



Formation Java JPA

Java Persistence API

Chapitre 1

JPA - Introduction

Sommaire

Couche de persistance sans ORM

ORM – Principe

ORM – Exemple

ORM – Un peu d'histoire

API JPA

Les objets JPA

Configuration

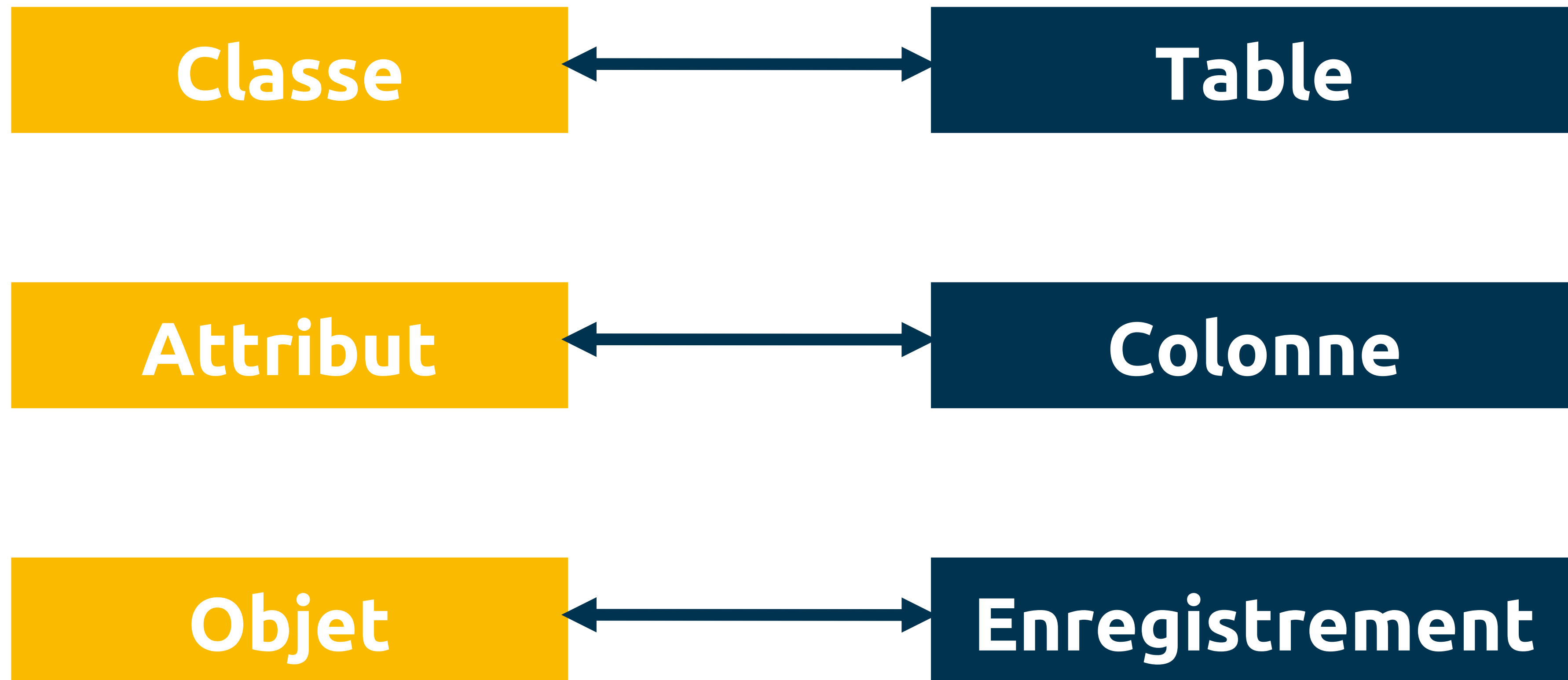
Mapping JPA

Les entités

Couche de persistance sans ORM

- ❑ Le développeur a la possibilité d'écrire manuellement la persistance des objets:
 - Toutes les étapes d'utilisation de JDBC apparaissent clairement dans le code des classes
 - La gestion des connexions
 - L'écriture manuelle des requêtes
 - Plus de dix lignes de codes sont nécessaires pour rendre persistante une classe contenant des propriétés simples
 - Développement objets "très coûteux"

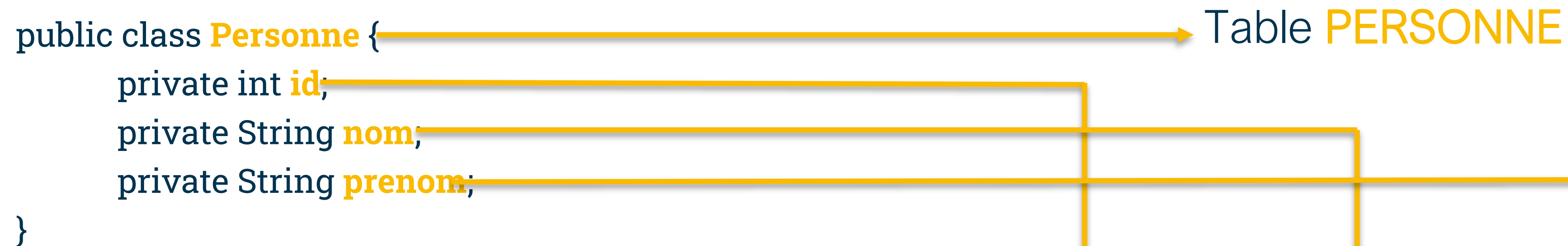
ORM - Principe



ORM - Exemple

```
public class Personne {  
    private int id;  
    private String nom;  
    private String prenom;  
}
```

Table **PERSONNE**



Personne p1 = new Personne(1, « Durand », « Paul »);

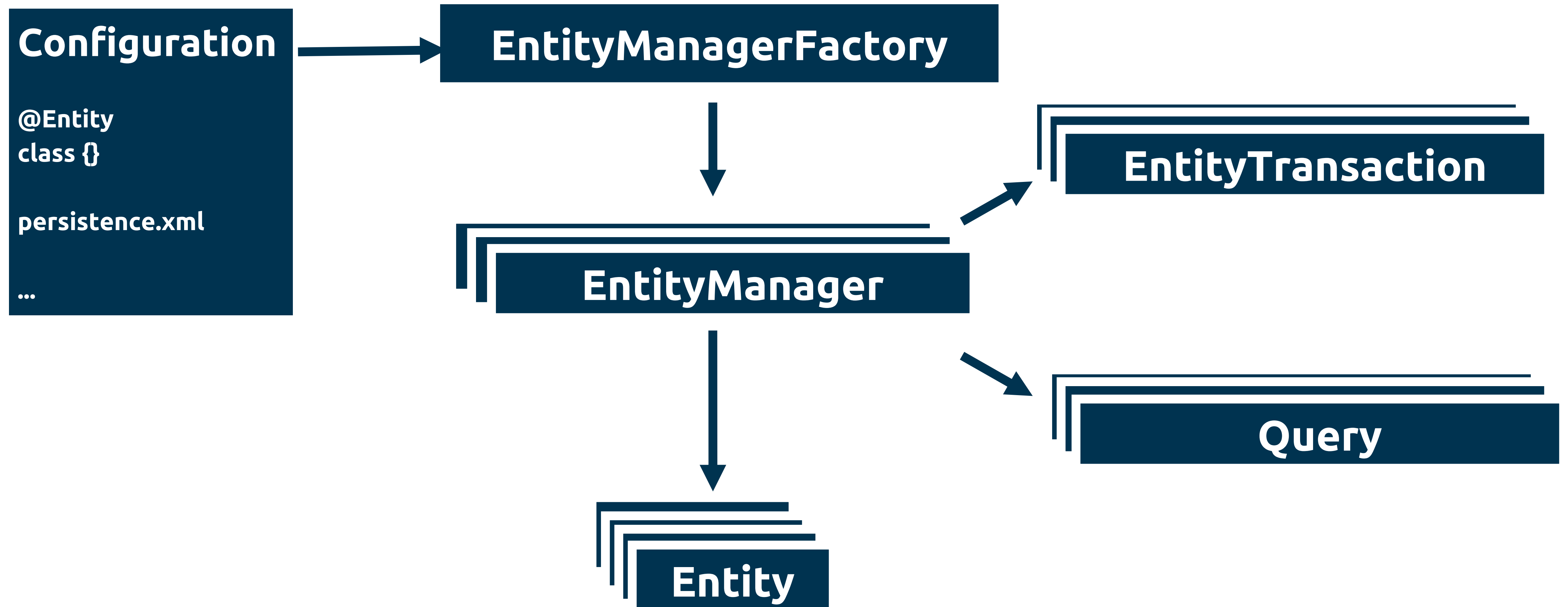
Personne p2 = new Personne(2, « Dupont », « Pierre »);

