Java Persistence Api

Origines

- Accès simple aux données
  - JDBC apparu dès les premières versions de Java
- Problème
  - L'objectif de Java est de modéliser les problématiques métier en objet
- Confrontation de deux mondes
  - Objet d'un coté, Relationnel de l'autre

ORM (Mapping Objet - Relationnel)

- Permet de mettre en correspondance le modèle de données relationnel et le modèle objets
  - Amélioration de l'architecture logicielle
  - Génération automatique du code de requêtage SQL
  - Abstraction de la base de données (travail sur les objets uniquement)
  - Gestion des "incompatibilités" (héritage, associations…)

Historique

- 1996 : Toplink, premier framework ORM pour le développement Java
- 1998 : sortie de l'API EJB Entity 1.0
- 2001 : sortie d'Hibernate permettant de pallier les lourdeurs des EJB
- 2003 : JBoss embauche Gavin King et les principaux développeurs
- 2006 : participation de King à la création de la norme JSR 220 (EJB 3.0 et JPA 1.0)
- 2007 : Oracle livre Toplink à la fondation Eclipse sous le nom EclipseLink
- 2009 : JPA 2.0
- 2013 : JPA 2.1
- 2017 : JPA 2.2

Fonctionnement

- C'est une interface de programmation
  - Elle a besoin d'une implémentation
- Technologie basée sur :
  - Des interfaces et des classes génériques
  - Des annotations
  - Un fournisseur de persistance
  - Un fichier de configuration au format XML

Les différentes implémentations

- EclipseLink,
  - L'implémentation de référence
- Hibernate
  - Projet porté par JBoss, appartenant à RedHat
  - De loin la plus utilisée
- OpenJPA,
  - Projet de la fondation Apache

Les entités

- Une entité est une classe dont les instances peuvent être persistantes
- Utilisation d'annotations
  - Sur la classe : correspondance avec la table associée
  - Sur les attributs : correspondance avec les colonnes de la table
- Structure
  - La classe est un JavaBean

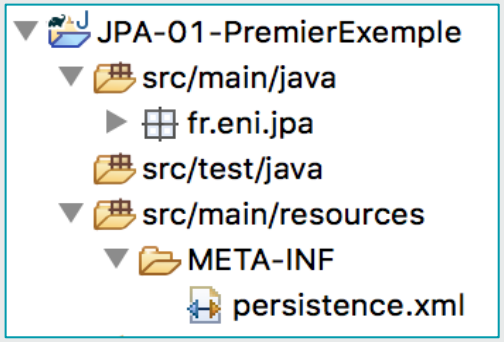
JPA

Exemple

```
package com.formation.jpa.entity;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class User implements Serializable
{
	@Id
	private int id;
	private String login;
	private String password;

	public User() {
	}
	...
```

Le fichier "persistence.xml"

- Fichier positionné dans un répertoire "META-INF" à la racine des sources



- Contient la configuration de une ou plusieurs "unités de persistance"

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

JPA

Exemple

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="Mysql_UP" transaction-type="RESOURCE_LOCAL">

        <class>fr.eni.jpa.entity.User</class>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/javaavance" />
            <property name="javax.persistence.jdbc.user" value="java" />
            <property name="javax.persistence.jdbc.password" value="avance" />

