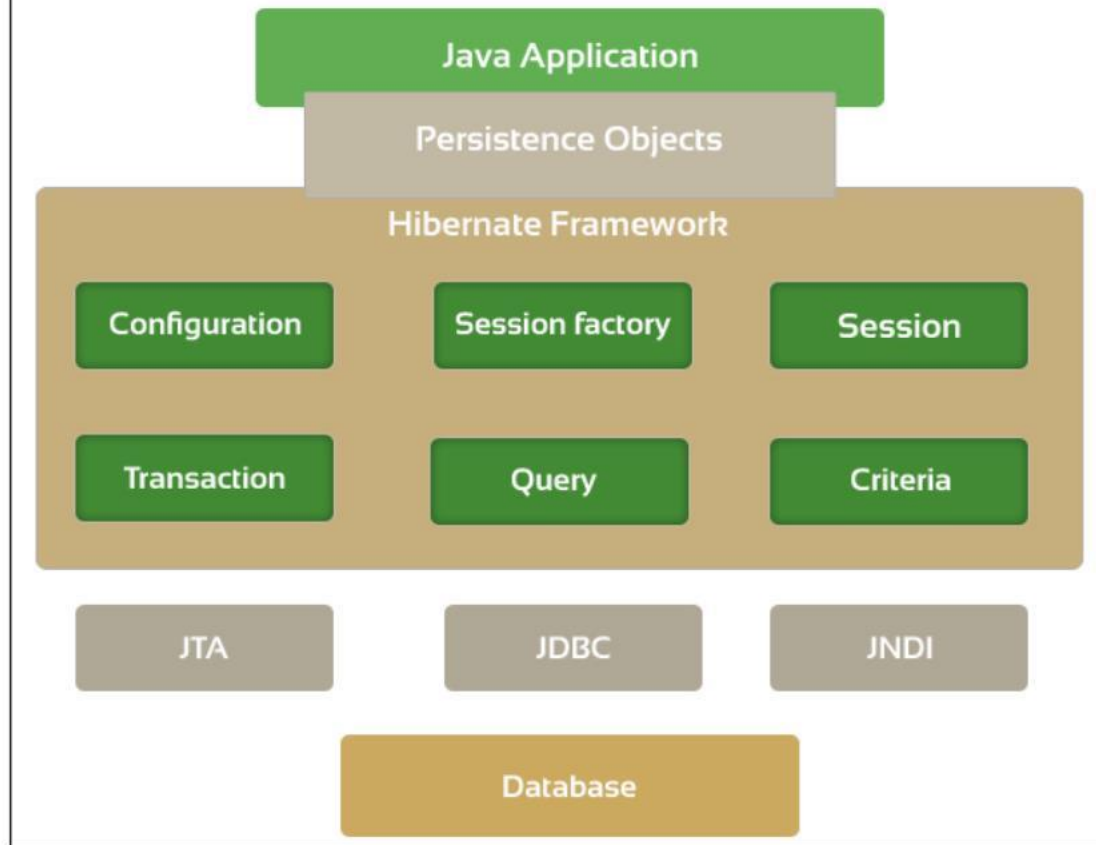


HIBERNATE

# UTILISATION D'HIBERNATE

---

## Hibernate Architecture



- Pour utiliser Hibernate dans le code, il est nécessaire de réaliser plusieurs opérations :
  1. création d'une instance de la classe **Configuration** (les propriétés sont définies dans le fichier **hibernate.cfg.xml**).
  2. création d'une instance de la classe **SessionFactory**, cet objet est obtenu à partir de l'instance du type **Configuration** en utilisant la méthode **buildSessionFactory()**.
  3. création d'une instance de la classe **Session** qui va permettre d'utiliser les services d'Hibernate, la classe **Session** est obtenu à partir d'une fabrique de type **SessionFactory** à l'aide de sa méthode **openSession()**.
  4. Par défaut, la méthode **openSession()** ouvre une connexion vers la base de données en utilisant les informations fournies par les propriétés de configuration.



```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import entity.Personne;
public class HibernateHellow {
public static void main(String[] args) {
//hibernate 3.0
SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();
// Hibernate 4.0 et plus, il est recommandé d'utiliser l'interface StandardServiceRegistry
StandardServiceRegistry registre= new StandardServiceRegistryBuilder().configure().build();
SessionFactory sessionFactory= new MetadataSources(registre). buildMetadata(). buildSessionFactory();

Session session=sessionFactory.openSession();
// suite de code
session.close();
sessionFactory.close();
}
}
```

- Pour créer **une nouvelle occurrence** dans la source de données, il suffit de :
  - créer une nouvelle instance de la classe encapsulant les données, de valoriser ces propriétés, et d'appeler la méthode **save()** de la session en lui passant en paramètre l'objet encapsulant les données.
  - La méthode **save()** n'a aucune action directe sur la base de données.
  - Pour enregistrer les données dans la base, il faut réaliser un commit sur la transaction de la classe Session.

```
SessionFactory sessionFactory=new Configuration().configure().buildSessionFactory();
Session session=sessionFactory.openSession();
session.beginTransaction();
Personne per=new Personne();
per.setNomPersonne("Ali");
per.setPrenomPersonne("Ahmed");
per.setAgePersonne(25);
session.save(per);
session.getTransaction().commit();
```

- La méthode **load()** ou **get()** de la classe Session permet **d'obtenir une instance de la classe des données** encapsulant les données de l'occurrence de la base dont l'identifiant est fourni en paramètre.
  - `Personne perChercher=session.load(Personne.class, 1);` // lève une exception si n'est pas trouvé
  - `Personne perChercher=session.get(Personne.class, 1);` // retourne null si n'est pas trouvé
  - `System.out.println("Nom :"+ perChercher.getNomPersonne());`
- Pour **supprimer une occurrence** encapsulée dans une classe, il suffit d'invoquer la méthode **delete()** de la classe Session en lui passant en paramètre l'instance de la classe.
  - `session.delete(perChercher);`
- Pour **mettre à jour une occurrence** dans la source de données, il suffit d'appeler la méthode **update()** de la session en lui passant en paramètre l'objet encapsulant les données. La méthode **saveOrUpdate()** laisse Hibernate choisir entre l'utilisation de la méthode `save()` ou `update()` en fonction de la valeur de l'identifiant dans la classe encapsulant les données.
  - Exemple :
    - `perChercher.setNomPersonne("Amal");`
    - `session.update(perChercher);`

```

public static void main(String[] args) {
SessionFactory factory=HibernateUtil.getFactory();
Session session=factory.openSession();
session.getTransaction().begin();
Personne persRechercher=session.load(Personne.class, 4);
persRechercher.setAge(20);
persRechercher.setNom("Mohamed");
session.update(persRechercher);
Personne persSupprimer=session.load(Personne.class, 5);
session.delete(persSupprimer);
session.getTransaction().commit();
session.close();
}

```

```

public class HibernateUtil {
private static SessionFactory factory=null;
private HibernateUtil(){}
public static SessionFactory getFactory(){
if (factory==null){
StandardServiceRegistry registre=
new StandardServiceRegistryBuilder().
configure().build();
factory=new MetadataSources(registre).
buildMetadata().
buildSessionFactory();
}
return factory;
}
}

```

```

Hibernate: select personne0_.id_person as id_perso1_0_0_, personne0_.person_age as person_a2_0_0_, personne0_.person_nom
as person_n3_0_0_, personne0_.person_prenom as person_p4_0_0_ from Personne personne0_ where personne0_.id_person=?
Hibernate: select personne0_.id_person as id_perso1_0_0_, personne0_.person_age as person_a2_0_0_, personne0_.person_nom
as person_n3_0_0_, personne0_.person_prenom as person_p4_0_0_ from Personne personne0_ where personne0_.id_person=?
Hibernate: update Personne set person_age=?, person_nom=?, person_prenom=? where id_person=?
Hibernate: delete from Personne where id_person=?

```



HIBERNATE

H Q L

HIBERNATE QUERY LANGUAGE

- Hibernate utilise plusieurs moyens pour obtenir des données de la base de données :
  1. **Hibernate Query Language (HQL)**
  2. **API Criteria** // deprecated par Hibernate 5, utiliser CriteriaQuery de API JPA .
  3. **SQL natives**

- **Hibernate propose son propre langage nommé HQL (Hibernate Query Language) pour interroger les bases de données.**
  - Le langage HQL est proche de SQL avec une *utilisation sous forme d'objets des noms de certaines entités*.
  - Il n'y a aucune référence aux tables ou aux champs car ceux-ci *sont référencés respectivement par leur classe et leurs propriétés*.
  - Hibernate qui se charge de générer la requête SQL à partir de la requête HQL en tenant compte du contexte.
- **Une requête HQL peut être composée :**
  - de clauses (from, select, where, ...)
  - de fonctions d'agrégation (Max, Count, Min, ...)
  - de sous requêtes

## Hibernate HQL vs JDBC SQL

```
List<Movie> movies = new ArrayList<Movie>();
Movie m = null;
try {
    Statement st = getConnection(). createStatement();
    ResultSet rs = st. executeQuery("SELECT * FROM MOVIES");
    while (rs. next()) {
        m = new Movie();
        m. setId(rs. getInt("ID"));
        m. setTitle(rs. getString("TITLE "));
        movies. add(m);
    }
} catch (SQLException ex) {
    System. err. println(ex. getMessage());
}
```

VS

```
List<Movie> movies=session. createQuery("from Movie" ).list(); // Hibernate HQL
```



## La classe Query : createQuery

- La mise en œuvre de HQL peut se faire de plusieurs manières.
  - Le plus courant est d'obtenir une instance de la classe **Query** en invoquant la méthode **createQuery()** de la **session** Hibernate courante.
  - **createQuery()** attend en paramètre la requête HQL qui devra être exécutée.

*// Get the current session*

**Session session = sessionFactory.getSession();**

*// Instance of the query is created from this session*

**Query query = session.createQuery("from **Personne**");**

### *La classe Query : liste() & iterator()*

- Pour parcourir la collection des occurrences trouvées, l'interface Query propose:
  - la méthode `list()` qui renvoie une collection de type List
  - la méthode `list().iterator()` qui renvoie un itérateur sur la collection

*// L'utilisation d'un iterator()*

```
Query query = session.createQuery("from Personne");  
Iterator personnes= query.iterate();  
while(personnes.hasNext()){  
    Personne per = (Personne) personnes.next();  
    System.out.println("Personne :"+ per);  
}
```

*// L'utilisation d'une liste()*

```
Query query = session.createQuery("from Personne");  
List<Personne> personnes= query.list();  
for (Personne per : personnes) {  
    System.out.println("Personne:"+ per);  
}
```

## La classe Query : les clauses

Les clauses sont les mots clés HQL qui sont utilisés pour définir la requête :

- **where condition** : précise une condition qui permet de filtrer les occurrences retournées. Doit être utilisé avec une clause select et/ou from
  - *from Personne where nom = 'Ali'*
- **order by propriété [asc|desc]** : précise un ordre de tri sur une ou plusieurs propriétés. L'ordre par défaut est ascendant
  - *from Personne where nom='ginfo' order by prenom desc, age asc*
- **group by propriete** : précise un critère de regroupement pour les résultats retournés. Doit être utilisé avec une clause select et/ou from
  - *from Personne order group by age*

## La classe Query : uniqueResult

Dans le cas où on attend qu'un seul occurrence (**uniqueResult**):

```
Query query = session.createQuery(" from Personne where id=1 ");  
Personne per = (Personne) query.uniqueResult();
```

## La classe Query : Paramètre nommé

- Il est également possible de définir des requêtes utilisant des **paramètres nommés**.
- Dans ces requêtes, les paramètres sont précisés avec un caractère « : » suivi d'un nom unique.
- L'interface Query propose de nombreuses méthodes **setXXX()** pour associer à chaque paramètre une valeur en fonction du type de la valeur (XXX représente le type). // deprecated dans Hibernate 5, utiliser **setParameter**(para,valeur)
- Chacune de ces méthodes possède deux surcharges permettant de préciser le paramètre à partir de son nom dans la requête et sa valeur.

```
Query query =  
session.createQuery("from Personne where id=:idP and nom=:nomP");  
query.setInteger("idP", 1);  
query.setString("nomP", "Ali");
```

## La classe Query : Paramètre positionné

- Il est également possible de définir des requêtes utilisant des **paramètres positionnés**.
- Dans ces requêtes, les paramètres sont précisés avec un caractère « ? ».
- les méthodes **setXXX()** ou **setParameter()** possède deux surcharges permettant de:
  - préciser l'index de paramètre dans la requête(à partir de 0) )
  - La valeur de paramètre.

```
Query query =  
session.createQuery("from Personne where id=? and nom=? ");  
query.setInteger(0, 1 );  
query.setString(1, "Ali");
```

## La classe Query : Alias

- Parfois, nous souhaitons donner un nom à l'objet que nous interrogeons. Les noms donnés à ces objets sont appelés **alias**, et ils sont particulièrement utiles lorsque nous construisons des jointures ou des sous-requêtes.

