digitalent
FORMATION

# Formation Java EE

## Java Persistence API

# JPA - Introduction

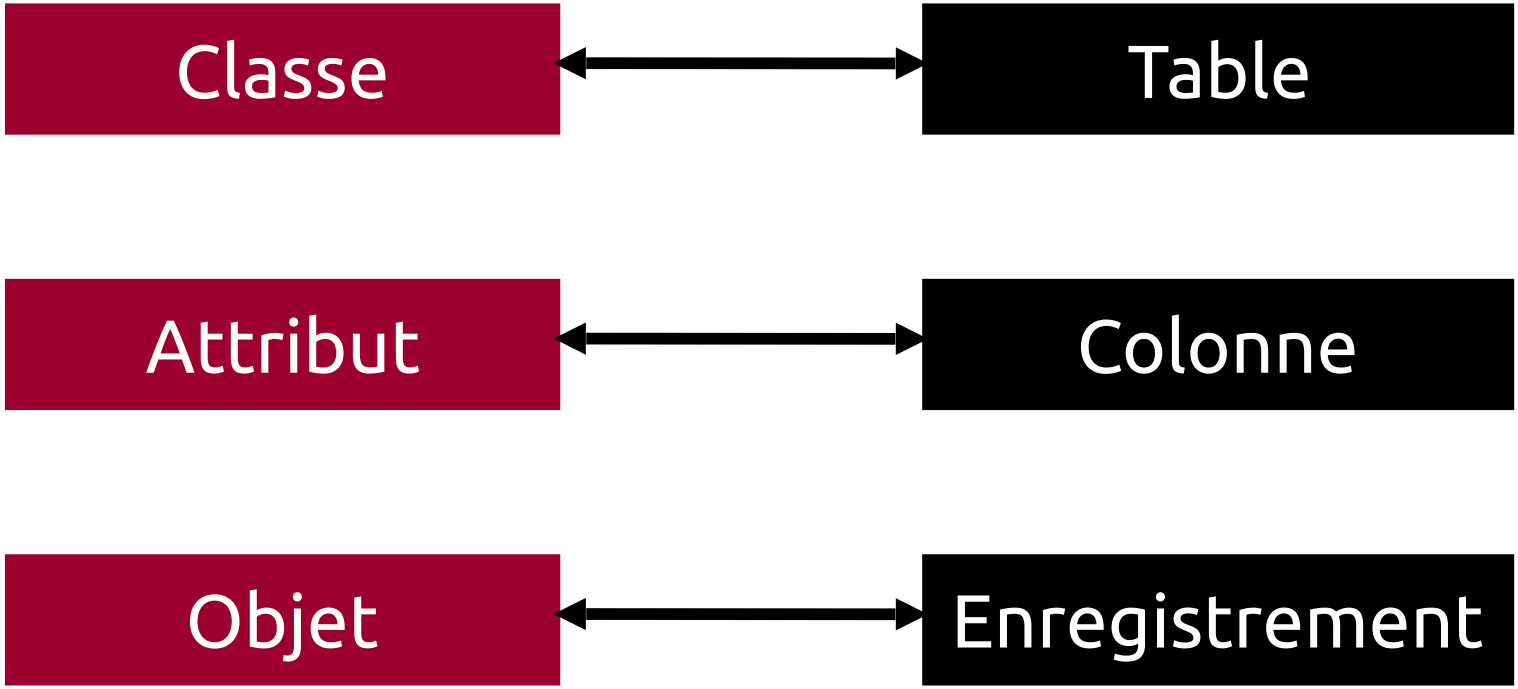
# Couche de persistance sans ORM

digitalent FORMATION

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 connexion
- 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 couteux"

# ORM - Principe



# ORM – Un peu d'histoire

digitalent FORMATION

1994 : TopLink, premier ORM au monde, en smallTalk
1996 : TopLink propose une version Java
1998 : EJB 1.0
2000 : EJB 2.0
2001 : Lancement d'Hibernate (alternative aux EJB 2)
2003 : Gavin King, créateur d'Hibernate, rejoint JBoss
2006 : EJB 3.0 et JPA 1.0
2007 : TopLink devient EclipseLink
2009 : JPA 2.0
2013 : JPA 2.1

