



JPA

Java Persistence API



Nantes – 8 et 9 juillet 2015



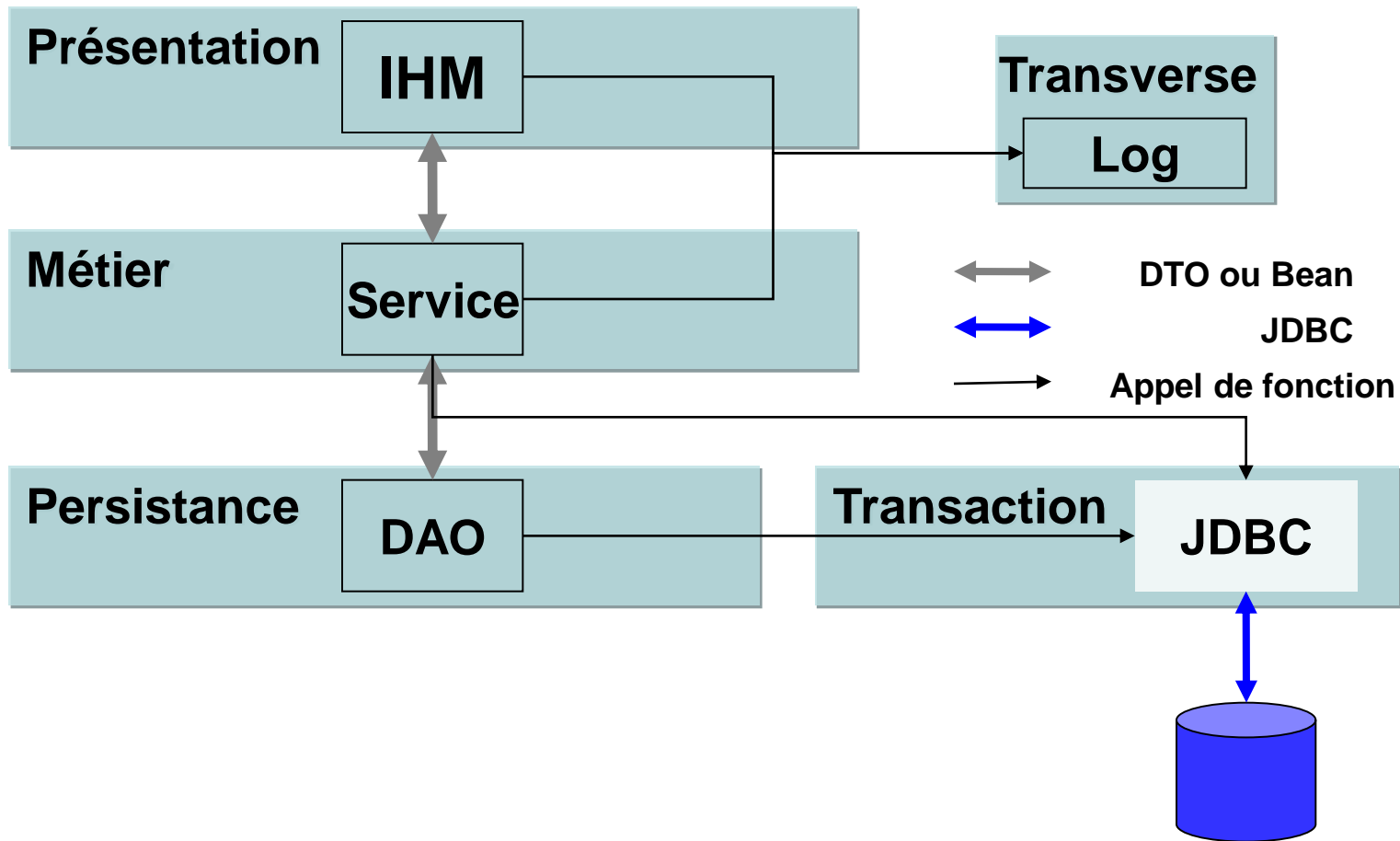
Fabien GUIBERT
Consultant / Architecte JEE
NEOBJECT SARL – www.neobject.fr

- ☐ Qui êtes-vous ?
- ☐ Pourquoi avoir choisi cette formation ?
- ☐ Quels conseils, quelles informations et compétences souhaitez-vous obtenir grâce à cette formation ?
- ☐ Les objectifs cités correspondent-ils à vos attentes ?
- ☐ Quelles sont les compétences qui vous semblent nécessaires pour suivre ce cours ?
- ☐ Êtes vous venu(e) avec des questions ?

- ☐ **Maitriser les concepts élémentaires des Entity Beans**
- ☐ **Maitriser l'exploitation du cycle de vie et de la transaction, le requêtage et le CRUD**
- ☐ **Mettre en œuvre un environnement JPA**
- ☐ **Mettre en Œuvre une couche performante d'accès aux données, basée sur le standard JPA**
- ☐ **Comprendre le Mapping Riche et les extensions**
- ☐ **Mettre en Œuvre un Mapping Riche**

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- 1. Introduction JPA : enjeux de la persistance et de l'ORM**
- 2. Les Entity Beans et le mapping ORM**
- 3. Opération CRUD avec l'EntityManager**
- 4. Relations 1-1, 1-N, N-1, N-M**
- 5. Héritage et polymorphisme**
- 6. Utilisation des collections et des Map**
- 7. Bidirectionnalité des relations**
- 8. Clés primaires composées**
- 9. Etats des objets et contexte de persistance**
- 10. Types de chargement : Lazy et Eager loading**
- 11. Langage de requête JPQL**
- 12. Gestion des transactions avec et sans JTA**
- 13. Gestion du cache Niveau 1 et 2**
- 14. Bonnes pratiques : performances JPA 2.1**

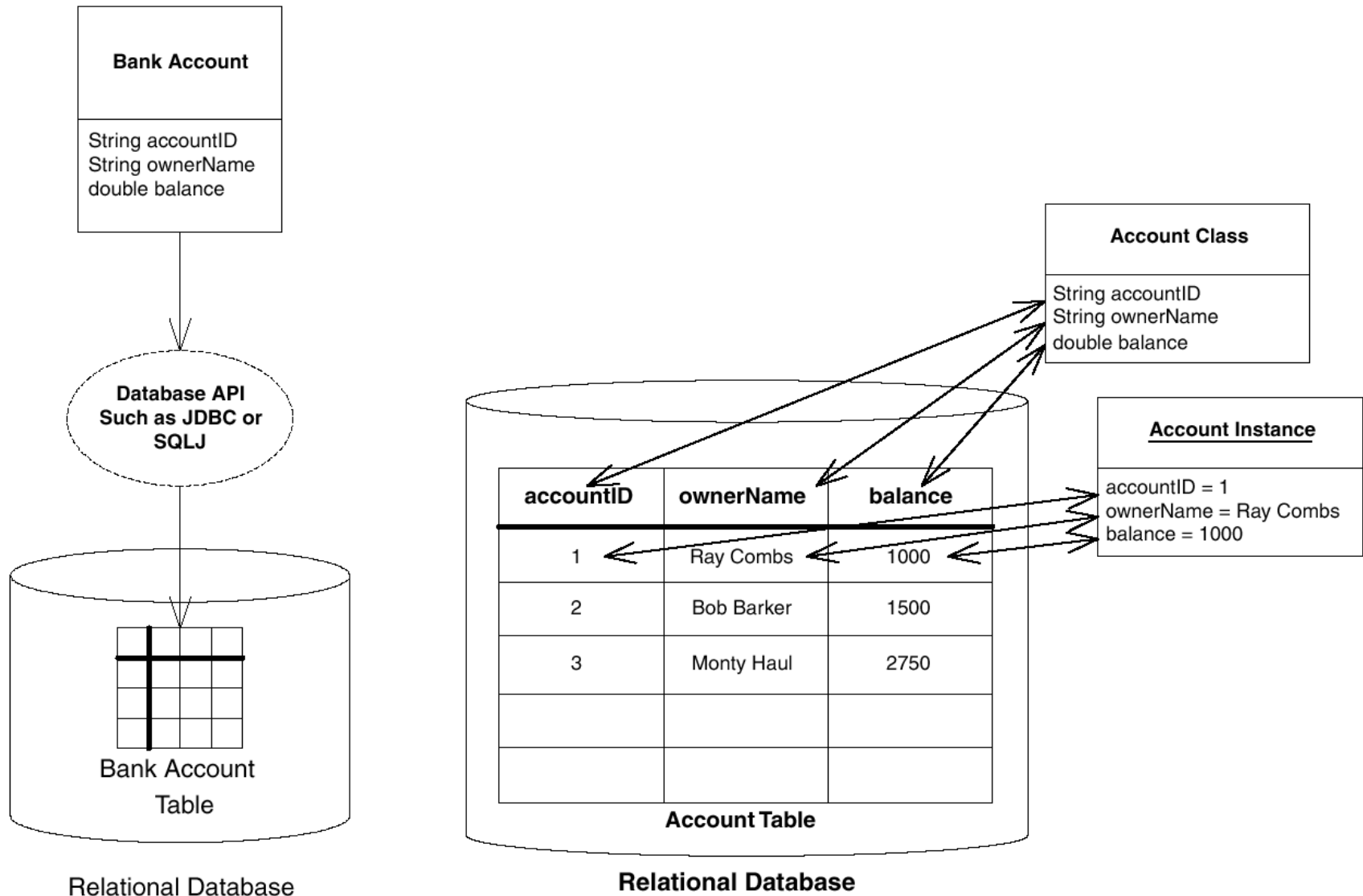


- Le développeur a la possibilité de coder la persistance des objets
- Toutes les étapes d'utilisation de JDBC apparaissent clairement dans le code des classes
 - La gestion des connexions
 - L'écriture manuelle des requêtes
- Plus de dix lignes de codes sont nécessaires pour rendre persistante une classe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

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

- ❑ **Défauts nombreux...**
- ❑ **Gestion des versions, maintenance...**
- ❑ **Pas de requêtes complexes...**
 - Ex : on sérialize mille comptes bancaires. Comment retrouver ceux qui ont un solde négatif ?
- ❑ **Solution : stocker les objets dans une base de donnée!**

- ❑ On stocke l'état d'un objet dans une base de donnée.
- ❑ Ex : la classe **Personne** possède deux attributs **nom** et **prenom**, on associe cette classe à *une table* qui possède deux colonnes : **nom** et **prenom**.
- ❑ On décompose chaque objet en une suite de variables dont on stockera la valeur dans une ou plusieurs tables.
- ❑ Permet des requêtes complexes.



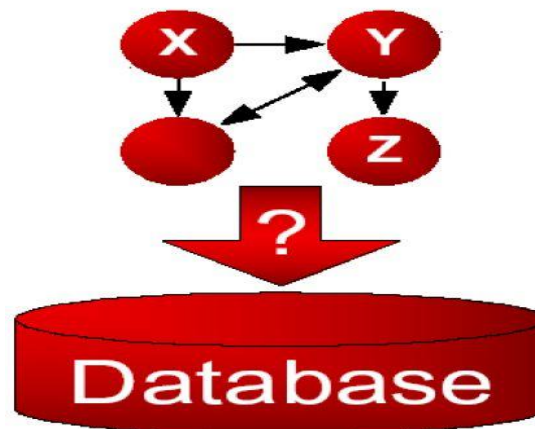
- **Persistance automatisée et transparente d'objets métiers vers une bases de données relationnelles**
- **Capacité à manipuler des données stockées dans une base de données relationnelles à l'aide d'un langage de programmation orienté-objet**
- **Techniques de programmation permettant de lier les bases de données relationnelles aux concepts de la programmation OO pour créer une "base de données orientées-objet virtuelle"**
- **Description à l'aide de métadonnées de la transformation réversible entre un modèle relationnel et un modèle de classes**