*// The per is the alias to the object Personne*

```
Query query = session.createQuery( "from Personne as per where per.nom=:nom  
and per.id=:id" );
```

## La classe Query : Select

- Pour préciser les propriétés à renvoyer. Nous utilisons la clause **Select**.

*// Une seule colonne*

```
Query query =
session.createQuery("SELECT nom from Personne");
List<String> listNoms= query.list();
System.out.println("Nom des personnes:");
// Loop through all of the result columns
for (String nom: listNoms) {
    System.out.println("\t" + nom);
}
```

```
String qr =
"SELECT nom, prenom from Personne";
Query query = session.createQuery(qr);
Iterator personnes= query.list().iterator();
while(personnes.hasNext()){
    Object[] p = (Object[])personnes.next();
    System.out.print("Nom: " + p[0] + "\t ");
    System.out.println("Prenom: " + p[1]);
}
```

```
String QUERY =
"SELECT new Chef(id, nom ) from Personne";
List<Chef> chefs = session.createQuery(QUERY).list();
for (Chef chef : chefs) {
    System.out.println("Les chefs: " + chef);
}
```



## La classe Query : Fonctions d'Agrégation

Les fonctions d'agrégation HQL ont un rôle similaire à celles de SQL, elles permettent de calculer des valeurs agrégeant des valeurs de propriétés issues du résultat de la requête.

- **count**( [distinct|all|\*] object | object.property )
- **sum**( [distinct|all] object.property )
- **avg**( [distinct|all] object.property )
- **max**( [distinct|all] object.property )
- **min**( [distinct|all] object.property )

*// Fetching the max age*

```
int ageMax= (int)session.createQuery("select max(age) from Personne").uniqueResult();
```

```
System.out.println("Age max:"+ageMax);
```

*// Getting the average age of persons*

```
double ageMoyen= (double)session.createQuery("select avg(age) from Personne").uniqueResult();
```

```
System.out.println("Age Moyenne:"+ageMoyen);
```

```
INFO: HHH000397: Using ASTQueryTranslatorFactory
```

```
Hibernate: select max(personne0_.person_age) as col_0_0_ from Personne personne0_
```

```
Age max:38
```

```
Hibernate: select avg(personne0_.person_age) as col_0_0_ from Personne personne0_
```

```
Age Moyenne:22.25
```

La classe Query : IN {ensemble}

Nous utilisons la clause **IN** pour extraire des données avec des critères correspondant à une liste sélective.

```
// Define a list and populate with our criteria
```

```
List villeList = new ArrayList();
```

```
villeList.add("Casablanca");
```

```
villeList.add("Agadir");
```

```
Query query = session.createQuery("from Personne where ville in (:villes) ");
```

```
query.setParameterList("villes", villeList);
```

```
List<Personne> personnes = query.list();
```

```
...
```

## La classe Query : Updates & Deletes

- Il existe un autre moyen de mettre à jour et de supprimer des données en utilisant la méthode **executeUpdate** de la requête.
- La méthode attend une chaîne de requête avec des paramètres de liaison.

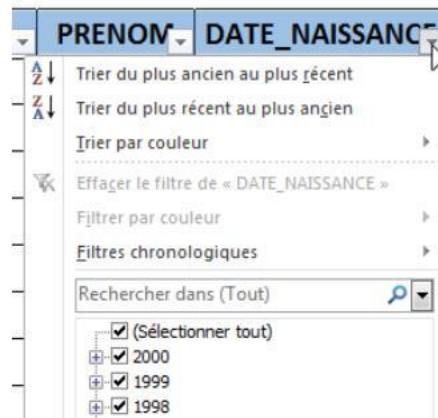
*// To update the record*

```
String update_query="update Personne set nom=:nomP where id=2";  
Query query = session.createQuery(update_query);  
query.setParameter("nomP", "Sirina");  
int success = query.executeUpdate();
```

*// To delete a record*

```
String delete_query="delete Personne where id=5";  
Query query = session.createQuery(delete_query);  
int success = query.executeUpdate();
```

- En plus du langage HQL, Hibernate propose l'API **Criteria** qui permet de construire des requêtes pour filtrer la base de données.
- Cette API permet facilement de combiner de nombreux critères (filtres) pour créer une requête : elle est particulièrement adaptée pour créer dynamiquement des requêtes à la volée comme c'est le cas par exemple pour des requêtes effectuant des recherches multicritères à partir d'informations fournies par l'utilisateur.
- Elle propose des classes et des interfaces qui encapsulent les fonctionnalités de SQL dont les principales sont :
  - **Criteria**
  - **Criterion**
  - **Restrictions**
  - **Projection**
  - **Order**



- Les critères de recherche - *permettant de restreindre les données retournées par la requête* - sont définis par l'interface **Criterion**.
- Le type **Criterion** encapsule un élément de la clause **where** de la requête SQL.
- La classe **Restrictions** est une fabrique qui propose des méthodes statiques pour créer des instances de type **Criterion**

```
Criteria criteria = session.createCriteria(Personne.class);
Criterion critere = Restrictions.eq("id", 2); // equal
criteria.add(critere);
List personnes = criteria.list();
```

```
Criteria criteria = session.createCriteria(Personne.class)
.add(Restrictions.eq("prenom", "Amal"))
.add(Restrictions.between("dateNaissance", fromDate, toDate))
.add(Restrictions.ne("ville", "Casablanca")); // not equal
```

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having firstName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));

// To check if the given property is null
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

- La classe **Projections** est une fabrique pour les instances de type Projection.

```
Criteria criteria = session.createCriteria(Personne.class);
Criterion critere = Restrictions.gt("age", 18);
criteria.add(critere);
criteria.setProjection(Projections.rowCount());
long nbr = (long) criteria.uniqueResult();
System.out.println("Nombre : "+nbr);
```

```
Hibernate: select count(*) as y0_ from Personne this_
where this_.person_age>?
Nombre : 7
```

*// Selecting all name columns*

```
List pers= session.createCriteria(Personne.class)
. setProjection(Projections.property("Nom"))
. list();
```

```
Criteria cr = session.createCriteria(Employee.class);
// To get total row count.
cr.setProjection(Projections.rowCount());
// To get average of a property.
cr.setProjection(Projections.avg("salary"));
// To get distinct count of a property.
cr.setProjection(Projections.countDistinct("firstName"));
// To get maximum of a property.
cr.setProjection(Projections.max("salary"));
// To get minimum of a property.
cr.setProjection(Projections.min("salary"));
// To get sum of a property.
cr.setProjection(Projections.sum("salary"));
```

- La classe **Order** encapsule une clause SQL **order by** .

```
Criteria criteria = session.createCriteria(Personne.class);
Criterion critere = Restrictions.gt("age", 20); // supérieure
criteria.add(critere);
criteria.addOrder(Order.asc("age"));
List<Personne> personnes = criteria.list();
```

```
Personne(6,Bourmi,Karima,19)
Personne(2,Sirina,Sirrin,20)
Personne(4,Mohamed,Ali,20)
Personne(3,Ameur,Mariem,27)
Personne(7,Amrani,Fatiha,27)
Personne(1,Boukouchi,Youness,28)
Personne(5,Boudil,Fadol,28)
```

```
Criteria criteria = session.createCriteria(Personne.class);
Criterion critere = Restrictions.gt("age", 18); // supérieure
criteria.add(critere);
criteria.addOrder(Order.asc("age"));
criteria.addOrder(Order.desc("nom"));
List<Personne> personnes = criteria.list();
```

```
Personne(6,Bourmi,Karima,19)
Personne(2,Sirina,Sirrin,20)
Personne(4,Mohamed,Ali,20)
Personne(7,Amrani,Fatiha,27)
Personne(3,Ameur,Mariem,27)
Personne(1,Boukouchi,Youness,28)
Personne(5,Boudil,Fadol,28)
```

## Native SQL

- Hibernate fournit également une fonctionnalité pour exécuter des requêtes SQL natives.
- La méthode `session.createQuery` renvoie un objet `SQLQuery`, similaire à la méthode utilisée par `createQuery` pour renvoyer un objet `Query`.
- Cette classe étend la classe `Query` vu précédemment.

```
SQLQuery query = session.createQuery("select * from PersonneTable");  
List employees = query.list();
```



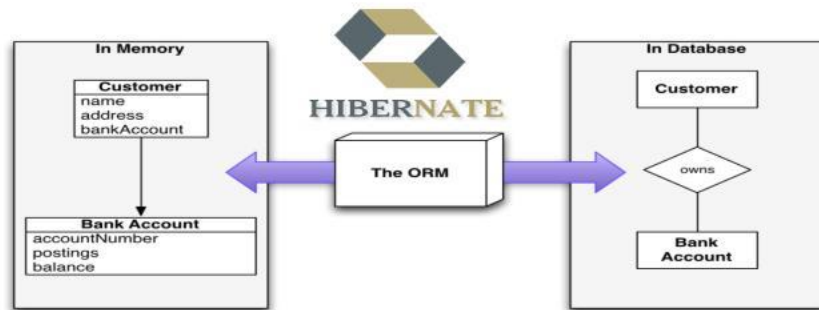


HIBERNATE

# LE MAPPING DES ASSOCIATIONS

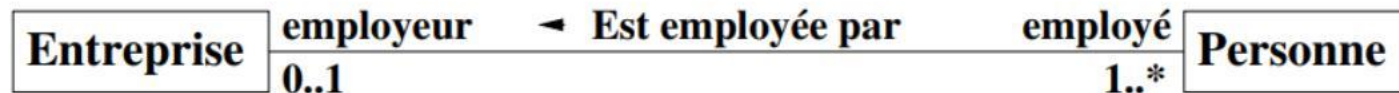
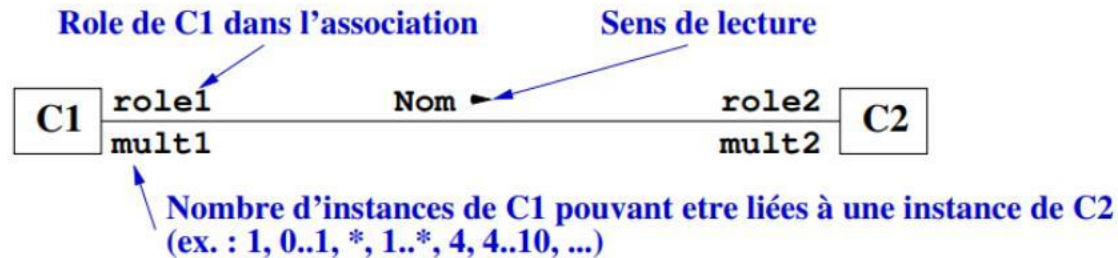
## Les associations

- Un des fondements du modèle de données relationnelles repose sur les relations qui peuvent intervenir entre les tables.
- Les relations utilisables dans le monde relationnel et le monde objet sont cependant différentes:
  - Les relations du monde objets :
    - Elles sont réalisées via des références entre objets
    - Elles peuvent mettre en œuvre l'héritage et le polymorphisme
  - Les relations du monde relationnel :
    - Elles sont gérées par des clés étrangères et des jointures entre tables
- Hibernate supporte plusieurs types de relations :
  - relation de type 1 - 1 (**one-to-one**)
  - relation de type 1 - n (**one-to-many**)
  - relation de type n - 1 (**many-to-one**)
  - relation de type n - n (**many-to-many**)



## La direction d'une associations

- Les associations permettent de manifester les liens entre les classes du modèle de persistance.
- Une association possède deux rôles, un à chacune de ses extrémités. La cardinalité d'une association dépend de la cardinalité de ces deux rôles.

