



**École Doctorale en Sciences Economique, Gestion et Informatique**  
**Faculté des Sciences Économiques et de Gestion de Sfax**  
**Département Informatique**

## **Auditoire**

**M**astère Professionnel Audit et Sécurité Informatique

# **Architectures des Systèmes d'Information (JavaEE)**

**Enseignante**

**AÏDA KHEMAKHEM**

**Année Universitaire**

**2019 – 2020**

# Chapitre 1 :

## JPA - EJB

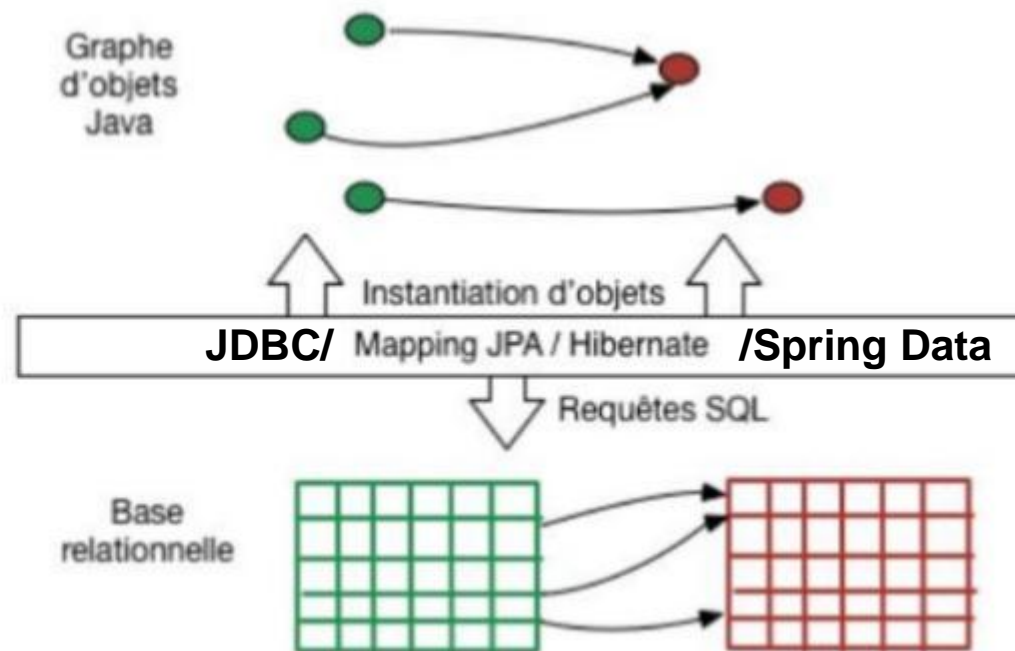
- I. Mapping Objet Relation
- II. Java Data Base Connectivity : JDBC
- III. Java Persistence Annotation : JPA
- IV. Hibernate
- V. Composants Enterprise Java Beans : EJB
- VI. Entity EJB

# I. Mapping Objet Relationnel des entités

**Mapping Objet/Relationnel** = correspondance entre le modèle de données relationnel et le modèle objet.

Un outil de mapping O/R doit proposer un certain nombre de fonctionnalités parmi lesquelles :

- 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.
- Proposer un langage de requêtes indépendant du SGBD cible et assurer une traduction en SQL natif.
- Supporter différentes formes d'identifiants générés automatiquement par les bases de données.
- Fonctionnalités pour améliorer les performances (cache, lazy loading, ...).
- Gestion des accès concurrents (verrou, dead lock, ...).
- Permettre l'héritage
- Support des transactions.