◦ Modèle relationnel

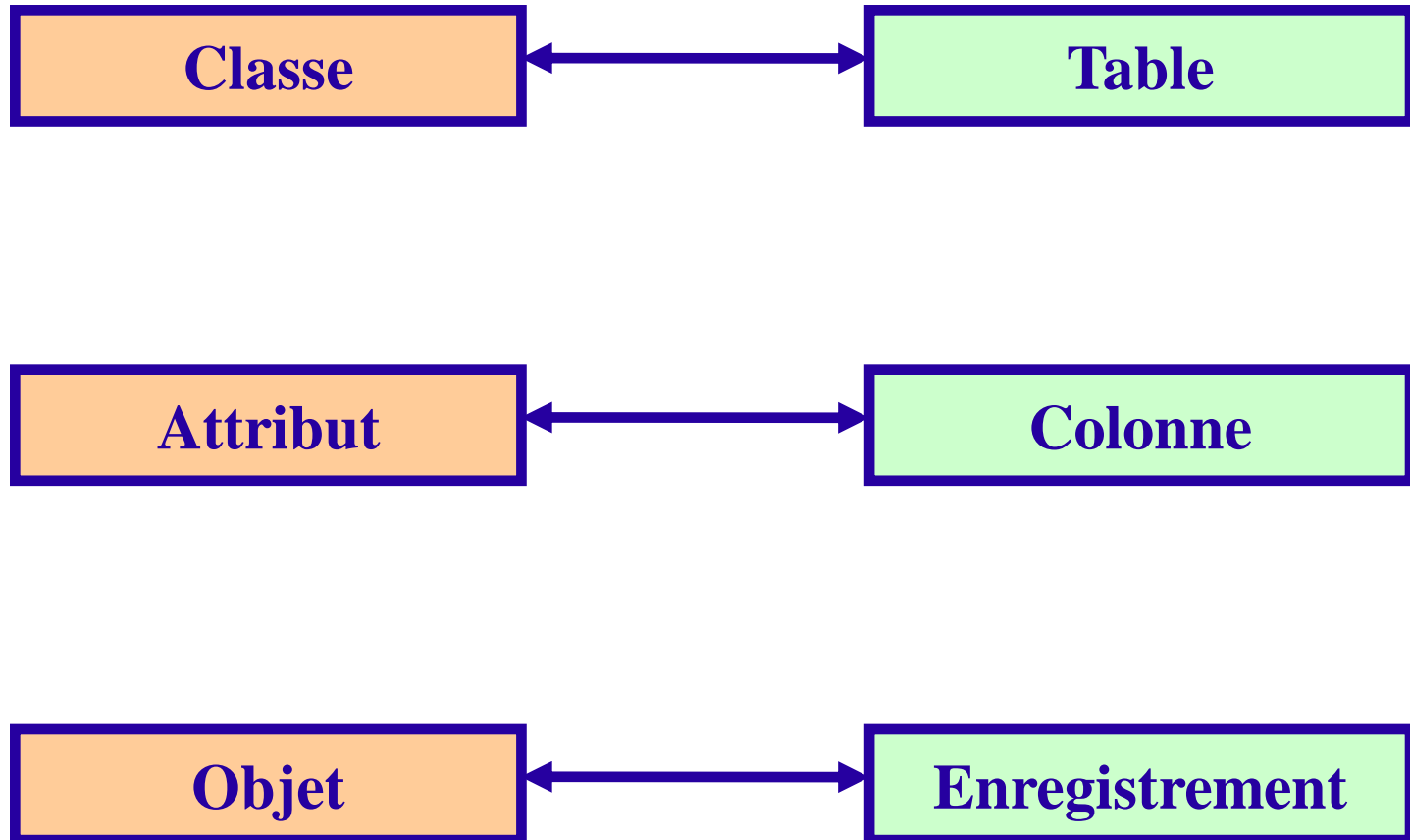
- ▶ Basé sur des principes mathématiques
- ▶ Pas de support natif des concepts objets (ex: polymorphisme)
- ▶ Mode de consultation: rapprochement des lignes de tables par jointure

◦ Modèle Objet

- ▶ Basé sur des principes de Génie Logiciel
- ▶ Mode de consultation: parcours de graphes d'instances



- **Objet = données (attributs) + comportement (méthodes)**
- **Enregistrement = données**
- **Les relations entre objets sont de différentes natures**
 - composition
 - Héritage
- **Un modèle objet est plus informatif qu'un modèle relationnel**
 - le passage « objet vers relationnel » nécessite de faire un choix parmi plusieurs solutions
 - le passage « relationnel vers objet » n'est pas souvent pertinent car le modèle objet résultant est dénaturé (contrainte, trigger..)



- **EJB 1.0, 1.1 et 2.0: Entity Beans**

- Incapacité de gérer les relation entre entités
- Performances
- Complexité d'utilisation
- Courbe d'apprentissage difficile à gravir

- **TopLink (1990)**

- Solution propriétaire et payante (WebGain puis Oracle)
- Puissance relationnelle
- Plus flexible et plus efficace que les EJB

- **JDO: Java Data Object (1999)**

- Abstraction du support de stockage
- Nouvelle logique d'interrogation éloignée du SQL

- **Hibernate**

- Framework de persistance Objet focalisé sur les bases de données relationnelles développé par Gavin King en 2001
- Implémenté en Java et possède des API Java
- Compatibilité avec tous les serveurs d'application J2EE du marché
- Compatibilité avec la plupart des SGBDR

- ❑ **199*** :
 - Développement de TopLink en SmallTalk par «The Object People»
- ❑ **1996-1998**
 - Ajout d'une version Java pour TopLink
- ❑ **2000**
 - Acquisition par WebGain
- ❑ **2002**
 - Acquisition par Oracle
- ❑ **2003-2004**
 - TopLink obtient plusieurs récompenses
- ❑ **2006**
 - Donation du code source au projet open-source « GlassFish » de java.net (Sun Microsystems)
 - Le projet est nommé « TopLink Essentials »
 - Le projet devient l'implémentation de référence de Java EE EJB 3.0 JPA (JSR 220)
- ❑ **2007**
 - Donation du code source pour la création du projet open-source « **EclipseLink** »
- ❑ **2008**
 - **EclipseLink** est sélectionné pour devenir l'implémentation de référence de JPA 2.0 (JSR 317)

❑ Annotations supplémentaires

- Beaucoup d'équivalents aux extensions Hibernate
- Liste complète :
[http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_\(ELUG\)](http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG))

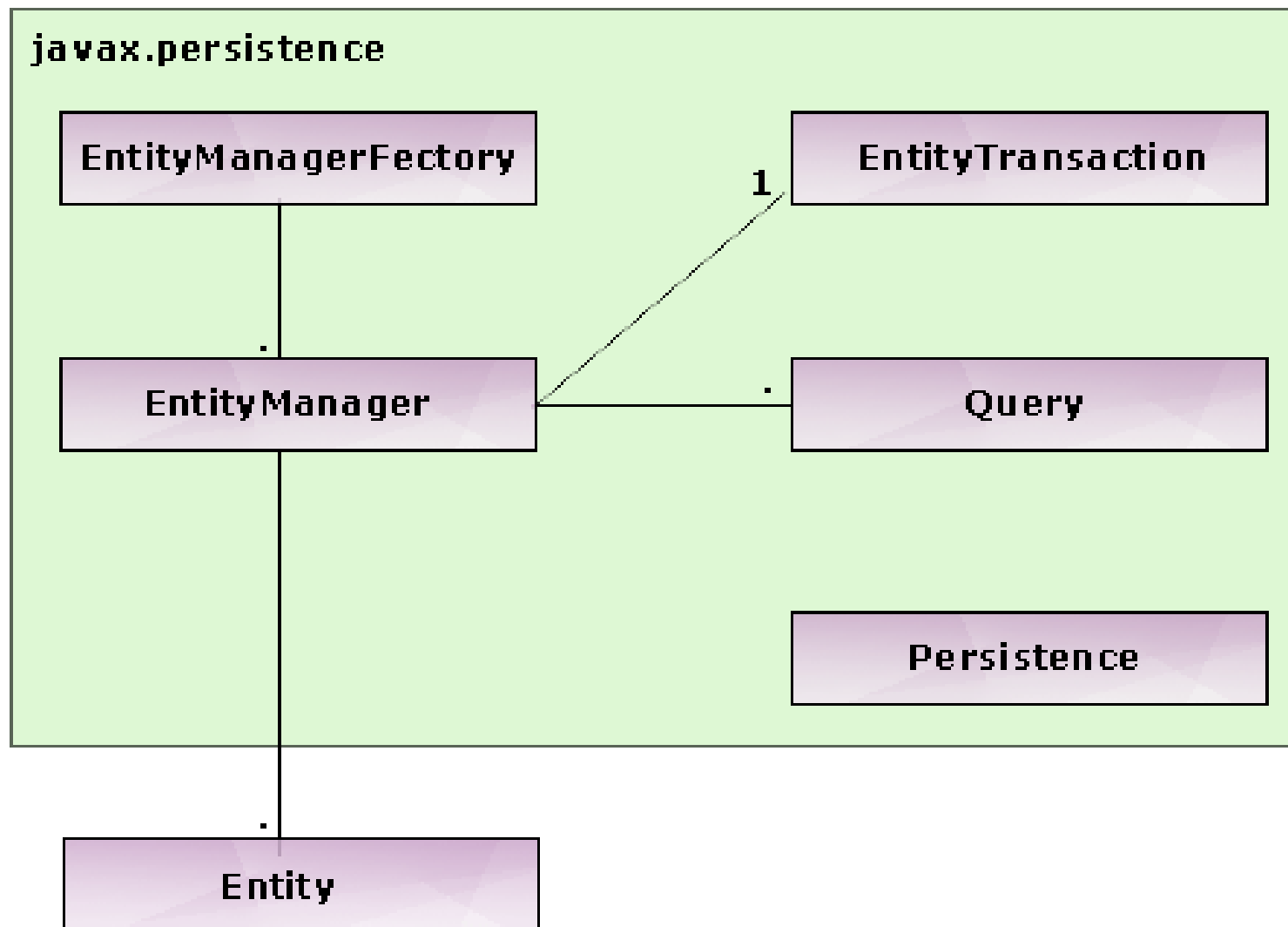
❑ Intégration de spécificités de la base de données Oracle

- LOB's
- NChar's
- XMLType's
- TIMESTAMP (TZ, LTZ)'s
- Native batch writing
- Structured object-relational data-types
- PLSQL data-types and stored procedures
- VPD, RAC, proxy authentication
- XDK XML parser
- Hierarchical selects (Select by prior)
- Returning clause
- Flashback history and queries
- Stored procedures, output parameters and output cursors
- Stored functions
- Oracle AQ

Standardiser une couche ORM via une spécification

- ❑ **Solution de persistance standardisée**
- ❑ **Facilité de mise en place**
- ❑ **S'inspirer des Frameworks largement adoptés par la communauté (notamment Hibernate)**
- ❑ **Proposer un Framework indépendant de la norme EJB**
 - ❑ le besoin de persistance est présent dans toutes les applications (JSE et JEE)

- ❑ Les entités persistantes sont des **POJO**
- ❑ Les entités ne se persistent pas elles-mêmes: Elles 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
- ❑ Java EE 6 repose à tous les niveaux sur de « **l'injection de code** » via des annotations de code : idem pour JPA



❑ La spécification Java Persistence API 1.0 ne couvre pas

- Les API Criteria et Example
- Mécanismes avancés d'attachement et de détachement d'objets
- Support des curseurs
- Filtrage dynamique des résultats
- Gestion du cache de second niveau
- Cascade « delete-orphan »
- ...

❑ Les implémentations propose des extensions

- Annotations supplémentaires
Compatibles avec les annotations standard JPA
- Possibilité d'accéder aux API natives de l'implémentation

❑ Annotations supplémentaires

- @org.hibernate.annotation.Entity
- @org.hibernate.annotations.CollectionsOfElements
- @org.hibernate.annotations.LazyToOne (PROXY | NO_PROXY | FALSE)
- @org.hibernate.annotations.Formula
- @org.hibernate.annotations.Type
- @org.hibernate.annotations.Where
- @org.hibernate.annotations.FilterDef
- @org.hibernate.annotations.Filter
- @org.hibernate.annotations.FilterJoinTables
- ...

**Gestion des
filtres**



❑ Accès à la session Hibernate

- API Criteria
- Méthodes de gestion du cycle de vie plus riches

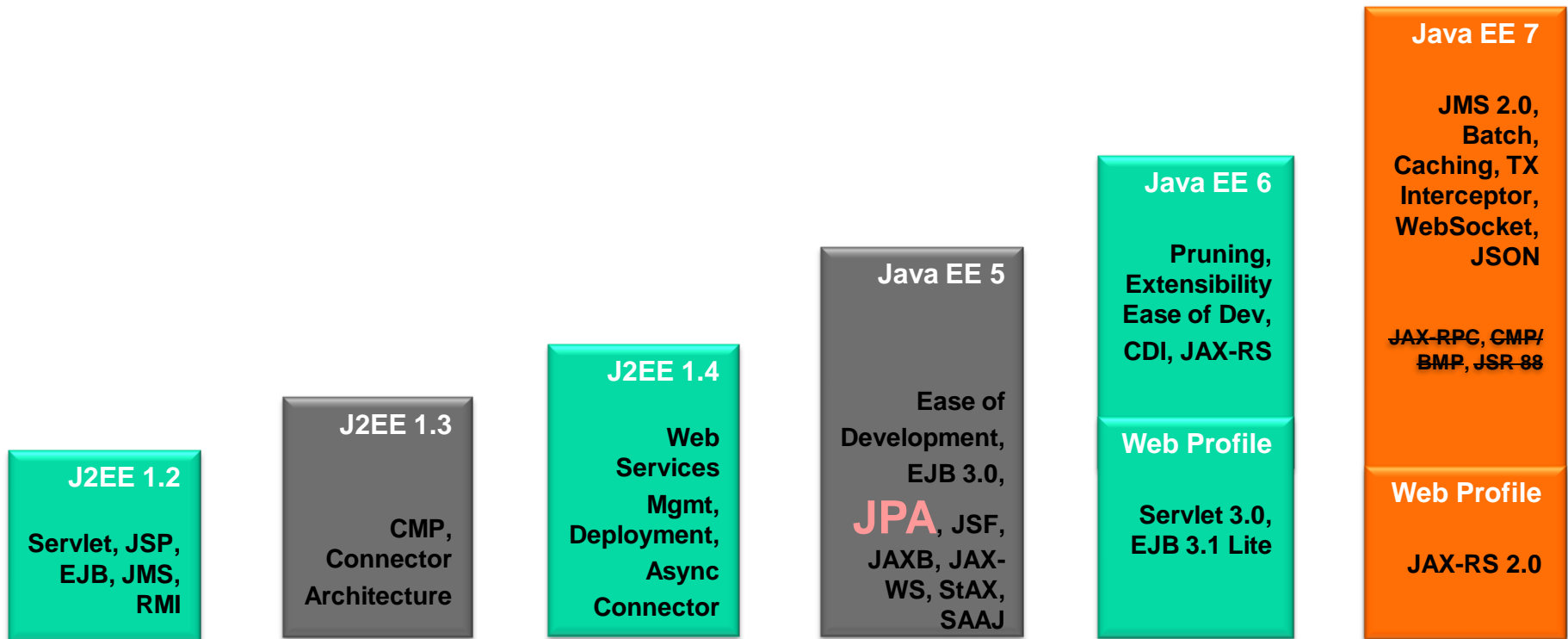
```
query.scroll()  
query.iterate()  
saveOrUpdate()  
evict()
```