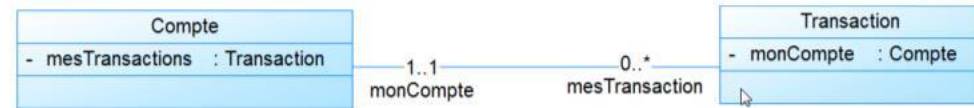


- **La navigabilité**
- Capacité d'une instance de C1 (resp. C2) à accéder aux instances de C2 (resp. C1)
- **Bidirectional** (Par défaut) : Navigabilité dans les deux sens, les objets peuvent traverser les deux coté de la relation:
  - C1 a un attribut de type C2 et C2 a un attribut de type C1
- **Unidirectional** : Spécification de la navigabilité : Orientation de la navigabilité, les objets peuvent uniquement traverser d'un côté de la relation, par exemple:
  - C1 a un attribut du type de C2, mais pas l'inverse



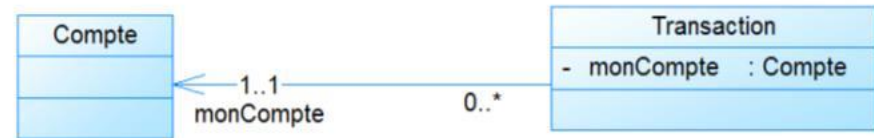
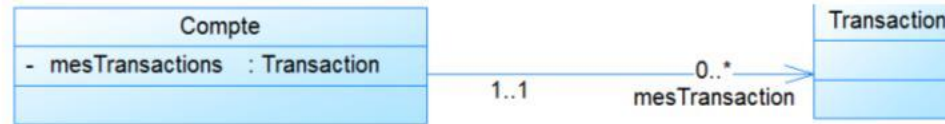
- **Bidirectional**, exemple:

- Compte et Transaction sont propriétaires de l'association
  - *Etant donné un objet Compte, il peut obtenir ses objets Transaction.*
  - *Etant donné un objet Transaction, il peut obtenir son propre objet Compte.*



- **Unidirectional**, exemple:

- Compte : propriétaire de l'association
  - *Etant donné un objet Compte, il peut obtenir ses objets Transaction.*
  - *Etant donné un objet Transaction, il ne peut pas obtenir son propre objet Compte.*
- Transaction: propriétaire de l'association
  - *Etant donné un objet Transaction, il peut obtenir son objet Compte.*
  - *Etant donné un objet Compte, il ne peut pas obtenir ses propres objets Transaction.*



## Les annotations



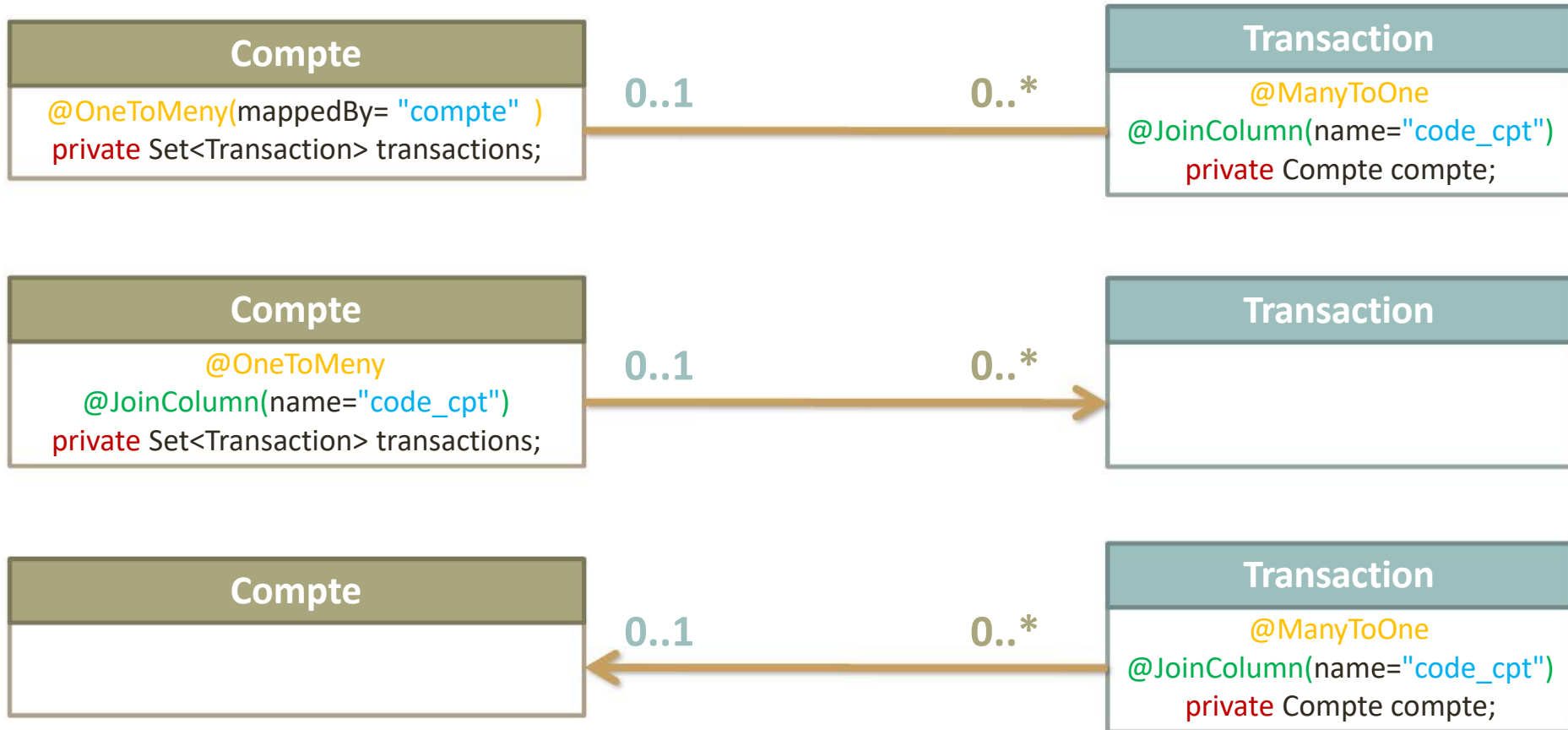
Classe 1	
@ maClasse 2	multi1
One To One	0..1
One To One   Not Null	1..1
Many To One	0..1
Many To One   Not Null	1..1
One To Many	0..*
One To Many   Not Null	1..*
Many To Many	0..*
Many To Many   Not Null	1..*

Classe 2	
multi2	@ maClasse 1
0..1	One To One
1..1	One To One   Not Null
0..*	One To Many
1..*	One To Many   Not Null
0..1	Many To One
1..1	Many To One   Not Null
0..*	Many To Many
1..*	Many To Many   Not Null

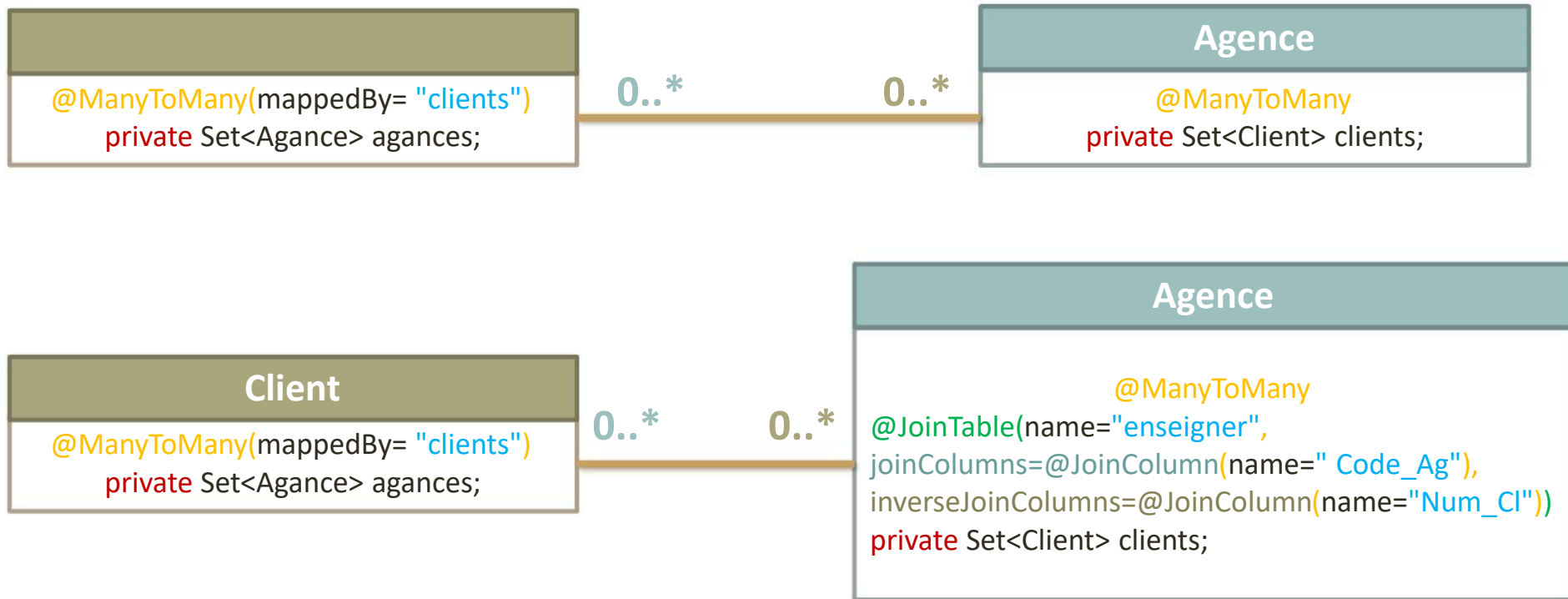
## La direction d'une associations

- La réduction de la portée de l'association est souvent réalisée en phase d'implémentation. C'est le développeur qu'est responsable de la gestion de la navigabilité des objets, c'est dire les deux bouts de l'association.
- Un des bouts est le propriétaire de l'association (Master-Slave):
  - pour les associations autres que M-N, ce bout correspond à la table qui contient la clé étrangère qui traduit l'association.
  - pour les associations M-N, le développeur choisit le bout propriétaire (de manière arbitraire).
  - l'autre bout (non propriétaire) est qualifié par l'attribut **mappedBy** qui donne le nom de l'association correspondante dans le bout propriétaire:
    - `@annotation( mappedBy= "ObjetAutreBout")`.