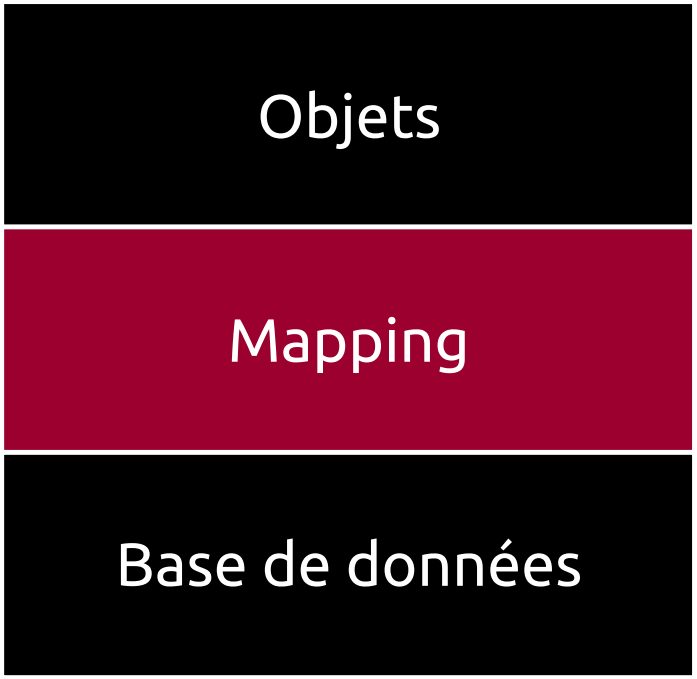
# Approches de mapping

**Bottom-up**

**Top-down**

**Meet in the middle**

Le plus fréquent



# JPA = Java Persistence API

digitalent FORMATION

Les entités persistantes sont des POJO

Les entités ne se persistent pas elles-mêmes : elle passent par l'EntityManager

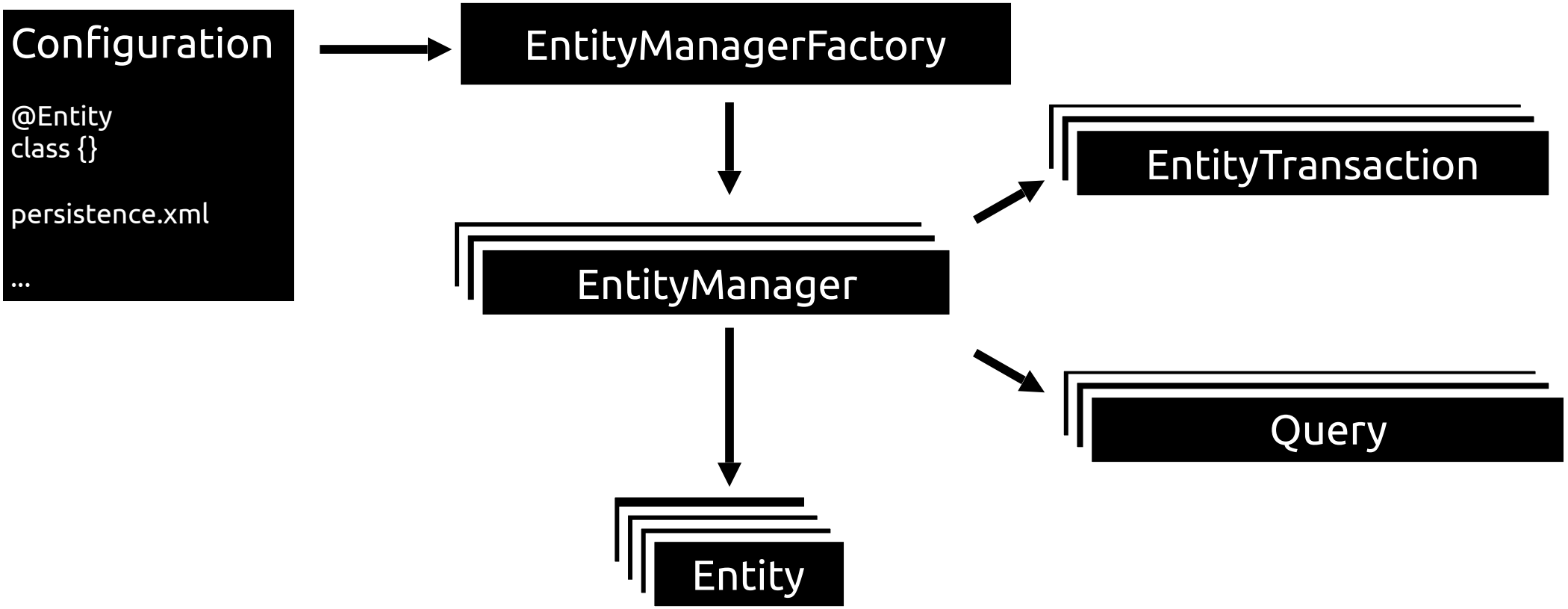
L'EntityManager propose des interfaces pour réaliser du CRUD basé sur un cycle de vie

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)

digitalent FORMATION

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation=http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/
persistence/persistence_2_1.xsd>


      <persistence-unit name="nantes-jpa" transaction-type="RESOURCE_LOCAL">
            <provider>org.hibernate.ejb.HibernatePersistence</provider>
            <properties>
                  <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:8889/nantes-bdd"/
>
                  <property name="javax.persistence.jdbc.user" value="nantes_user"/>
                  <property name="javax.persistence.jdbc.password" value="nantes_pass"/>
                  <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            </properties>
      </persistence-unit>