ID	NOM	PRENOM
1	Durand	Paul
2	Dupont	Pierre

JPA = Java Persistence API

- ❑ Les entités persistantes sont des POJO
- ❑ Les entités ne se persistent pas elles-mêmes : elle passent par l'**EntityManager**
- ❑ L'EntityManager se charge de persister les entités en s'appuyant sur les annotations qu'elles portent
- ❑ La classe **Query** encapsule les résultats d'une requête
- ❑ Le langage **JPQL** : (Java Persistence Query Language). Une requête JPQL est analogue à une requête SQL sur des objets

Les objets JPA



Configuration (persistence.xml) (1/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
  version="3.0">
  <persistence-unit name="nantes-jpa" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/nantes-bdd"/>
      <property name="jakarta.persistence.jdbc.user" value="nantes_user"/>
      <property name="jakarta.persistence.jdbc.password" value="nantes_pass"/>
    </properties>
  </persistence-unit>
</persistence>
```

Mapping JPA

- ❑ JPA doit faire un pont entre le monde relationnel de la base de données et le monde objet
- ❑ Ce pont est fait par configuration :
 - Avec des fichiers XML. C'était quasiment l'unique façon de faire jusqu'à l'avènement du JDK 1.5
 - Avec des annotations Java depuis le JDK 1.5

Les entités

- ❑ Le bean entity est composé de propriétés mappées sur les champs de la table de la base de données.
- ❑ Ce sont de simples POJO (Plain Old Java Object).
- ❑ Un POJO n'implémente aucune interface particulière ni n'hérite d'aucune classe mère technique.

Conditions pour les classes entités

❑ Un bean entité doit obligatoirement

- Avoir un constructeur **sans argument**
- Être déclaré avec l'annotation **@jakarta.persistence.Entity**
- Posséder au moins une propriété déclarée comme clé primaire avec l'annotation **@Id**

Exemple d'entité (1/4)

@Entity Indique que la classe doit être gérée par JPA

@Entity

public class Personne {

private Integer id;

private String nom;

private String prenom;

// constructeur sans argument

public Personne() {
}

Constructeur sans argument

// getters and setters

}

Exemple d'entité (2/4)

@Entity

@Entity Indique que la classe doit être gérée par JPA

@Table(name="personne")

@Table (facultatif) désigne la table à mapper

public class Personne {

@Id

@Id Clé primaire

private Integer id;

private String nom;

private String prenom;

// constructeur sans argument

public Personne() {
}

Constructeur sans argument

// getters and setters

}

Exemple d'entité (3/4)

@Entity

@Entity Indique que la classe doit être gérée par JPA

@Table(name="personne")

@Table (facultatif) désigne la table à mapper

public class Personne {

@Id

@Id Clé primaire

private Integer id;

@Column(name = "NOM", length = 30, nullable = false, unique = true)

private String nom;

@Column
pour Associer un champ à une
colonne de la table

@Column(name = "PRENOM", length = 30, nullable = false)

private String prenom;

// constructeur sans argument

public Personne() {
}

Constructeur sans argument

// getters and setters

}

Exemple d'entité avec id généré

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

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

    public Personne() {
    }

}
```

Stratégie de gestion de l'id

Chapitre 2

JPA - CRUD

Sommaire

Cycle de vie d'une entité

EntityManager

Insérer une donnée

Modifier une donnée

Fusionner une donnée

Supprimer une donnée

Le fichier persistence.xml

- ❑ Le fichier **persistence.xml** peut contenir un nombre indéfini de **persistence-unit**
- ❑ Chaque **persistence-unit** fait référence à une base de données, éventuellement d'éditeurs différents.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
  version="3.0">
  <persistence-unit name="maconfig1" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/magasin"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value="admin54!"/>
    </properties>
  </persistence-unit>

  <persistence-unit name="maconfig2" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/compta"/>
      <property name="jakarta.persistence.jdbc.user" value="user"/>
      <property name="jakarta.persistence.jdbc.password" value="user256_ret"/>
    </properties>
  </persistence-unit>
</persistence>
```

Création d'un EntityManager

```
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("maConfig1");  
EntityManager em = entityManagerFactory.createEntityManager();
```

❑ Qu'est-ce que **maConfig1** ?

- C'est le nom de la **configuration** telle que vous l'avez nommée dans le fichier **persistence.xml**
- La classe **Persistence** charge le fichier **persistence.xml**
- Dans ce fichier, il y a une configuration nommée **maConfig1**

```
<persistence-unit name="maConfig1" transaction-type="RESOURCE_LOCAL">  
    ...  
</persistence-unit>
```

Contexte de persistance

- ❑ Le **contexte de persistance** est lié à un **EntityManager**
- ❑ Il contient tous les objets extraits, mis à jour ou insérés via l'entityManager.
- ❑ Un **objet** qui **est dans le contexte de persistance** est dit « **Managé** »
- ❑ Un **objet** qui **n'est pas dans le contexte de persistance** est soit:
 - « **Transient** », c'est le cas des nouvelles instances d'objet
 - « **Détachés** », c'est le cas des objets qui ont été sortis du contexte de persistance.

Rôle d'une transaction

- ❑ L'ouverture d'une **transaction** est indispensable pour l'insertion, la modification et la suppression de données.
- ❑ Une transaction unique suffit pour un traitement complet : insertion, mise à jour et suppression de données dans différentes tables.
- ❑ Il est nécessaire de commiter la transaction en fin de traitement.

```
EntityManager em = entityManagerFactory.createEntityManager();
```

```
EntityTransaction transaction = em.getTransaction();
```

```
transaction.begin();
```

```
// Mises à jour, insertions et suppressions ici
```

```
transaction.commit();
```

- ❑ Dans les diapositives suivantes, la gestion de la transaction est implicite.

Insérer une donnée

- ❑ La méthode `persist (...)` permet de demander l'enregistrement d'une nouvelle **entité JPA**

```
EntityManager em = entityManagerFactory.createEntityManager();
```

```
Hotel h = new Hotel();  
h.setNom("BeauRegard");  
h.setVille("Lisbonne");  
em.persist(h);
```

- ❑ Pour récupérer l'identifiant généré (ex: 13465):
 - récupérer l'ID depuis l'objet « sauvé »

```
long id = h.getId();
```


Insertion et contexte de persistance

- ❑ Une instance insérée via la méthode persist est dans le contexte de persistance.