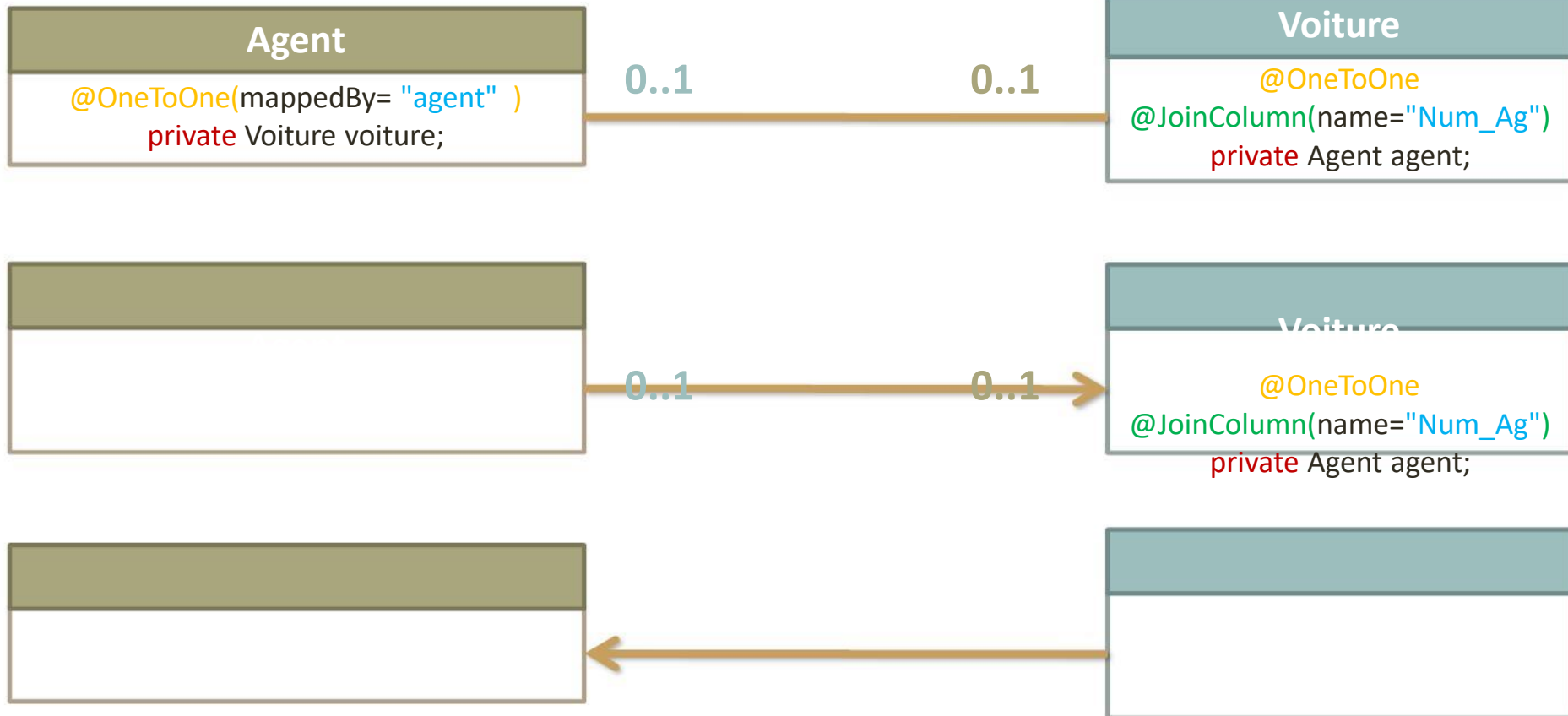
## Les annotations







## Les annotations



## Récupération d'associations

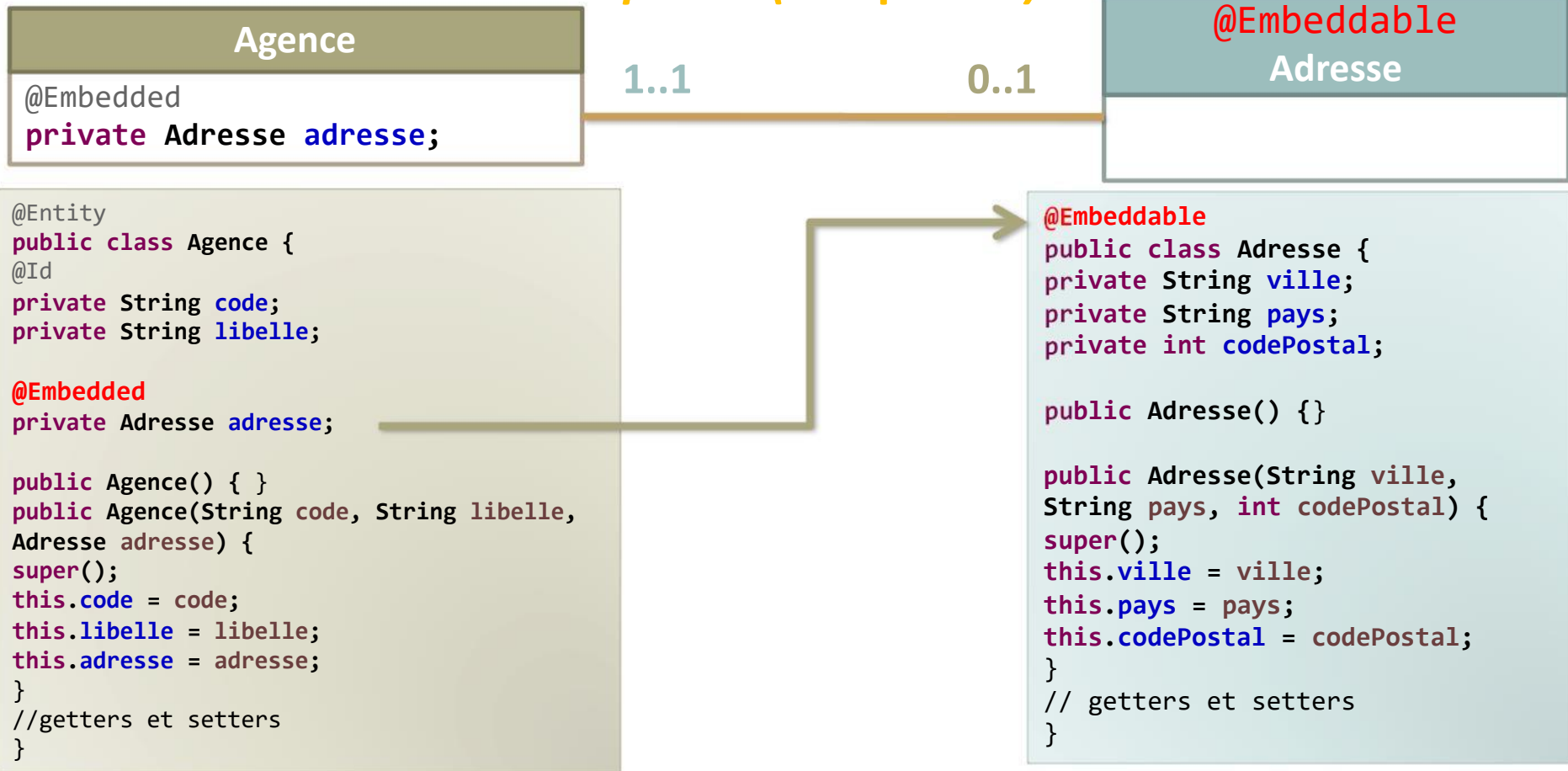
- Vous avez la possibilité de récupérer les entités associées:
  - soit immédiatement ("eager"),
  - soit à la demande ("lazy").
- Le paramètre fetch peut être positionné à FetchType.LAZY ou à FetchType.EAGER.

```
@Entity
public class Agence {
    @Id
    private String code;
    private String libelle;
    @ManyToMany(fetch=FetchType.EAGER)
    private Set<Client> clients = new
    HashSet<>();
    ...
}
```

```
@Entity
public class Agence {
    @Id
    private String code;
    private String libelle;
    @ManyToMany(fetch=FetchType.LAZY)
    private Set<Client> clients = new
    HashSet<>();
    ...
}
```

- **Les composants (component)**
  - **Une *entité*** existe par elle-même indépendamment de toute autre entité, et peut être rendue persistante par insertion dans la base de données, avec un identifiant propre.
- **Un *composant*** est un objet sans identifiant, qui ne peut être persistant que par rattachement à une entité.
  - La notion de composant résulte du constat qu'une ligne dans une base de données peut *parfois* être décomposée en plusieurs sous-ensemble dotés chacun d'une logique autonome. Cette décomposition mène à une granularité fine de la représentation objet, dans laquelle on associe *plusieurs* objets à *une* ligne de la table.
  - **@Embedded** nous indique que ce composant est embarqué.
  - **@Embeddable** nous indique que cette classe sera utilisée comme composant.

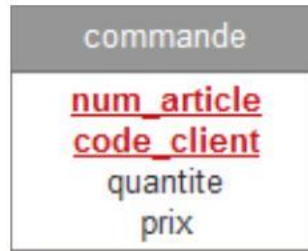
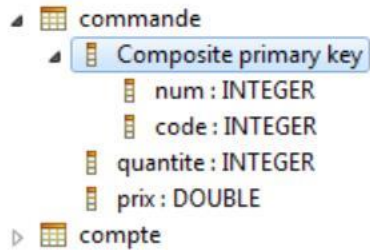
## Uncomposant (component)



- **Les clefs primaires composées**

- Les clefs primaires composées utilisent une classe embarquée comme représentation de la clef primaire,
- 3 possibilités:
  - **@Id** et **@Embeddable**: un seul attribut Id dans la classe entité
  - **@IdClass**: correspond à plusieurs attributs Id dans la classe entité
  - **@EmbeddedId** un seul attribut Id dans la classe entité
- Dans les trois cas, la clé doit être représentée par une classe Java:
  - les attributs correspondent aux composants de la clé
  - la classe doit être publique
    - la classe doit avoir un constructeur sans paramètre
    - la classe doit être sérialisable.
  - La classe doit redéfinir equals() et hashCode().

## Une Clé composée : @IdClass(ClassePK.class)



```
@Entity
@IdClass(CommandePK.class)
public class Commande {
```

```
@Id
private int num;
@Id
private int code;
```

```
private int quantite;
private double prix;
```

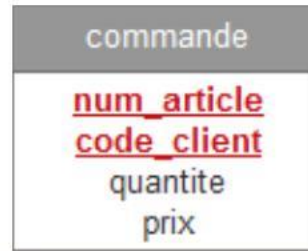
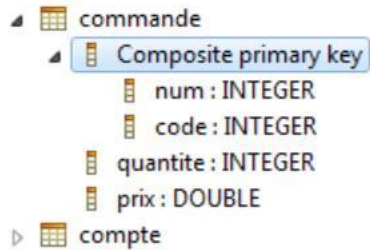
```
public Commande() {}
// les getters et les setters
}
```

```
public class CommandePK implements Serializable {

    private int num;
    private int code;

    public CommandePK() {}
    // les getters et les setters
    public int hashCode() {
        ...
    }
    public boolean equals(Object obj) {
        ...
    }
}
```

## Une Clé composée: @Id et @Embeddable



```
@Entity
public class Commande {

    @Id
    private CommandePK pk;

    private int quantite;
    private double prix;

    public Commande() {}
    // les getters et les setters

}
```

@Embeddable

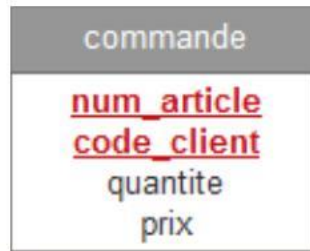
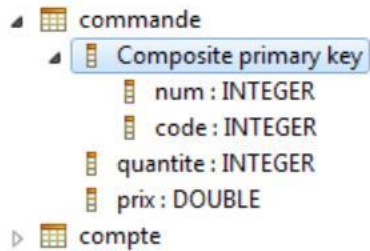
```
public class CommandePK implements Serializable {

    Private int num;
    Private int code;

    public CommandePK() {}
    // les getters et les setters
    public int hashCode() {
        ...
    }
    public boolean equals(Object obj) {
        ...
    }
}
```



## Une Clé composée : @EmbeddedId



```
@Entity
public class Commande {

    @EmbeddedId
    private CommandePK pk;

    private int quantite;
    private double prix;

    public Commande() {}
    // les getters et les setters

}
```

```
public class CommandePK implements Serializable {

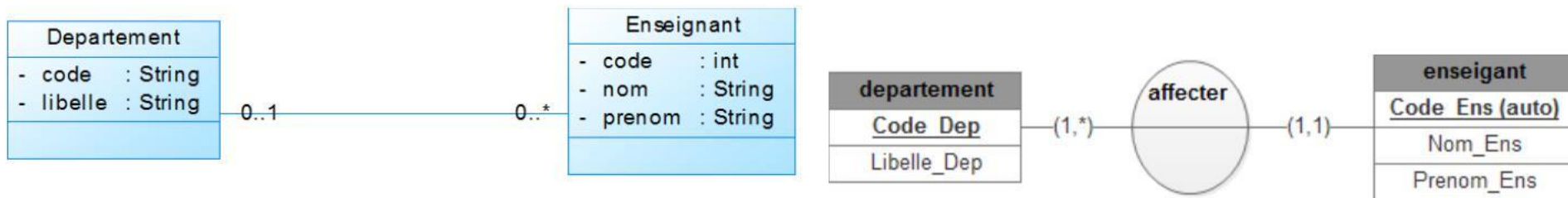
    private int num;
    private int code;

    public CommandePK() {}
    // les getters et les setters
    public int hashCode() {
        ...
    }
    public boolean equals(Object obj) {
        ...
    }
}
```