</persistence>
```

org.hibernate.jpa.HibernatePersistenceProvider

RESOURCE_LOCAL => utilisation en dehors d'un conteneur Java EE

# Configuration (persistence.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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"
             version="2.0">
   <persistence-unit name="manager1" transaction-type="JTA">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <mapping-file>ormap.xml</mapping-file>
     <class>fr.nantes.Employee</class>
      <class>fr.nantes.Person</class>
      <class>fr.nantes.Address</class>
      <properties>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

JTA => utilisation dans conteneur Java EE

# 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 sous jacente.

Une propriété encapsule les données d'un champ d'une table.

Elle est utilisable au travers de simple accesseurs (getter/setter).

Ce sont de simples POJO (Plain Old Java Object).

Un POJO n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.

# Conditions pour les classes entités

digitalent FORMATION

Un bean entité doit obligatoirement

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

# Exemple d'entité

@Entity Indique que la classe doit être gérée par JPA

@Table (facultatif) désigne la table à mapper

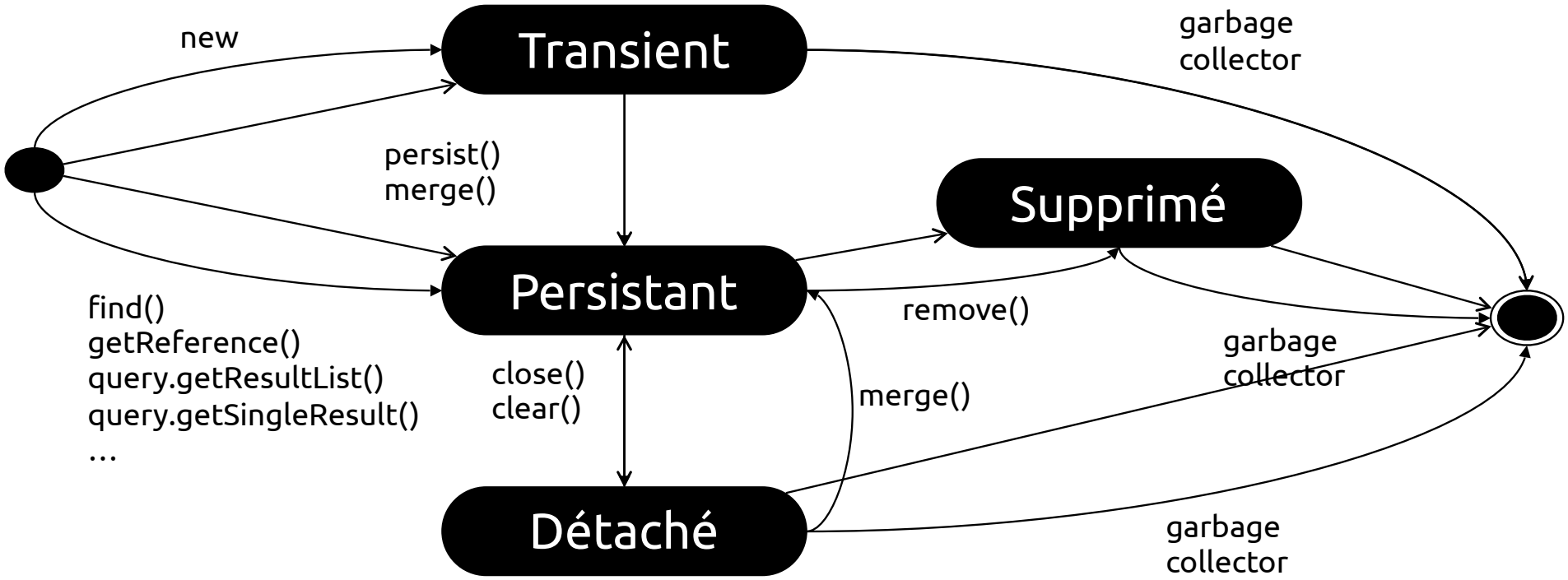
```
@Entity
@Table(name="personne")
public class Personne {
   @Id
   private Integer id;
   @Column(name = "NOM", length = 30, nullable = false, unique = true)
   private String nom;
   @Column(name = "PRENOM", length = 30, nullable = false)
   private String prenom;

   // constructeurs
   public Personne() {
   }

   // getters and setters
}
```

@Id Clé primaire

@Column Associer un champ à une colonne de la table

# JPA - CRUD

# Cycle de vie d'une entité



# Insertion

La méthode persist (…) de l'interface *javax.persistence.EntityManager* permet de demander l'enregistrement d'un objet

```
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();
```

# Récupérer une instance

La recherche par la clé primaire est faite avec les méthodes :

- **find()**: Renvoie null si l'occurrence n'est pas trouvée

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

- **getReference()**: Lève une exception javax.persistence.EntityNotFoundException si l'occurrence n'est pas trouvée

```
try {
      Hotel h = em.getReference(Hotel.class, 120);
} catch (EntityNotFoundException e) {
      //hôtel non trouvée
}
```

# Récupérer une instance

La recherche par requête est réalisée grâce aux:

- Méthodes createQuery(), createNamedQuery() et createNativeQuery() de EntityManager
- Langage de requêtes JPQL

```
Query query = em.createQuery("select h from Hotel h
                                        where h.nom='nom2'");
Hotel h1 = (Hotel) query.getResultList().get(0);
```

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

Hotel h2 = query2.getResultList().get(0);
```

Faire plutot celle-ci

Ca evite un cast

# Modifier une instance

Toutes les requetes --> devienent des objets persistants.