            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
    </persistence-unit>

</persistence>
```

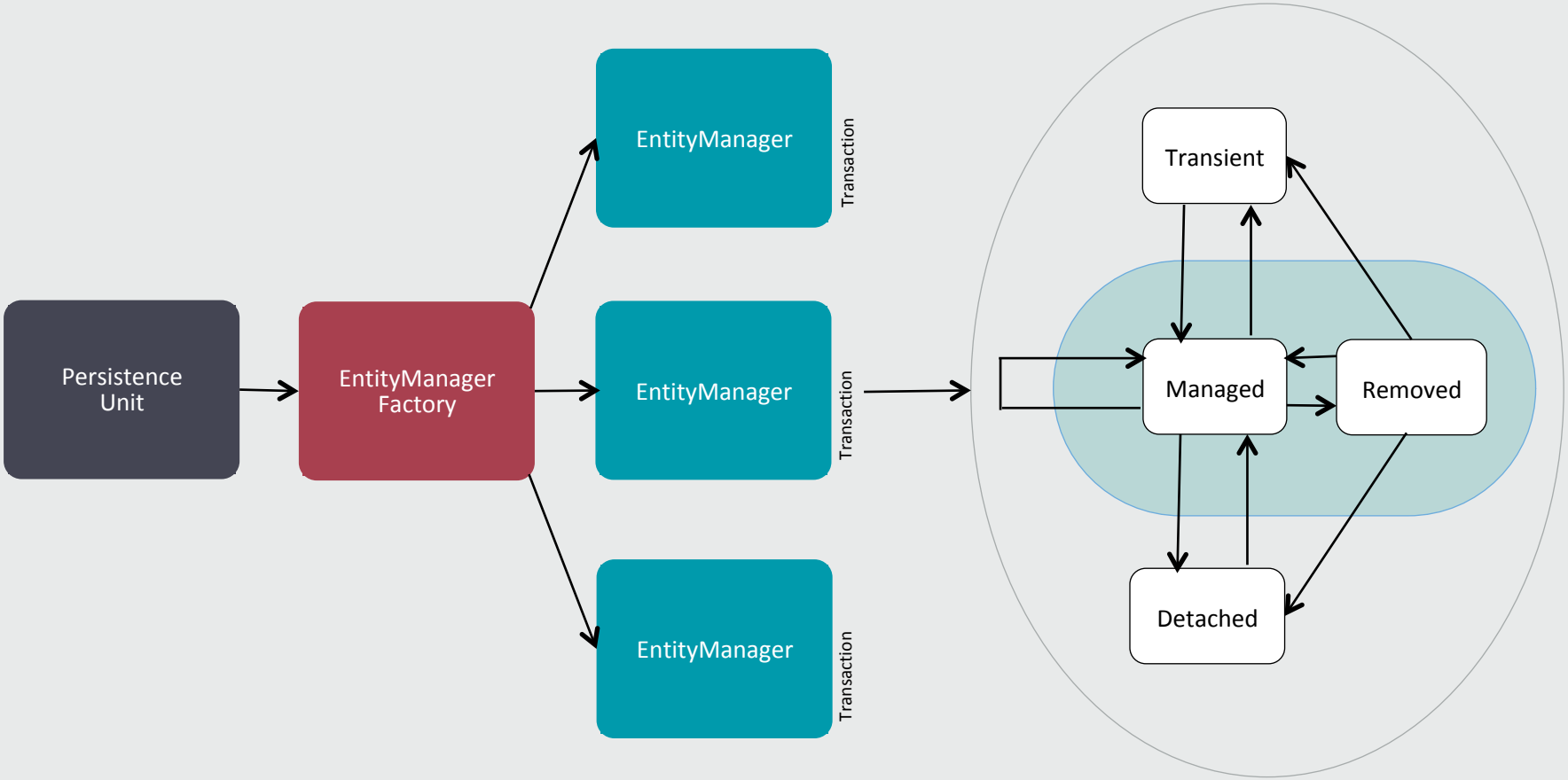
EntityManagerFactory et EntityManager

- EntityManagerFactory : la fabrique d'EntityManager
- EntityManager : gestionnaire d'entités

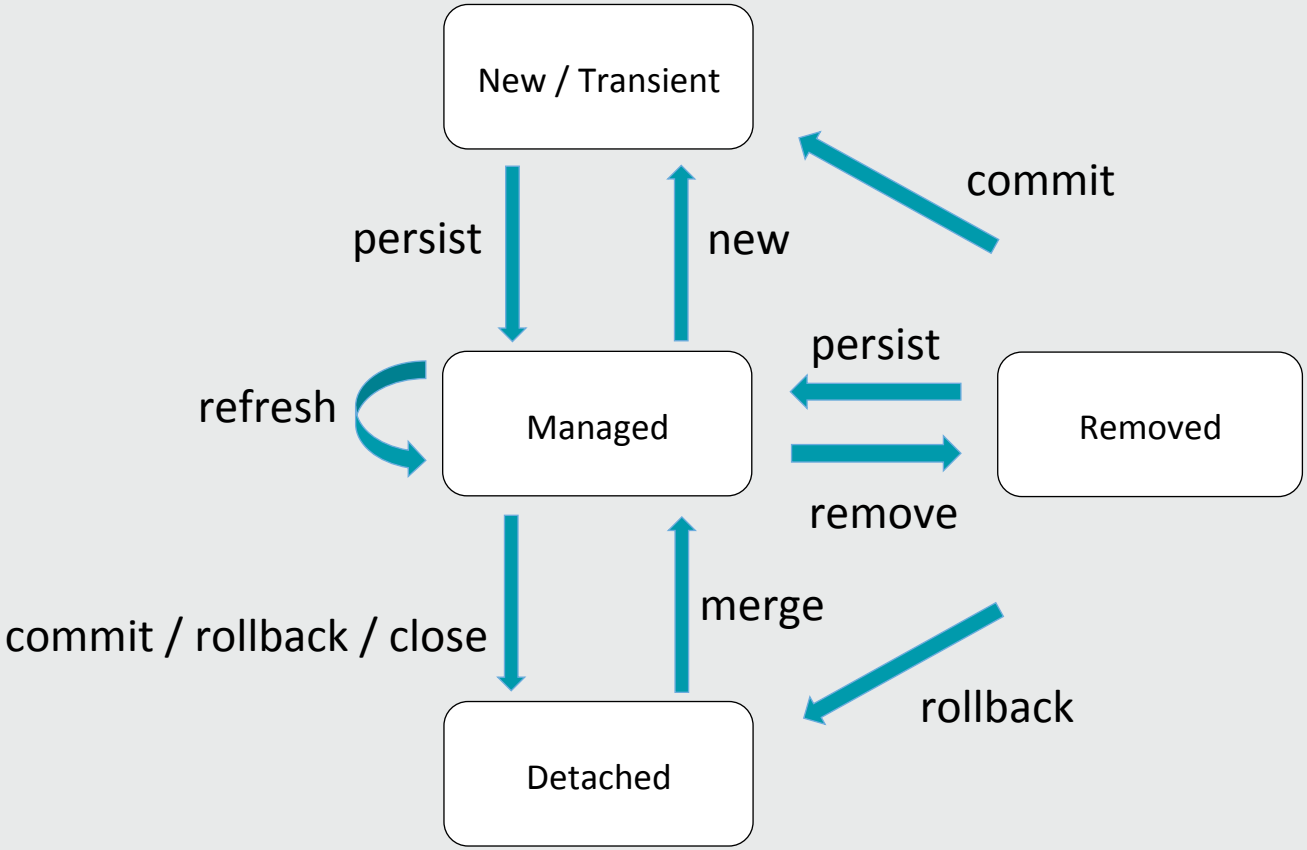
```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Mysql_UP");
EntityManager em = emf.createEntityManager();

// ...