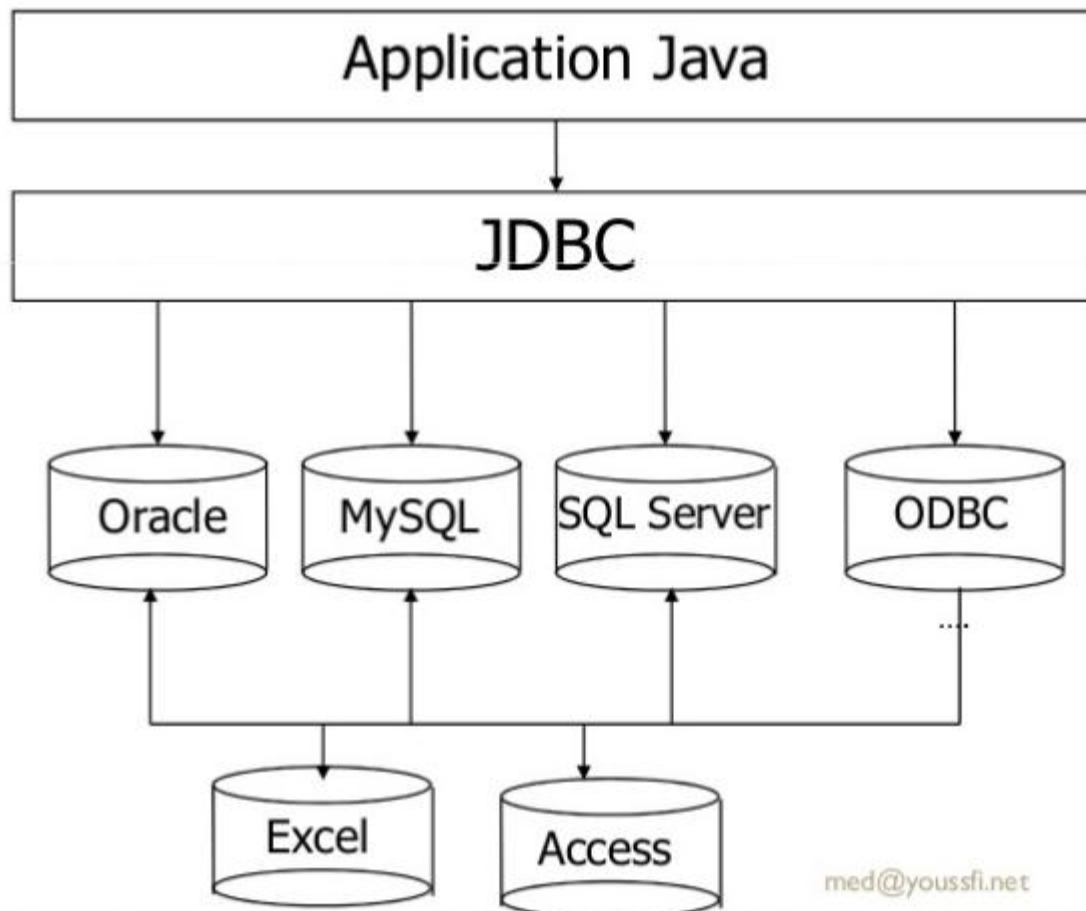


## II. Java Data Base Connectivity : JDBC

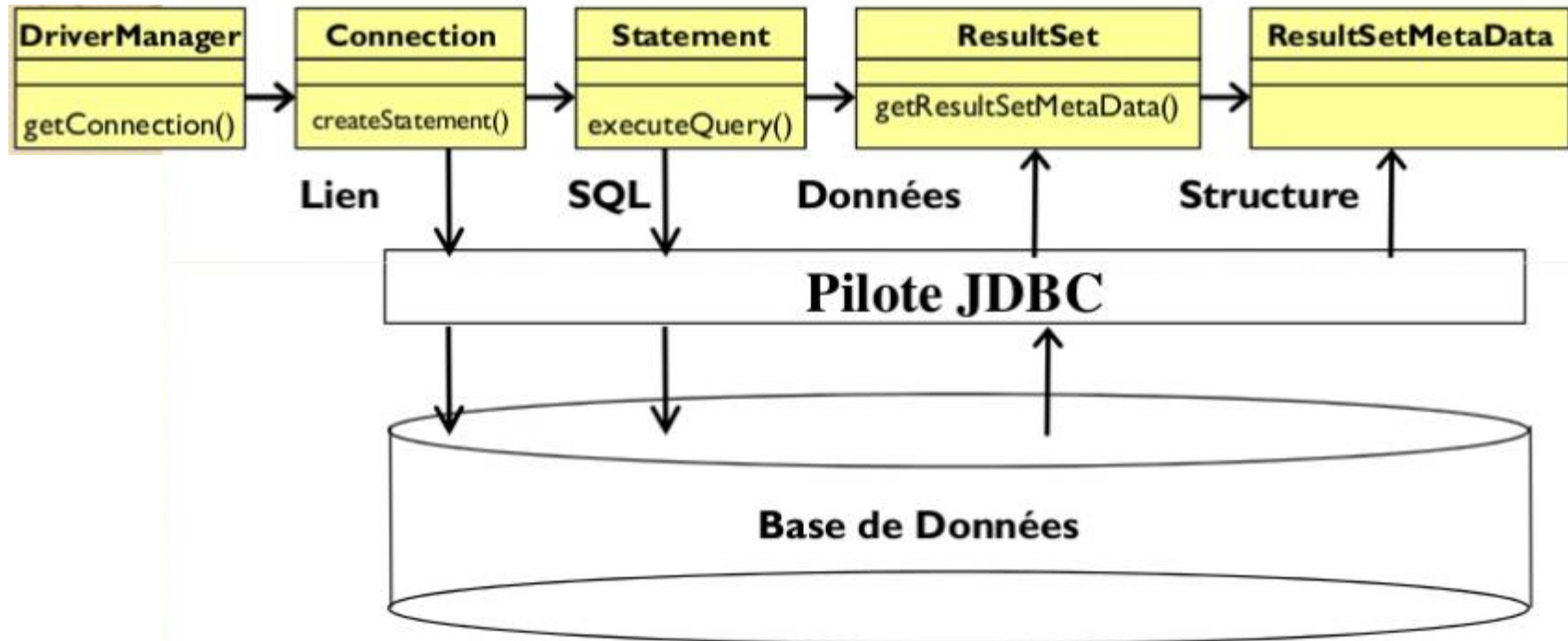
- JDBC (*Java Data Base Connectivity*) est l'**API** de base pour l'accès à des **bases de données relationnelles** avec le langage SQL, depuis un programme en Java
- Il est fourni par le paquetage **java.sql** qui contient un grand nombre d'**interfaces** et quelques classes
- JDBC **ne fournit pas** les classes qui implémentent les interfaces
- L'API JDBC est presque totalement **indépendante** des SGBDRs

# JDBC

- **JDBC** (Java Data Base Connectivity) : Pilotes java pour permettre l'accès aux bases de données.
- Chaque SGBD possède ses propres pilotes JDBC.