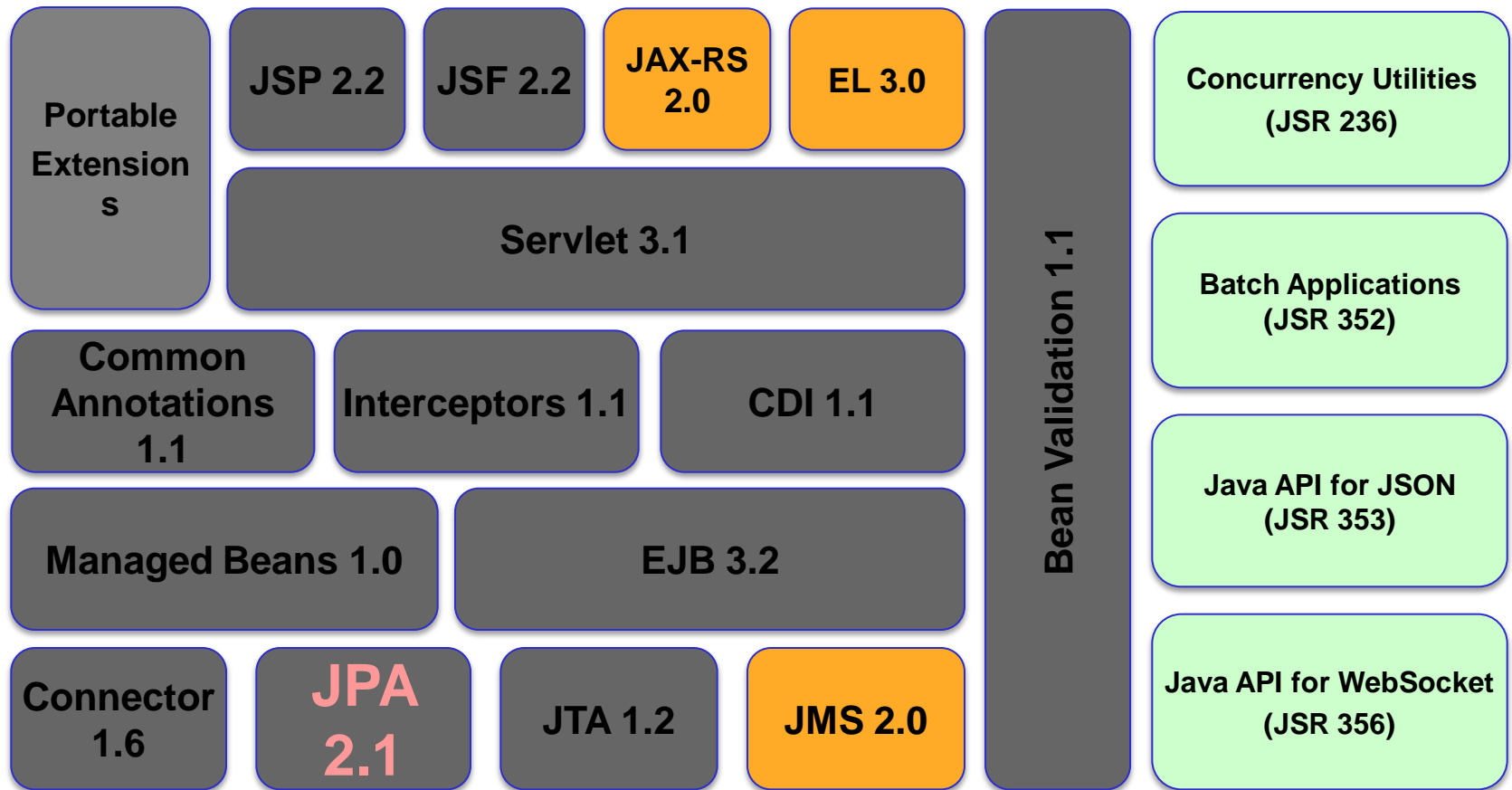
```
EntityManager em = ...  
  
Session session = (Session)em.getDelegate();
```


- ❑ À l'origine, JPA est une API visant à normaliser la gestion de la persistance dans une application JAVA EE face à une base relationnelle.
- ❑ JPA évolue pour intégrer les notions de persistances non-relationnelles (noSQL, etc.) et est de plus en plus couplé à la problématique de la validation (« *Bean validation* », JSR-303)

Historique :

- JPA 1.0 :
 - ❑ En 2006 via JSR-220
 - ❑ Périmètre initial
- JPA 2.0 :
 - ❑ En 2009 via JSR-317
 - ❑ Améliorations notables : Criteria, Validation, Mapping extensions, ...
- JPA 2.1 :
 - ❑ En 2013 via JSR-338
 - ❑ Améliorations notables : Custom converters, entity graphs, stored procedures, ...





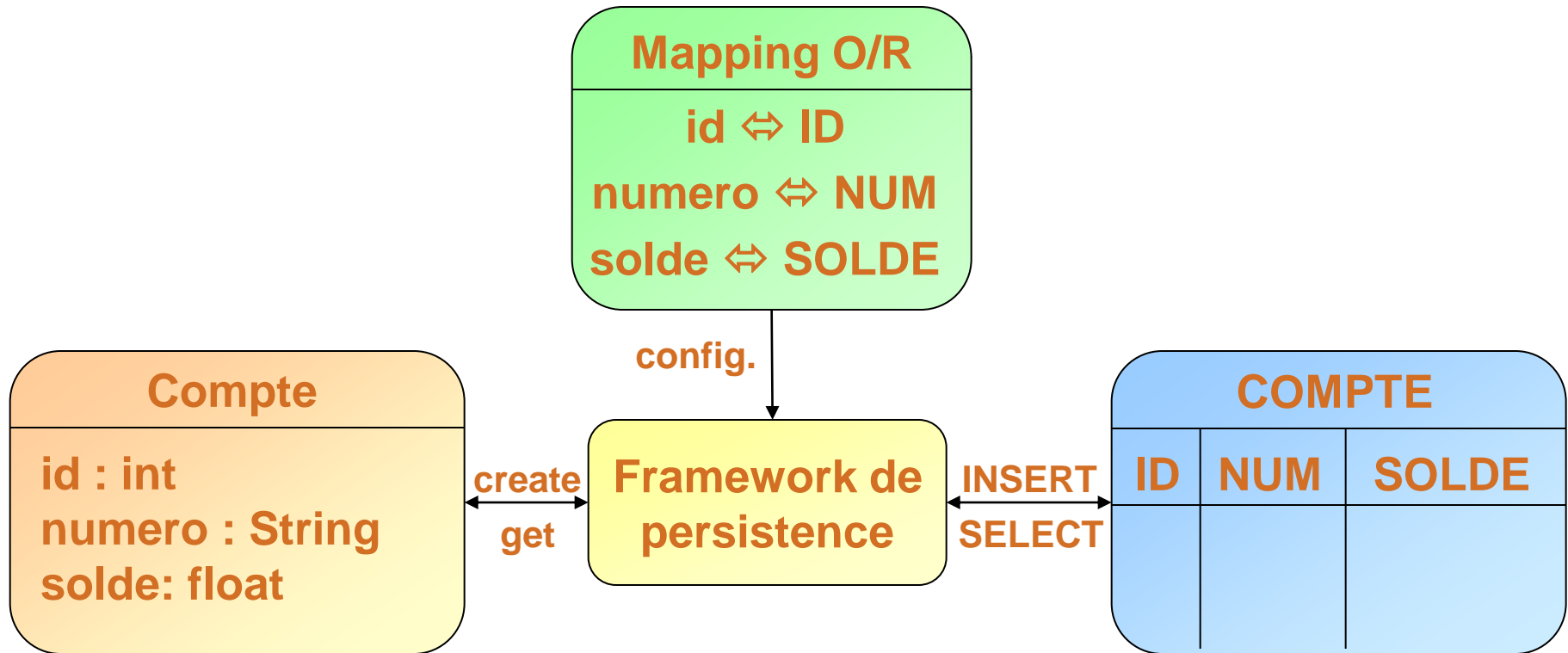
 Nouveau
 Release Majeure
 Mise à jour

- ❑ **JAVA EE fournit le cadre (contrats) uniquement via son JDK. Les implémentations peuvent être fournies par d'autres éditeurs. Dans le cas de JPA, ces implémentations sont appelées « JPA PROVIDER »**
- ❑ **Exemples :**
 - Hibernate
 - OpenJPA
 - EclipseLink
 - ...
- ❑ **Quel que soit l'implémentation retenue pour un projet, le JPA PROVIDER manipulera la notion de « session de persistance »**
- ❑ **Cette notion permet de connaître, à un instant donné de l'exécution d'un service défini en utilisant JPA, quels sont les objets qui seront affectés par la persistance**
- ❑ **Le JPA PROVIDER permet une abstraction de la source de données. C'est lui qui choisit notamment quand et comment son modèle mémoire est synchronisé avec la base de données**

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

Monde Objet

Monde Relationnel



- ❑ **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

- ❑ **Le bean entity est composé de propriétés mappés 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.

❑ 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`

```
@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
    ...
}
```

...

❑ @Entity

- Indique que la classe doit être gérée par JPA

❑ @Table

- Désigne la table de la base de données dont la classe est une représentation.
- Si l'attribut « name » est absent, l'entité est liée à la table de la base de données correspondant au nom de la classe de l'entité.

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

❏ @Id

- permet d'associer un champ de la table à la propriété en tant que clé primaire
- Les types java candidats sont :
 - *types Java primitifs : **byte, int, short, long, char***
 - *Classes Wrapper des types primitifs : **Byte, Integer, Short, Long, Character***
 - *Arrays de types primitifs ou wrappers: **int[], Integer[], etc.***
 - *Strings, nombres et dates: **java.lang.String, java.math.BigInteger, java.util.Date, java.sql.Date***

❑ @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

❑ @GeneratedValue

- on parle d'une clé « **surrogate** » (clé artificielle) → délégation du choix de la valeur au SGBD

❑ Stratégies @GeneratedValue

❑ Les types possibles sont les suivants :

- **AUTO**: laisse l'implémentation générer la valeur de la clé primaire.
- **IDENTITY**: utilise un type de colonne spécial de la base de données.
- **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*

- ❑ **@Column:** Associé à un getter, il permet d'associer un champ de la table à la propriété.
 - **unique=true** indique que l'attribut doit être unique. Cela se traduit dans la base de données par l'ajout d'une contrainte d'unicité sur la colonne associée
 - **length=30** fixe à 30 le nombre de caractères de la colonne.

- ❑ **@Basic:** représente la forme de mapping la plus simple.
 - Comportement implicite pour les types suivants : *Java.lang.String*, *java.math.BigInteger*, *java.math.BigDecimal*, *java.util.Date*, *java.util.Calendar*, *java.sql.Date*, *java.sql.Time*, *java.sql.Timestamp*, *byte[]*, *char[]*, *enums*
 - L'attribut *fetch* permet de préciser le mode de chargement d'une propriété:
 - LAZY:** valeur chargée uniquement lors de son utilisation
 - EAGER:** valeur toujours chargée

- ❑ **@Version :** Gère les accès concurrents à une même ligne de la table

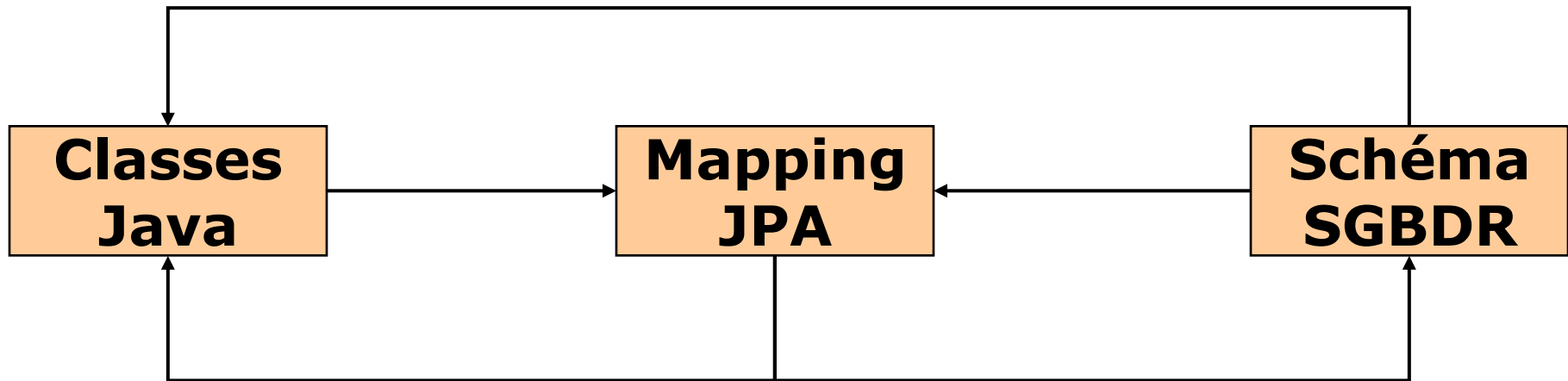
- ❑ **@Temporal**: Indique quel type est le type SQL d'une colonne / champ de type date / heure.
 - *TemporalType.DATE*: désigne une date seule sans heure associée
 - *TemporalType.TIME*: désigne une heure
 - *TemporalType.TIMESTAMP*: désigne une date avec une heure

- ❑ **@Transient** : Indique de ne pas tenir compte du champ lors du mapping (son accesseur : `@Transient getTotal() {return sum1 + sum2 }`)

- ❑ **@Enumerated**: utiliser pour mapper des énumération de type entier ou chaîne de caractères