Pour modifier un objet il faut

1. Le lire
   - Récupération de l'objet en lecture/écriture
2. Le modifier
   - Aucune méthode particulière n'est invoquée lors de la modification d'un objet !

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

# Fusionner les données

```
Query query = em.createQuery("select h from Hotel h where
h.nom='nom2'");

Hotel hotel = (Hotel) query.getSingleResult();

if(hotel != null){
    Hotel hotel2 = new Hotel();
    hotel2.setId(hotel.getId());
   hotel2.setNom(hotel.getNom());
   hotel2.setVille("nouvelle ville");
   em.merge(hotel2);
}
```

# Supprimer les données

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

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

# 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 automatiquement. Contient plusieurs attributs

- Strategy : Précise le type de générateur à utiliser : TABLE, SEQUENCE, IDENTITY ou AUTO. La valeur par défaut est AUTO
- Generator : nom du générateur à utiliser

on a a mettre IDENTY quelques parts

# Générateurs d'identifiants

Des implémentations sont fournies

Les types possibles sont les suivants

- AUTO: laisse l'implémentation générer la valeur de la clé primaire. Cette valeur est unique pour toute la base de données.
- IDENTITY: similaire à AUTO sauf que la valeur générée est unique par type.
- TABLE: utilise une table dédiée qui stocke les clés des tables générées. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.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 @javax.persistence.SequenceGenerator

# 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

Permet de dire que cette propriété ne correspond à rien en base

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

Ca permet de ne pas mapper le champs.

@Enumerated : utiliser pour mapper des énumération de type entier ou chaine de caractères

# Transaction avec JPA

digitalent FORMATION

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

...
em.persist(p1);
...
em.persist(p2);
...
et.commit();// et.rollback()
em.close();
```

# @NamedQuery

digitalent FORMATION

```
@Entity
@NamedQuery(name="person.findByLastname", query="select p from Person p
where p.lastname=:name")
public class Person {

@Entity
@NamedQueries({
    @NamedQuery(name="person.findByLastname", query="select p from
Person p where p.lastname=:name"),
    @NamedQuery(name="person.findByFirstname", query="select p from
Person p where p.firstname=:name")
})
public class Person {
```

# @NamedQuery

Pour être d'avoir les noms --> on peu faire deja des prérequetes des requetes... Ca permet de gagner du temps.

```
public Person findByLastname(String lastname) {
        return em.createNamedQuery("person.findByLastname",
                    Person.class)
                .setParameter("name", lastname)
                .getSingleResult();
}
```

# JPA - Relations

# Relations

Une relation peut-être unidirectionnelle ou bidirectionnelle.

Les associations entre entités correspond à une jointures 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>

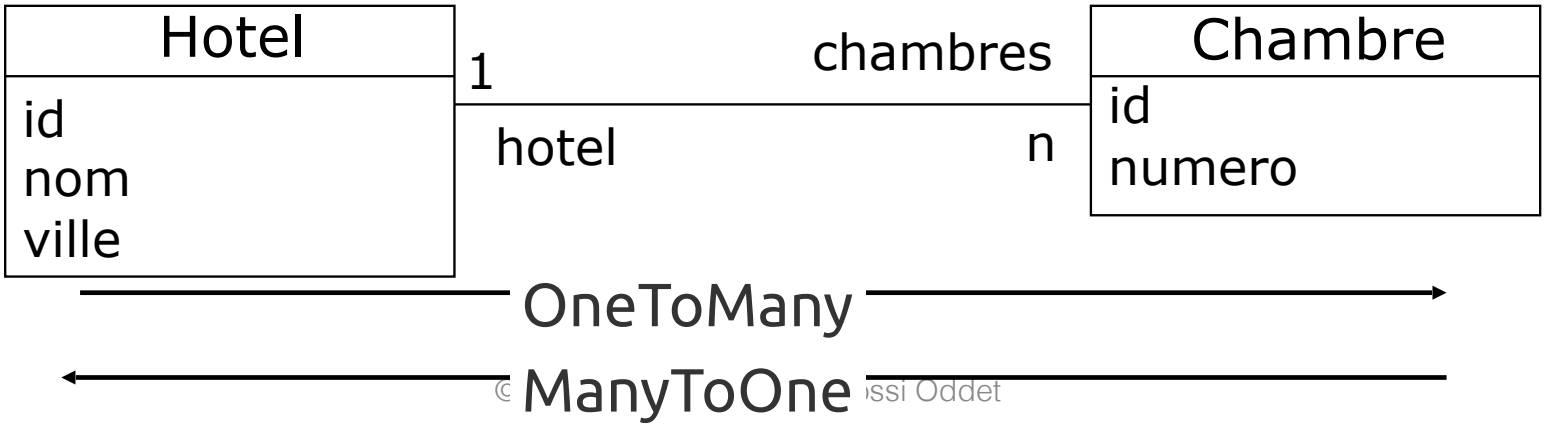
Attributs :

- 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



# @OneToMany

digitalent FORMATION