# JDBC

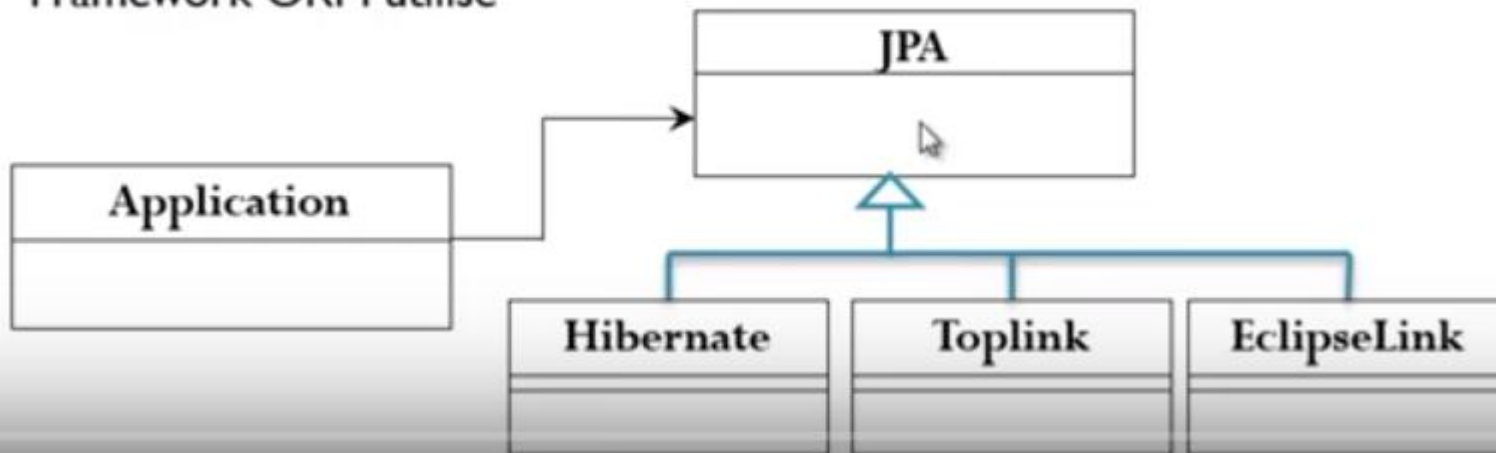


## II. Java Persistence Annotation : JPA

- JPA (Java Persistence Annotation) est une couche d'abstraction de la couche JDBC
- Elle permet notamment de faire du Mapping Relationnel-Objet (ORM, Object-Relationnal Mapping en anglais) qui consiste à modéliser la base de données sous forme d'objets pour une manipulation plus simple à travers le code Java (requêtes pré-écrites, gestion des liens entre les tables,...).
- Généralement
  - la couche métier contient une classe (entité) par table,
  - et la couche dao contient une classe qui regroupe la manipulation d'une table, en spécifiant les fonctions de base (ajout, recherche...) et les gestionnaires d'exceptions.

## II.1. les implémentations de JPA

- JPA est une spécification créée par Sun pour standardiser le mapping Objet relationnel.
- JPA définit un ensemble d'interfaces, de classes abstraites et d'annotations qui permettent la description du mapping objet relationnel
- Il existe plusieurs implémentation de JPA:
  - Hibernate
  - Toplink
  - IBatis
  - EclipseLink
- L'utilisation de JPA permet à votre application d'être indépendante du Framework ORM utilisé





## II.2. Persistence via JPA

- JPA 2 propose un modèle standard de persistance à l'aide des Entity beans
- Le serveur d'application fournit un Framework implémentant la spécification JPA pour assurer le mapping objet relation, comme Hibernate, Toplink, EclipseLink,...
- Ce framework assureront la persistance des entiy et devront être compatibles avec la norme JPA 2
  - JavaEE 6 repose à tous les niveaux sur de «l'injection de code» via des annotations @ de code
  - Souvent on ne fera pas de « new », les variables seront créées/initialisées par injection de code

## II.3. Les moyens de mapping

- Il existe deux moyens pour mapper les entités :
  - Créer des fichiers XML de mapping : Hibernate
  - Utiliser les annotations JPA (@Table, @Column...)
- La création des fichiers XML de mapping a l'avantage de séparer le code java du mapping objet relationnel
- L'utilisation des annotations JPA laisse votre code indépendant de Hibernate.

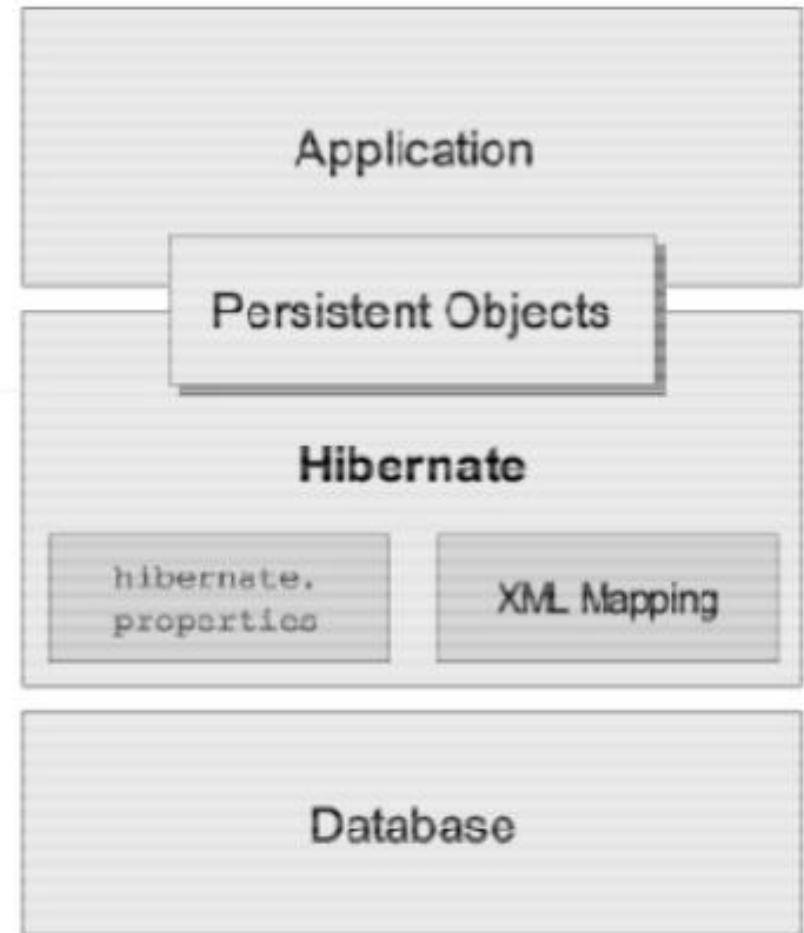
**Dans cette formation, nous allons utiliser les annotations JPA**

### III. Hibernate : présentation

- Hibernate s'occupe du transfert des objets Java dans les tables de la base de données
- En plus, il permet de requêter les données et propose des moyens de les récupérer.
- Il peut donc réduire de manière significative le temps de développement qui aurait été autrement perdu dans une manipulation manuelle des données via SQL et JDBC

# L'architecture d'Hibernate

- Hibernate permet d'assurer la persistance des objets de l'application dans un entrepôt de données.
- Cet entrepôt de données est dans la majorité des cas une base de données relationnelle, mais il peut être un fichier XML.
- Le mapping des objets est effectuée par Hibernate en se basant sur des fichiers de configuration en format texte ou souvent XML.