- ❑ L'API JPA permet de mapper des types complexes
- ❑ Pour mapper les objets larges utiliser `@Lob` qui est un complément de l'annotation `@Basic`
- ❑ Le type de l'objet BLOB ou CLOB est déduit du type de la propriété Java
 - BLOB pour les tableaux de byte ou Byte ou les objets sérializables
 - CLOB pour les chaînes de caractères et les tableaux de caractères char ou Char
- ❑ Fréquemment ce type de propriété est chargé de façon Lazy

```
public class Collaborateur {  
    @Id  
    @GeneratedValue  
    private int id;  
    @Basic(fetch = FetchType.LAZY, optional = false)  
    private String prenom;  
    private String nom;  
    @Lob  
    @Basic(fetch = FetchType.LAZY)  
    private JPEG photo;  
    ...  
}
```



❑ Différents outils existent

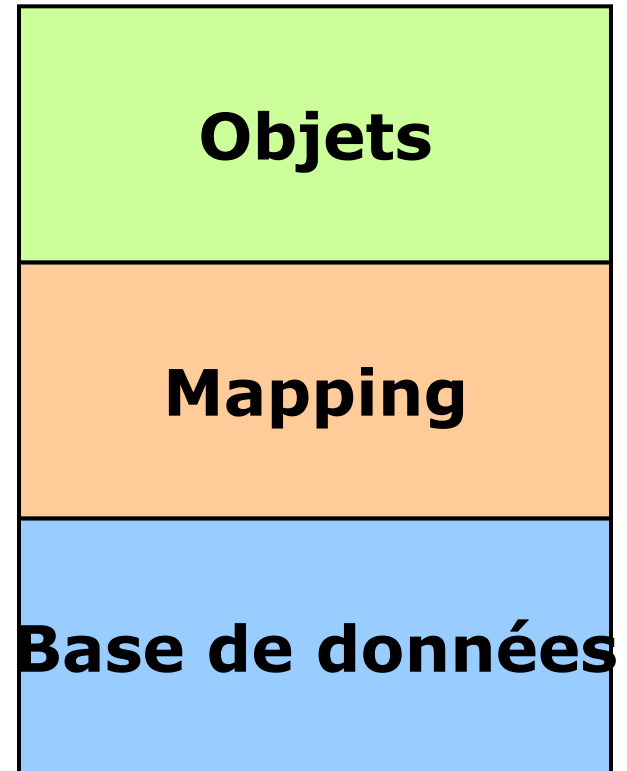
- Proposés avec les différentes implémentations de JPA
- Indépendants

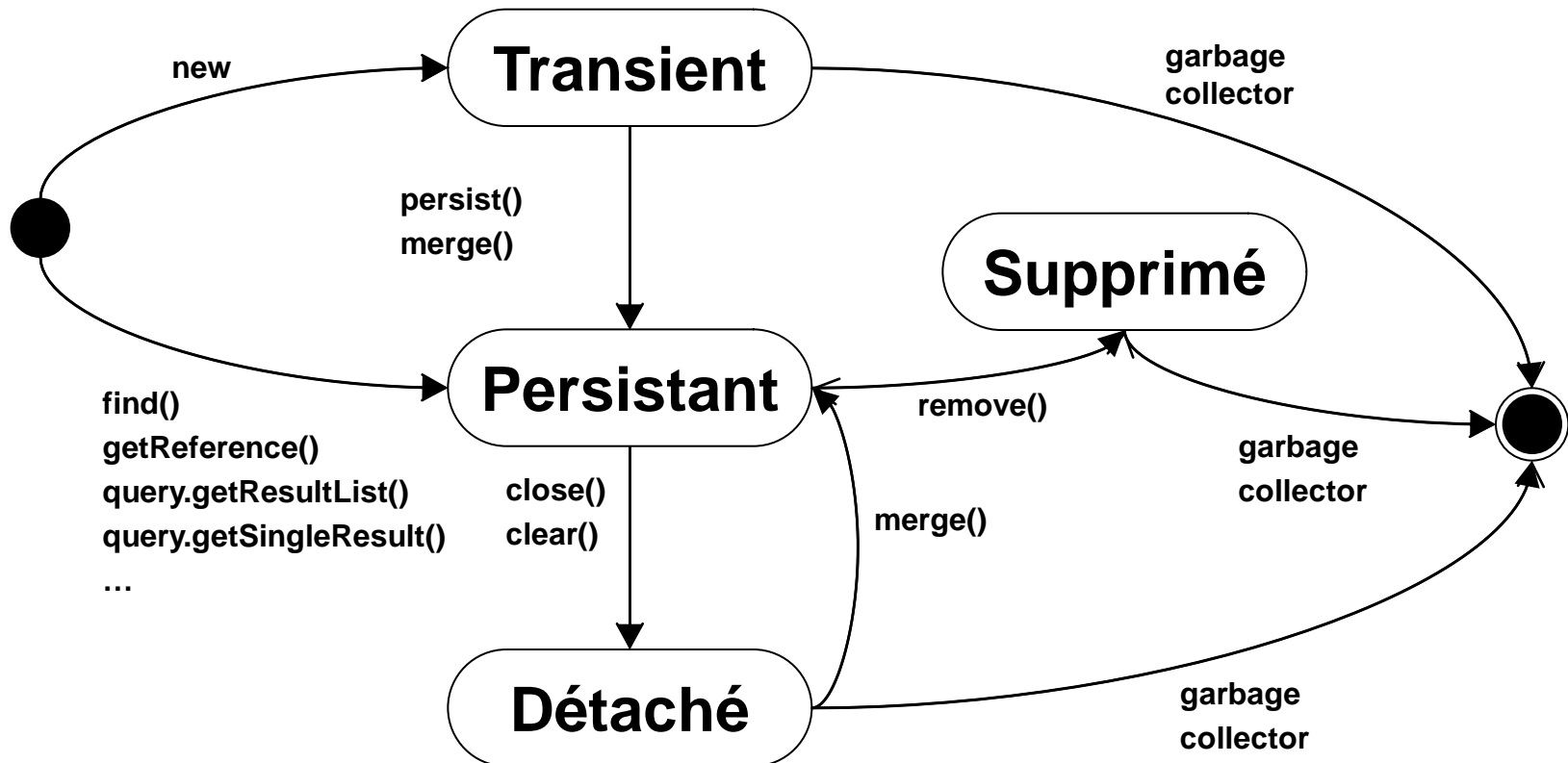
❑ Cas de l'utilisation des annotations

- Cas le plus courant avec JPA
- « Classes Java » et « Mapping JPA » ne font qu'un

➤ Génération de la base de données

- ❑ JPA est capable de **générer un script de création ou de mise à jour de la base de données cible** (script DDL) en fonction des annotations décrites dans les Entity.
- ❑ Ce script est normalisé au maximum en respectant la norme SQL99, mais des différences entre SGBD existent toujours. Il est alors nécessaire de spécifier le SGBD cible dans un fichier de configuration (équivalent du « dialect » dans Hibernate)
- ❑ Dans un environnement de production, les scripts SQL sont souvent écrits et/ou vérifiés par des DBA. La génération automatique peut donc être à proscrire.

Bottom-up**Top-down****Meet in the middle****Le plus fréquent**



- **Conceptuellement :**

- ▶ Chaque objet devrait avoir son cycle de vie

- **Dans la pratique :**

- ▶ Une ligne en base de données peut correspondre à plusieurs objets
- ▶ Certains objets n'ont pas d'identifiant

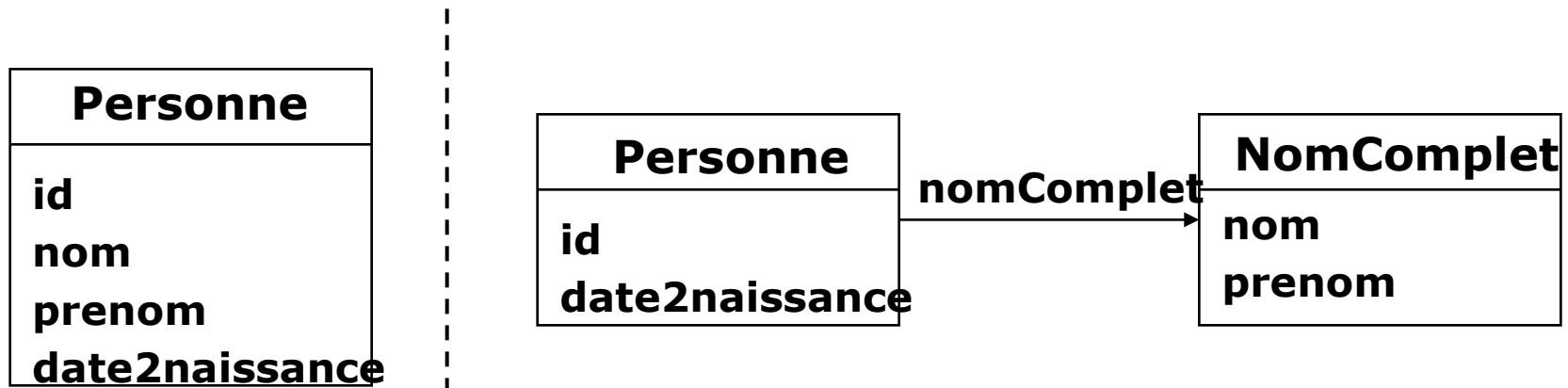
Entité

- ▶ Objet ayant sa propre identité
- ▶ Son ajout/suppression est explicitement demandé

Valeur

- ▶ Objet dépendant d'une entité
- ▶ Son cycle de vie est dépendant de celui de l'entité

Ex : Une personne et son nom complet



❑ Les Entity Bean sont déployés dans des « persistence Units »,

- Spécifiés dans le fichier « persistence.xml » qui est dans le jar contenant les EJBs.
- Exemple le plus simple :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="intro"/>
</persistence>
```

- Mais on peut ajouter de nombreux paramètres :
 - <description>, <provider>, <transaction type>, <mapping file> etc.

❑ Ce fichier « configure » le persistence manager en JEE

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0"
  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 persistence_1_0.xsd">
  <persistence-unit name="IGift-ejbPU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/igift</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

❑ Ce fichier « configure » le persistence manager en JSE

- Pas de serveur JNDI en JSE : déclaration de la datasource

```
<?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 persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="JPA2_TUTORIEL"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.bankonet.model.TestEntity</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql:///banque" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```

□ Tutorial

➤ Création d'un projet Eclipse JPA sur MySQL

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

❑ 4 opérations de base

❑ CRUD

- **Create** : création
- **Read** : lecture
- **Update** : mise à jour
- **Delete** : suppression

- INSERT en la base de données
- SELECT en base de données
- UPDATE en base de données
- DELETE en base de données



Toutes les opérations d'écriture doivent avoir lieu dans une transaction

- ❑ 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();
```

❑ **L'EntityManager propose deux mécanismes pour rechercher une instance :**

- La recherche à partir de la **clé primaire**
- La recherche à partir **d'une requête**

❑ 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) {  
    System.out.println("Hôtel non trouvé");  
}
```

- ❑ 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);
```

❑ Pour modifier un objet il faut

1. Le lire

Récupération de l'objet en lecture/écriture

2. Le modifier

Son nouvel état est persisté automatiquement

❑ 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");  
}
```

**Lecture
nécessaire**

- ❑ **Méthode `merge()` : Fusionner les données d'une entité non gérée avec la base de données.**
- ❑ **Exemple d'utilisation**
 - Sériailisation d'une entité pour l'envoyer au client
 - Le client modifie l'entité et la renvoie
 - Synchronisation de ses données avec celles de la base de 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);  
}
```

- ❑ Il s'agit de rafraichir les données d'une entité avec celle de la base de données
- ❑ La méthode *refresh()* de l'interface *EntityManager* est responsable de cette opération

```
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("jpa");  
EntityManager em = emf.createEntityManager();  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
Hotel hotel = (Hotel) em.find(Hotel.class, 120);  
if (hotel != null) {  
    em.refresh(hotel);  
}  
tx.commit();  
em.close();  
emf.close();
```

- ❑ **Paradoxe: pour supprimer un objet il faut d'abord le lire**
- ❑ **La méthode `remove(...)` de l'interface `EntityManager`, permet la demande de suppression d'un objet**

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

❑ Toute modification doit avoir lieu dans une Transaction

➤ Nom qualifié: *javax.persistence.EntityTransaction*

```
EntityTransaction tx = em.getTransaction();  
tx.begin();  
// traitement  
...  
tx.commit(); // ou tx.rollback();  
em.close();
```

○ Une transaction se termine:

- Soit par un **commit** → validation de toutes les opérations effectuées
- Soit par un **rollback** → annulation de toutes les opérations effectuées
- Soit par « **rien** » → équivaut au rollback

La clôture de la session est obligatoire (sinon pb de connexions JDBC)

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("jpa");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

// début transaction
tx.begin();
Hotel hotel = (Hotel) em.find(Hotel.class, 120);
if (hotel != null) {
    em.remove(hotel);
}

// fin transaction
tx.commit();
// fin EntityManager
em.close();
// fin EntityManagerFactory
emf.close();
```


□ TP0 : mapping

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ Cardinalité (1-1, 1-n, n-n...),
- ❑ Direction des relations (bi-directionnelles, uni-directionnelles),
- ❑ Agrégation vs composition et destructions en cascade,
- ❑ Relations récursives, circulaires, aggressive-load, lazy-load,
- ❑ Intégrité référentielle,
- ❑ Accéder aux relations depuis un code client, via des Collections,
- ❑ Comment gérer tout ça !