```
Hotel hotel = new Hotel();  
hotel.setId(2);  
hotel.setNom("Untel");  
em.persist(hotel);
```

- ❑ Cette instance est désormais mappée sur un enregistrement en base de données.
- ❑ Toute modification portée sur l'objet provoque une mise à jour automatique.

```
hotel.setNom("Untel");
```

Provoque un update automatique

Récupérer une instance

❑ La recherche par la clé primaire est faite avec la méthode :

- **find(..., ...)**

```
Hotel h = em.find(Hotel.class, 120);  
if (h != null){  
    //traitement  
}
```

Si dans la classe Hotel, la clé primaire est du type **int** alors le second argument est de type **int**

- Renvoie **null** si l'occurrence n'est pas trouvée

❑ Une instance extraite via la méthode **find** est dans le **contexte de persistance**.

Récupérer une instance

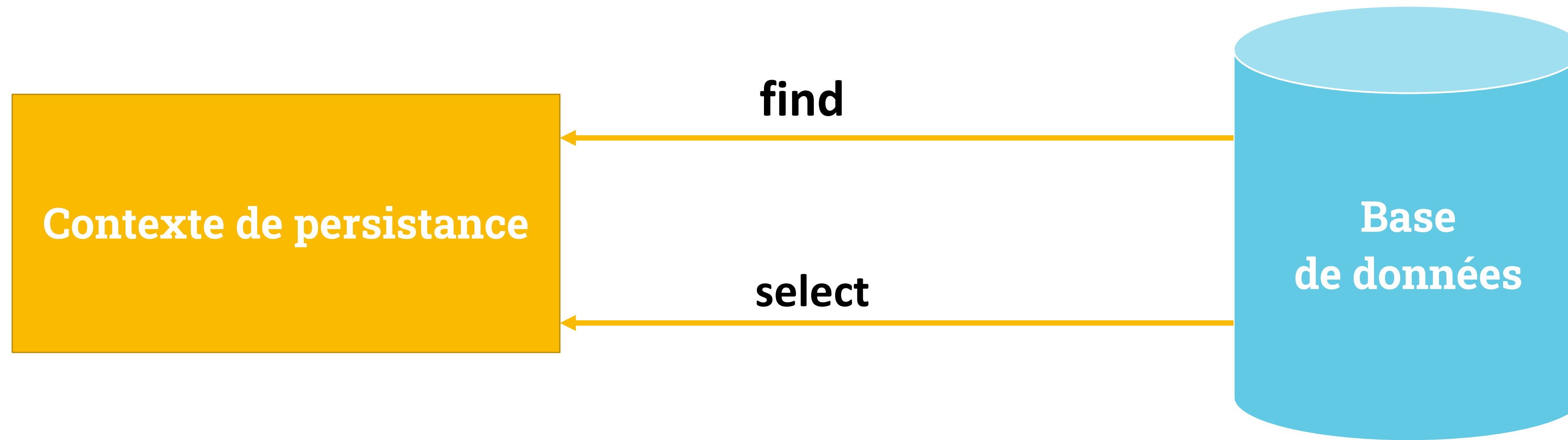
- ❑ La recherche par requête est réalisée grâce aux:
 - Langage de requêtes JPQL
 - Méthodes **createQuery()** de EntityManager

```
TypedQuery<Hotel> query2 = em.createQuery("select h from Hotel h where h.nom='nom2'",  
Hotel.class);
```

```
List<Hotel> hotels = query2.getResultList();
```

Contexte de persistance

- ❑ Tous les objets extraits de la base sont mis dans cet espace mémoire.
- ❑ On ne peut pas modifier ou supprimer un objet qui n'est pas dans le contexte de persistance.



Modifier une instance (1/2)

- ❑ La première manière de faire est d'extraire l'objet de la base puis d'utiliser un set

```
Hotel h = em.find(Hotel.class, 120);  
If (h != null){  
    h.setNom("Beauregard");  
}
```

- ❑ L'invocation d'un setter sur un objet situé dans le contexte de persistance provoque automatiquement un update

Modifier une instance (2/2)

- ❑ Il est également possible d'utiliser la méthode **merge**

```
Hotel hotel = new Hotel();  
hotel.setId(2);  
hotel.setNom("Untel");  
hotel.setVille("nouvelle ville");  
em.merge(hotel);
```

- ❑ La méthode **merge** va plus loin. Si l'objet d'identifiant 2 n'existe pas en base de données, il est automatiquement créé.
- ❑ Le merge est donc une sorte d'update or create.

Supprimer les données

```
Hotel h = em.find(Hotel.class, 120);
```

```
If (h != null){  
    em.remove(h);  
}
```

- ❑ L'objet supprimé **doit faire partie du contexte de persistance.**
- ❑ Il faut d'abord l'extraire de la base pour le supprimer.

Atelier (TP)

OBJECTIFS : Extraire des informations d'une base de données

DESCRIPTION :

- Dans le TP n°1 vous allez vous connecter à une base de données et en extraire des informations.

Mapping Clé Primaire (1)

- ❑ **@Id** : il permet d'associer un champ de la table à la propriété en tant que clé primaire
- ❑ **@GeneratedValue** : indique que la clé primaire est générée via une stratégie. Contient plusieurs attributs :
 - Strategy : Précise le type de générateur à utiliser :
 - *TABLE*,
 - *SEQUENCE*,
 - *IDENTITY*
 - *AUTO*. La valeur par défaut est *AUTO*
 - Generator : nom du générateur à utiliser

Générateurs d'identifiants

- ❑ Attention, les stratégies de génération de valeur de clé primaire ne marche pas sur toutes les bases de données.
- ❑ Les types possibles sont les suivants :
 - **AUTO**: laisse la base de données choisir la stratégie de génération par défaut pour générer la valeur de la clé primaire. Exemple: IDENTITY pour MySQL.
 - **IDENTITY**: utilise une colonne pour générer la clé primaire (AUTO_INCREMENT pour MySQL).
 - **TABLE**: utilise une table dédiée qui stocke les clés des tables générées. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @jakarta.persistence.TableGenerator
 - **SEQUENCE**: utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @jakarta.persistence.SequenceGenerator

Exemple

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

Exemple pour un ID en AUTO_INCREMENT (MySQL ou MariaDB)

- ❑ Evidemment la colonne ID en base doit avoir l'option AUTO_INCREMENT.

Autres annotations basiques

@Temporal: Indique quel type est le type SQL d'une colonne / champ de type date / heure.

TemporalType.DATE: désigne une date seule sans heure associée

TemporalType.TIME: désigne une heure

TemporalType.TIMESTAMP: désigne une date avec une heure (type par défaut)

!

Ne fonctionne qu'avec des objets `java.util.Date` (java 7)

@Transient : Indique de ne pas tenir compte du champ lors du mapping

@Enumerated : utiliser pour mapper des énumérations de type entier ou chaîne de caractères

Java 8 : mapper les LocalDate et LocalDateTime

- ❑ **@Temporal** ne fonctionne pas avec les classes LocalDate et LocalDateTime.
- ❑ **@Column** doit être utilisée sur les attributs de type LocalDate et LocalDateTime

Transaction avec JPA

```
EntityManager et = em.getTransaction();  
et.begin();
```

...

```
em.persist(p1);
```

...

```
em.persist(p2);
```

...