## IV. Composants Business (EJB)

### ■ EJB : Enterprise Java Beans

- “Written in the Java programming language, an Enterprise Bean is a server-side component that encapsulates the business logic of an application.”  
[ORACLE]
- Un composant autonome du côté serveur qui encapsule la logique métier de l'application
  - on se focalise sur la logique applicative (fonctionnels)
  - les services systèmes (non-fonctionnels) sont fournis par le conteneur
  - la logique de présentation est du ressort du client

## IV.1. Quand on utilise les EJB?

On choisit de développer une application avec les EJBs si :

- L'application doit être évolutive. Afin de gérer un nombre important d'utilisateurs, on doit distribuer les composants de l'application dans de multiples machines. Par conséquent, le Bean Entreprise peut s'exécuter sur différentes machines, tout en gardant une transparence par rapport à l'utilisateur final.
- Les transactions doivent assurer l'intégrité des données. Les Beans Entreprises supportent les transactions, qui est un mécanisme qui gère les accès concurrents aux objets partagés.
- L'application peut avoir une variété de clients. Avec peu de lignes de code, un client distant peut facilement localiser les Beans Entreprises. Les clients peuvent être légers, variés et nombreux.

## IV.2. Objectifs de EJB

- Fournir un modèle standard pour la construction d'applications distribuées en Java
  - Développement (Ing./Prod.)
  - Déploiement
  - Exécution (conteneur)
- Simplifier l'écriture de composants serveurs
- Portabilité (Java, inter-conteneurs)

## IV.3. Les types de EJB

- **Entity** : représente un composant métier qui est enregistré d'une manière permanente (Modèle)
- **Session** : assure une tâche au client
- **Message-Driven** : se base sur la notification par des messages asynchrones



## IV.4. Les annotations selon type de l'EJB

### ■ Les annotations :

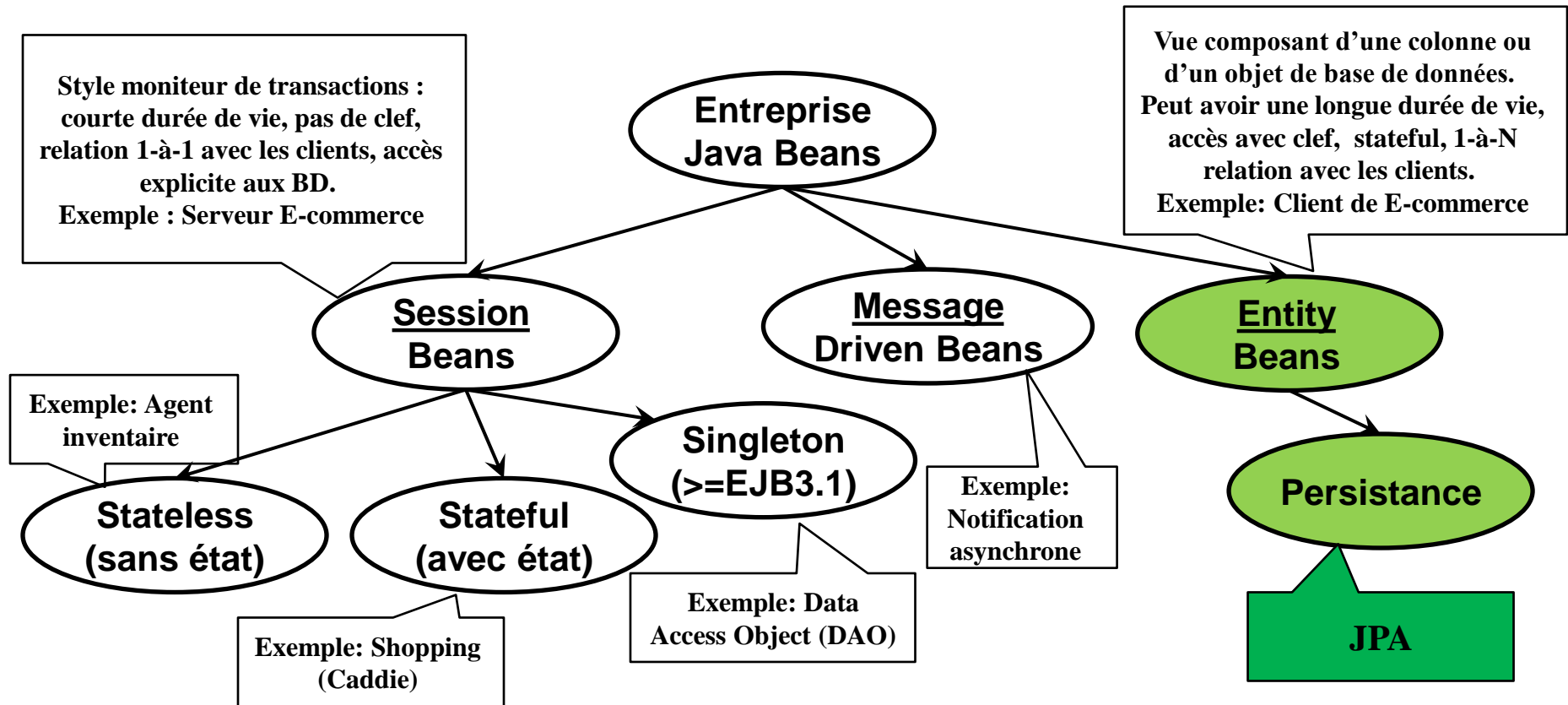
- Les annotations obligatoires : Spécification du type du bean
- Toutes les annotations ont une valeur par défaut (Valeur typique d'utilisation)
- Utilisation du descripteur de déploiement pour des besoins très particuliers