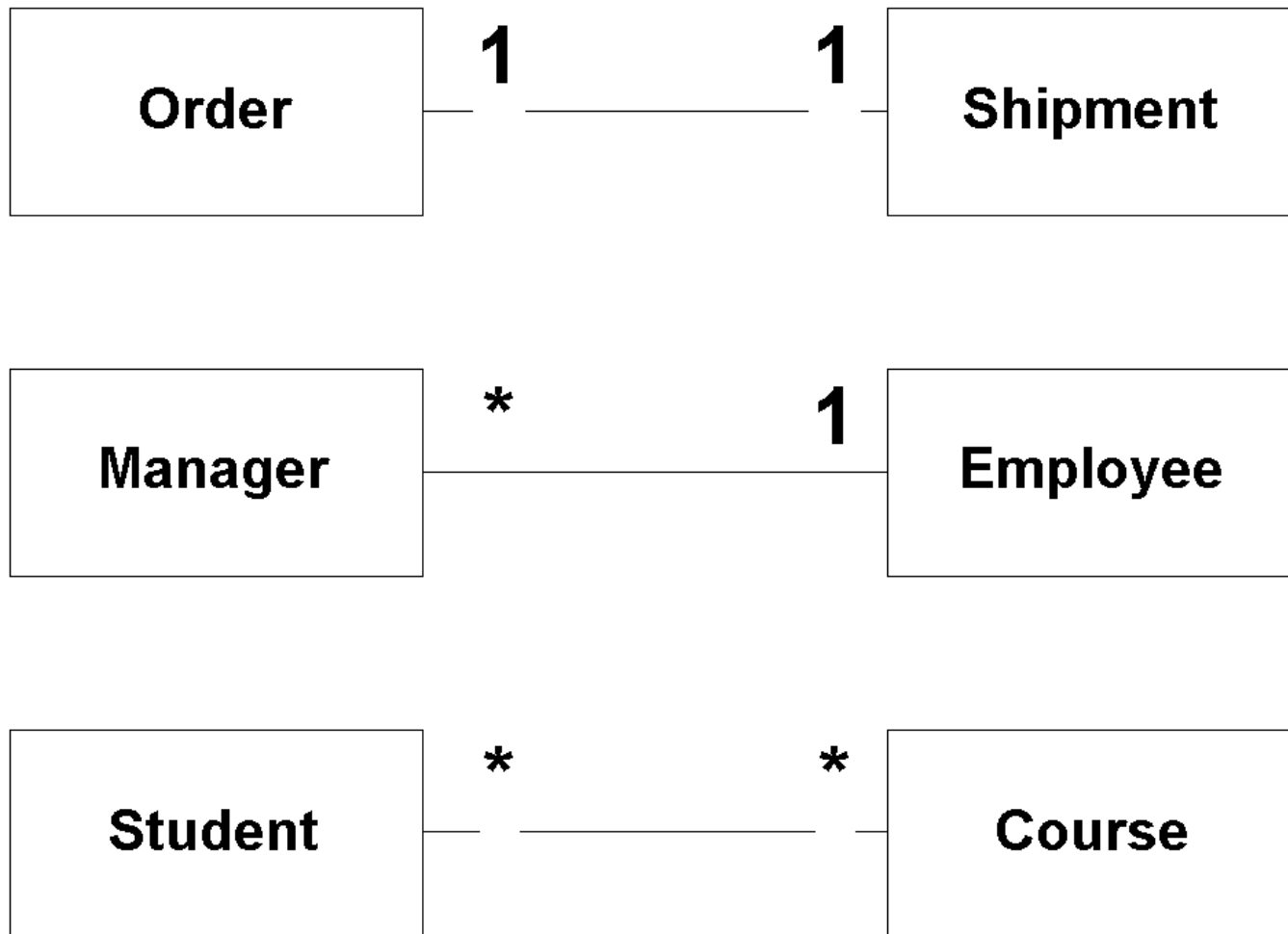
❑ Unidirectionnelle

- On ne peut aller que du bean A vers le bean B

❑ Bidirectionnelle

- On peut aller du bean A vers le bean B et inversement

- ❑ **La cardinalité indique combien d'instances vont intervenir de chaque côté d'une relation**
- ❑ **One-to-One (1:1)**
 - Un employé a une adresse...
- ❑ **One-to-Many (1:N)**
 - Un PDG et ses employés...
- ❑ **Many-to-Many (M:N)**
 - Des étudiants suivent des cours...



- ❑ **Une relation peut-être unidirectionnelle ou bidirectionnelle.**
- ❑ **Les associations entre entités correspondent généralement 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**
- ❑ **Règles :**
 - **@JoinColumn** : définit la colonne de jointure.
 - **@JoinTable** : définit la table de jointure

- ❑ 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 la plus souvent utilisée**
- ❑ **Seule relation correspondant réellement au Modèle Physique de Données (MPD)**
- ❑ **Différence entre OneToMany et ManyToOne suivant le « sens » d'observation de la relation**

❑ Exemple

- Une **Chambre** appartient à un **Hôtel**
- Un **Hôtel** regroupe plusieurs **Chambres**



————— **OneToMany** —————→

←———— **ManyToOne** —————

```
public class Chambre {  
    private int id;  
    private Hotel hotel; // reference vers l'hotel  
    ...  
    public Hotel getHotel() { return hotel; }  
    public void setHotel(Hotel hotel) { this.hotel = hotel; }  
}
```

```
public class Hotel {  
    private int id;  
    // pas de reference vers l'ensemble des chambres  
    ...  
}
```

❑ Annotation @ManyToOne

- Attribut *optional* : défini si l'association est optionnelle ou pas. (Par défaut true).

```
public class Chambre {  
    private int id;  
    @ManyToOne  
    @JoinColumn(name="HOT_ID")  
    private Hotel hotel;  
    ...  
}
```

❑ Utilisation

```
Hotel h1 = ...;  
Chambre ch1 = ...;  
Chambre ch2 = ...;  
ch1.setHotel(h1);  
ch2.setHotel(h1);
```

```
public class Hotel {  
    private int id;  
    private Set chambres; // référence vers les chambres  
    ...  
    public Hotel() { chambres = new HashSet(); }  
    public Collection getChambres() { return chambres; }  
    public void setChambres(Set chambres) { this.chambres =  
        chambres; }  
}
```

```
public class Chambre {  
    private int id;  
    // pas de référence vers l'hotel  
    ...  
}
```



La Collection peut être de type: Set, List, Array, Collection, Map



Ne pas oublier d'initialiser la Collection à « vide » (et pas « null »)

- ❑ Annotation **@OneToMany**
- ❑ Selon la spécification, le schéma relationnel passe par une table de jointure
- ❑ Peut être réalisé par une clé étrangère si l'annotation **@JoinColumn** est précisée

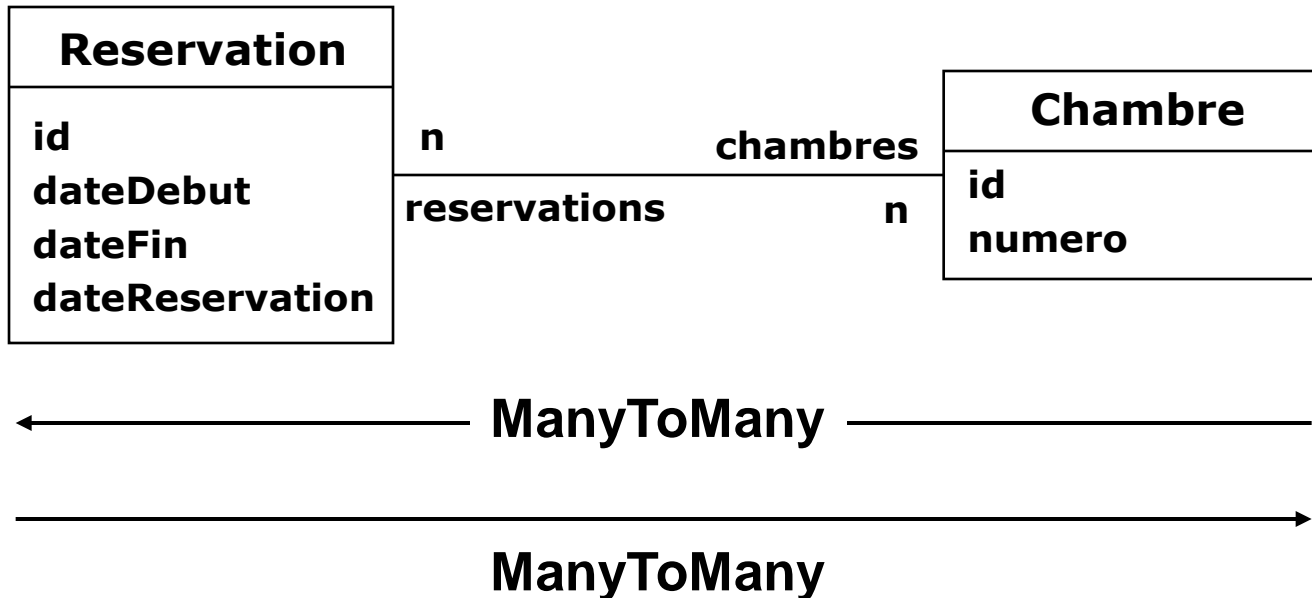
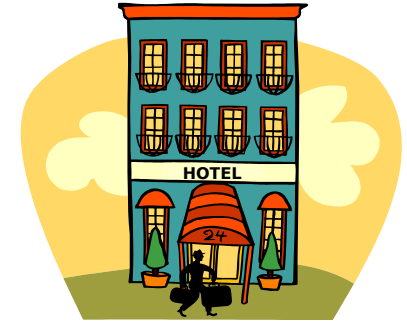
```
public class Hotel {  
    private long id;  
  
    @OneToMany  
    private Set<Chambre> chambres; // référence vers les chambres  
    public Hotel() {  
        chambres = new HashSet<Chambre>();  
    }  
    ...  
}
```

❑ Utilisation

```
Hotel h1 = ...;  
Chambre ch1 = ...;  
Chambre ch2 = ...;  
    h1.getChambres().add(ch1);  
h1.getChambres().add(ch2);
```

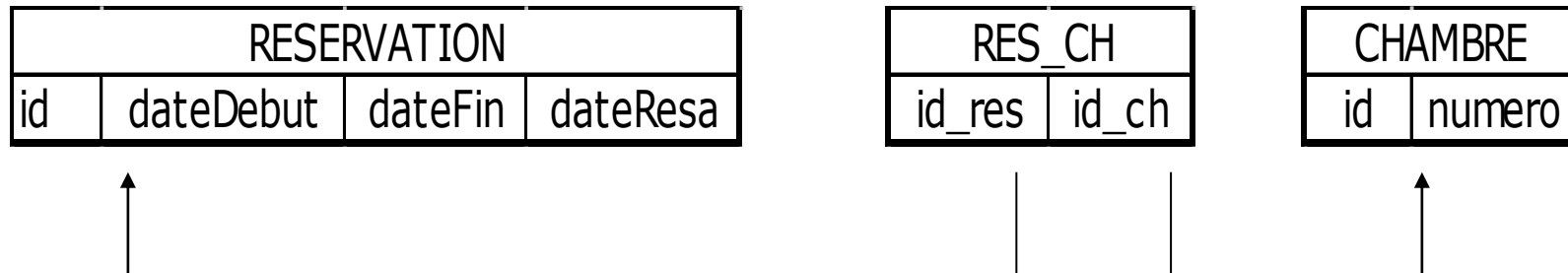
❑ Exemple

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



❑ En base de données, utilisation d'une table intermédiaire

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



```
public class Reservation {  
    private int id;  
    private Set chambres; // référence vers les chambres  
    ...  
    public Reservation() { chambres = new HashSet(); }  
    public Set getChambres() { return chambres; }  
    public void setChambres(Set chambres) { this.chambres =  
        chambres; }  
}
```

```
public class Chambre {  
    private int id;  
    // pas de référence vers les réservations  
}
```



La Collection peut être de type: Set, List, Array, Collection, Map



Ne pas oublier d'initialiser la Collection à « vide » (et pas « null »)

❑ Annotation @ManyToMany

```
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;  
    ...  
}
```

○ Utilisation

```
Reservation rel = ...;  
Chambre ch1 = ...;  
rel.getChambres().add(ch1);
```

❑ En base de données, cette relation se traduit par une clé étrangère

- Avec contrainte d'unicité

Exemple : une commande et un colis

OrderPK	OrderName	Shipment ForeignPK
12345	Software Order	10101

ShipmentPK	City	ZipCode
10101	Austin	78727

❑ L'une des contreparties peut exister sans l'autre

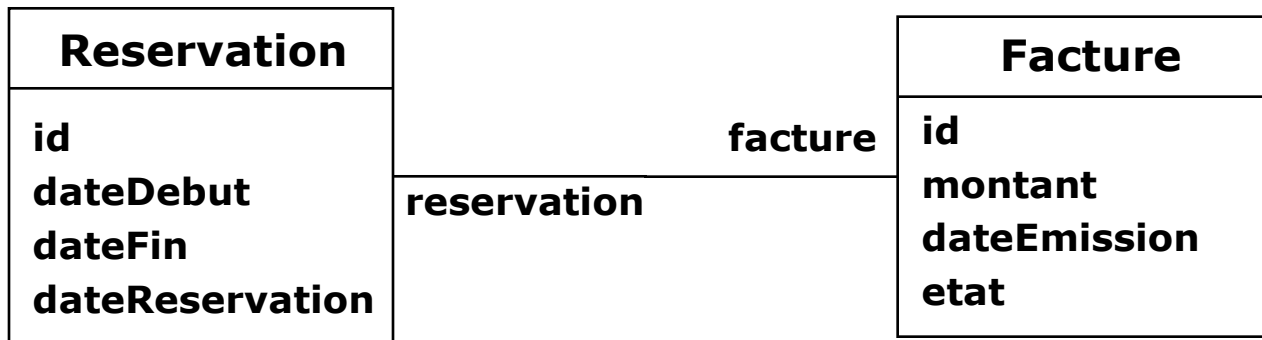
- Il peut y avoir une commande sans colis
- Il ne peut pas y avoir de colis sans commande