```
et.commit();    // ou et.rollback()  
em.close();
```

Chapitre 2

Pour aller plus loin

Ce que vous ne pouvez pas faire pour insérer...

- ❑ Cas d'erreur « detached entity » : vous essayez de **valoriser l'identifiant de la nouvelle entité** alors que vous avez positionné l'annotation `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

```
Hotel hotel = new Hotel();  
hotel.setId(2);  
hotel.setNom("Untel");  
em.persist(hotel);
```

jakarta.persistence.PersistenceException: org.hibernate.PersistentObjectException: detached entity passed to persist: fr.diginamic.entite.Hotel

- ❑ Si vous voulez imposer l'id, retirez l'annotation `@GeneratedValue`

Merge vs persist

- ❑ Comme on l'a vu précédemment le merge permet d'insérer en base de données également les nouvelles entités.
- ❑ Mais, à la différence du persist, le **merge** réalise **systematiquement un SELECT** en base de données pour vérifier si l'instance existe ou non avant de l'insérer.

Ce que vous ne pouvez pas faire pour modifier...

- ❑ Vous ne **pouvez pas** utiliser la méthode **persist** pour **modifier**.
- ❑ Cas d'erreur « detached entity » : vous essayez de persister une information qui existe déjà en base alors qu'elle n'est pas dans le contexte de persistance.

```
Hotel hotel = new Hotel();  
hotel.setId(2);  
hotel.setNom("Untel");  
em.persist(hotel);
```

Attention au merge

- ❑ A la différence de la méthode `persist` qui retourne `void`, la méthode `merge` renvoie l'objet managé.
- ❑ Le **merge** regarde si l'objet passé en paramètre est dans le persistence contexte.
 - S'il y est, la méthode retourne l'objet en question. Cet objet est forcément managé puisqu'il était dans le persistence contexte.
 - S'il n'y est pas, l'objet passé en paramètre sera créé mais il ne sera pas mis dans le persistence contexte.

```
Hotel hotel = new Hotel();
```

```
hotel.setId(2);
```

```
hotel.setNom("Untel");
```

```
hotel.setVille("nouvelle ville");
```

```
Hotel hotel2 = em.merge(hotel);
```

Managé

Pas managé

```
hotel.setVille("nouvelle ville 2"); // Pas d'effet
```

```
hotel2.setVille("nouvelle ville 2"); // Mise à jour
```

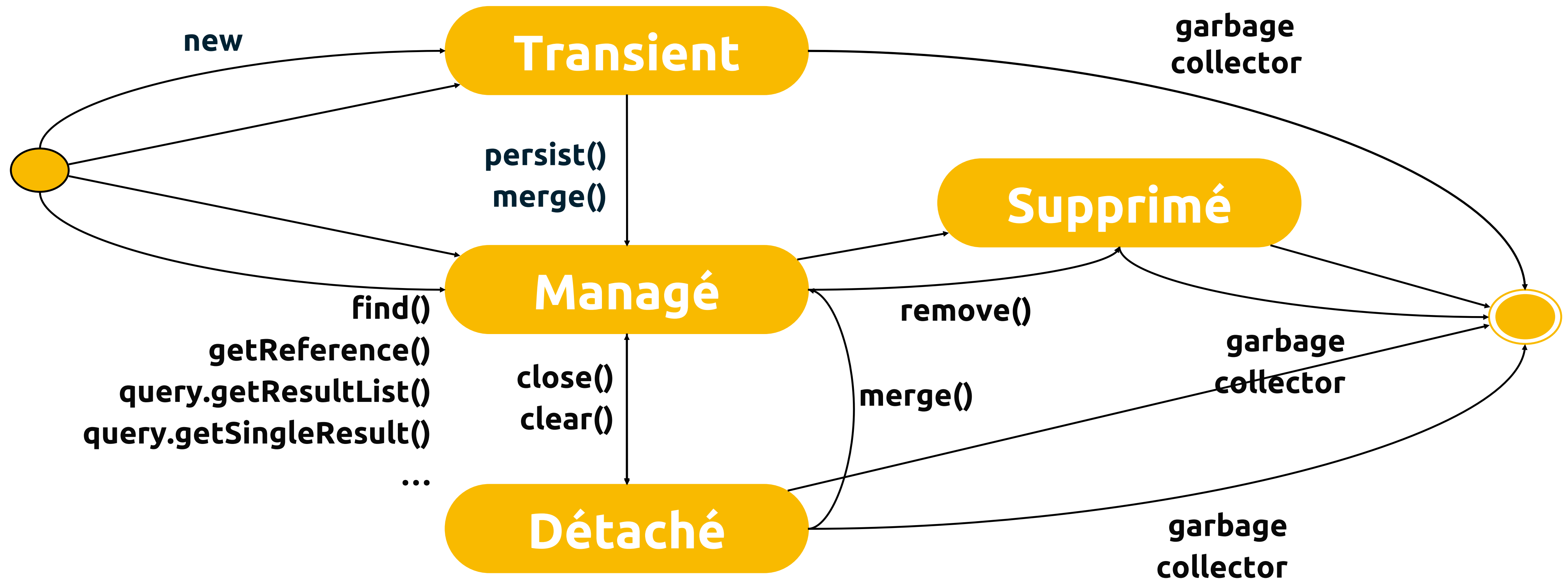
Différence entre merge et persist

- ❑ Le persist ne fait que de l'insertion, pas de mise à jour
- ❑ Le merge fait aussi bien de l'insertion que de la mise à jour. Par contre un objet mergé n'est pas « managé »

Le persist	
Hotel hotel = new Hotel(); hotel.setId(2); hotel.setNom("Untel"); em.persist(hotel);	Si l'objet n'existe pas en base de données, exception si l'Id est géré manuellement.
hotel.setNom("Untel2");	Requête update générée automatiquement

Le merge	
Hotel hotel = new Hotel(); hotel.setId(2); hotel.setNom("Untel"); em.merge(hotel);	Si l'objet n'existe pas en base de données, il est créé en base de données même si l'identifiant est géré manuellement
	Si l'objet existait en base de données, la mise à jour est effectuée.
hotel.setNom("Untel2");	Sans effet

Cycle de vie d'une entité



Chapitre 3

JPA – Les relations

Sommaire

Relations

@JoinColumn

Relation 1-n

Relation n-1

Relation n-n

Relation 1-1

Relations

- ❑ Une relation peut-être unidirectionnelle ou bidirectionnelle.
- ❑ Les associations entre entités correspondent à une jointure entre tables de la base de données:
 - Les tables sont liées grâce à une contrainte de clé étrangère
- ❑ Annotations :
 - @OneToOne
 - @OneToMany
 - @ManyToOne
 - @ManyToMany
- ❑ Règle :
 - @JoinColumn : définit la colonne de jointure.
 - @JoinTable : définit la table de jointure

@JoinColumn

Sans cette annotation le nom est défini par défaut:
<nom_entité>_<clé_primaire_entité>

Attributs :

name

Nom de la colonne correspondant à la clé étrangère.

referencedColumnName

Nom de la clé étrangère.

unique

Permet de définir la contrainte d'unicité de l'attribut. (Par défaut, false)

S'ajoute aux contraintes définies avec l'annotation @Table.

nullable

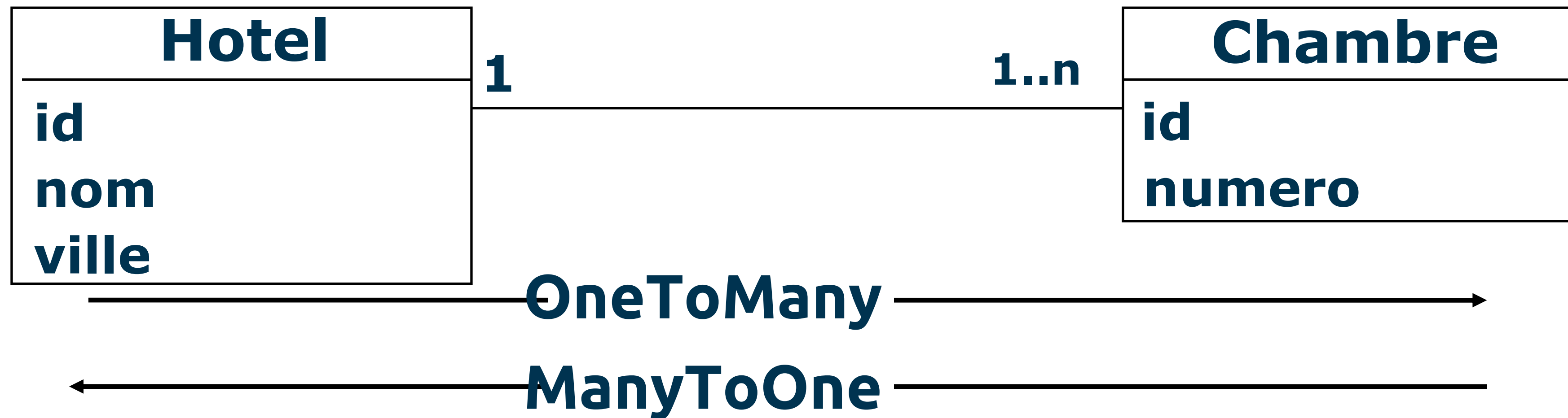
Définit si une colonne accepte la valeur nulle. (Par défaut, true)

Relation 1-n

Exemple

Une Chambre appartient à un Hôtel

Un Hôtel regroupe plusieurs Chambres



@ManyToOne

@Entity

@Table (name=« CHAMBRE »)

public class Chambre {

@Id

private int id;

@Column(name="NUMERO")

private int numero;

@ManyToOne

@JoinColumn(name="HOT_ID")

private Hotel hotel;

}

@ManyToOne associé à l'annotation @JoinColumn

@OneToMany