### ■ EntityBean: **@Entity**

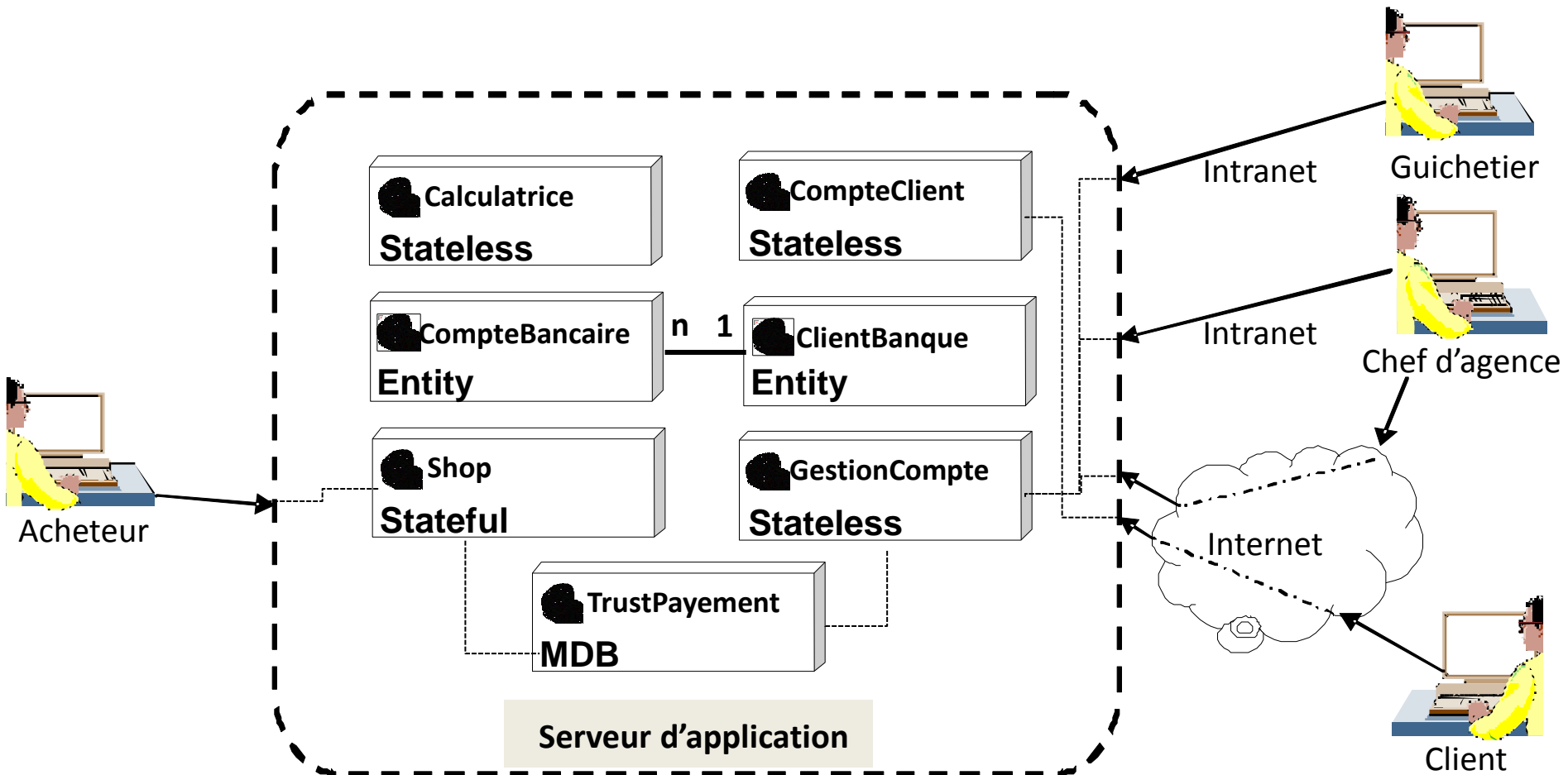
### ■ SessionBean : @Stateless, @Stateful, @Singleton

### ■ MessageDrivenBean : @MessageDriven

## IV.5. Taxonomie des composants EJB3.2



## IV.6. Application de gestion bancaire



# V. Entity EJB : Définition

- Représentation de données manipulées par l'application :
  - Enregistre les données dans un SGBD (ou tout autre support accessible en JDBC)
  - Assure une correspondance objet–tuple relationnel (*mapping O/R*)
  - Offre la possibilité de définir des clés, des relations, des recherches
- **Avantage :**
  - Manipulation d'objets Java plutôt que de requêtes SQL
- Mis en œuvre :
  - Annotations (@)
  - JPA (Java Persistence API)

# V. 1. Les annotations au niveau de la classe

- **@Entity** : indique que le bean (la classe) ainsi annoté est un *Entity Bean* (EB). Le bean sera manipulé et persisté par JPA.
- L'utilisation de @Entity **impose** que le bean possède
  - un identifiant (voir @Id),
  - un constructeur sans argument et tous les accesseurs (getters et setters)
- **Chaque classe de l'EB est mise en correspondance avec une table:**
  - **par défaut**, la table porte le nom de la classe
  - sauf si on spécifie le nom à travers l'annotation **@Table(name="...")**
- **@Table** : cette annotation s'utilise au niveau de la classe
  - Elle possède l'attribut "name" permettant de spécifier le nom de la table en BD
  - On utilise @Table lorsque le nom de la table est différent de celui du bean

```
@Entity
@Table (name="TMusicien")
public class Musicien implements Serializable
{ @Id
  private Long id;
  private String nomPrenom;...}
```