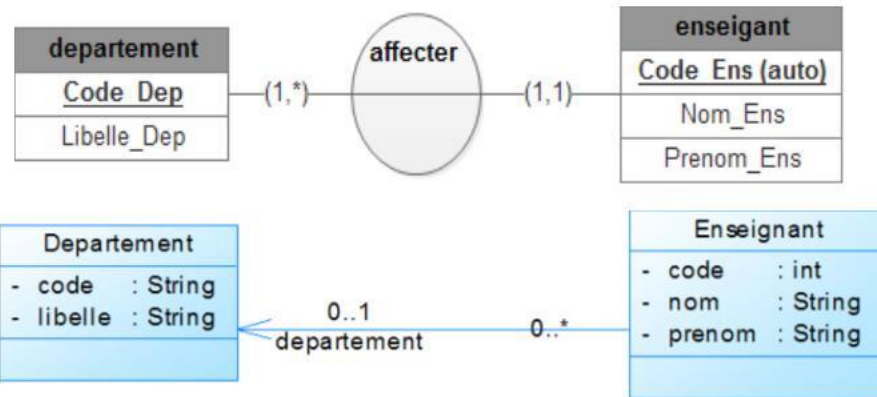
# Association un-à-plusieurs

# Associations 1-N et N-1

- les associations “un à plusieurs” et “plusieurs à un”
- Annotations @OneToMany et @ManyToOne
- Représentée par une clé étrangère dans la table qui correspond au côté propriétaire (obligatoirement le côté Many)
- Exemple : Enseignant-Département, cette association peut se représenter de 3 manières :
  - dans un Enseignant, on place un lien vers le Département (unidirectionnel); **@ManyToOne**
  - dans un Département , on place des liens vers les Enseignants qu’il lui sont affectés (unidirectionnel); **@OneToMany**
  - Ou bien, on représente les liens des deux côtés (bidirectionnel). **mappedBy**
- Généralement, dans une base relationnelle, une association représentée par une clé



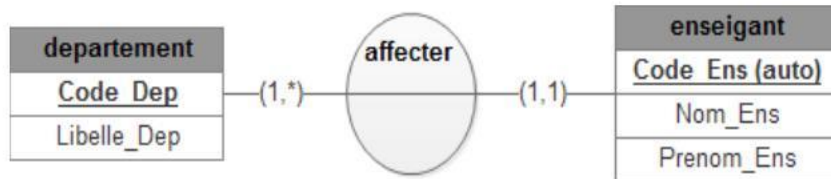
- *Association Unidirectionnelle.*
- => Enseignant est le maître de l'association
- Dans la classe **Enseignant**, **@ManyToOne** nous indique qu'un objet lié nommé **departement** (clé étrangère) est de la classe **Departement**.
- L'annotation **@JoinColumn** indique la clé étrangère dans la table **Enseignant** qui permet de rechercher le Département concerné.



```

@Entity
@Table(name="enseignant")
public class Enseignant {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="Code_Ens")
    private int code;
    @Column(name="Nom_Ens", length=30)
    private String nom;
    @Column(name="Prenom_Ens", length=30)
    private String prenom;
    @ManyToOne
    @JoinColumn(name="Code_Dep")
    private Departement departement;
    public Enseignant() {}
    public Departement getDepartement() {
        return departement;
    }
    public void setDepartement(Departement departement)
    {this.departement = departement;}
    // les getters et les setters
}
  
```

- Après l'exécution



Département			
Column	Type	Null	Primary
Code_Dep	varchar(6)	No	yes
Libelle_Dep	varchar(50)	Yes	

Enseignant			
Column	Type	Null	Primary
Code_Ens	int(11)	No	AUTO_INCREMENT
Nom_Ens	varchar(30)	Yes	
Prenom_Ens	varchar(30)	Yes	
Code_Dep	varchar(6)	Yes	FK

```

public class MappingTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Session session=HibernateUtil.getSessionFactory()
            .openSession();

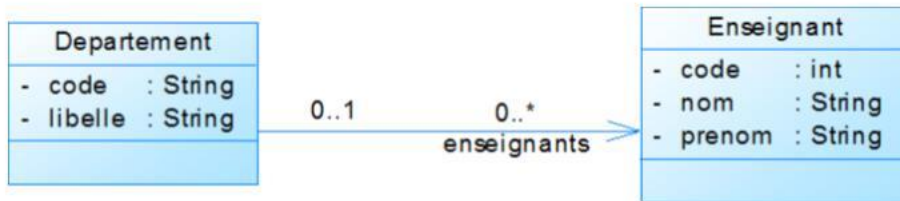
        Transaction tx=session.getTransaction();
        tx.begin();
        Enseignant ens=new Enseignant("Youness",
            "Boukouchi");

        session.save(ens);
        Departement dep=new Departement("GINFO",
            "Génie Informatique");
        ens.setDepartement(dep);
        session.save(dep);
        tx.commit();
    }
}
  
```

Code_Dep	Libelle_Dep
GINFO	Génie Informatique

Code_Ens	Nom_Ens	Prenom_Ens	Code_Dep
1	Youness	Boukouchi	<a href="#">GINFO</a>

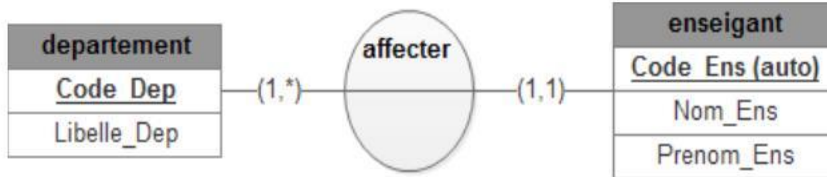
- *Association Unidirectionnelle.*
- => Département est le maître de l'association
- Dans la classe **Département**, **@OneToMany** nous indique que la collection des objets liés nommé **enseignants**, est de la classe **Enseignant**.
- L'annotation **@JoinColumn** indique la **clé étrangère** dans la table **Enseignant** qui permet de rechercher le Département concerné.
- On ajoute le getter et le setter de la collection.
- On ajoute la méthode **ajouterEnseignant** pour lier un objet Enseignant au département.



```

@Entity
@Table(name="departement")
public class Departement {
    @Id
    @Column(name="Code_Dep", length=6)
    private String code;
    @Column(name="Libelle_Dep", length=50)
    private String libelle;
    @OneToMany
    @JoinColumn(name="Code_Dep")
    Set<Enseignant> enseignants =new HashSet<>();
    public Set<Enseignant> getEnseignants() {
        return enseignants;
    }
    public void setEnseignants(Set<Enseignant>
    enseignants) {
        this.enseignants = enseignants;
    }
    public void ajouterEnseignant(Enseignant ens){
        this.enseignants.add(ens);
    }
    //Les constructeurs
    //les getters et les setters
}
  
```

- Après l'exécution



Département			
Column	Type	Null	Primary
Code_Dep	varchar(6)	No	yes
Libelle_Dep	varchar(50)	Yes	

Enseignant			
Column	Type	Null	Primary
Code_Ens	int(11)	No	AUTO_INCREMENT
Nom_Ens	varchar(30)	Yes	
Prenom_Ens	varchar(30)	Yes	
Code_Dep	varchar(6)	Yes	FK

```

public class MappingTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Session session=HibernateUtil.getSessionFactory()
            .openSession();
  
```

```

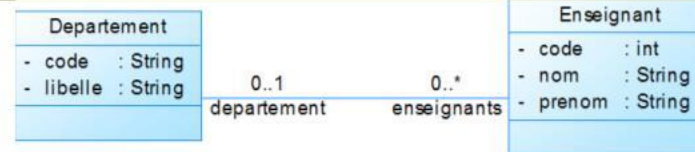
        Transaction tx=session.getTransaction();
        tx.begin();
        Enseignant ens=new Enseignant("Youness",
            "Boukouchi");
  
```

```

        session.save(ens);
        Departement dep=new Departement("GINFO",
            "Génie Informatique");
        dep.ajouterEnseignant(ens);
        session.save(dep);
        tx.commit();
    }
}
  
```



- *Association bidirectionnelle.*
- => Département et Enseignant sont les maîtres de l'association
- Dans la classe **Enseignant**, **@ManyToOne** nous indique qu'un objet lié nommé **departement** est de la classe **Département**.
- L'annotation **@JoinColumn** indique la **clé étrangère** dans la table **Enseignant** qui permet de rechercher le Département concerné.
- Dans la classe **Département**, **@OneToMany** nous indique que la collection des objets liés nommé **enseignants**, est de la classe **Enseignant**.
- L'attribut **mappedBy** nous indique l'attribut de mapping dans la classe **Enseignant**.

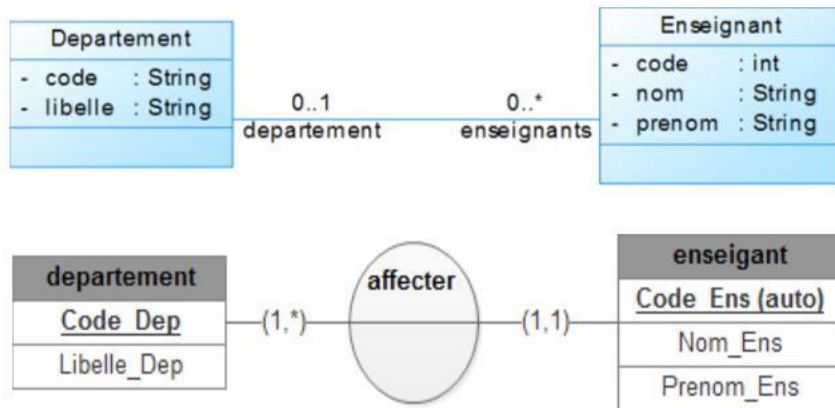


```
@Entity
@Table(name="departement")
public class Departement {
    // les champs
    @OneToMany(mappedBy="departement")
    Set<Enseignant> enseignants =new HashSet<>();
    public Set<Enseignant> getEnseignants() {
        return enseignants;
    }
    public void setEnseignants(Set<Enseignant>
        enseignants) {
        this.enseignants = enseignants;
    }
    public Departement() {}
} // les getters et les setters
```

```
@Entity
@Table(name="enseignant")
public class Enseignant {
    //les champs
    @ManyToOne
    @JoinColumn(name="Code_Dep")
    private Departement departement;
    public Departement getDepartement() {
        return departement;
    }
    public void setDepartement(Departement
        departement) {this.departement = departement;}
    public Enseignant() {}
} //les getters et les setters
```



- Après l'exécution



#### Console Hibernate:

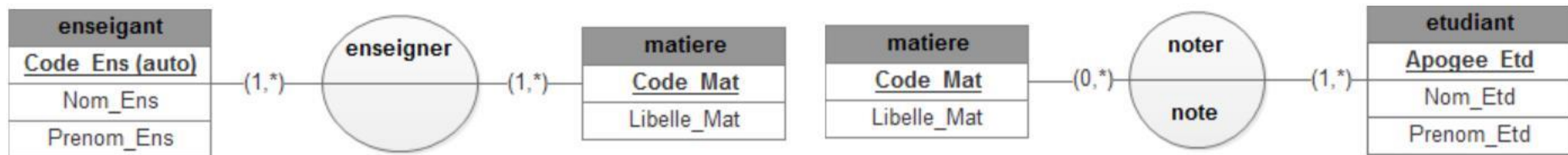
```
Hibernate: insert into enseignant (Code_Dep, Nom_Ens, ...
Hibernate: insert into enseignant (Code_Dep, Nom_Ens, ...
Hibernate: insert into departement (Libelle_Dep, Code_Dep) ...
Hibernate: update enseignant set Code_Dep=?, Nom_Ens=?,...
Hibernate: update enseignant set Code_Dep=?, Nom_Ens=?, ...
Hibernate: select departemen0_.Code_Dep as ...
Hibernate: select enseignant0_.Code_Dep as ...
size :2
```

```
public class MappingTest {
    public static void main(String[] args) {
        Session
        session=HibernateUtil.getSessionFactory().openSession();
        Transaction tx=session.getTransaction();
        tx.begin();
        Enseignant ens1=new Enseignant("Youness","Boukouchi");
        session.save(ens1);
        Enseignant ens2=new Enseignant("Mohammed","Ahmed");
        session.save(ens2);
        Departement dep=new Departement("GINFO","Génie
        Informatique");
        ens1.setDepartement(dep);
        ens2.setDepartement(dep);
        session.save(dep);
        tx.commit();
        session.refresh(dep);
        System.out.println("size :"+dep.getEnseignants().size());

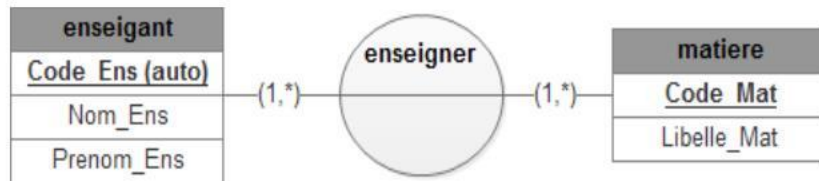
        session.close();
    }
}
```

# Association plusieurs-à-plusieurs

- Quand on représente une association plusieurs-plusieurs en relationnel, l'association devient une table dont la clé est la concaténation des clés des deux entités de l'association.
- La valeur par défaut du nom de la table association est la concaténation des 2 tables, séparées par \_
- Annotation @ManyToMany est représentée par une table association
- Deux cas se présentent :
  - l'association n'est pas porteuse d'aucun attribut.
  - l'association est porteuse des attributs (traitée comme @OneToMany dans les deux cotés de l'association).
- Dans notre exemple:
  - un enseignant enseigne **plusieurs** matières.
  - Une matière est enseignée par **plusieurs** enseignants.