```
public class Hotel {  
    private long id;
```

@OneToMany porte l'attribut mappedBy

```
    @OneToMany(mappedBy="hotel")
```

```
    private Set<Chambre> chambres; // référence vers les chambres
```

```
    public Hotel() {
```

```
        chambres = new HashSet<Chambre>();
```

```
    }
```

```
    ...
```

```
}
```

Récapitulatif : Relation 1-n / n-1



```
public class Hotel {
```

```
    @OneToMany(mappedBy="hotel")
```

```
    private Set<Chambre> chambres;
```

```
}
```

```
public class Chambre {
```

```
    @ManyToOne
```

```
    @JoinColumn(name="HOT_ID")
```

```
    private Hotel hotel;
```

```
}
```

Relation n-n

Exemple

Une Reservation peut concerner plusieurs Chambre

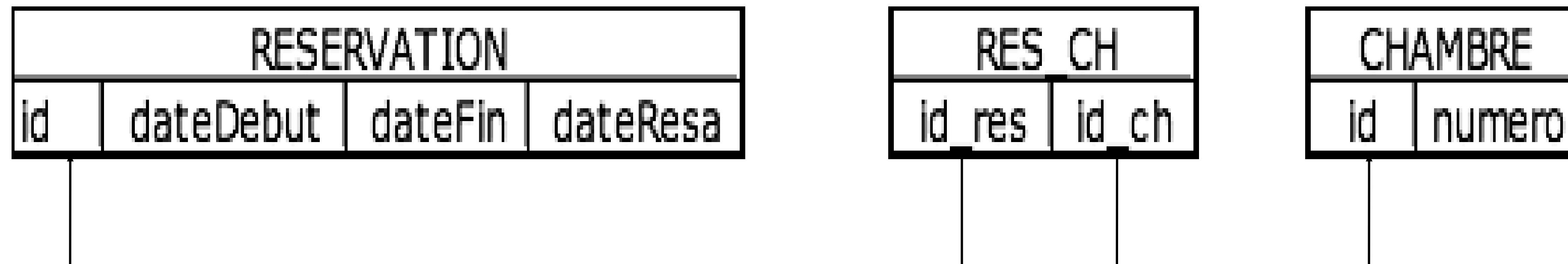
Une Chambre peut faire l'objet de plusieurs Reservation



Relation n-n

En base de données, utilisation d'une table intermédiaire

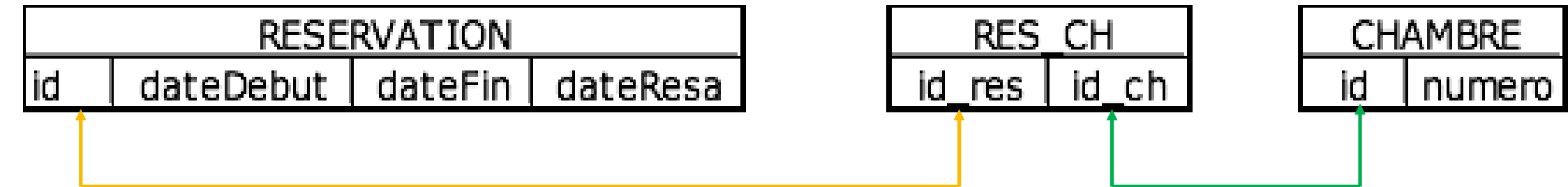
D'un point de vue du MCD, correspond à 2 relations 1:n



@ManyToMany

```
@Table (name="RESERVATION")
public class Reservation {
    @Id
    @Column(name="ID")
    private int id;

    @ManyToMany
    @JoinTable(name="RES_CH",
        joinColumns= @JoinColumn(name="ID_RES", referencedColumnName="ID"),
        inverseJoinColumns= @JoinColumn(name="ID_CH", referencedColumnName="ID")
    )
    private Set<Chambre> chambres;
}
```



La propriété **joinColumns** décrit la jointure entre la table « RESERVATION » et la table RES_CH

La propriété **inverseJoinColumns** décrit la jointure entre la table « CHAMBRE » et la table RES_CH

@ManyToMany

```
@Table (name="CHAMBRE")  
public class Chambre {  
  
    @ManyToMany(mappedBy="chambres")  
    private Set<Reservation> reservations;  
  
}
```

Dans ce cas la classe Chambre est esclave.

Si on déclare @JoinTable dans les 2 classes, les 2 relations sont maîtres.

Relation 1-1



@OneToOne par clé étrangère

```
public class Facture {  
    @Id  
    private int id;  
  
    @OneToOne  
    private Reservation reservation; // référence vers la réservation  
    ...  
    public Reservation getReservation() { ... }  
    public void setReservation(Reservation res) { ... }  
}
```

Atelier (TP)

OBJECTIFS : Utiliser des annotations

DESCRIPTION :

- Dans le TP n°2 vous allez mettre en place des relations au niveau des entités.

Chapitre 4

Embeddable et embedded

@Embeddable, Embedded

PERSONNES

ID	NOM	PRENOM	NUMERO	RUE	VILLE

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

```
    @Id
    private Long id;
    private String nom;
    private String prenom;
```