## V.2. Les annotations de l'identifiant

- **@Id** : C'est la clé primaire : indique que le champ ainsi annoté est l'identifiant de l'entité
- **@GeneratedValue** : indique que la valeur du champ ainsi annoté est générée automatiquement par la base de données
  - GenerationType.AUTO : les numéros de séquence sont choisis automatiquement
  - GenerationType.SEQUENCE : un générateur de numéros de séquence est à fournir

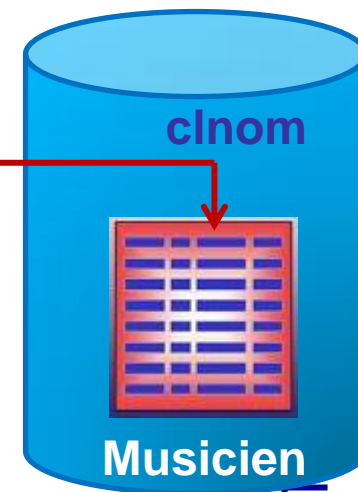
**Remarque** : Si l'identifiant est généré automatiquement, il faut avoir un constructeur sans identifiant

```
@Entity
public class Musicien implements Serializable
{ @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String nomPrenom;
  public Musicien(String nom) {this.nomPrenom = nom;}
  ...}
```

## V.3. Les annotations au niveau des attributs

- Nom de la colonne
  - par défaut, la colonne porte le nom du **l'attribut**
  - sauf si on spécifie le nom à travers l'annotation **@Column**(name="...")
- Types supportés: Types primitifs (String, Date, etc) et les types Class correspondants
- **@Column** : permet de faire quelques précisions sur le champ ainsi annoté. Cette annotation possède :
  - L'attribut "**name**" permettant de spécifier le nom du champ en base, si celui-ci diffère de celui du champ dans le bean.
  - L'attribut "**nullable**" spécifie si le champ peut être nul.
  - L'attribut "**length**" spécifie la taille du champ.

```
@Entity
public class Musicien implements Serializable
{...
@Column (name="clnom", length=100, nullable=false)
private String nomPrenom;...}
```



## V.3. Les annotations au niveau des attributs (suite)

- **@Transient** : attribut transitoire (non persistant) :  
`@javax.persistence.Transient`. Pour ne pas tenir compte de ce champs lors du mapping
- **@Enumerated** : indique que le champ ainsi annoté est un enum Java, dont la représentation en base pourra être un numérique ou un string.
- **@Temporal** : indique que le champ ainsi annoté est une date. Cette annotation possède des attributs pour préciser le format à utiliser



## VI. Mise en relation d'entités

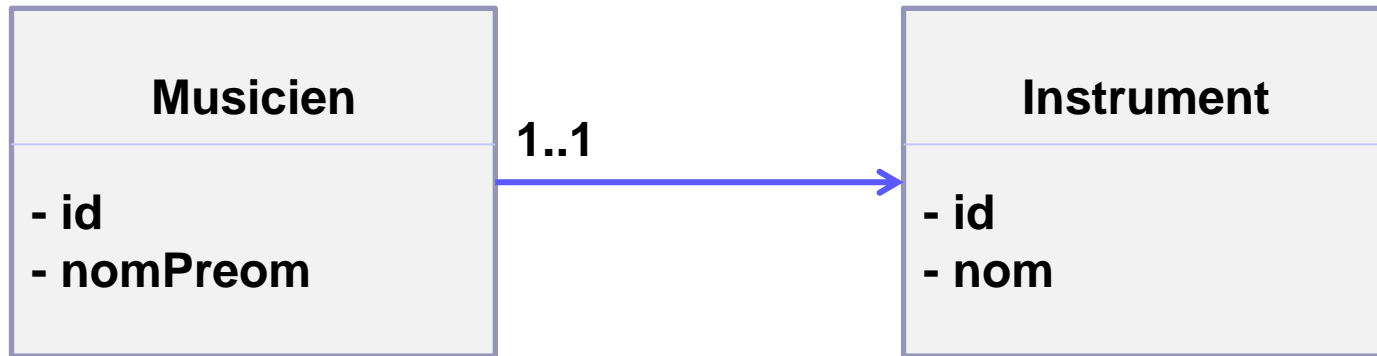
- Comme en SQL, on peut définir quatre types de relations entre les entités JPA :
  - relation **1:1** : annotée par **@OneToOne**
  - relation **n:1** : annotée par **@ManyToOne**
  - relation **1:p** : annotée par **@OneToMany**
  - relation **n:p** : annotée par **@ManyToMany**
- Bien qu'exprimées entre des classes Java, les relations sont définies entre des entités JPA. Il n'est donc pas légal d'annoter une relation qui pointerait vers une classe qui ne serait pas une entité.
- Cette annotation possède :
  - L'attribut "**mappedBy**" permettant de spécifier le nom du champ en base, si celui-ci diffère de celui du champ dans le bean.
  - L'attribut "**fetch**" permet de spécifier la méthode de chargement
    - FetchType.EAGER indique que la relation doit être chargée en même temps que l'entité qui la porte.
    - FetchType.LAZY : indique que la relation doit être chargée à la demande