```
public class Hotel {
    private long id;

    @OneToMany(mappedBy="hotel")
    private Set<Chambre> chambres; // référence vers
                                   les chambres
    public Hotel() {
        chambres = new HashSet<Chambre>();
    }
    …
}
```

# @ManyToOne

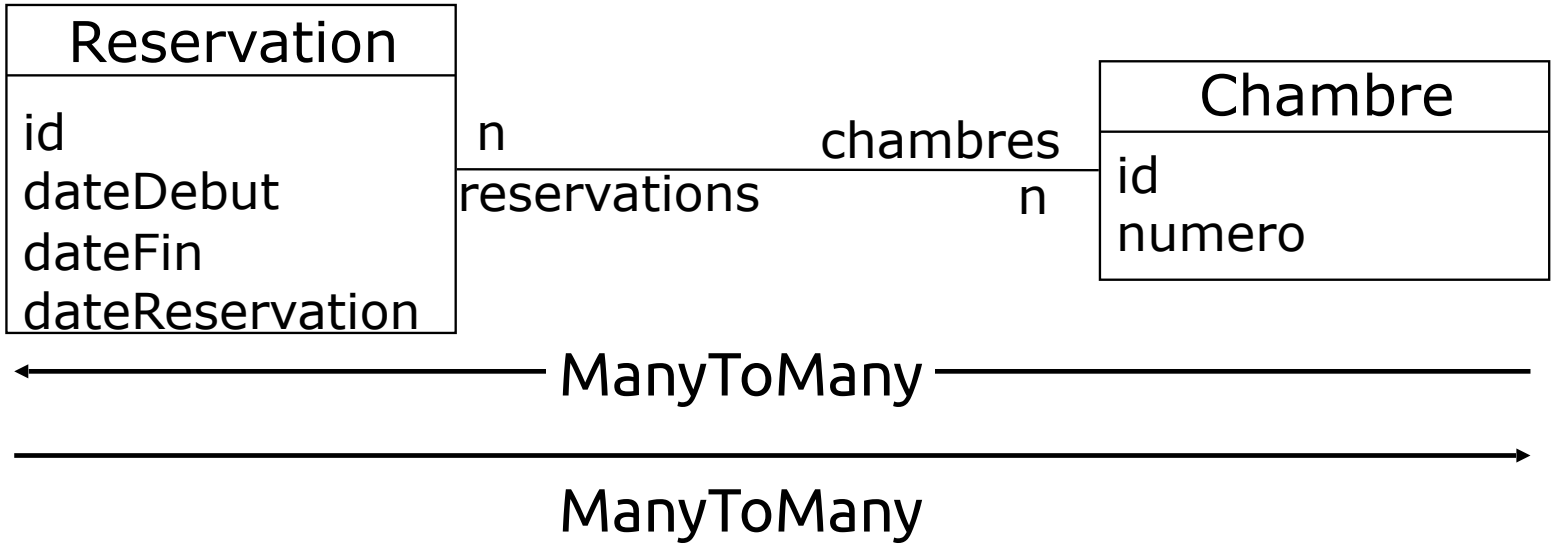
```
public class Chambre {

    private int id;

    @ManyToOne
    @JoinColumn(name="HOT_ID")
    private Hotel hotel;