Facture n'a pas d'identifiant propre

❑ D'un point de vue du MPD, correspond à une 1:n

❑ Exemple

- Pour chaque **Reservation** une **Facture** est émise
Une réservation peut ne pas avoir de Facture

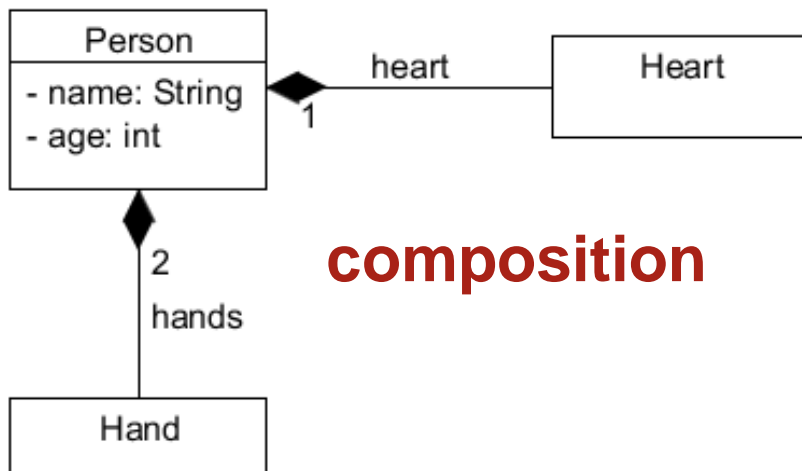


```
@Entity
public class Facture {
    @Id
    private int id;
    @OneToOne
    private Reservation reservation; // référence vers la
    réservation
    ...
    public Reservation getReservation() { return
    reservation; }
    public void setReservation(Reservation res) {
    this.reservation = res;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

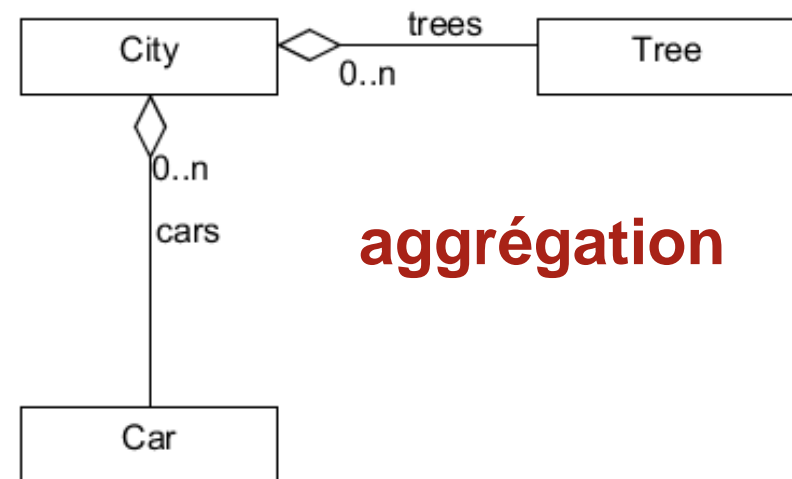
```
@Entity
public class Reservation {
    @Id
    @GeneratedValue
    private int id;
    // pas de référence vers la facture
    ...
}
```

❑ Le modèle objet définit 2 types de relations 1-N et 1-1

- **L'agrégation** : un objet A **utilise** un ou N objet B, le cycle de vie de l'objet B est **indépendant** de celui de l'objet A
- **La composition** : un objet A **est composé** d'un ou N objet B, le cycle de vie de l'objet B est **dépendant** de celui de l'objet A

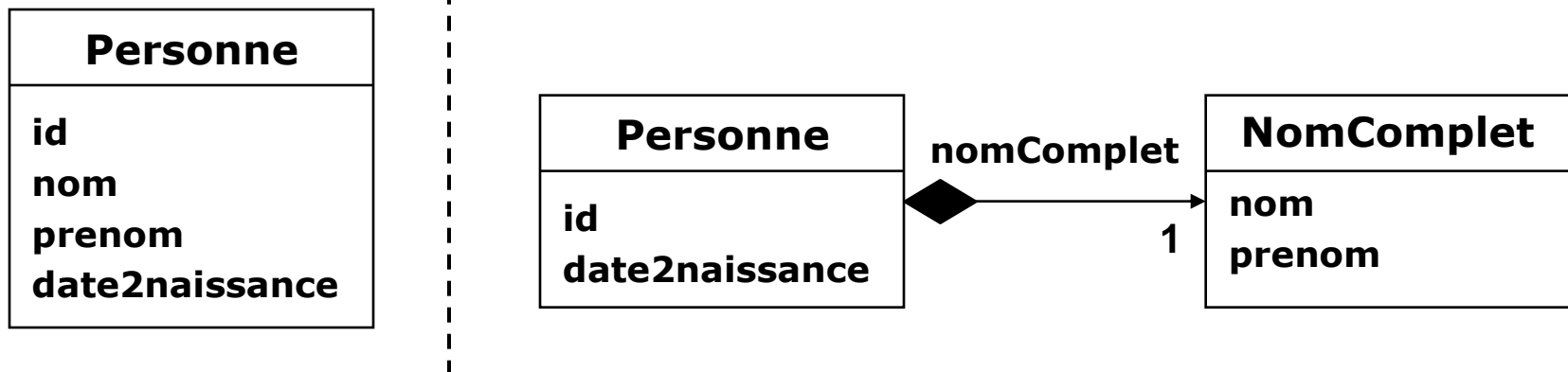


composition



agrégation

- ❑ **Un composant est un objet associé par composition et persisté dans la même table que l'objet lui-même**
 - Les cycles de vie des objets sont les mêmes
- ❑ **Le terme « composant »**
 - N'a rien à voir avec « architecture à base de composants »
 - Correspond au concept objet de composition (UML)



- ❑ `@Embeddable` : Déclare une classe comme composition d'une autre classe
- ❑ Ses propriétés sont stockées dans la même table que la classe primaire
- ❑ Elle ne doit pas contenir d'annotation `@Id`

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

- ❑ **@Embedded** : Déclare qu'une propriété d'une entité est un objet inclus

```
public class Personne {  
  
    @Embedded  
    private NomComplet nomComplet;  
  
}
```

- Les propriétés d'une composition peuvent être surchargées par l'entité qui la référence avec @AttributeOverride

❑ Utiliser des colonnes composites :

```
@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;
    @Embedded
    Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    String zip;
    protected String plusFour;
}
```

□ TP1 : relations

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

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

- Encapsulation
- Héritage
- Polymorphisme

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

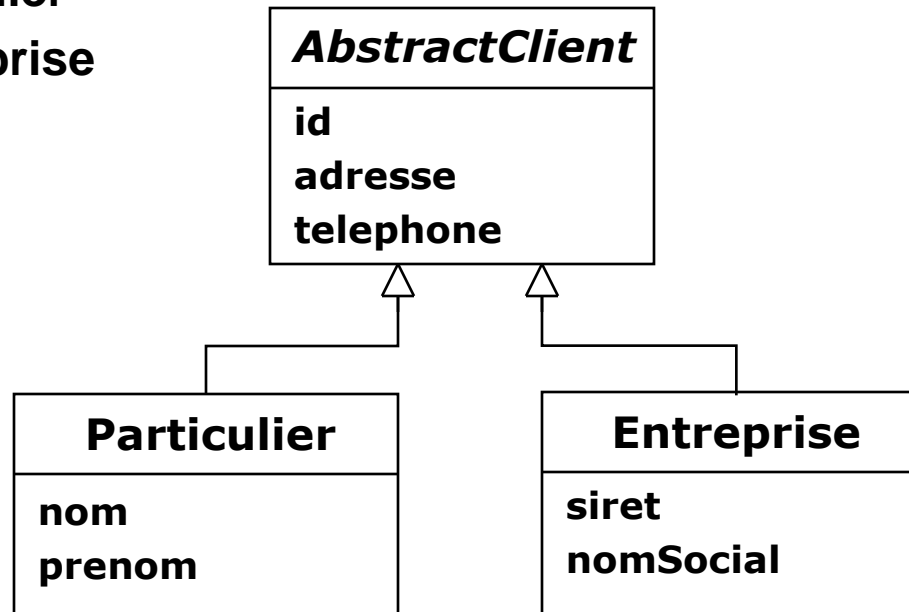
- Chaque stratégie a ses avantages et ses inconvénients

❑ JPA permet l'utilisation de ces 3 stratégies générales

- Choix avec l'annotation `@Inheritance`
- Attribut : `strategy`
 - Le type de stratégie à utiliser
 - Valeurs possibles: `InheritanceType.SINGLE_TABLE`,
`InheritanceType.JOINED`, `InheritanceType.TABLE_PER_CLASS`

□ Un Client peut être

- soit un **Particulier**
- soit une **Entreprise**



Nous considérerons que la classe **AbstractClient** est abstraite

Théorique	+-
Technique	--

❑ Appellation: « **SINGLE_TABLE** » ou « flat mapping »

❑ 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, dénormalisation du schéma SQL)
- (-) Des enregistrements quasiment vides (espace perdu)
- (+) Pas de clés étrangères, pas de jointure au requêtage

❑ 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

```

/**
 * 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
servant de
discriminant**




```

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

```

**Valeur du
discriminant
pour cette
sous-classe**



```

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

```

Utilisation

```
/**
 * 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("Neobject");
    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 route de la halte des gauthiers");
    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);
}
```

Recherche sans
connaissance
du type réel

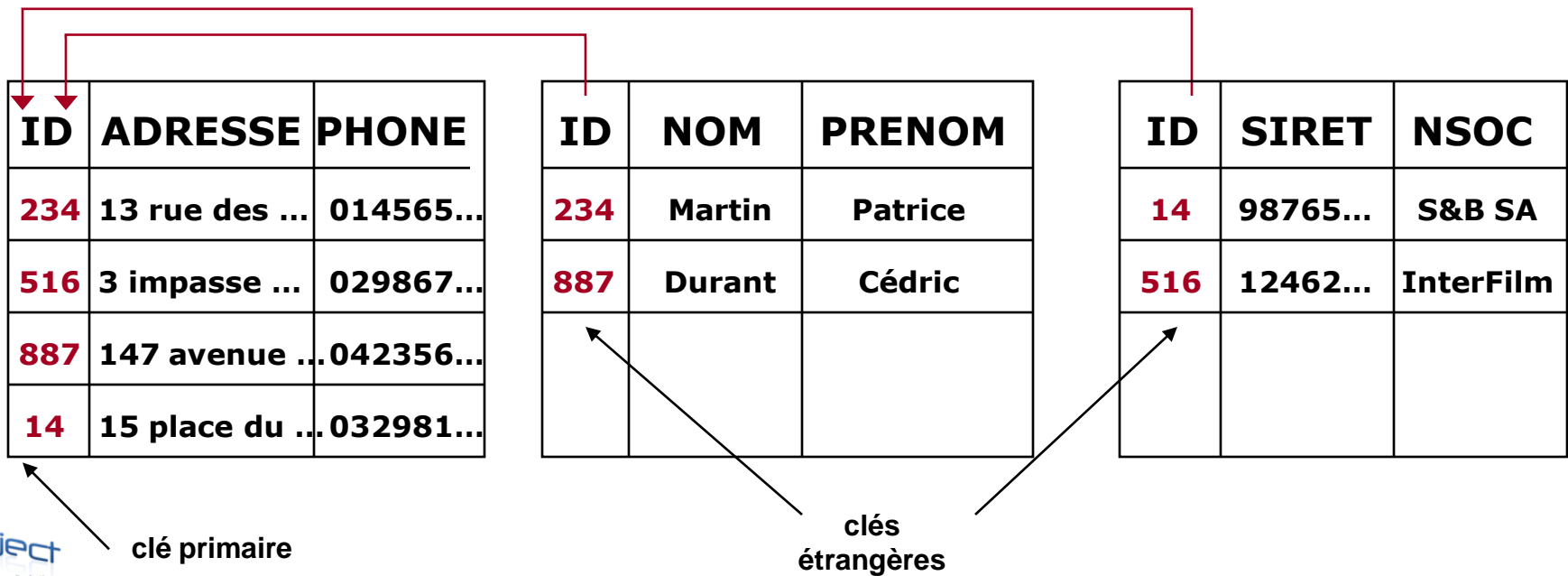


Théorique	++
Technique	--

- ❑ **Appellation: « Table per subclass » 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 → performances non optimales



- ❑ 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



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

    ...
}
```

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

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

Utilisation

```
/**
 * 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("Neobject");
    e.setSiret("44019981800013");
    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();
    em.persist(e);

    Particulier p = new Particulier(); // création d'un particulier
    p.setAdresse("14 route de la halte des gauthiers");
    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);
}
```

Polymorphisme :
Recherche sans
connaissance du
type réel



❑ Appellation: « Table per concrete class »

Théorique	--
Technique	+-

❑ 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

- ❑ 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

```
/**
 * 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 rendues impossibles par cette annotation

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

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

Utilisation

```
/**
 * 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("Neobject");
    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 route de la halte des gauthiers");
    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);
}
```

Gestion des ids doit être effectuées manuellement

Connaissance du type réel nécessaire



Pas de recherche, avec find(), possible sans connaître le type réel (polymorphisme)

❑ Variante du cas précédent

Théorique	--
Technique	+--

❑ Même principe mais

- Clé primaire partagée entre les tables
- Un mapping pour la hiérarchie

❑ Théoriquement

- (-) Clé primaire partagée (peu ordinaire)