## VI.1. Relation unidirectionnelle et bidirectionnelle

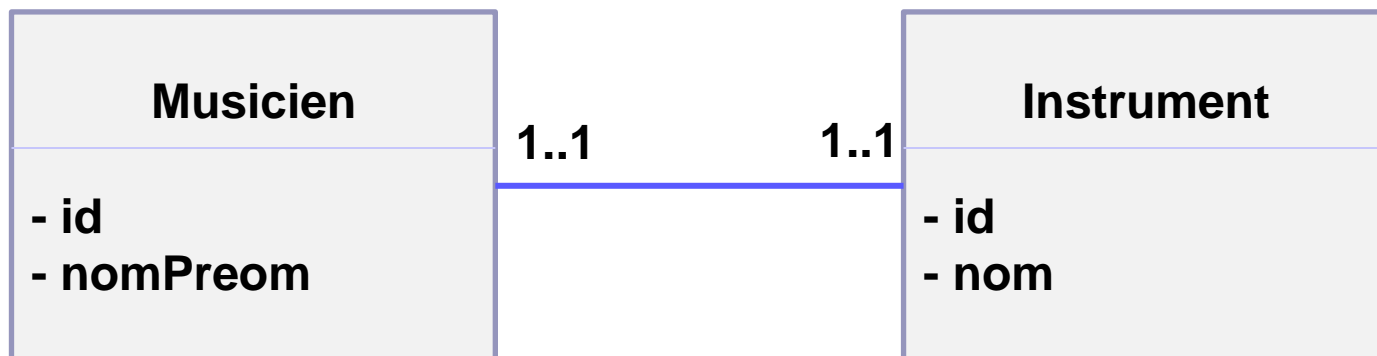
- Les relations entre entités, telles que définies en JPA peuvent être unidirectionnelles ou bidirectionnelles.
- Dans ce second cas, l'une des deux entités doit être **maître** et l'autre **esclave**.
- Dans le cas des relations 1:1 et n:p, on peut choisir le côté **maître** comme on le souhaite.
- Dans le cas des relations 1:p et n:1, l'entité du côté 1 est l'entité **esclave**.

## VI.2. Relation 1:1

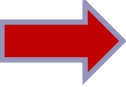
### ■ Unidirectionnelle



### ■ Bidirectionnelle



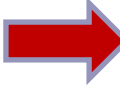
## a. Unidirectionnelle



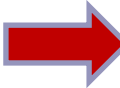
```
@Entity
public class Musicien implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nomPrenom
@OneToOne
private Instrument instrument;}
```

```
@Entity
public class Instrument implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nom
// reste de la classe
}
```

## b. Bidirectionnelle



```
@Entity
public class Musicien implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nomPrenom
@OneToOne
private Instrument instrument;}
```

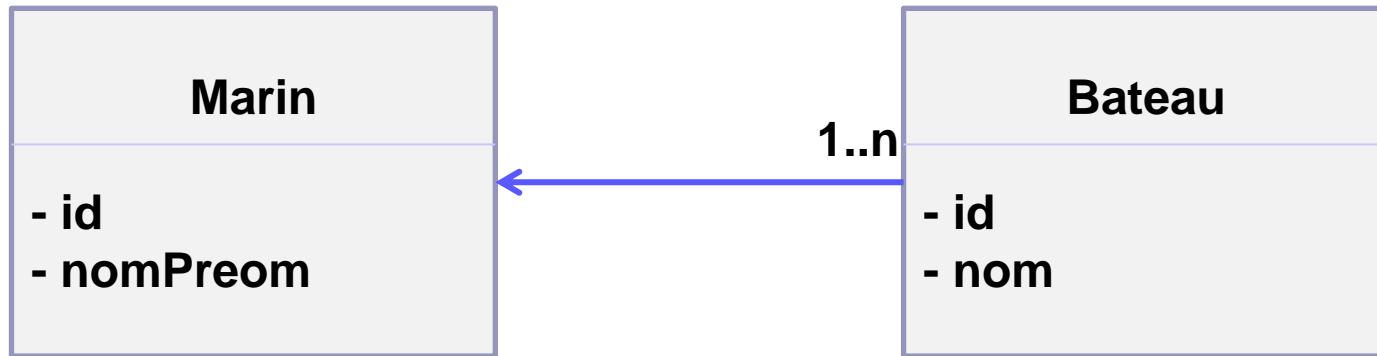


```
@Entity
public class Instrument implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nom
@OneToOne (mappedBy="instrument")
private Musicien musicien;
// reste de la classe
}
```

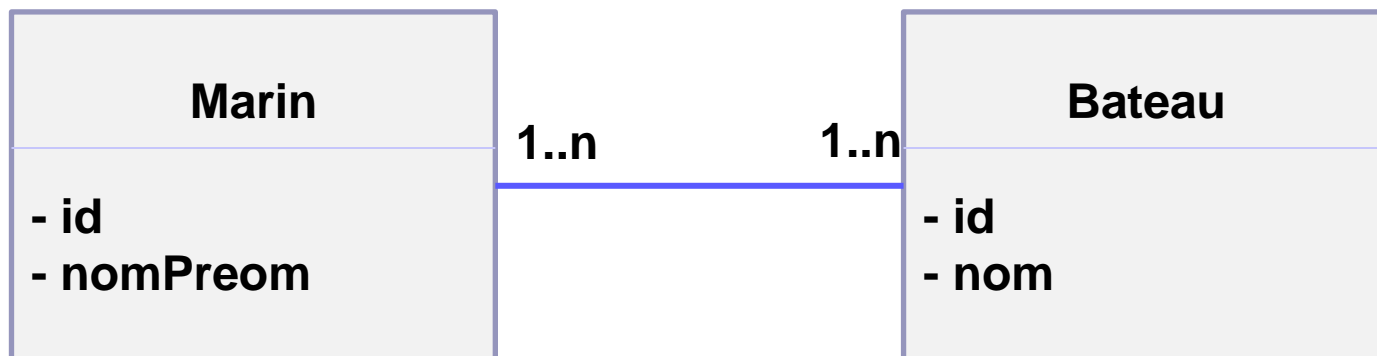
Référence la  
relation dans  
la classe  
Musicien