```
    @Embedded
    private Adresse adresse;
```

```
}
```

```
@Embeddable
public class Adresse {
    private Integer numero;
    private String rue;
    private String ville;
}
```

1 Table => 2 classes.

Chapitre 5

L'héritage

Sommaire

Héritage

Héritage – exemple

1 table par classe

1 table pour toute la hiérarchie

1 table par classe concrète

Héritage

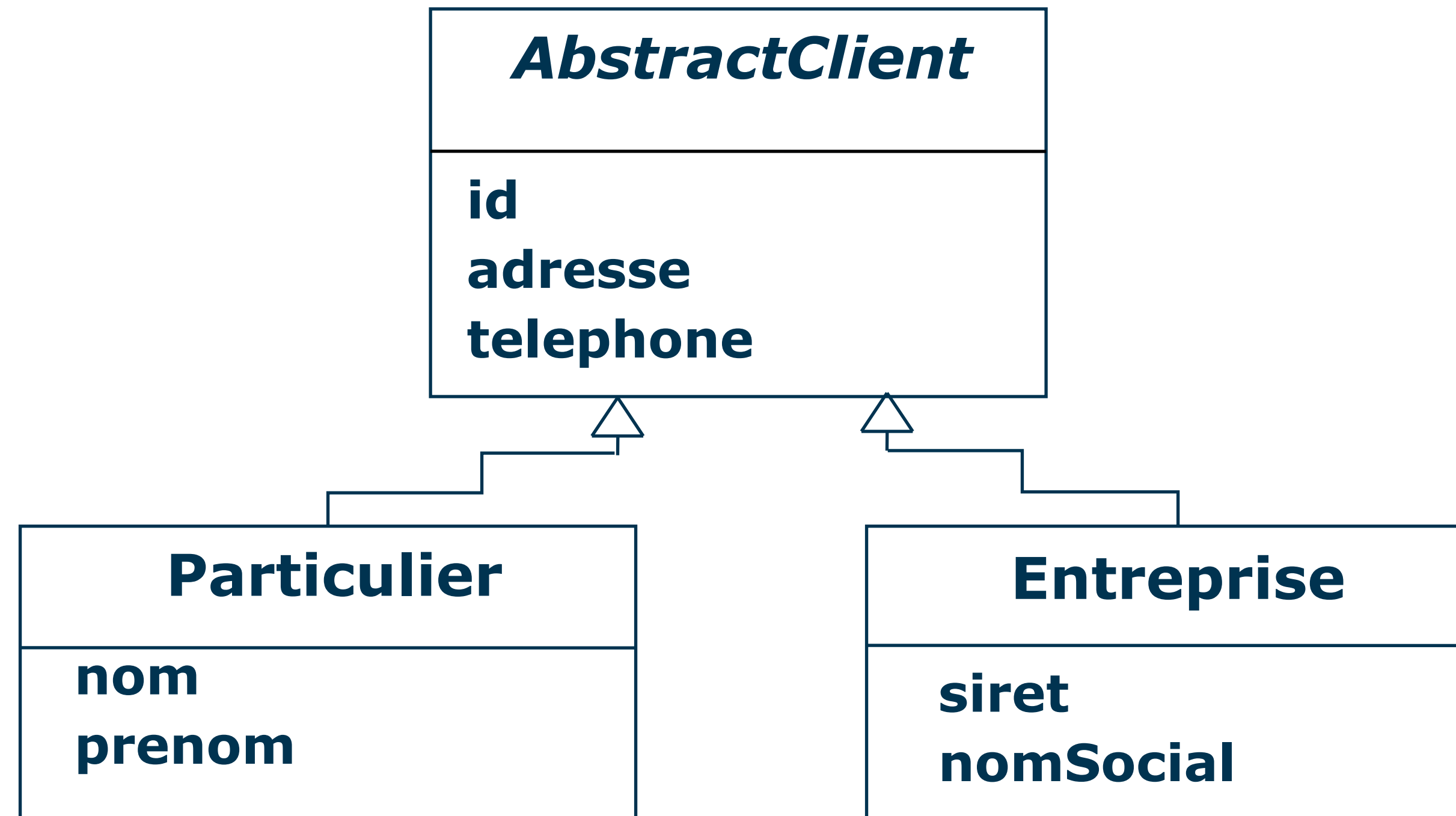
- ❑ L'héritage est l'un des 3 grands principes de l'objet parmi:
 - Abstraction
 - Héritage
 - Polymorphisme

- ❑ 3 stratégies de stockage permettent le mapping ORM de l'héritage
 - Chaque stratégie a ses avantages et ses inconvénients

- ❑ JPA permet l'utilisation de ces 3 stratégies générales

Héritage : Exemple

- ❑ Un Client peut être
 - soit un Particulier
 - soit une Entreprise



1 table par classe

- ❑ Appellation: « **Table per class** » ou "**Joined**"
- ❑ Idée générale
 - A chaque classe du modèle objet correspond une table
 - 3 tables dans l'exemple CLIENT-PARTICULIER-ENTREPRISE
 - Chaque table contient les attributs de l'objet + l'identifiant
 - L'héritage entre classes est modélisé par des clés étrangères (qui sont généralement aussi clés primaires)
- ❑ Conceptuellement
 - (+) Solution simple et efficace
- ❑ Techniquement
 - (-) Jointures lors de chaque requête à performances non optimales

1 table par classe

- ❑ 1 table pour l'objet AbstractClient
- ❑ 1 table pour l'objet Particulier
- ❑ 1 table pour l'objet Entreprise
- ❑ 2 clés étrangères pour représenter les deux relations d'héritage

ID	ADRESSE	PHONE
234	13 rue des ...	014565...
516	3 impasse ...	029867...
887	147 avenue ...	042356...
14	15 place du ...	032981...

ID	NOM	PRENOM
234	Martin	Patrice
887	Durant	Cédric

ID	SIRET	NSOC
14	98765...	S&B SA
516	12462...	InterFilm

clé primaire

clés étrangères

67

1 table par classe

```
/**
 * Client abstrait: Entreprise ou particulier.
 */
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class AbstractClient {

    @Id
    @GeneratedValue
    private Integer id;

    ...

}
```

```
/**
 * Modélise une entreprise.
 */
@Entity
public class Entreprise extends AbstractClient {

    private String numeroSiret;

    ..

}
```

```
/**
 * Un particulier.
 */
@Entity
public class Particulier extends AbstractClient {

    private String nom;
    private String prenom;

    ...

}
```

1 table par classe

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasse() {

    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();

    Entreprise e = new Entreprise(); // création d'une entreprise
    e.setAdresse("1");
    e.setTelephone("0141220300");
    e.setNomSocial("SQLi");
    e.setSiret("44019981800012");
    em.persist(e);

    Particulier p = new Particulier(); // création d'un particulier
    p.setAdresse("14 rue de la Foret");
    p.setTelephone("0112341234");
    p.setNom("Durant");
    p.setPrenom("Cédric");
    em.persist(p);

    transaction.commit();
}
```

1 table pour toute la hiérarchie

❑ Appellation: « SINGLE_TABLE »

❑ Idée générale

- Toutes les informations de toutes les classes sont stockées dans une seule et unique table
- Toutes les colonnes ne servent pas à toutes les classes, seules certaines colonnes sont utilisées par chaque classe
- Chaque enregistrement est « typé » avec un indicateur permettant de retrouver le type de l'instance → discriminateur (discriminator)

❑ Conceptuellement

- (+) Simple à mettre en place
- (-) Une solution faible, pas très évolutive

❑ Techniquement

- (-) « NOT-NULL » inutilisable sur les colonnes (problèmes d'intégrité)
- (-) Des enregistrements quasiment vides (espace perdu)
- (+) Pas de clés étrangères, pas de jointure au requêtage