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

JPA

Cycle de vie des entités



Cycle de vie des entités



JPA

Exemple

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

User u1 = new User(1, "java", "avance");

em.getTransaction().begin();
try {
    em.persist(u1);
    em.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
    em.getTransaction().rollback();
}

String requete = "from User u";
List<User> listeU = em.createQuery(requete).getResultList();

System.out.println("Liste des User :");
for (User user : listeU) {
    System.out.println(user);
}
em.close();
emf.close();
```

Démonstration

Les annotations

- @Entity
  - Obligatoire, sur la classe
- @Table (name="nomTable")
  - Facultatif, sur la classe
  - Mapper les objets de la classe avec la table dont le nom est redéfini
  - Si omis, la table prend le nom de la classe

```
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="USERS")
public class User {
```

JPA

Les annotations

- @Id
  - La déclaration d'une clé primaire est obligatoire
  - Sur un attribut ou sur le getter

- @GeneratedValue
  - Facultatif, sur l'attribut ou sur le getter annoté avec @Id
  - Définit la manière dont la base gère l'auto-incrément de la clé primaire
  - Attribut "strategy" obligatoire pouvant avoir comme valeur :
    - AUTO
    - IDENTITY
    - SEQUENCE
    - TABLE

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

Les annotations

- @Column
  - Facultatif, sur un attribut
  - Paramètres : name, length, nullable, unique
- @Transient
  - Facultatif, sur un attribut
  - Indique que l'attribut ne sera pas mappé (et donc non persisté) dans la table
- @Basic
  - Parametres : fetch, optional

Démonstration

Les clés primaires composites (méthode 1)

- Utilisation des annotations @Id et @IdClass
  - @Id sur les attributs composant la clé composite
  - Création d'une classe décrivant la clé primaire
    - Attributs, getters et setters de la clé primaire identiques à la classe principale
    - Implémentation de Serializable
    - Constructeur sans paramètre
  - Ajout de l'annotation @IdClass(nomClassePK.class) sur la classe principale

Démonstration

Les clés primaires composites (méthode 2)

- Utilisation des annotations @EmbeddedId et @Embeddable
  - Création d'une classe décrivant la clé primaire
    - Attributs, getters et setters composant la clé primaire
    - Implémentation de Serializable
    - Constructeur sans paramètre
    - Annoté avec @Embeddable
  - Déclaration d'un attribut instance de classe embarquée
    - @EmbeddedId sur l'attribut composant la clé composite

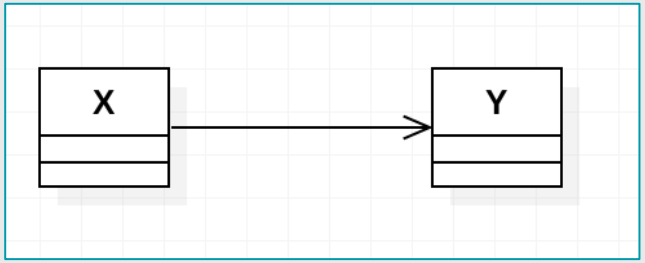
Attention :

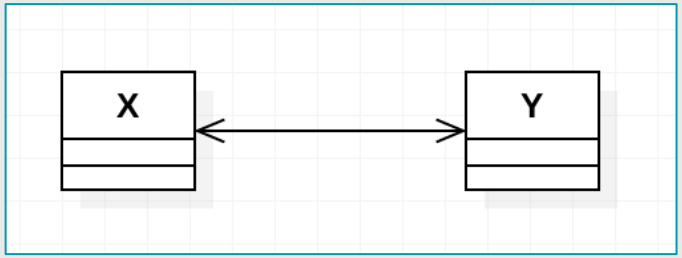
Modification de la structure de la classe Entité

Démonstration

Direction des relations et cardinalité

- Direction
  - Unidirectionnelle : le bean X possède une référence vers le bean Y
  - Bidirectionnelle : le bean X possède une référence vers le bean Y et réciproquement

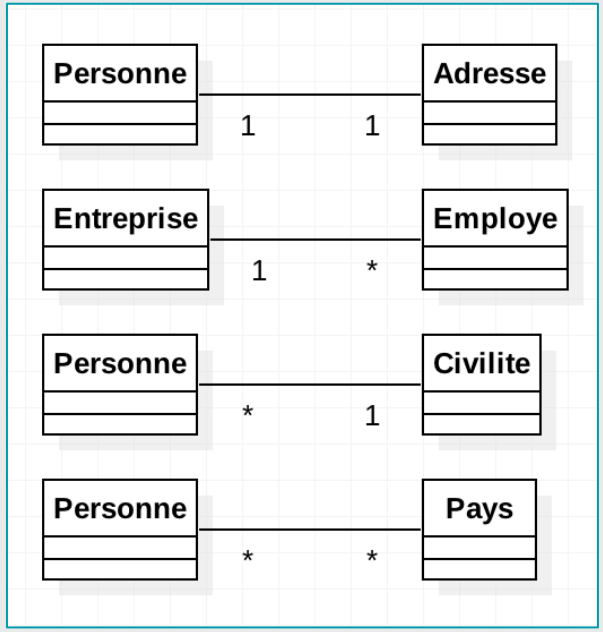




Direction des relations et cardinalité

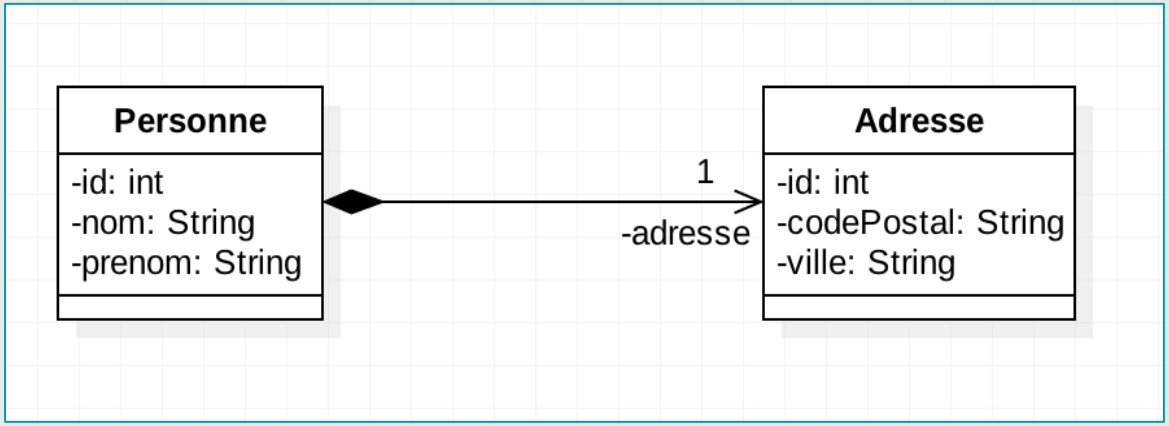
- Cardinalité
  - Indique combien d'instances vont intervenir de chaque côté d'une relation

  - One to one (1:1)
    - Une personne a une adresse
  - One to Many (1:N)
    - Une entreprise a des employés
  - Many to One (N:1)
    - Une référence est partagée par plusieurs beans
    - Une personne a une civilité
  - Many to Many (M:N)
    - Des personnes ont visité des pays



Relation 1-1 unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Personne
  - Adresse
  - Chaque classe sera mappée avec sa propre table



JPA

Relation 1-1 unidirectionnelle

- Déclaration d'un attribut Adresse dans la classe Personne
  - Annoté avec @OneToOne
    - Paramètres possibles : Cascade, orphanRemoval
  - @Basic(fetch = LAZY ou EAGER)
  - @JoinColumn

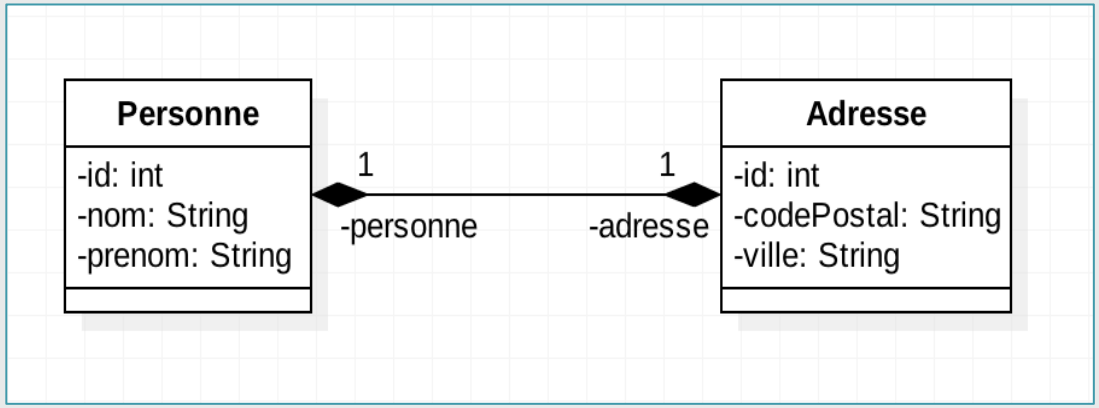
```
@Entity
@Table(name = "AdresseOTO")
public class Adresse {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    // ...
```