## VI.2. Relation 1:n

### ■ Unidirectionnelle

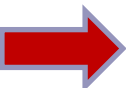


### ■ Bidirectionnelle



## a. Unidirectionnelle

```
@Entity
public class Marin implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nomPrenom
}
```



```
@Entity
public class Bateau implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nom
@OneToMany
private Collection<Marin> marins ;
// reste de la classe
}
```

## b. Bidirectionnelle

```
@Entity
public class Marin implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nomPrenom
→ @ManyToOne
private Bateau bateau;}
```

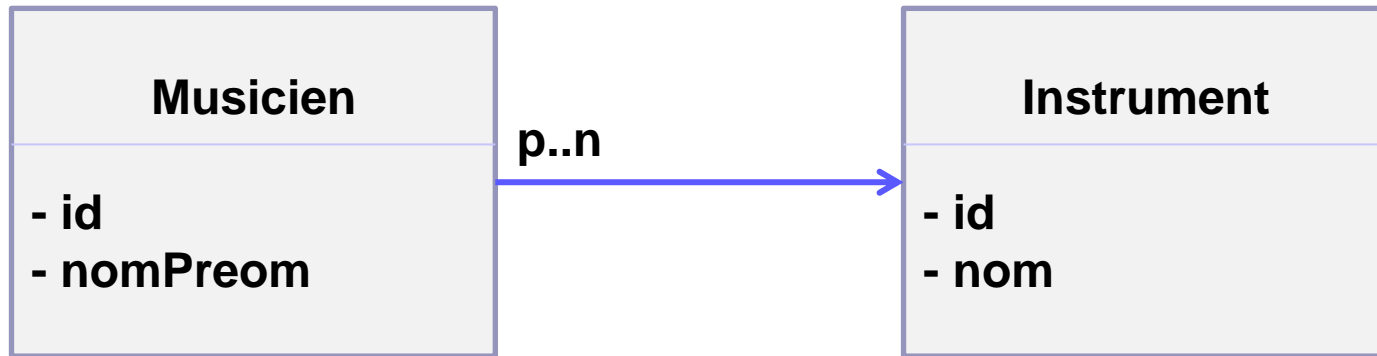
```
@Entity
public class Bateau implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nom
→ @OneToMany (mappedBy="bateau", fetch=FetchType.LAZY)
private Collection<Marin> marins;
// reste de la classe
}
```

Référence la  
relation dans  
la classe  
Marin

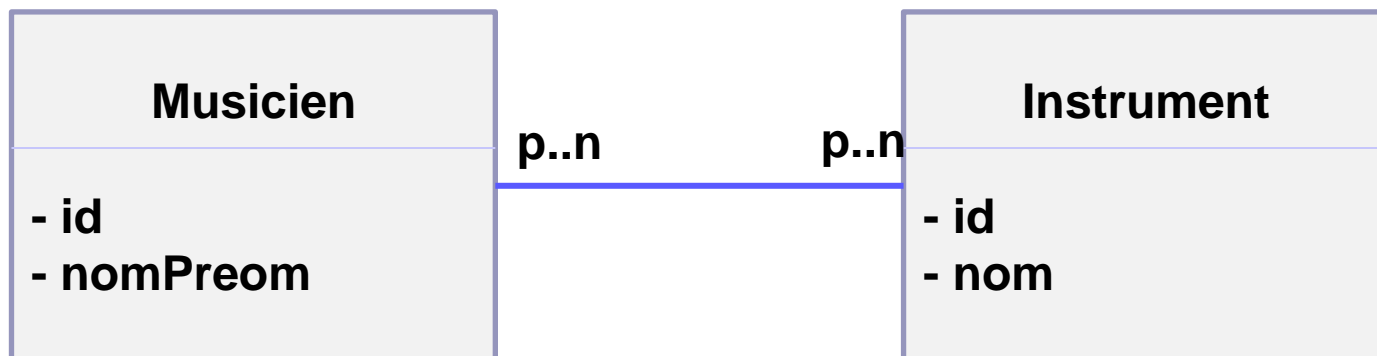


## VI.4. Relation p:n

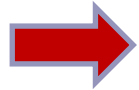
### ■ Unidirectionnelle



### ■ Bidirectionnelle



## a. Unidirectionnelle

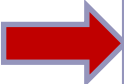


```
@Entity
public class Musicien implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
@ManyToMany
private Collection<Instrument> instruments ;}
```

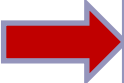
```
@Entity
public class Instrument implements Serializable
{ @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String nom
// reste de la classe

}
```

## b. Bidirectionnelle



```
@Entity
public class Musicien implements Serializable
{ @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  @ManyToOne
  private Collection<Instrument> instruments ;
  // reste de la classe }
```



```
@Entity
public class Instrument implements Serializable
{ @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String nom
  @ManyToMany (mappedBy="instruments")
  private Collection<Musicien> musiciens ;
  // reste de la classe
}
```

## VII. Autres annotations

- Il existe de nombreuses autres annotations,
  - Voir par exemple : JPA Reference  
<http://www.objectdb.com/api/java/jpa>
- Il se peut que les TPs en introduisent certaines
- Les curieux peuvent consulter la spécification JPA

**?**  
**Héritage entre les entités**  
**Les annotations**  
**&**  
**Les attributs**

## VIII. La persistance par sérialisation

- Sérialisation = sauvegarde de l'état d'un objet sous forme d'octets.
  - Rappel : l'état d'un objet peut être quelque chose de très compliqué.
  - Etat d'un objet = ses attributs, y compris les attributs hérités.
  - Si les attributs sont eux-même des instances d'une classe, il faut sauvegarder aussi les attributs de ces instances, etc...
- A partir d'un état sérialisé, on peut reconstruire l'objet
- En java, au travers de l'interface `java.io.Serializable`, des méthodes de `java.io.ObjectInputStream` et `java.io.ObjectOutputStream`