1 table pour toute la hiérarchie

❑ Une seule table dans laquelle sont stockées

- ET les informations des Particuliers
- ET les informations des Entreprises

❑ La colonne TYPE contient le discriminateur

- P → Particulier E → Entreprise

ID	TYPE	ADRESSE	PHONE	NOM	PRENOM	SIRET	NSOC
234	P	13 rue des ...	014565...	Martin	Patrice	<i>null</i>	<i>null</i>
516	E	3 impasse ...	029867...	<i>null</i>	<i>null</i>	12462...	InterFilm
887	P	147 avenue ...	042356...	Durant	Cédric	<i>null</i>	<i>null</i>
14	E	15 place du ...	032981...	<i>null</i>	<i>null</i>	98765...	S&B SA

← clé primaire

1 table pour toute la hiérarchie

```
/**
 * Client abstrait: Entreprise ou particulier.
 */
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
public class AbstractClient {

    @Id
    @GeneratedValue
    private Integer id;

    ...
}
```

Nom de la colonne servant de discriminant

```
/**
 * Modélise une entreprise.
 */
@Entity
@DiscriminatorValue("E")
public class Entreprise extends AbstractClient {

    private String nomSocial;
    private String siret;

    ...
}
```

```
/**
 * Un particulier.
 */
@Entity
@DiscriminatorValue("P")
public class Particulier extends AbstractClient {

    private String nom;
    private String prenom;

    ...
}
```


1 table pour toute la hiérarchie

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasse() {

    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();

    Entreprise e = new Entreprise(); // création d'une entreprise
    e.setAdresse("1");
    e.setTelephone("0141220300");
    e.setNomSocial("SQLi");
    e.setSiret("44019981800012");
    em.persist(e);

    Particulier p = new Particulier(); // création d'un particulier
    p.setAdresse("14 rue de la Foret");
    p.setTelephone("0112341234");
    p.setNom("Durant");
    p.setPrenom("Cédric");
    em.persist(p);

    transaction.commit();
}
```

1 table par classe concrète

- ❑ Appellation: « Table per subclass »
- ❑ Idée générale
 - Chaque classe concrète est stockée dans une table différente
 - Duplication des colonnes dans le schéma pour les propriétés communes
- ❑ Théoriquement
 - (-) Équivaut à dire: « ne considérons pas la relation d'héritage, ce sont des classes différentes sans lien particulier entre elles »
- ❑ Techniquement
 - (-) Pas d'unicité globale des clés primaires sur toutes les tables
 - (+) Pas de problème de colonnes vides
 - (+) Pas de clés étrangères → pas de problème de jointure

1 table par classe concrète

- ❑ Client est abstraite → Pas de table
- ❑ Particulier est concrète → 1 table PARTICULIER
 - des colonnes sont ajoutées pour les attributs de Client
- ❑ Entreprise est concrète → 1 table ENTREPRISE
 - des colonnes sont ajoutées pour les attributs de Client

ID	ADRESSE	PHONE	NOM	PRENOM
234	13 rue des ...	014565...	Martin	Patrice
887	147 avenue ...	042356...	Durant	Cédric

ID	ADRESSE	PHONE	SIRET	NSOC
14	15 place du ...	032981...	12462...	S&B SA
516	3 impasse ...	029867...	98765...	InterFilm

clés
primaires

75

1 table par classe concrète

```
/**
 * Client abstrait: Entreprise ou particulier.
 */
@MappedSuperclass
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class AbstractClient {

    @Id
    private Integer id;
}
```

L'entité mère n'est pas mappée à une table. Les requêtes polymorphes sont rendus impossibles par cette annotation

```
/**
 * Modélise une entreprise.
 */
@Entity
public class Entreprise extends AbstractClient {

    private String nomSocial;
    private String siret;
}
```

```
/**
 * Un particulier.
 */
@Entity
public class Particulier extends AbstractClient {

    private String nom;
    private String prenom;
}
```

1 table par classe concrète

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasseConcrete() {

    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();

    Entreprise e = new Entreprise(); // création d'une entreprise
    e.setAdresse("1");
    e.setTelephone("0141220300");
    e.setNomSocial("SQLi");
    e.setSiret("44019981800012");
    em.persist(e);

    Particulier p = new Particulier(); // création d'un particulier
    p.setAdresse("14 rue de la Foret");
    p.setTelephone("0112341234");
    p.setNom("Durant");
    p.setPrenom("Cédric");
    em.persist(p);

    transaction.commit();
}
```

Chapitre 5 - Annexes

L'héritage – FAQ

Mixage des stratégies

- ❑ **Question:** Supposons que vous ayez une hiérarchie d'objet avec plusieurs couches et que vous souhaitiez mettre en œuvre des stratégies différentes. Est-ce possible ?
- ❑ **Réponse :** Non ce n'est pas possible. La stratégie s'applique à toute la hiérarchie définie par la classe mère.

Mise en commun d'attributs

- ❑ **Question:** Comment faire pour éviter par exemple de répéter l'attribut @Id dans chaque classe.
- ❑ **Réponse :** Il faut définir une classe mère qui portera l'attribut. Cette classe doit porter l'annotation @MappedSuperClass
- ❑ L'annotation @MappedSuperClass désigne une classe dont les informations de mapping s'appliquent à toutes les classes filles de type @Entity.

Peut-on contourner la 3^{ème} stratégie ?

- ❑ Une **@MappedSuperClass** n'est pas considérée comme une entité
- ❑ De ce fait il est **impossible** d'avoir une relation @OneToOne, @OneToMany, @ManyToOne ou @ManyToMany sur une @MappedSuperClass
- ❑ Exemple de cas qui ne fonctionne pas si AbstractClient porte l'annotation @MappedSuperClass

```
@Entity
@Table(name="COMPTE_BANCAIRE")
public class CompteBancaire {

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

    @OneToMany(mappedBy="compteBancaire")
    private List<AbstractClient> clients = new ArrayList<>();
```



Contournement pour la 3^{ème} stratégie

- ❑ Ne pas déclarer la classe **AbstractClient** comme une **@MappedSuperClass** mais comme une **@Entity**
- ❑ Mais dans ce cas il y a obligation d'utiliser la stratégie **GenerationType.Table** pour l'id.

```
@Entity
```

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

```
public class AbstractClient {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.TABLE)
```

```
    private Integer id;
```

La stratégie TABLE et la table hibernate_sequences

- ❑ Lorsque vous choisissez la stratégie TABLE pour gérer un identifiant, Hibernate crée automatiquement une table hibernate_sequences pour stocker la valeur de l'identifiant suivant:

	sequence_name	next_val
▶	default	1

- ❑ L'identifiant est géré par une **séquence** appelée **default**

Cas particuliers: plusieurs stratégies TABLE

- ❑ Dans le cas où plusieurs hiérarchies d'objets utilisent cette stratégie, il va falloir utiliser des séquences différentes
- ❑ **@TableGenerator** permet d'indiquer que la séquence **abstract_client** est stockée dans la table **hibernate_sequences**, sinon par défaut une nouvelle table est créée.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@TableGenerator(
    name="abstract_client",
    table="hibernate_sequences",
    allocationSize = 1,
    initialValue = 1
)
public class AbstractClient {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "abstract_client")
    private Integer id;
```

allocationSize : incrément

	sequence_name	next_val
▶	default	1
	abstract_client	2

Les requêtes polymorphes

❑ Exemple de requête polymorphe

```
TypedQuery<AbstractClient> query = em.createQuery("From AbstractClient",  
AbstractClient.class);
```

```
List<AbstractClient> clients = query.getResultList();
```

- ❑ Cette requête permet de récupérer tous les clients, que ce soit des particuliers ou des entreprises
- ❑ Hibernate gère les jointures entre différentes tables

Chapitre 6

Clé composite

Exemple de clé composite

@Entity

```
public class Project {  
    @EmbeddedId  
    private CleComposite id;  
}
```

@Embeddable

```
class CleComposite {  
    private int departmentId;  
    private long projectId;  
}
```

Chapitre 7

JPQI

Sommaire

JPQL: Pourquoi ?

Paramètre nommé

Les divers types de retour

Opérateur new

Clause JOIN

Update et DELETE

Utilisation des fonctions size et is (not) empty

JPQL: pourquoi ?

- ❑ La méthode **find** est vite limitée (recherche par id uniquement).
- ❑ JPQL, langage standardisée indépendant de la base de données
 - Proche du SQL
 - S'utilise avec les **entités JPA** et leurs **attributs**.