```
@Entity
@Table(name = "PersonneOTO")
public class Personne {

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

    private String nom;
    private String prenom;

    @OneToOne(cascade=CascadeType.ALL)
    @Basic(fetch=FetchType.LAZY)  // fetch=FetchType.EAGER
    private Adresse adresse;
```

Démonstration

Relation 1-1 bidirectionnelle

- Création de deux classes annotées avec @Entity
  - Personne
  - Adresse
  - Chaque classe sera mappée avec sa propre table



JPA

Relation 1-1 bidirectionnelle

- Déclaration d'un attribut Adresse dans la classe Personne
  - Annoté avec @OneToOne
    - Paramètres possibles : Cascade, orphanRemoval
- Déclaration d'un attribut Personne dans la classe Adresse
  - Annoté avec @OneToOne(mappedBy="…")

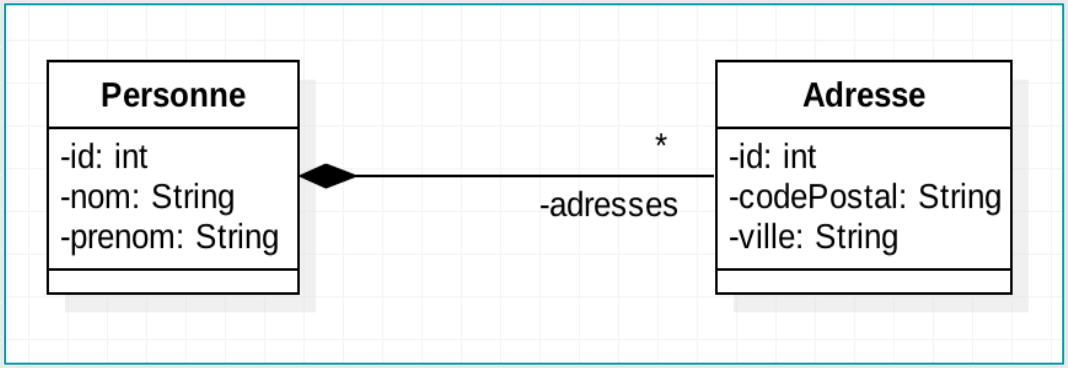
```java
@Entity
@Table(name = "PersonneOTOBi")
public class Personne {