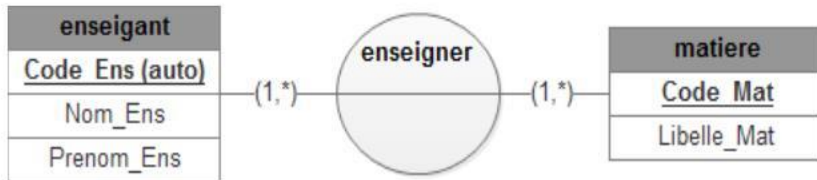
- *Association Unidirectionnelle.*
- => Matière est le maître de l'association
- Dans la classe **Matière**, **@ManyToMany** nous indique que la collection des objets liés nommé **enseignants**, est de la classe **Enseignant**.



```

@Entity
@Table(name="matiere")
public class Matiere {
    @Id
    @Column(name="Code_Mat", length=6)
    private String code;
    @Column(name="Libelle_Mat", length=100)
    private String libelle;
    @ManyToMany
    private Set<Enseignant> enseignants=new HashSet<>();
    public Set<Enseignant> getEnseignants() {
        return enseignants;
    }
    public void setEnseignants(Set<Enseignant> enseignants) {
        this.enseignants = enseignants;
    }
    public void ajouterEnseignant(Enseignant ens){
        this.enseignants.add(ens);}
    //les constructeurs
    //les getters et les setters
}
  
```

- Après l'exécution



Enseignant			
Column	Type	Null	Primary
Code_Ens	int(11)	No	PK
Nom_Ens	varchar(30)	Yes	NULL
Prenom_Ens	varchar(30)	Yes	NULL

matiere_enseignant			
Column	Type	Null	Primary
Matiere_Code_Mat	varchar(6)	No	FK
enseignants_Code_Ens	int(11)	No	FK

```

public class MappingTest {
    public static void main(String[] args) {
        Session
        session=HibernateUtil.getSessionFactory().openSession();
        Transaction tx=session.getTransaction();
        tx.begin();
        Enseignant ens1=new Enseignant("Youness","Boukouchi");
        session.save(ens1);
        Matiere mat1=new Matiere("JEE",
                                "Java Enterprise Edition");
        mat1.ajouterEnseignant(ens1);
        Matiere mat2=new Matiere("SOA",
                                "Architecture Orientée Services");
        mat2.ajouterEnseignant(ens1);
        session.save(mat1);
        session.save(mat2);
        tx.commit();
        System.out.println("size
        :"+mat1.getEnseignants().size());
        session.close();
    }
}
  
```

Matière			
Column	Type	Null	Primary
Code_Mat	varchar(6)	No	PK
Libelle_Mat	varchar(100)	Yes	NULL

- attribut **name** donne le nom de la table association : **enseigner**
- attribut **joinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté propriétaire de l'association: **Matiere**
- attribut **inverseJoinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté qui n'est pas propriétaire de l'association : **Enseignant**

enseigner			
Column	Type	Null	Primary
Code_Mat	varchar(6)	No	FK
Code_Ens	int(11)	No	FK

Console Hibernate:

```
Hibernate: insert into enseignant (Nom_Ens, Prenom_Ens) values (?, ?)
Hibernate: insert into matiere (Libelle_Mat, Code_Mat) values (?, ?)
Hibernate: insert into matiere (Libelle_Mat, Code_Mat) values (?, ?)
Hibernate: insert into enseigner (Code_Mat, Code_Ens) values (?, ?)
Hibernate: insert into enseigner (Code_Mat, Code_Ens) values (?, ?)
size :1
```

```
@Entity
@Table(name="matiere")
public class Matiere {
    @Id
    @Column(name="Code_Mat", length=6)
    private String code;
    @Column(name="Libelle_Mat", length=100)
    private String libelle;
    @ManyToMany
    @JoinTable(name="enseigner",
        joinColumns=@JoinColumn(name="Code_Mat"),
        inverseJoinColumns=@JoinColumn(name="Code_Ens"))
    private Set<Enseignant> enseignants=new HashSet<>();
    public Set<Enseignant> getEnseignants() {
        return enseignants;
    }
    public void setEnseignants(Set<Enseignant> enseignants) {
        this.enseignants = enseignants;
    }
    public void ajouterEnseignant(Enseignant ens){
        this.enseignants.add(ens);}
    //les constructeurs
    //les getters et les setters
}
```

- *Association Bidirectionnelle.*
- => Matière et Enseignant sont les propriétaires de l'association
- Dans la classe Matière, **@ManyToMany** nous indique que la collection des objets liés nommé enseignants, est de la classe Enseignant.
- Dans la classe Enseignant, **@ManyToMany** avec l'attribut **mappedBy** qui indique le mapping dans la classe Enseignant.

```

@Entity
@Table(name="enseignant")
public class Enseignant {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="Code_Ens")
    private int code;
    @Column(name="Nom_Ens", length=30)
    private String nom;
    @Column(name="Prenom_Ens", length=30)
    private String prenom;
    @ManyToMany(mappedBy="enseignants")
    private Set<Matiere> matieres=new HashSet<>();
    public Set<Matiere> getMatieres() {
        return matieres;}
    public void setMatieres(Set<Matiere> matieres) {
        this.matieres = matieres;}
    //les getters et les setters
}

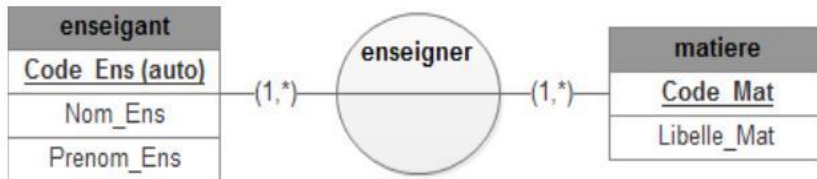
```

```

@Entity
@Table(name="matiere")
public class Matiere {
    @Id
    @Column(name="Code_Mat", length=6)
    private String code;
    @Column(name="Libelle_Mat", length=100)
    private String libelle;
    @ManyToMany
    @JoinTable(name="enseigner",
        joinColumns=@JoinColumn(name="Code_Mat"),
        inverseJoinColumns=@JoinColumn(name="Code_Ens"))
    private Set<Enseignant> enseignants=new HashSet<>();
    public Set<Enseignant> getEnseignants() {
        return enseignants;}
    public void setEnseignants(Set<Enseignant>
        enseignants) {
        this.enseignants = enseignants;}
    public void ajouterEnseignant(Enseignant ens){
        this.enseignants.add(ens);}
    //les getters et les setters
}

```

- Après l'exécution



```

public class MappingTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Session
        session=HibernateUtil.getSessionFactory().openSession();
        Transaction tx=session.getTransaction();
        tx.begin();
        Enseignant ens1=new Enseignant("Youness","Boukouchi");
        session.save(ens1);
        Matiere mat1=new Matiere("JEE","Java Entreprise
        Edition");
        mat1.ajouterEnseignant(ens1);
        Matiere mat2=new Matiere("SOA","Architecture Orientée
        Services");
        mat2.ajouterEnseignant(ens1);
        session.save(mat1);
        session.save(mat2);
        tx.commit();
        session.refresh(ens1);
        System.out.println("size :"+ens1.getMatiere().size());
        session.close();
        session.close();
    }
}
  
```

#### Console Hibernate:

```

Hibernate: insert into enseignant (Nom_Ens, Prenom_Ens) values (?, ?)
Hibernate: insert into matiere (Libelle_Mat, Code_Mat) values (?, ?)
Hibernate: insert into matiere (Libelle_Mat, Code_Mat) values (?, ?)
Hibernate: insert into enseigner (Code_Mat, Code_Ens) values (?, ?)
Hibernate: insert into enseigner (Code_Mat, Code_Ens) values (?, ?)
Hibernate: select enseignant0_.Code_Ens as ... where
enseignant0_.Code_Ens=?
Hibernate: select matieres0_.Code_Ens as ... from enseigner matieres0_ inner
join matiere matiere1_ on matieres0_.Code_Mat=matiere1_.Code_Mat
where matieres0_.Code_Ens=?
size :2
  
```

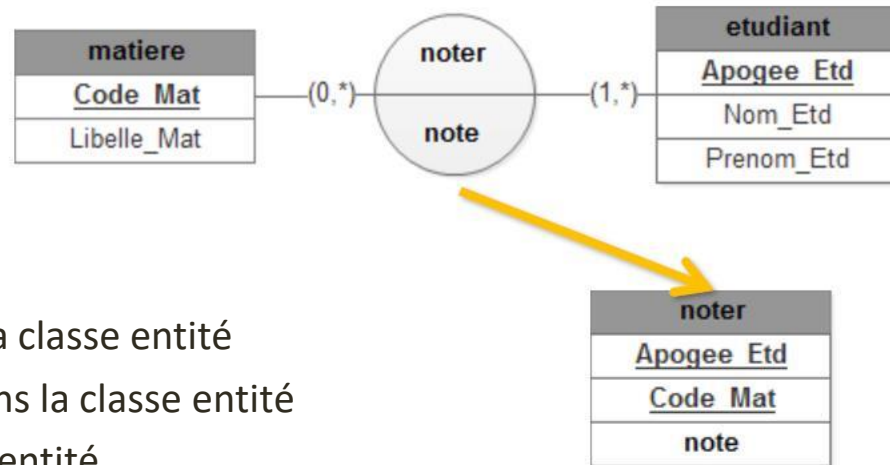


- Clé composé :

- Table existante avec clé multi-attributs
- Association N-M (Many-to-Many)

- 3 possibilités:

- **@Id** et **@Embeddable**: un seul attribut Id dans la classe entité
- **@IdClass**: correspond à plusieurs attributs Id dans la classe entité
- **@EmbeddedId** un seul attribut Id dans la classe entité
- Dans les trois cas, la clé doit être représentée par une classe Java:
- les attributs correspondent aux composants de la clé
- la classe doit être publique et avoir un constructeur sans paramètre
- la classe doit être sérializable et redéfinir equals et hashCode.