}
```

==colonne de jointure

# Relation n-n

Exemple

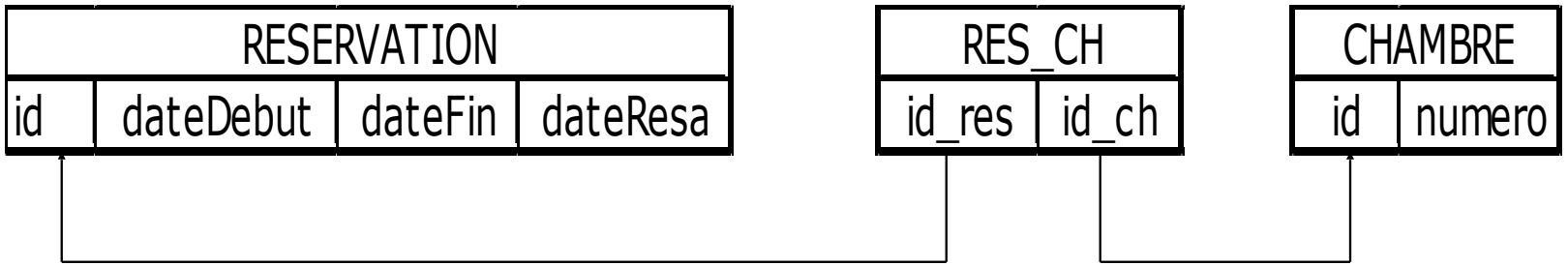
- Une Reservation peut concerner plusieurs Chambre
- Une Chambre peut faire l'objet de plusieurs Reservation



# 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



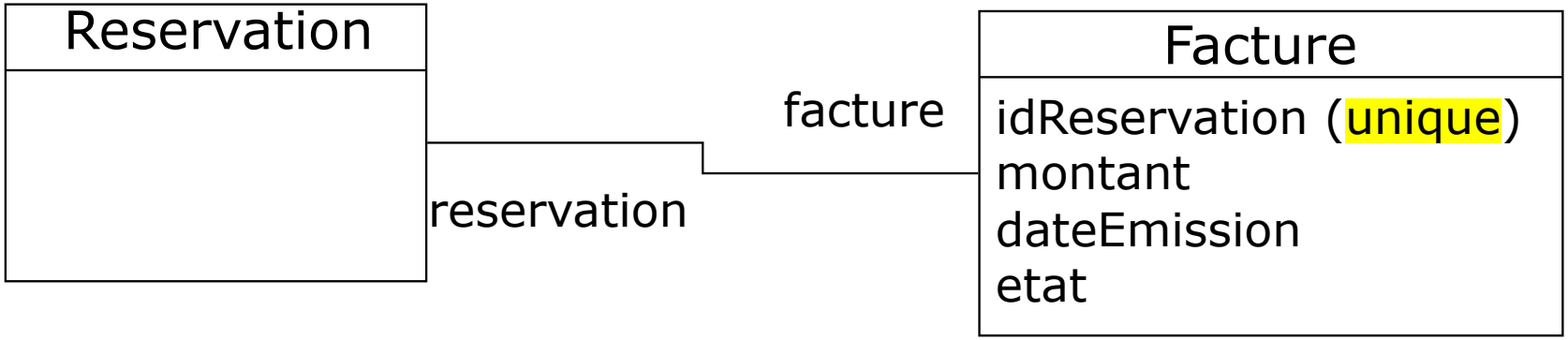
# @ManyToMany

Par defaut il est en mode lazy

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

# Relation 1-1

evite d'avoir une facture pour une meme reservation



# @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) { ...}
    }
```

# @Embeddable, @Embedded

# @Embeddable, Embedded

```
@Embeddable
public class NomComplet {
    private String nom;
    private String prenom;
}

public class Personne {

    @Embedded
    private NomComplet nomComplet;
}
```

sans auqu'un lien d'héritage

on reutilise la table --> il n'y a pas de lien

On peut modéliser une façon afin de creer un Héritage

# Héritage

# Héritage

L'héritage est l'un des 3 grands principes de l'objet

- Encapsulation
- Héritage
- Polymorphisme

3 stratégie pour mapper

3 stratégies permettent le mapping O/R de l'héritage

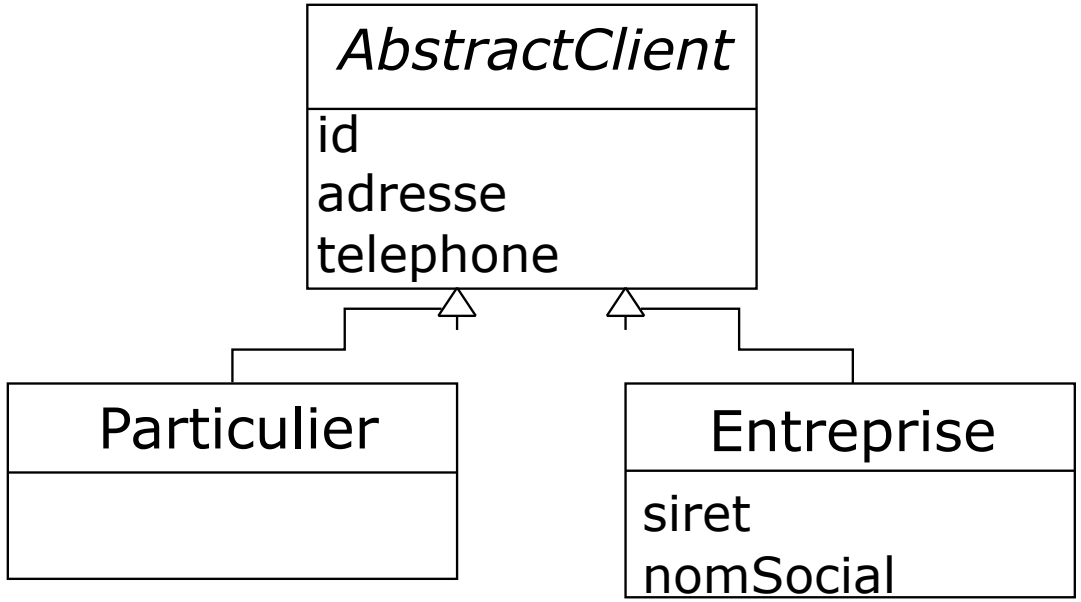
- Chaque stratégie a ses avantages et ses inconvénients