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

    private String nom;
    private String prenom;

    @OneToOne(cascade=CascadeType.ALL)
    @Basic(fetch=FetchType.LAZY)  // fetch=FetchType.EAGER
    private Adresse adresse;
```

```java
@Entity
@Table(name = "AdresseOTOBi")

public class Adresse {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String codePostal;
    private String ville;

    @OneToOne(mappedBy="adresse", cascade=CascadeType.ALL)
    private Personne personne;
```

Démonstration

Relation 1-N unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Personne
  - Adresse
  - Chaque classe sera mappée avec sa propre table
  - Création d'une colonne de jointure



JPA

Relation 1-N unidirectionnelle

- Déclaration d'un attribut Adresse dans la classe Personne
  - Annoté avec @OneToMany
    - Paramètres possibles : Cascade, orphanRemoval
  - @Basic(fetch = LAZY ou EAGER)
  - @JoinColumn

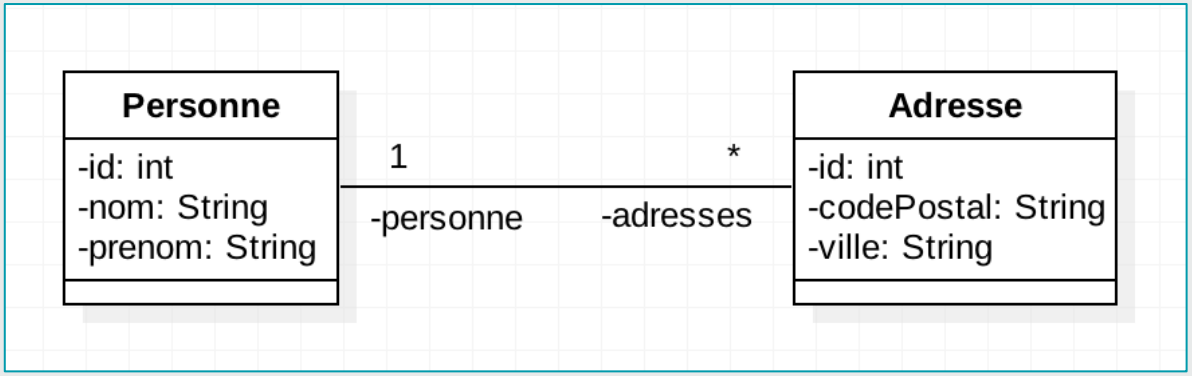
```
@Entity
@Table(name = "AdresseOTM")
public class Adresse {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    // ...
```

```
@Entity
@Table(name = "PersonneOTM")
public class Personne {

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

    private String nom;
    private String prenom;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, orphanRemoval = true)
    @JoinColumn(name="Personne_id")
    private List<Adresse> listeAdresses;
```

Démonstration

Relation 1-N bidirectionnelle

- Création de deux classes annotées avec @Entity
  - Personne
  - Adresse
  - Chaque classe sera mappée avec sa propre table
  - Création d'une colonne de jointure



JPA

Relation 1-N bidirectionnelle

- Déclaration d'un attribut Adresse dans la classe Personne
  - Annoté avec @OneToMany(mappedBy="…")
- Déclaration d'un attribut Personne dans la classe Adresse
  - Annoté avec @ManyToOne

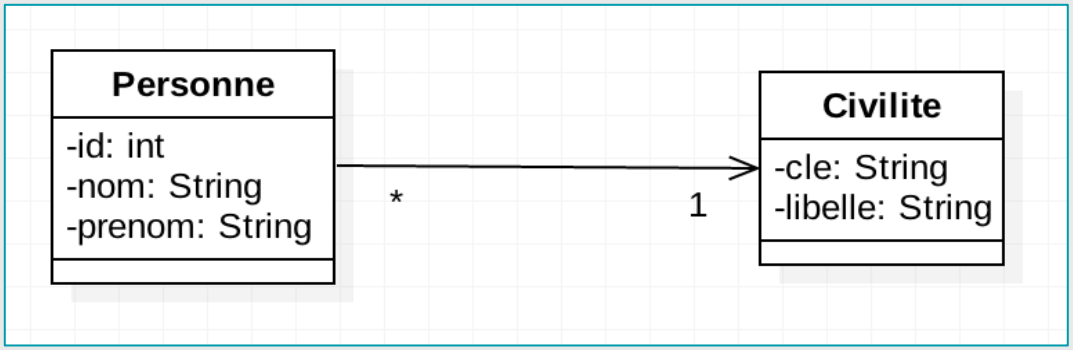
```java
@Entity
@Table(name="PersonneOTMBi")
public class Personne implements Serializable {
    private static final long serialVersionUID = 1L;