## Clé composite : @Id

- Correspond au cas où la classe entité comprend un seul attribut annoté **@Id**
- La classe clé primaire est annotée par **@Embeddable**

```
@Entity
@Table(name="noter")
public class Noter {
    @Id
    private Noter_PK pk;
    private double note;
    public Noter() {super();}
    public Noter(int apogee, String matiere, double
note) {this.pk = new Noter_PK(matiere, apogee);
this.note = note;}
    public Noter_PK getPk() {return pk;}
    public void setPk(Noter_PK pk) {this.pk = pk;}
    public double getNote() {return note;}
    public void setNote(double note) {this.note =
note;}
}
```


**@Embeddable**

```
public class Noter_PK implements
Serializable{
    private static final long serialVersionUID =
1L;
    @Column(name="Code_Mat")
    private String codeMat;
    @Column(name="Apogee_Mat")
    private int apogeeEtd;
    public Noter_PK(String codeMat, int
apogeeEtd) {this.codeMat = codeMat;
this.apogeeEtd = apogeeEtd;}
    public Noter_PK() {super();}
    public String getCodeMat() {
return codeMat;}
    public void setCodeMat(String codeMat) {
this.codeMat = codeMat;}
    public int getApogeeEtd() {
return apogeeEtd;}
    public void setApogeeEtd(int apogeeEtd) {
this.apogeeEtd = apogeeEtd;}
}
```

## Clé composite : @EmbeddedId

- Correspond au cas où la classe entité comprend un seul attribut annoté **@EmbeddedId**
- La classe clé primaire n'est pas annotée par @Embeddable

```
@Entity
@Table(name="noter")
public class Noter {
    @EmbeddedId
    private Noter_PK pk;
    private double note;
    public Noter() {super();}
    public Noter(int apogee, String matiere, double note) {this.pk = new Noter_PK(matiere, apogee); this.note = note;}
    public Noter_PK getPk() {return pk;}
    public void setPk(Noter_PK pk) {this.pk = pk;}
    public double getNote() {return note;}
    public void setNote(double note) {this.note = note;}
}
```




```
public class Noter_PK implements
Serializable{
    private static final long serialVersionUID =
    1L;
    @Column(name="Code_Mat")
    private String codeMat;
    @Column(name="Apogee_Mat")
    private int apogeeEtd;
    public Noter_PK(String codeMat, int
    apogeeEtd) {this.codeMat = codeMat;
    this.apogeeEtd = apogeeEtd;}
    public Noter_PK() {super();}
    public String getCodeMat() {
    return codeMat;}
    public void setCodeMat(String codeMat) {
    this.codeMat = codeMat;}
    public int getApogeeEtd() {
    return apogeeEtd;}
    public void setApogeeEtd(int apogeeEtd) {
    this.apogeeEtd = apogeeEtd;}
}
```

## Clé composite : @IdClass

- Correspond au cas où la classe entité comprend plusieurs attributs annotés par **@Id**
- La classe entité est annotée par **@IdClass** qui prend en paramètre le nom de la classe clé primaire
  - La classe clé primaire n'est pas annotée: ses attributs ont les mêmes noms et mêmes types que les attributs annotés @Id dans la classe entité

```
@Entity
@IdClass(Noter_PK.class)
@Table(name="noter")
public class Noter {
    @Id @Column(name="Code_Mat", length=6)
    private String codeMat;
    @Id @Column(name="Apogee_Etd")
    private int apogeeEtd;
    private double note;
    public Noter() {}
    //les getters et les setters
}
```



```
public class Noter_PK implements
Serializable{
    private static final long serialVersionUID =
    1L;
    private String codeMat;
    private int apogeeEtd;
    public Noter_PK(String codeMat, int
    apogeeEtd) {this.codeMat = codeMat;
    this.apogeeEtd = apogeeEtd;}
    public Noter_PK() {super();}
    public String getCodeMat() {
    return codeMat;}
    public void setCodeMat(String codeMat) {
    this.codeMat = codeMat;}
    public int getApogeeEtd() {
    return apogeeEtd;}
    public void setApogeeEtd(int apogeeEtd) {
    this.apogeeEtd = apogeeEtd;}
}
```

- Après l'exécution

noter
<u>Apogee Etd</u>
<u>Code Mat</u>
note

Apogee_Mat	Code_Mat	note
2300	JEE	16.75
5376	JEE	18

```
public class MappingTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Session
        session=HibernateUtil.getSessionFactory()
        .openSession();
        Transaction tx=session.getTransaction();
        tx.begin();
        Noter note1=new Noter(2300, "JEE", 16.75);
        session.save(note1);
        Noter note2=new Noter(5376, "JEE", 18.00);
        session.save(note2);
        tx.commit();
        session.close();
    }
}
```

#### Console Hibernate:

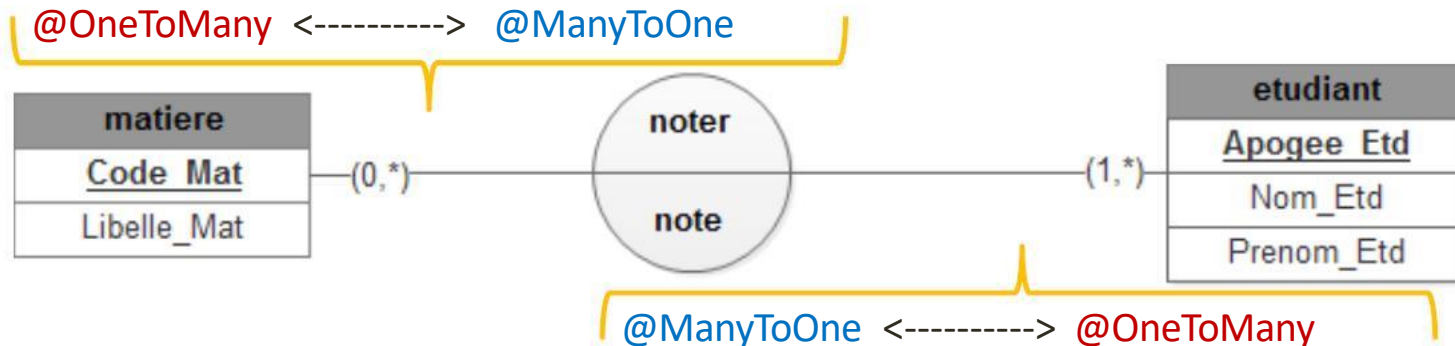
Hibernate: drop table if exists noter

Hibernate: create table noter (Apogee\_Mat integer not null, Code\_Mat varchar(255) not null, note double precision not null, primary key (Apogee\_Mat, Code\_Mat))

Hibernate: insert into noter (note, Apogee\_Mat, Code\_Mat) values (?, ?, ?)

Hibernate: insert into noter (note, Apogee\_Mat, Code\_Mat) values (?, ?, ?)

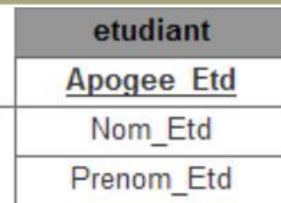
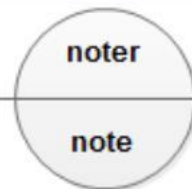
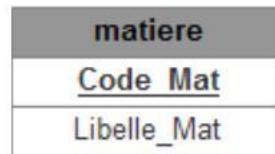
- *Association Bidirectionnelle.*
- Dans ce cas on a trois classes à mapper:
  - Matière avec la clé primaire Code\_Mat
  - Etudiant avec la clé primaire Apogee\_Etd
  - Noter avec deux clés étrangères (Code\_Mat, Apogee\_Etd)
- Dans cette situation on fait le mapping de deux associations :
  - Matière-Noter avec annotation @ManyToOne et @OneToMany
  - Etudiant-Noter avec annotation @ManyToOne et @OneToMany



```

@Entity
@Table(name="noter")
public class Noter {
@Id
private Noter_PK pk;
private double note;
public Noter() {}
public Noter(Etudiant etudiant, Matiere matiere,
double note) {
this.pk = new Noter_PK(matiere,etudiant);
this.note = note;}
public Noter_PK getPk() {return pk;}
public void setPk(Noter_PK pk) {this.pk = pk;}
public Etudiant getEtudiant() {
return this.pk.getEtudiant();}
public void setEtudiant(Etudiant etudiant) {
this.pk.setEtudiant(etudiant);}
public Matiere getMatiere() {
return this.pk.getMatiere();}
public void setMatiere(Matiere matiere) {
this.pk.setMatiere(matiere);}
public double getNote() {return note;}
public void setNote(double note) {
this.note = note;}
}

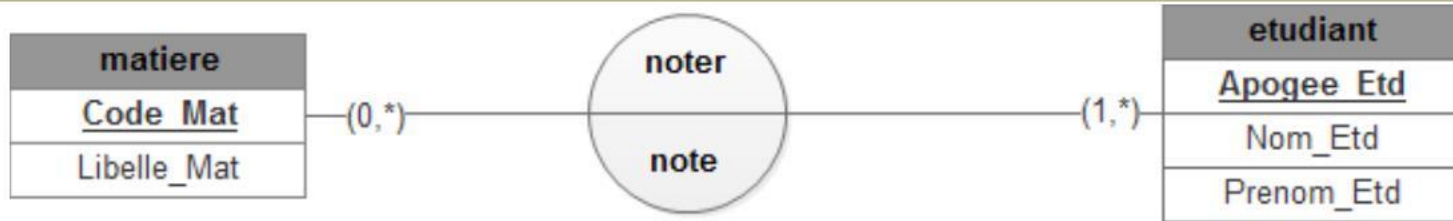
```



```

@Embeddable
public class Noter_PK implements Serializable{
private static final long serialVersionUID = 1L;
@ManyToOne
@JoinColumn(name="Code_Mat")
private Matiere matiere;
@ManyToOne
@JoinColumn(name="Apogee_Mat")
private Etudiant etudiant;
public Noter_PK() {}
public Noter_PK(Matiere matiere, Etudiant etudiant)
{this.matiere = matiere;
this.etudiant = etudiant;}
public Etudiant getEtudiant() {return etudiant;}
public void setEtudiant(Etudiant etudiant) {
this.etudiant = etudiant;}
public Matiere getMatiere() {return matiere;}
public void setMatiere(Matiere matiere) {
this.matiere = matiere;}
}

```



```

@Entity
@Table(name="matiere")
public class Matiere {
    @Id
    @Column(name="Code_Mat", length=6)
    private String code;
    @Column(name="Libelle_Mat", length=100)
    private String libelle;
    @OneToMany (mappedBy="pk.matiere")
    private Set<Noter> notes=new HashSet<>();
    public Set<Noter> getNotes() {
        return notes;
    }
    public void setNotes(Set<Noter> notes) {
        this.notes = notes;
    }
}
  
```

```

@Entity
@Table(name="etudiant")
public class Etudiant {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="Apogee_Etd")
    private int apogee;
    @Column(name="Nom_Etd", length=30)
    private String nom;
    @Column(name="Nom_Prenom", length=30)
    private String prenom;
    @OneToMany (mappedBy="pk.etudiant")
    private Set<Noter> notes=new HashSet<>();
    public Set<Noter> getNotes() {return notes;}
    public void setNotes(Set<Noter> notes) {
        this.notes = notes;}
    //les getters, les setters et les constructeurs
}
  
```



- Apres l'exécution

#### Console Hibernate:

```

Hibernate: create table etudiant (Apogee_Etd...,
Hibernate: create table matiere (Code_Mat ...
Hibernate: create table noter (note ..., primary key (Apogee_Mat,
Code_Mat))
Hibernate: alter table noter add constraint ... foreign key (Code_Mat)
references matiere (Code_Mat)
Hibernate: alter table noter add constraint ... foreign key (Apogee_Mat)
references etudiant (Apogee_Etd)
Hibernate: insert into etudiant (Nom_Etd, Nom_Prenom) values ...
Hibernate: insert into matiere (Libelle_Mat, Code_Mat) values ...
Hibernate: insert into matiere (Libelle_Mat, Code_Mat) values ...
Hibernate: insert into noter (note, Apogee_Mat, Code_Mat) values ...
Hibernate: insert into noter (note, Apogee_Mat, Code_Mat) values ...
Hibernate: select etudiant0_.Apogee_Etd ... where ...
Hibernate: select notes0_.Apogee_Mat ... inner join matiere ... where
notes0_.Apogee_Mat=?
Size : 2