❑ 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 (UNION)

- ❑ 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
- ❑ Les identifiants des entités sont gérés manuellement
- ❑ Les requêtes polymorphe sont possibles et les résultats agrégés avec un UNION en SQL



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 utilisant la même séquence

1 table par classe concrète (UNION) exemple

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

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

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

Utilisation

```
/**
 * Test de la persistance de l'héritage avec une table par
 * classe en union.
 */
@SuppressWarnings("unchecked")
public void testUneTableParClasseConcreteUnion() {
    Entreprise e = new Entreprise(); // création d'une entreprise
    e.setAdresse("1");
    e.setTelephone("0141220300");
    e.setNomSocial("Neobject");
    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.setId(4);
    p.setAdresse("14 route de la halte des gauthiers");
    p.setTelephone("0112341234");
    p.setNom("Durant");
    p.setPrenom("Cédric");
    em.persist(p);
    transaction.commit();
    StringBuilder sb = new StringBuilder(" select c from AbstractClient c ");
    Query query = em.createQuery(sb.toString());
    List<AbstractClient> clients = query.getResultList();
    LOGGER.debug("List des clients" + clients);
}
```

Polymorphisme possible



□ Les éléments à prendre en compte

- Besoin de polymorphismes ou non (associations/requêtes)
- Nombres de propriétés
- Différences entre les classes filles
 - **Structurelles (propriétés)**
 - Comportementales (méthodes)

Polymorphisme	Peu de propriétés	Structure proche	Choix
Nécessaire	Indifférent	Indifférent	Table by subclass
Nécessaire	Oui	Oui	Table by class hierarchy
Pas nécessaire	Indifférent	Indifférent	Table per concrete class
Nécessaire	Indifférent	Indifférent	Table by concrete class avec sequence

❑ Table per hierarchy pour les problèmes simples

- Facile à mettre en œuvre, bonnes performances

❑ Pour des cas plus complexes : table per subclass ou table per classe UNION

- Selon les performances JOIN et UNION

❑ Penser reconcevoir sans héritage

- Utiliser la délégation
- Penser aux composants

□ TP2 : héritage

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ **Une collection se décrit de façon très proche du Java**
- ❑ **Plusieurs types de collections**
 - Set: ensemble non indexé, non typé, sans doublon
 - List: ensemble indexé, non typé, avec doublon
 - Array: ensemble indexé, typé, avec doublon
 - Map: dictionnaire (clef / valeur), non typé, sans doublon de clef

- ❑ A la place d'une collection il est possible d'utiliser une *map*
- ❑ Dans ce cas, il faut annoter l'association avec `@MapKey` qui précise la clé de la *map* qui doit être un des attributs de l'entité contenue dans la *map*
- ❑ La spécification n'autorise qu'une clé simple comme index d'une map
- ❑ Si l'association est traduite par une *map* mais n'a pas d'annotation `@KeyMap`, la clé sera considérée être l'identificateur de l'entité

- ❑ Les collaborateurs d'un hotel peuvent être enregistrés dans une *map* dont les clés sont les noms des collaborateurs (on suppose que 2 collaborateurs n'ont pas le même nom)

```
public class Hotel {  
    private long id;  
  
    @OneToMany(mappedBy="nom")  
    public Map<String, Collaborateur>  
    getCollaborateurs() {  
        ...  
    }  
    ...  
}
```

- ❑ L'ordre d'une liste n'est pas nécessairement préservé dans la base de données
- ❑ De plus, l'ordre en mémoire doit être maintenu par le code (pas automatique)
- ❑ Ce qui est possible est de spécifier l'ordre des éléments d'une collection au chargement avec `@OrderBy`

- ❑ Cette annotation indique dans quel ordre sont récupérées les entités associées
- ❑ Chaque attribut peut être précisé par ASC ou DESC (ordre ascendant ou descendant);
 - ASC par défaut
- ❑ Les différents attributs sont séparés par une virgule
- ❑ Si aucun attribut n'est précisé, l'ordre sera celui de la clé primaire

```
public class Hotel {  
    private long id;  
    @OneToMany(mappedBy="nom")  
    @OrderBy("nom DESC, poste ASC")  
    public List<Collaborateur> getCollaborateurs() {  
        ...  
    }  
    ...  
}
```

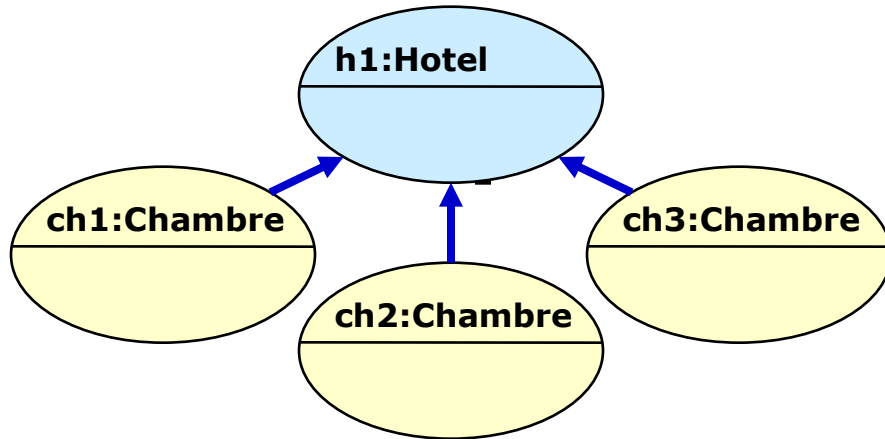
1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ **Un concept courant dans les modèles de classe métiers**
- ❑ **La navigabilité est activée dans les deux classes qui sont les deux extrémités de l'association**
- ❑ **Au niveau de la base de données y a pas de notion de navigabilité**
 - Les tables sont simplement liées par une clé étrangère
 - La colonne de la clé étrangère est gérée par deux classes
- ❑ **L'une des deux extrémité est appelé propriétaire de l'association**

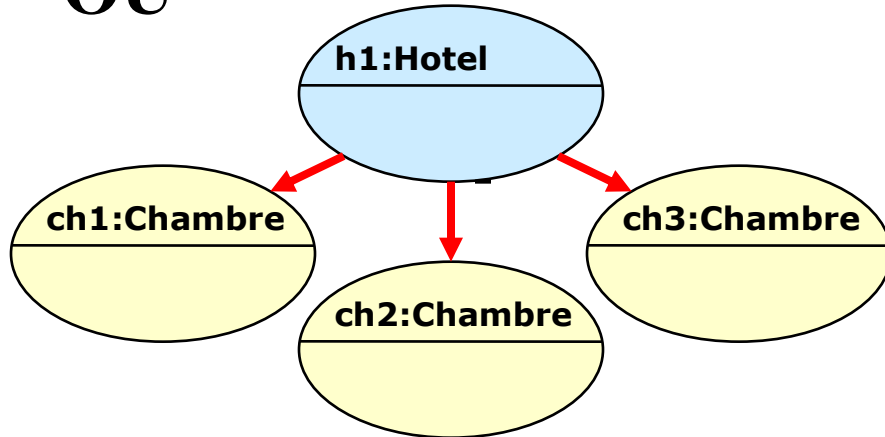
- ❑ Correspond à l'entité contenant la clé étrangère de l'association pour les relation autre que ManyToMany
- ❑ Pour les associations ManyToMany le développeur peut choisir arbitrairement le bout propriétaire
- ❑ L'autre bout (non propriétaire) est qualifié par l'attribut *mappedBy* qui donne le nom de l'attribut dans le bout propriétaire qui correspond à la même association

```
public class Hotel {  
    private long id;  
    @OneToMany(mappedBy="hotel")  
    private Set<Chambre> chambres;  
    ...  
}
```

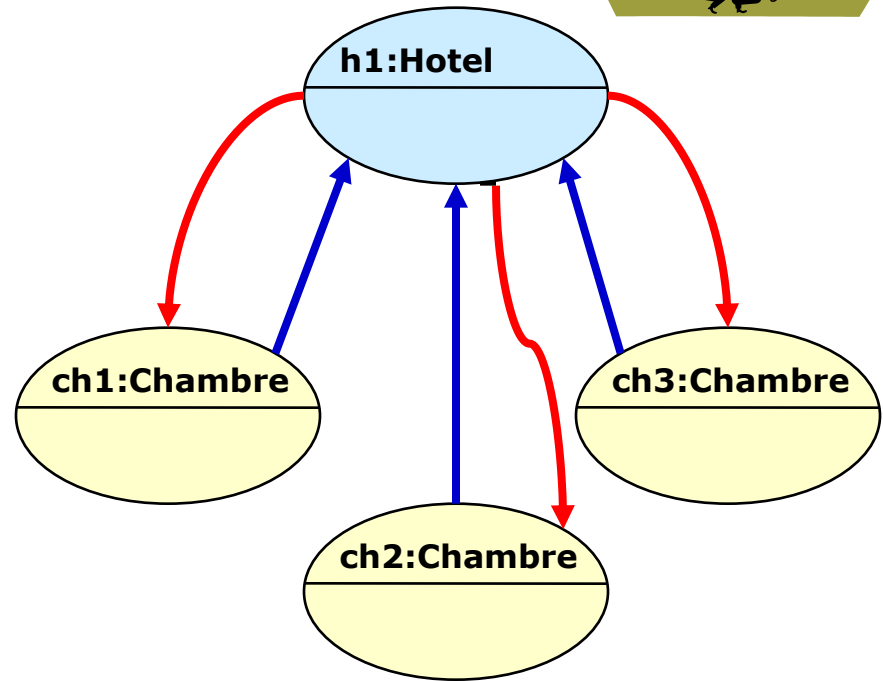
```
public class Chambre {  
    private int id;  
    @ManyToOne  
    private Hotel hotel;  
    ...  
}
```



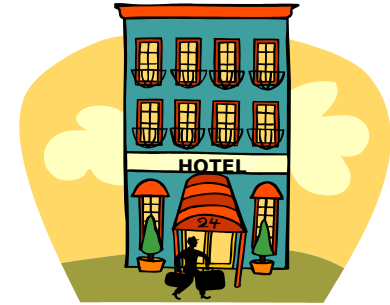
OU



Uni-directionnel



Bi-directionnel




❑ Uni-directionnel

- (+) Pas de problème de synchro de références
- (+) Pas de problème de responsabilité lors de la persistance
- (-) Problème lors des parcours des graphes: un unique sens

❑ Bi-directionnel

- (+) Possibilité de naviguer dans les deux sens d'une relation
- (-) Désynchronisation possible des références lors des MAJ
- (-) Problème de responsabilité lors de la persistance

- ❑ **Le développeur doit maintenir les références manuellement**
- ❑ **1 idée commune**
 - Modifier les références de manière simultanée
- ❑ **2 méthodes**
 - 1ère méthode
 - Dans le code, lors de la modification d'une référence, toujours penser à modifier la « référence inverse »
 - 2ème méthode
 - Factoriser les modifications de références dans une seule méthode

 **La 1ère méthode est souvent source d'erreurs (difficiles à localiser)**

❑ Première méthode: « Penser aux MAJ des références »

➤ Méthode simpliste

```
Hotel h1 = ...;  
Chambre ch1 = ...;  
// mise à jour de la liste des nouvelles références  
ch1.setHotel(h1);  
h1.getChambres().add(ch1);
```

➤ Méthode plus aboutie

```
Hotel h1 = ...;  
Chambre ch1 = ...;  
// mise à jour de la liste des anciennes références  
Hotel h2 = ch1.getHotel();  
if (h2 != null) { h2.getChambres().remove(ch1); }  
// mise à jour de la liste des nouvelles références  
ch1.setHotel(h1);  
h1.getChambres().add(ch1);
```



La méthode simpliste, ayant de nombreux défauts, est à proscrire

- Deuxième méthode: « Factoriser les MAJ des références »

```
public class Hotel {  
    ...  
    public void ajouterChambre(Chambre ch) {  
        Hotel hBis = ch.getHotel();  
        if (hBis != null) { hBis.retirerChambre(ch); }  
        ch.setHotel(this);  
        this.chambres.add(ch);  
    }  
    public void retirerChambre(Chambre ch) {  
        ch.setHotel(null);  
        this.chambres.remove(ch);  
    }  
}
```

```
Hotel h1 = ...;  
Chambre ch1 = ...;  
// invocation de la méthode de mise à jour  
h1.ajouterChambre(ch1);
```

- o **Attention aux noms des méthodes**

- o **1 idée commune**

- ▶ Il s'agit d'un ajout et non-pas d'une création
- ▶ Quelques exemples :
 - ▶ 1:n → ajouterChambre(...) & retirerChambre(...)
 - ▶ n:n → ajouterReservation(...) & retirerReservation(...)
 - ▶ 1:1 → changerDirecteur(...)

◦ Choix 1

- ▶ Utiliser uniquement des relations uni-directionnelles et INTERDIRE les relations bi-directionnelles

◦ Choix 2

- ▶ Mettre en place initialement uniquement des relations uni-directionnelles puis mettre en place AU BESOIN les relations bi-directionnelles (en pensant aux méthodes ajouter/retirer)

◦ Choix 3

- ▶ Mettre en place initialement des relations bi-directionnelles (en pensant aux méthodes ajouter/retirer)
- ▶ Ne jamais établir de relation bi-directionnelle sans avoir mis en place les méthodes ajouter/retirer !!!