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

    private String nom;
    private String prenom;

    @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true, mappedBy="personne")
    private List<Adresse> listeAdresses;
```

```java
@Entity
@Table(name="AdresseOTMBi")
public class Adresse implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String codePostal;
    private String ville;

    @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private Personne personne;
```

Démonstration

Relation N-1 unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Personne
  - Civilite
  - Chaque classe sera mappée avec sa propre table
  - Pas de table de jointure (une colonne de jointure dans la table Personne)



Relation N-1 unidirectionnelle

- Déclaration d'un attribut Civilite dans la classe Personne
  - Annoté avec @ManyToOne
    - Paramètres possibles : cascade, fetch, optional

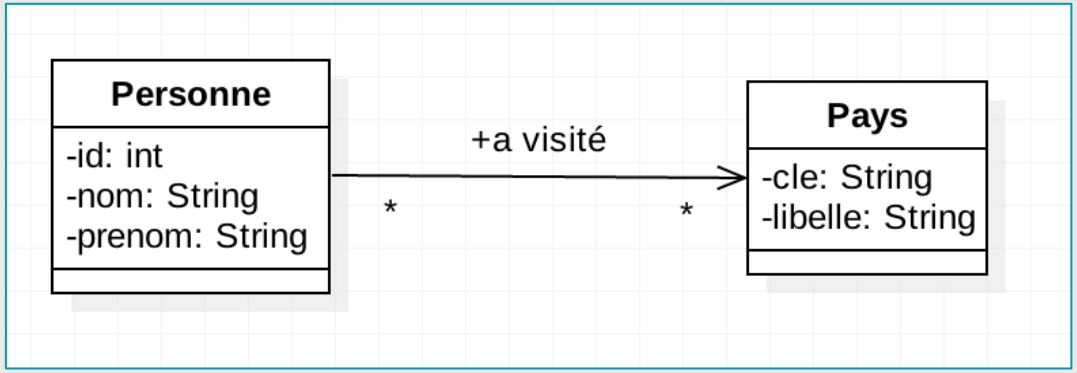
```java
@Entity
@Table(name="CiviliteMTO")
public class Civilite {

    @Id
    private String cle;
    private String libelle;
```

```java
@Entity
@Table(name="PersonneMTO")
public class Personne {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String nom;
    private String prenom;

    @ManyToOne
    private Civilite civilite;
```

Démonstration

Relation M-N unidirectionnelle

- Création de deux classes annotées avec @Entity
  - Personne
  - Pays
  - Chaque classe sera mappée avec sa propre table
  - Création d'une table de jointure



Relation M-N unidirectionnelle

- Déclaration d'un attribut Adresse dans la classe Personne
  - Annoté avec @ManyToMany
    - Paramètres possibles : Cascade, orphanRemoval
  - @Basic(fetch = LAZY ou EAGER)
  - @JoinColumn

```java
@Entity
@Table(name="PersonneMTM")
public class Personne implements Serializable {

    private static final long serialVersionUID = 1L;

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

    private String nom;
    private String prenom;

    @ManyToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(name="PersonnePays",
                joinColumns={@JoinColumn(name="personneId")},
                inverseJoinColumns={@JoinColumn(name="paysId")}
            )
    private List<Pays> paysVisites;
```

```java
@Entity
@Table(name="PaysMTM")
public class Pays {

    @Id
    @Column(name="cle")
    private String key;
    private String libelle;
```

Démonstration

JPA

Héritage

- Trois stratégies pour enregistrer une hiérarchie de classes en base :
  - SINGLE_TABLE :
    - Chaque hiérarchie d'entités JPA est enregistrée dans une table unique
    - Stratégie efficace pour les modèles de faible profondeur d'héritage
  - JOINED :
    - Chaque entité JPA est enregistrée dans sa propre table
    - Les entités d'une hiérarchie sont en jointure les unes des autres
    - Stratégie inefficace dans le cas de hiérarchies trop importantes
  - TABLE_PER_CLASS :
    - Seules les entités associées à des classes concrètes sont enregistrées dans leur propre table
    - Efficace, notamment dans le cas des hiérarchies importantes

Héritage – SINGLE_TABLE

- Toute la hiérarchie de classes est enregistrée dans une seule table
  - @Entity sur chaque classe
  - @Inheritance(strategy=InheritanceType.SINGLE_TABLE) sur la classe mère
- Autant de colonnes que de champs persistants différents
- Utilisation d'une colonne supplémentaire discriminante
  - @DiscriminatorColumn(name="TYPE_ENTITE") sur la classe mère
  - @DiscriminatorValue("...") sur chacune des classes de la hiérarchie

JPA

Héritage – SINGLE_TABLE

```java
@Entity
@Table(name="SingleVoiture")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCR")
@DiscriminatorValue(value="V")
public class Voiture {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String marque;
```

```java
@Entity
@DiscriminatorValue(value="B")
public class Berline extends Voiture {

    private String couleurCuir;
```

```java
@Entity
@DiscriminatorValue(value="C")
public class VoitureDeCourse extends Voiture {

    private String ecurie;
```

DISCR	id	marque	ecurie	couleurCuir
V	1	Renault Clio	NULL	NULL
B	2	BMW	NULL	Rouge
C	3	Ferrari	Scuderia Ferrari	NULL

Héritage – TABLE_PER_CLASS

- Autant de tables qu'il y a de classes concrètes annotées @Entity dans la hiérarchie
  - @Entity sur chaque classe
  - @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS) sur la classe mère
- Chaque table possède
  - sa propre clé primaire
  - les colonnes correspondant aux attributs issus de l'héritage
  - ses propres attributs
- Pas de colonne discriminante

JPA

Héritage – TABLE_PER_CLASS