```

```

public class MappingTest {
    public static void main(String[] args) {
        Session session=HibernateUtil.getSessionFactory().
            .openSession();

        Transaction tx=session.getTransaction();
        tx.begin();
        Etudiant etd1=new Etudiant(2020, "BOUKOUCHI",
            "Youness");
        session.save(etd1);
        Matiere mat1=new Matiere("JEE", "Java Entreprise
            Edition"); session.save(mat1);
        Matiere mat2=new Matiere("SOA", "Architecture
            orientée services");session.save(mat2);
        Noter note1=new Noter(etd1,mat1,18.75);
        session.save(note1);
        Noter note2=new Noter(etd1,mat2,17.25);
        session.save(note2);
        tx.commit();
        session.refresh(etd1);
        System.out.println("Size : "+
            etd1.getNotes().size());

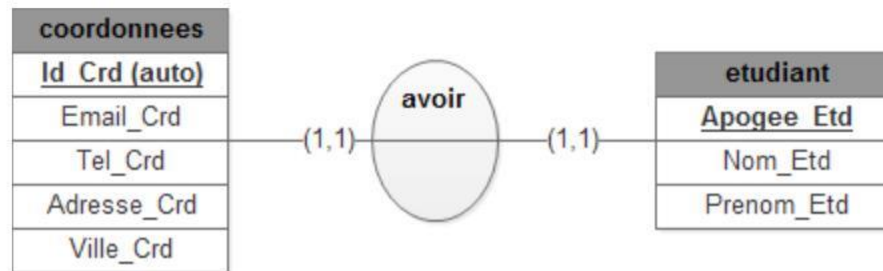
        session.close();
    }
}

```

# Association un-à-un

## • Annotation @OneToOne représente l'association 1-1

- Dans l'exemple ci-dessous, chaque « étudiant » ne peut avoir qu'une seule «Coordonnées» et une « coordonnées » ne peut appartenir qu'à un seul « étudiant ».
- Cette relation peut se traduire de plusieurs manières dans la base de données :
  - Une seule table qui contient les données de l'étudiant et ses coordonnées.
  - Deux tables, une pour les étudiants et une pour les coordonnées avec une clé primaire partagée
  - Deux tables, une pour les étudiants et une pour les coordonnées avec une seule clé étrangère
- Il y a plusieurs façons de traiter ce cas avec une ou deux tables dans la base de données et Hibernate :



- Dans la classe **Etudiant**, **@Embedded** nous indique que ce composant est embarqué (**@Embeddable**).
- L'annotation **@Embeddable** permet de préciser que la classe sera utilisée comme un component. Un tel élément n'a pas d'identifiant puisque celui utilisé sera celui de l'entité englobant.

etudiant
<u>Apogee Etd</u>
Nom_Etd
Prenom_Etd
Email_Crd
Tel_Crd
Adresse_Crd
Ville_Crd

```

@Entity
@Table(name="etudiant")
public class Etudiant {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="Apogee_Etd")
    private int apogee;
    @Column(name="Nom_Etd", length=30)
    private String nom;
    @Column(name="Nom_Prenom", length=30)
    private String prenom;
    @Embedded
    private Coordonnees coordonnees;
    //les constructeurs
    //les getters et les setters
}

```

```

@Embeddable
public class Coordonnees {
    @Column(name="adresse_Crd")
    private String adresse;
    @Column(name="ville_Crd")
    private String ville;
    @Column(name="email_Crd")
    private String email;
    @Column(name="tel_Crd")
    private String tel;
    public Coordonnees() {}
    public Coordonnees(String adresse,
        String ville, String tel, String
        email) {
        this.adresse = adresse;
        this.ville = ville;
        this.tel = tel;
        this.email = email;
    }
}

```



## L'annotation @Embedded & @Embeddable

- Apres l'exécution

etudiant			
Column	Type	Null	Default
Apogee_Etd	int(11)	No	PK Auto
adresse_Crd	varchar(255)	Yes	NULL
email_Crd	varchar(255)	Yes	NULL
tel_Crd	varchar(255)	Yes	NULL
ville_Crd	varchar(255)	Yes	NULL
Nom_Etd	varchar(30)	Yes	NULL
Nom_Prenom	varchar(30)	Yes	NULL

```
public class MappingTest {
    public static void main(String[] args) {
        Session
        session=HibernateUtil.getSessionFactory().openSession();
        Transaction tx=session.getTransaction();
        tx.begin();
        Coordonnees crd=new Coordonnees("ENSA-agadir",
                                         "Agadir", "0525008800",
                                         "y.boukouchi@gmail.com");
        Etudiant etd1=new Etudiant(2020, "BOUKOUCHI",
                                    "Youness", crd);

        session.save(etd1);

        tx.commit();}
    }
```

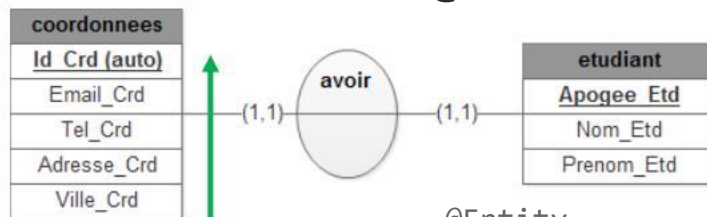
### Console Hibernate:

Hibernate: drop table if exists etudiant

Hibernate: create table etudiant (Apogee\_Etd integer not null auto\_increment, adresse\_Crd varchar(255), email\_Crd varchar(255), tel\_Crd varchar(255), ville\_Crd varchar(255), Nom\_Etd varchar(30), Nom\_Prenom varchar(30), primary key (Apogee\_Etd))

Hibernate: insert into etudiant (adresse\_Crd, email\_Crd, tel\_Crd, ville\_Crd, Nom\_Etd, Nom\_Prenom) values (?, ?, ?, ?, ?, ?)

- Deux table avec une seul clé étrangère
- C'est la table étudiant qui contient la clé étrangère (Id\_Crd)
- On va faire un mapping bidirectionnelle avec l'annotation @OneToOne et l'attribut mappedBy



```

@Entity
@Table(name="coordonnees")
public class Coordonnees {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="Id_Crd")
    private int id;
    @Column(name="adresse_Crd")
    private String adresse;
    @Column(name="ville_Crd")
    private String ville;
    @Column(name="email_Crd")
    private String email;
    @Column(name="tel_Crd")
    private String tel;
    @OneToOne(mappedBy="coordonnees")
    private Etudiant etudiant;
    //les constructeurs, les getters et les setters
}
  
```

```

@Entity
@Table(name="etudiant")
public class Etudiant {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="Apogee_Etd", length=5)
    private int apogee;
    @Column(name="Nom_Etd", length=30)
    private String nom;
    @Column(name="Nom_Prenom", length=30)
    private String prenom;
    @OneToOne
    @JoinColumn(name="Id_Crd")
    private Coordonnees coordonnees;
    //les constructeurs, les getters et les setters
}
  
```

- Après l'exécution

coordonnees			
Column	Type	Null	Default
Id_Crd	int(11)	No	PK
adresse_Crd	varchar(255)	Yes	NULL
email_Crd	varchar(255)	Yes	NULL
tel_Crd	varchar(255)	Yes	NULL
ville_Crd	varchar(255)	Yes	NULL

```

public class MappingTest {
    public static void main(String[] args) {
        Session session=HibernateUtil.getSessionFactory()
                                .openSession();
        Transaction tx=session.getTransaction(); tx.begin();
        Coordonnees crd=new Coordonnees("ENSA-agadir", "Agadir",
                                "0525008800", "y.boukouchi@gmail.com");
        Etudiant etd1=new Etudiant(2020, "BOUKOUCHI",
                                "Youness", crd);
        session.save(crd);session.save(etd1);
        tx.commit();} }

```

etudiant			
Column	Type	Null	Default
Apogee_Etd	int(11)	No	PK
Nom_Etd	varchar(30)	Yes	NULL
Nom_Prenom	varchar(30)	Yes	NULL
Id_Crd	int(11)	Yes	FK

Console Hibernate:

Hibernate: create table coordonnees (Id\_Crd integer not null auto\_increment, ..., primary key (Id\_Crd))

Hibernate: create table etudiant (Apogee\_Etd integer not null auto\_increment, ..., primary key (Apogee\_Etd))

Hibernate: alter table etudiant add constraint ... foreign key (Id\_Crd) references coordonnees (Id\_Crd)

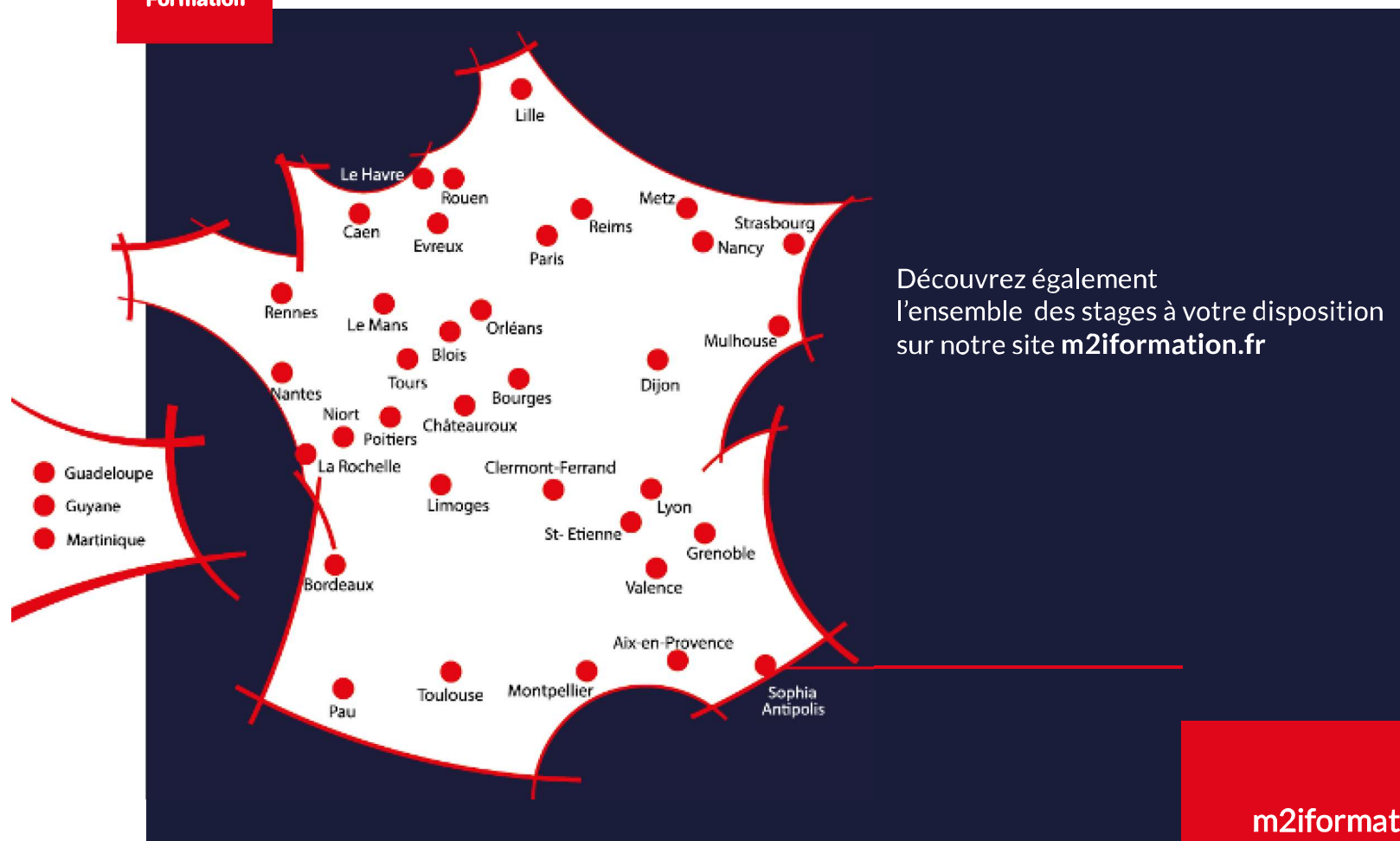
Hibernate: insert into coordonnees (adresse\_Crd, email\_Crd, tel\_Crd, ville\_Crd) values (?, ?, ?, ?)

Hibernate: insert into etudiant (Id\_Crd, Nom\_Etd, Nom\_Prenom) values (?, ?, ?)

THE END

---





Découvrez également  
l'ensemble des stages à votre disposition  
sur notre site [m2information.fr](https://m2information.fr)

[m2information.fr](https://m2information.fr)