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

Statégie n°1

Une table par classe

Appellation: « Table per subclass »

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

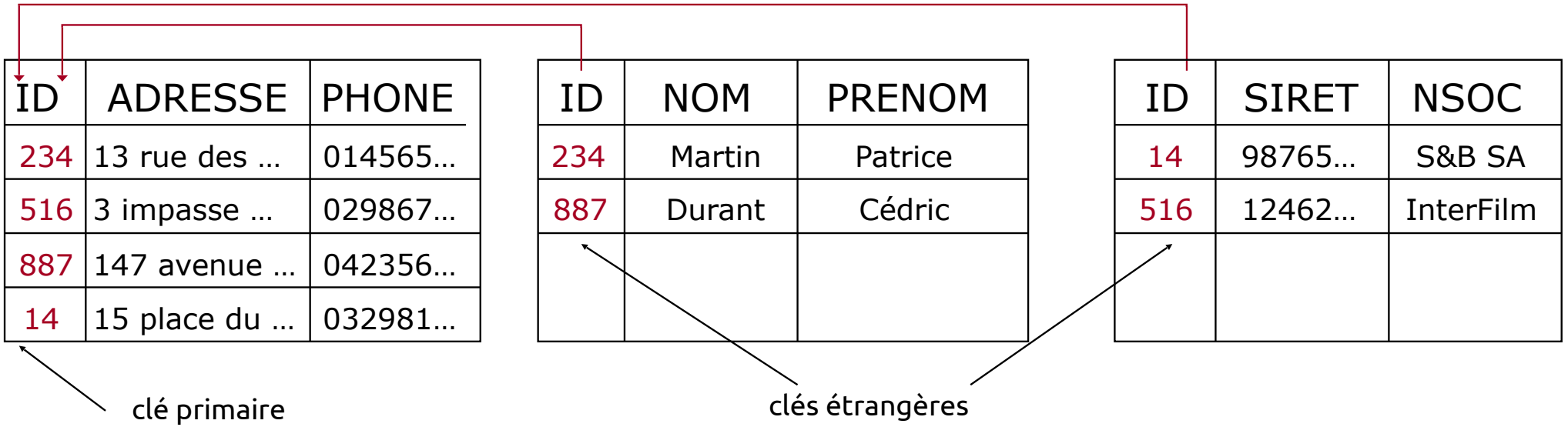
- (+) Solution simple et efficace

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



# 1 table par classe

```java
/**
 * Client abstrait: Entreprise ou particulier.
 */
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class AbstractClient {
            /**
             * L'identifiant
             */
            @Id
            @GeneratedValue
            private Integer id;
...
}
```

```java
/**
 * Un particulier.
 *
 */
@Entity
public class Particulier extends AbstractClient {
            /**
             * Le nom.
             */
            private String nom;
...
}
```

```java
/**
 * Une entreprise.
 *
 */
@Entity
public class Entreprise
             extends AbstractClient {
            /**
             * La raison sociale.
             */
            ...
}
```

# 1 table par classe

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasse() {
            Entreprise e = new Entreprise(); // création d'une entreprise
            e.setAdresse("1");
            e.setTelephone("0141220300");
            e.setNomSocial("SQLi");
            e.setSiret("44019981800012");
            EntityManager em = emf.createEntityManager();
            EntityTransaction transaction = em.getTransaction();
            transaction.begin();
            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();
            Entreprise e2 = (Entreprise) em.find(AbstractClient.class, 3);
            Particulier p2 = (Particulier) em.find(AbstractClient.class, 4);
            LOGGER.debug("Client: " + p2);
            LOGGER.debug("Client: " + e2);
}
```

Les informations seront bien répartit

# 1 table pour toute la hiérarchie

Statégie n°2 : Une table pour tout garder

## 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




Solution intéréssante quand les données sont déja mapper de cette manière.

# 1 table pour toute la hiérarchie

digitalent FORMATION

## 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 | *null* | *null* |
| 516 | E | 3 impasse … | 029867… | *null* | *null* | 12462… | InterFilm |
| 887 | P | 147 avenue … | 042356… | Durant | Cédric | *null* | *null* |
| 14 | E | 15 place du … | 032981… | *null* | *null* | 98765… | S&B SA |

clé primaire

# 1 table pour toute la hiérarchie

```java
/**
 * Client abstrait: Entreprise ou particulier.
 */
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
public class AbstractClient {
            /**
             * L'identifiant
             */
            @Id
            @GeneratedValue
            private Integer id;
...
}
```

Nom de la colonne discriminante

Nom de la colonne servant de discriminant