```java
@Entity(name="TPCVoitureEntity")
@Table(name="TPCVoiture")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Voiture {

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private int id;
    private String marque;
```

```java
@Entity(name="TPCBerlineEntity")
@Table(name="TPCBerline")
public class Berline extends Voiture {

    private String couleurCuir;
```

```java
@Entity(name="TPCVoitureDeCourseEntity")
@Table(name="TPCVoitureDeCourse")
public class VoitureDeCourse extends Voiture {

    private String ecurie;
```

id	marque
1	Renault Clio

id	marque	couleurCuir
2	BMW	Rouge

id	marque	ecurie
3	Ferrari	Scuderia Ferrari

Héritage – JOINED

- Autant de tables qu'il y a de classes annotées @Entity dans la hiérarchie
  - @Entity sur chaque classe
  - @Inheritance(strategy=InheritanceType.JOINED) sur la classe mère
- Chaque table possède
  - Ses propres champs
- Les tables "filles" possèdent
  - Leurs propres champs
  - Une colonne référence à la table mère
- Possibilité de définir une colonne discriminante

JPA

Héritage – JOINED

```java
@Entity(name="JoinedVoitureEntity")
@Table(name="JoinedVoiture")
@DiscriminatorColumn(name="DISCR")
@DiscriminatorValue(value="V")
@Inheritance(strategy=InheritanceType.JOINED)
public class Voiture {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String marque;
```

```java
@Entity(name = "JoinedBerlineEntity")
@Table(name = "JoinedBerline")
@DiscriminatorValue(value="B")
public class Berline extends Voiture {

    private String couleurCuir;
```

```java
@Entity(name="JoinedVoitureDeCourseEntity")
@Table(name="JoinedVoitureDeCourse")
@DiscriminatorValue(value="C")
public class VoitureDeCourse extends Voiture {

    private String ecurie;
```

DISCR	id	marque
V	1	Renault Clio
B	2	BMW
C	3	Ferrari

couleurCuir	id
Rouge	2

ecurie	id
Scuderia Ferrari	3

Démonstration

Gestion des collections de base

- Possibilité d'enregistrer une collection d'éléments simple (String, Date, Integer…) sans avoir besoin de créer une nouvelle classe Entity
- Utilisation de l'annotation @ElementCollection
- Possibilité de redéfinir le nom de la table de jointure ainsi que les colonnes
  - @CollectionTable (
    name = "…",
    joinColumns=@JoinColumn(name = "…", referencedColumnName = "...")
  - @Column(name="…")

JPA

Gestion des collections de base

```java
@Entity
@Table(name = "PersonneCollection")
public class Personne implements Serializable{

    private static final long serialVersionUID = 1L;

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

    private String nom;
    private String prenom;

    @ElementCollection
    @CollectionTable(    name = "Sports",
                         joinColumns=
                             @JoinColumn(name="id_spo", referencedColumnName="id"))
    @Column(name = "sport")
    List<String> sports;
```

PersonneCollection

id	nom	prenom
1	Legrand	Lucie
2	Lepetit	Marc

Sports

id_spo	sport
1	Athlétisme
1	Judo
2	Football
2	Judo

Démonstration

JPQL

- Java Persistence Query Language (JPQL) est un langage de requêtage spécifique à JPA (issu du HQL d'Hibernate), indépendant de la base de données
  - Il ressemble à du SQL
  - Cependant, il s'utilise sur des objets Java (et non les tables)

- Deux types de requêtes
  - Named Queries
  - Dynamic Queries

JPQL - Named Queries

- Requêtes statiques pouvant être réutilisées
- Parsées et compilées une seule fois au chargement de la classe
- Utilisation d'annotations directement sur la classe
  - @NamedQueries
  - @NamedQuery
- Leur nom doit être unique pour l'application
- Elles retournent une liste typée
- Elles peuvent avoir des paramètres

JPQL - Named Queries

- Déclarations :

```
@Entity
@Table(name="PersonneMTO")

@NamedQueries({
    @NamedQuery (name = "findTous",
                 query="SELECT p FROM Personne p"),

    @NamedQuery( name = "findNomCommencePar",
                 query="SELECT p FROM Personne p WHERE p.nom LIKE :var"),

    @NamedQuery( name = "findMessieurs",
                 query="SELECT p FROM Personne p WHERE p.civilite.cle = 'M'"),
})
public class Personne {
```

JPA

JPQL - Named Queries

- Appels :

```
public static List<Personne> findTous(){
    TypedQuery<Personne> query = em.createNamedQuery("findTous", Personne.class);

    return query.getResultList();
}

public static List<Personne> findNomCommencePar(String debut){
    TypedQuery<Personne> query = em.createNamedQuery("findNomCommencePar", Personne.class);

    return query
                .setParameter("var", debut+"%")
                .getResultList();
}

public static List<Personne> findMessieurs(){
    TypedQuery<Personne> query = em.createNamedQuery("findMessieurs", Personne.class);

    return query.getResultList();
}
```

JPQL – Dynamic Queries

- Générées à la volée
- Dépendent du contexte
- Non réutilisées
- Moins performantes
- Syntaxe identique aux Named Queries

JPQL – Dynamic Queries

- Création d'un objet de type Query