```
SELECT user FROM User user WHERE user.login LIKE :LOGIN
```

```
SELECT count(user) FROM User user
```

Paramètre nommé

❑ Paramètre nommé

```
Query query = em.createQuery("SELECT p FROM Person p WHERE p.name= :name");  
query.setParameter("name", "Doe");
```

❑ Pas de paramètre positionnel comme en JDBC

Les divers types de retour (1/2)

❑ Liste typée

```
TypedQuery<Person> query = em.createQuery("SELECT p FROM Person p WHERE p.age > 50", Person.class);  
List<Person> persons = query.getResultList();
```

❑ Résultat unique

```
TypedQuery<Person> query = em.createQuery("SELECT p FROM Person p WHERE p.id = ?1", Person.class);  
query.setParameter(1, 12);  
Person person = query.getSingleResult();
```

Les divers types de retour (2/2)

❑ Valeur unique

```
TypedQuery<Integer> query = em.createQuery("SELECT MAX(p.age) FROM Person p", Integer.class);  
Integer ageMax = query.getSingleResult();
```

❑ Ensemble de valeurs

```
TypedQuery<String> query = em.createQuery("SELECT p.firstName FROM Person p", String.class);  
List<String> firstNames = query.getResultList();
```

Opérateur new

- ❑ Permet d'instancier un objet dans une requête JPQL

```
Query query = em.createQuery("SELECT new fr.person.Customer(p.firstName, p.lastName) FROM Person p");  
List<Customer> customers = query.getResultList();
```

- ❑ Remarques

- La classe Customer **n'est pas forcément** une entité.
- Attention en cas de refactoring: **déconseillé**.

Clause JOIN

❑ Pour réaliser une jointure sur un attribut

❑ Exemple 1

```
Query query = em.createQuery("SELECT p FROM Person p JOIN p.address a WHERE a.city= 'Nantes'");  
List<Person> persons = query.getResultList();
```

❑ Exemple 2

```
Query query = em.createQuery("SELECT p FROM Person p WHERE p.address.city= 'Nantes'");  
List<Person> persons = query.getResultList();
```

Clause JOIN

❑ Pour réaliser une jointure sur une relation OneToMany (attribut de type liste)

❑ Exemple

```
Query query = em.createQuery("SELECT l FROM Livre l JOIN l.emprunts e WHERE e.delai > 5");  
List<Livre> livres = query.getResultList();
```

❑ On ne peut pas écrire ceci car délai n'est pas un attribut de la liste emprunts:

```
Query query = em.createQuery("SELECT l FROM Livre l WHERE l.emprunts.delai > 5");  
List<Livre> livres = query.getResultList();
```


UPDATE et DELETE

❑ Pour réaliser un update

```
Query query = em.createQuery("UPDATE Livre l SET l.titre= :newTitre WHERE l.titre=:oldTitre");  
query.setParameter("newTitre", "Germinal");  
query.setParameter("oldTitre", "Germinol");  
query.executeUpdate();
```

❑ Pour réaliser un delete

```
Query query = em.createQuery("DELETE FROM Livre l WHERE l.titre=:titre");  
query.setParameter("titre", "Germinol");  
query.executeUpdate();
```

FONCTIONS

❑ Fonction SIZE

```
Query query = em.createQuery("SELECT l FROM Livre l WHERE SIZE(l.emprunts)=0");  
List<Livre> livres = query.getResultList();
```

❑ Fonction IS (NOT) EMPTY

```
Query query = em.createQuery("SELECT l FROM Livre l WHERE l.emprunts IS EMPTY");  
List<Livre> livres = query.getResultList();
```

Atelier (TP)

OBJECTIFS : Utiliser JPQL

DESCRIPTION :

- Dans les TP n°6 vous allez devoir mettre au point des requêtes JPQL pour extraire des données de la base.

Annexe Cache L2

Sommaire

JPA et Cache L2

Cache L2 - dépendances

Cache L2 - configuration

Cache L2 – annotation des entités

Fichier ehcache.xml

Mise en cache – vérification programmatique

JPA et cache L2

- ❑ Par défaut JPA fonctionne avec un cache de niveau 1 associé à l'EntityManager et donc au contexte de persistance.
- ❑ JPA peut fonctionner avec un **cache de niveau 2**, associé à l'EntityManagerFactory, et donc actif durant toute la durée de vie de l'application.
- ❑ On va stocker dans ce cache prioritairement des informations de type "référentiel", i.e. qui changent peu.
- ❑ Exemples de données "référentiel":
 - *Une liste de salles de formation pour une application de gestion d'une société de formation.*
 - *La liste des départements français*
 - *La liste des pays du monde*
 - *etc.*

Cache L2 - dépendances

- ❑ La première étape est d'ajouter une dépendance dans le fichier pom.xml.
- ❑ Ici on choisit l'implémentation ehcache:

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-ehcache</artifactId>  
  <version>...</version>  
</dependency>
```

Cache L2 - configuration

❑ La seconde étape est d'activer le cache dans le fichier persistence.xml.

❑ Ici on choisit l'implémentation ehcache:

```
...  
<property name="hibernate.cache.use_second_level_cache" value="true" />  
<property name="jakarta.persistence.sharedCache.mode" value="ENABLE_SELECTIVE"/>  
<property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcache.EhCacheRegionFactory" />  
...
```

Seules les entités marquées de l'annotation @Cacheable sont prises en compte.

❑ Autre mode possible pour sharedCache.mode:

- NONE aucune entité en cache
- ALL toutes les entités en cache
- DISABLE_SELECTIVE toutes les entités sauf celles avec @Cacheable(false)
- UNSPECIFIED mode par défaut.

Cache L2 – annotation des entités

La dernière étape est d'annoter les entités avec une annotation particulière.

```
@Entity
@Table(name = "LIVRE")
@Cacheable
public class Livre {
    ...
}
```

Cache L2 – fichier ehcache.xml

❑ Il est possible de configurer le cache plus finement avec un fichier de configuration à placer dans src/main/resources.

❑ Dans ce cas une nouvelle propriété doit être ajoutée dans persistence.xml:

```
<property name="hibernate.cache.provider_configuration_file_resource_path" value="ehcache.xml"/>
```

❑ Exemple:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="ehcache.xsd"
    updateCheck="true" monitoring="autodetect" dynamicConfig="true">
```

```
    <defaultCache name="default" maxElementsInMemory="5000" eternal="true"
        overflowToDisk="false" diskPersistent="false"
        memoryStoreEvictionPolicy="LRU"/>
```

```
</ehcache>
```

Cache L2 – vérification programmatique

- ❑ Il est possible de vérifier si un objet est dans le cache ou non.

```
Cache cache = entityManagerFactory.getCache();  
boolean isInCache = cache.contains(Livre.class, 1);
```



Clé primaire (id)