```java
/**
 * Modélise une entreprise.
 *
 */
@Entity
@DiscriminatorValue("E")
public class Entreprise extends AbstractClient {
    /**
     * La raison sociale.
     */
    private String nomSocial;
```

```java
/**
 * Un particulier.
 *
 */
@Entity
@DiscriminatorValue("P")
public class Particulier extends
                                AbstractClient {
    /**
     * Le nom.
     */
     private String nom;
```

# 1 table pour toute la hiérarchie

digitalent FORMATION

```
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasse() {
            Entreprise e = new Entreprise(); // création d'une entreprise
            e.setAdresse("1");
            e.setTelephone("0141220300");
            e.setNomSocial("SQLi");
            e.setSiret("44019981800012");
            EntityManager em = emf.createEntityManager();
            EntityTransaction transaction = em.getTransaction();
            transaction.begin();
            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();
            Entreprise e2 = (Entreprise) em.find(AbstractClient.class, 3);
            Particulier p2 = (Particulier) em.find(AbstractClient.class, 4);
            LOGGER.debug("Client: " + p2);
            LOGGER.debug("Client: " + e2);
}
```

# 1 table par classe concrète

Statégie n°3 : Une table par classe concrête

Appellation: « Table per class »

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

# 1 table par classe concrète

```java
/**
 * Client abstrait: Entreprise ou particulier.
 */
@MappedSuperclass
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class AbstractClient {
    /**
     * L'identifiant
     */
    @Id
    private Integer id;
```

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

```java
/**
 * Modélise une entreprise.
 *
 */
@Entity
public class Entreprise extends AbstractClient {
   /**
    * La raison sociale.
    */
    private String nomSocial;
```

```java
/**
 *Un particulier.
 *
 */
@Entity
public class Particulier extends AbstractClient {
    /**
     * Le nom.
     */
    private String nom;
```

# 1 table par classe concrète

```java
/**
 * Test de la persistance de l'héritage avec une table par classe.
 */
public void testUneTableParClasseConcrete() {
    Entreprise e = new Entreprise(); // création d'une entreprise
    e.setId(3);
    e.setAdresse("1");
    e.setTelephone("0141220300");
    e.setNomSocial("SQLi");
    e.setSiret("44019981800012");
    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();
    em.persist(e);
    Particulier p = new Particulier(); // création d'un particulier
    e.setId(3);
    p.setAdresse("14 rue de la Foret");
    p.setTelephone("0112341234");
    p.setNom("Durant");
    p.setPrenom("Cédric");
    em.persist(p);
    transaction.commit();
    Entreprise e2 = em.find(Entreprise.class, 3);
    Particulier p2 = em.find(Particulier.class, 4);
    LOGGER.debug("Client: " + p2);
    LOGGER.debug("Client: " + e2);
}
```

On peut plus de faire une requête sur toute les clients... exemple ~~Select * From Client~~

# Clé composite

# Clés composites

Les clés composites peuvent être gérées de deux manières :

- @IdClass
  - non recommandé
  - requêtes JPQL plus simple
- @EmbeddedId
  - dans les requêtes JPQL, il faut passer systématiquement par la classe représentant la clé composite

# Exemple de clé composite

digitalent FORMATION

```
@Entity
public class Project {
    @EmbeddedId ProjectId id;
}

@Embeddable
class ProjectId {
    int departmentId;
    long projectId;
}
```

# Intercepteurs JPA

# Evénements

Les intercepteurs JPA donnent la possibilité d'ajouter des traitements transverses dans le cycle de vie des entités.

Les annotations utilisées :

```
@PrePersist
@PostPersist
@PostLoad
@PreUpdate
@PostUpdate
@PreRemove
@PostRemove
```

# Evénement au sein d'une entité

Permet de faire une action à chaque moment d'un....

```
@Entity
public class PizzaOrder {

    @PrePersist private void onPrePersist(){}
    @PostPersist private void onPostPersist(){}
    @PostLoad private void onPostLoad(){}
    @PreUpdate private void onPreUpdate(){}
    @PostUpdate private void onPostUpdate(){}
    @PreRemove private void onPreRemove(){}
    @PostRemove private void onPostRemove(){}
```




On a une pizza, a chaque fois que l'on veut le afficher, on veut stocker la pizza... on fait un traitement...

# Ecouteurs externes

digitalent FORMATION

```
public class AuditInterceptor {
        @PrePersist private void onPrePersist(Object o){}
        @PostPersist private void onPostPersist(Object o){}
        @PostLoad private void onPostLoad(Object o){}
        @PreUpdate private void onPreUpdate(Object o){}
        @PostUpdate private void onPostUpdate(Object o){}
        @PreRemove private void onPreRemove(Object o){}
        @PostRemove private void onPostRemove(Object o){}
}

@Entity
@EntityListeners({AuditInterceptor.class, LogInterceptor.class})
public class Customer {
    @Id @GeneratedValue private Long id;
```

>Exemple :si on veut logger a chaque fois q'une pizza est créé dans la base.