```
Query query = em.createQuery(
"SELECT p FROM Personne p WHERE p.nom = 'Legrand' ");
```

- Création d'un objet de type TypedQuery

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p WHERE p.nom = 'Lepetit'", Personne.class);
```

JPQL – Dynamic Queries

- Passage de paramètres nommés

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p WHERE p.nom = :nom", Personne.class);
query.setParameter("nom", "Legrand");
```

- Passage de paramètres de position

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p WHERE p.nom = ?1", Personne.class);
query.setParameter(1, "Legrand");
```

JPQL – Dynamic Queries

- Recherche d'une liste d'éléments

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p WHERE p.nom = 'Legrand' ", Personne.class);
List<Personne> liste = query.getResultList();
```

- Recherche d'un seul élément :

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p WHERE p.id = :id ", Personne.class);
query.setParameter("id", 1);
Personne p = query.getSingleResult();
```

Attention :

Renvoie une exception si aucun résultat trouvé

JPQL – Dynamic Queries

- Recherche d'une valeur

```
TypedQuery<Long> query = em.createQuery(
"SELECT COUNT(p.id) FROM Personne p", Long.class);
long nb = query.getSingleResult();
```

- Recherche d'une liste de valeurs

```
TypedQuery<String> query = em.createQuery(
"SELECT p.prenom FROM Personne p", String.class);
List<String> liste = query.getResultList();
```

JPQL – Dynamic Queries

- Jointure

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p JOIN p.civilite c WHERE c.cle='Mlle'",
Personne.class);

List<Personne> liste = query.getResultList();
```

- Jointure avec tri

```
TypedQuery<Personne> query = em.createQuery(
"SELECT p FROM Personne p JOIN p.civilite c ORDER BY c.libelle",
Personne.class);

List<Personne> liste = query.getResultList();
```

JPQL – Dynamic Queries

- Suppression

```
Query query = em.createQuery(
              "DELETE FROM Personne p WHERE p.nom = 'Legrand'");
query.executeUpdate();
```

- Modification

```
Query query = em.createQuery(
              "UPDATE Personne p SET p.prenom = :prenom WHERE p.id = :id");
...
query.executeUpdate();
```

Démonstration

Criteria

- Criteria API est une alternative à JPQL
- Cette API permet de construire des critères de requête à l'aide d'objets Java
- Objectifs de Criteria API
  - s'affranchir des chaînes de caractères qui définissent les requêtes en JPQL (pas de contrôle à la compilation Java, erreurs de syntaxe décelées à l'exécution)
  - apporter un contrôle de type ("type-safe")
- Criteria API repose essentiellement sur 2 classes
  - CriteriaBuilder
  - CriteriaQuery

JPA

Criteria

- CriteriaBuilder
  - Interface principale de l'API Criteria
  - Elle s'obtient via un EntityManager
  - Elle permet de faire des requêtes typées

- CriteriaQuery
  - Représente une requête SELECT en base de données
  - Elle représente toutes les clauses de la requête
  - Ses éléments peuvent être réutilisés pour plusieurs CriteriaQuery différentes
  - On y retrouve les clauses habituelles
    - from(Class) la clause FROM
    - select() la clause de sélection
    - distinct() pour supprimer les doublons
    - where(), orderBy(), groupBy(), having()...

JPA

Criteria

- Root
  - Correspond à la racine de la requête
  - Similaire à la clause FROM d'une requête JPQL
  - Nécessaire si on désire faire des jointures, des prédicats, etc.
  - Instance récupérée via l'objet CriteriaQuery

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Personne> query = cb.createQuery(Personne.class);

Root<Personne> liste = query.from(Personne.class);

query.select(liste);

TypedQuery<Personne> tq = em.createQuery(query);

return tq.getResultList();
```

JPA

Criteria

- Tri
  - Utilisation de la classe
    - javax.persistence.criteria.Order
  - Et de la méthode
    - orderBy(Order o)

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Personne> query = cb.createQuery(Personne.class);

Root<Personne> pers = query.from(Personne.class);
Order ordre = cb.asc(pers.get("nom"));

query.select(pers);
query.orderBy(ordre);

TypedQuery<Personne> tq = em.createQuery(query);

return tq.getResultList();
```

JPA

Criteria

- Restriction du résultat
  - Utilisation de la méthode where sur l'objet CriteriaQuery et des méthodes de restriction sur l'objet CriteriaBuilder
  - Utilisation des méthodes equal, exists, and, or, like, notlike…

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Personne> query = cb.createQuery(Personne.class);

Root<Personne> pers = query.from(Personne.class);
Predicate clauseWhere = cb.equal(pers.get("nom"), "Legrand");
query.select(pers);
query.where(clauseWhere);

TypedQuery<Personne> tq = em.createQuery(query);

return tq.getResultList();
```

JPA

Criteria

- Jointures
  - Utilisation de la classe `Join<Source,Target>` pour faire une jointure entre deux entités liées par une association
  - La méthode `join` prend pour paramètre le nom de l'association (classe source)

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Personne> query = cb.createQuery(Personne.class);

Root<Personne> pers = query.from(Personne.class);
Join<Personne, Civilite> civ = pers.join("civilite");

Predicate clauseWhere = cb.equal(civ.get("cle"), "Mlle");
query.select(pers);
query.where(clauseWhere);

TypedQuery<Personne> tq = em.createQuery(query);

return tq.getResultList();
```

Démonstration