❑ Exemple de bidirectionnalité sur une relation 1-N : classe Company

```
...
@Entity(name="CompanyOMBid")
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;

    ...

    @OneToMany(cascade={CascadeType.ALL},
        fetch=FetchType.EAGER,
        mappedBy="company")
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }

    ...
}
```

❑ Classe Employee

```
...
@Entity(name="EmployeeOMBid")
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;
    private Company company;

    ...

    @ManyToOne
    public Company getCompany() {
        return company;
    }

    public void setCompany(Company company) {
        this.company = company;
    }
}
...
```

- TP 2 bis : relations N-N bidirectionnelle

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ Une clé primaire peut être composée de plusieurs colonnes
- ❑ Peut arriver quand la BD existe déjà ou quand la classe correspond à une table association (association ManyToMany)
- ❑ 2 possibilités :
 - **@IdClass**
 - **@EmbeddedId** et **@Embeddable**
- ❑ La clé primaire doit être représentée par une classe

```
@Entity @Table(name="LINEITEM")  
@IdClass(LineItemId.class)  
public class LineItem implements Serializable {
```

Une clé primaire composée représentée sous la forme d'une classe DOIT :

- ☐ 1 - être publique et fournir un constructeur public par défaut, sans argument
- ☐ 2 - être sérialisable (*implements Serializable*)
- ☐ 3 - implémenter *equals()* et *hashCode()* de manière rigoureuse

❑ Choix 1

- Définir la classe de clé primaire comme objet inclus grâce à `@Embeddable`

```
@Embeddable
public class NomComplet implements Serializable{
    ...
}
```

- Déclarer la clé primaire avec l'annotation `@Id` dans l'entité

```
public class Personne {
    @Id
    private NomComplet nomComplet;
}
```

❑ Choix 2

- Déclarer la clé primaire avec l'annotation `@EmbeddedId` dans l'entité : indique que la clé primaire est dans une autre classe

```
public class Personne {  
    @EmbeddedId  
    private NomComplet nomComplet;  
}
```

- Ne pas stéréotyper la classe de la clé primaire

```
@Embeddable  
public class NomComplet implements Serializable{  
    ...  
}
```

❑ Choix 3

- Dupliquer les propriétés de la classe de clé primaire dans l'entité et les annoter avec @Id
- Annoter l'entité avec @IdClass

```
@Entity
@IdClass(NomComplet)
public class Personne {
    @Id
    private String nom;
    @Id
    private String prenom;
    ...
}
```

- Ne pas stéréotyper la classe de la clé primaire

```
public class NomComplet implements Serializable{
    private String nom;
    private String prenom;
    ...
}
```

- ❑ Deux entités sont souvent créées lorsque on a deux tables liées avec une clé étrangère
- ❑ On peut fusionner les deux tables dans le même objet
- ❑ `@SecondaryTable` et `@SecondaryTables` permettent de mapper une entité à plusieurs tables

❑ Utilisée pour mapper une table secondaire

- Définition de la table
- Déclaration de la jointure entre les tables

```
@Entity
@SecondaryTable (
    name="PERSONNE_CONTACT",
    pkJoinColumns=@PrimaryKeyJoinColumn (name="PERSONNE_ID")
)
public class Personne {
    ...
}
```


- Utilisée pour mapper plusieurs tables secondaires

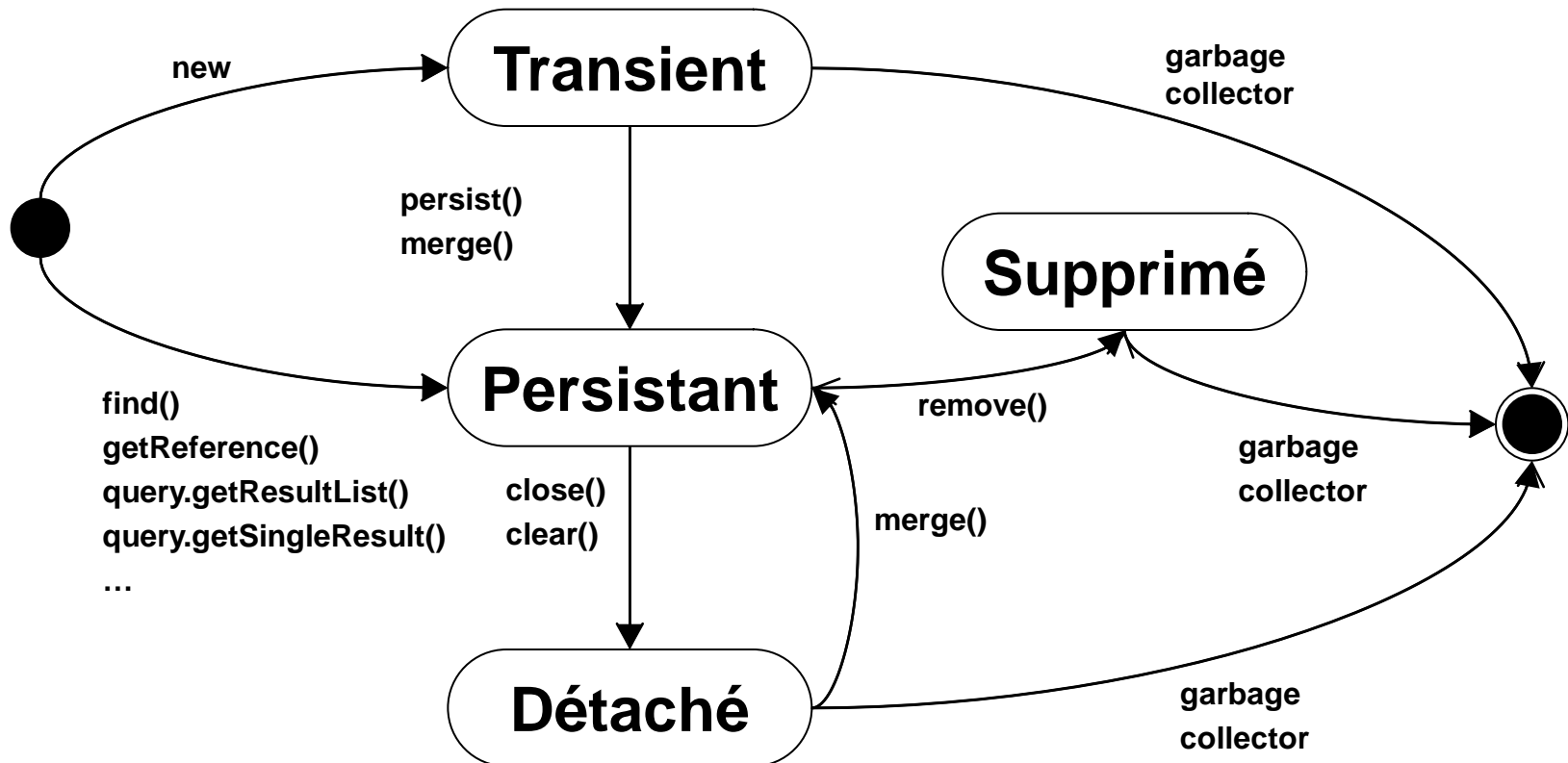
```
@Entity
@SecondaryTables ({
    @SecondaryTable (
        name="PERSONNE_CONTACT",
        pkJoinColumns=@PrimaryKeyJoinColumn (name="PERSONNE_ID") ,
    @SecondaryTable (
        name="PERSONNE_ADRESSE",
        pkJoinColumns=@PrimaryKeyJoinColumn (name="PERSONNE_ID")
    })
})
public class Personne {
    ...
}
```

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ **Mapping des associations = structurel**
 - Résout les aspects statiques de l'antagonisme objet / relationnel
- ❑ **Aspect comportemental important aussi**
- ❑ **Raisonner en terme de cycle de vies et d'états des objets**
 - Plutôt que de raisonner en terme de requêtes SQL

❑ JPA définit 4 états pour les objets

- **Transient** : instancié avec `new`, pas attaché à une Session, pas de représentation correspondante en base de données
- **Persistent** : a une identité en base de données, rattaché à une Session
- **Supprimé** : programmé pour être supprimé en base
- **Détaché** : a été persistant, mais dont la session a été fermée



❑ Garantie de JPA :

- Objets persistants : toujours synchronisés avec le SGBD
- Objets détachés : aucune garantie

❑ Les objets détachés peuvent être

- Fusionnés (« merge ») : retour à l'état persistant

❑ La gestion des états est faite via le contexte de persistance

- EntityManager

- ❑ L' EntityManager JPA « contient » le contexte de persistance
- ❑ Equivalent à un cache d'entités totalement gérées
- ❑ Le contexte de persistance :
 - *Est propre à une unité de travail (≈ transaction)*
 - Vérifie si les objets persistants ont été modifiés
 - « Automatic dirty checking »
 - Effectue les requêtes « en fond »
 - « transactionnal write-behind »
 - Cache de premier niveau (L1 cache)
 - Garantie l'identité des objets

- ❑ **Idée : optimiser les accès à la base de données**
- ❑ **L'état du contexte est propagé vers la base de données à la fin de l'unité de travail**
 - INSERT, UPDATE, DELETE (DML)
- ❑ **Maintient un « historique » de chaque entités**
 - Pour savoir si un ordre SQL est nécessaire
- ❑ **Bénéfices**
 - Diminue les latences réseau
 - Diminue les lock-time de la base de données
 - Ordres SQL envoyés en mode batch (JDBC), donc plus rapide

- ❑ **Mise en cache des entités chargées, pour l'unité de travail**
- ❑ **Améliore les performances**
- ❑ **Garantit l'isolation au sein d'une unité de travail**
 - repeatable-read au sein de l'unité
 - pas de conflits à la fin de l'unité (1 instance = 1 enregistrement)
 - = garantie de l'identité
- ❑ **Evite les Stack Overflow en cas de références cycliques**

❑ Signification des méthodes essentielles de `EntityManager`

- `persist()` : attachement, rend persistant, programmé pour un INSERT
- `find()` : attachement, récupération dans la session ou dans le SGBD
- `merge()` : attachement, le gestionnaire d'entités va « suivre » l'objet pour savoir si un UPDATE est nécessaire
- `remove()` : programme un delete sur l'enregistrement

- ❑ « Flush » du contexte = synchronisation avec le SGBD
- ❑ Quand le flush est-il effectué ? (comportement par défaut)
 - Commit d'une transaction
 - Avant d'effectuer une requête
 - En appelant `EntityManager.flush()`
- ❑ Réglages possible, avec `EntityManager.setFlushMode()`
 - AUTO : cf. ci-dessus
 - COMMIT : seulement lors du commit de la transaction

- ❑ **Assure un fonctionnement optimal**
- ❑ **Performances**
 - Ordres SQL seulement si nécessaire
 - Appels optimisés (mode batch)
- ❑ **Intégrité et sécurité de la persistance**
 - Garantie de l'identité dans une unité de travail
 - Répercussion des modifications dans l'unité

❑ Annotations de callbacks :

- JPA laisse la possibilité aux développeurs de rajouter des traitements **avant** ou **après** qu'un changement d'état d'un bean se produise.

❑ Sept annotations correspondent à ces différents points d'attache :

`@javax.persistence.PrePersist`

`@javax.persistence.PostPersist`

`@javax.persistence.PreRemove`

`@javax.persistence.PostRemove`

`@javax.persistence.PreUpdate`

`@javax.persistence.PostUpdate`

`@javax.persistence.PostLoad`

- ❑ Avant d'insérer un entity bean en base, JPA exécutent les méthodes annotées par **@PrePersist**.
- ❑ Si l'insertion ne rencontre pas de problèmes ou d'exception, les méthodes annotées **@PostPersist** sont exécutées.
- ❑ Il en est de même pour les mises à jour (**@PreUpdate**, **@PostUpdate**) et les suppressions (**@PreRemove**, **@PostRemove**).
- ❑ Par contre, l'annotation **@PostLoad** est appelée lorsqu'un entity bean est chargé à partir de la base de données via une requête ou une association.

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

❑ Lazy loading

- Le chargement est effectué **au plus tard**, uniquement lorsque les données sont effectivement accédées dans le code Java (par un getter)
- Des requêtes SQL supplémentaires sont effectuées lors de l'accès effectif

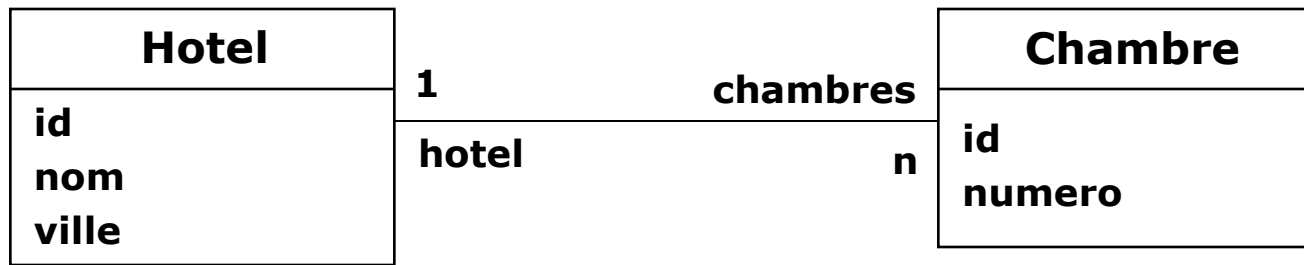
❑ Eager fetching

- Le chargement est effectué **au plus tôt**, dès que les données sont potentiellement accessible dans le code Java
- Une seule requête SQL ramène l'ensemble des données dès le départ

- ❑ **Lazy / Eager loading : choix par défaut des relations :**

Annotation	Default Fetching Strategy
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

❑ Par défaut, fonctionnement en lazy-loading



```
Hotel hotel = em.find(Hotel.class, 501);
```

```
Set<Chambre> chambres = hotel.getChambres();
```

```
for (Iterator<Chambre> ite=chambres.iterator(); ite.hasNext();) {
    Chambre c = ite.next();
}
```

} 1 requête

} 1 autre
requête

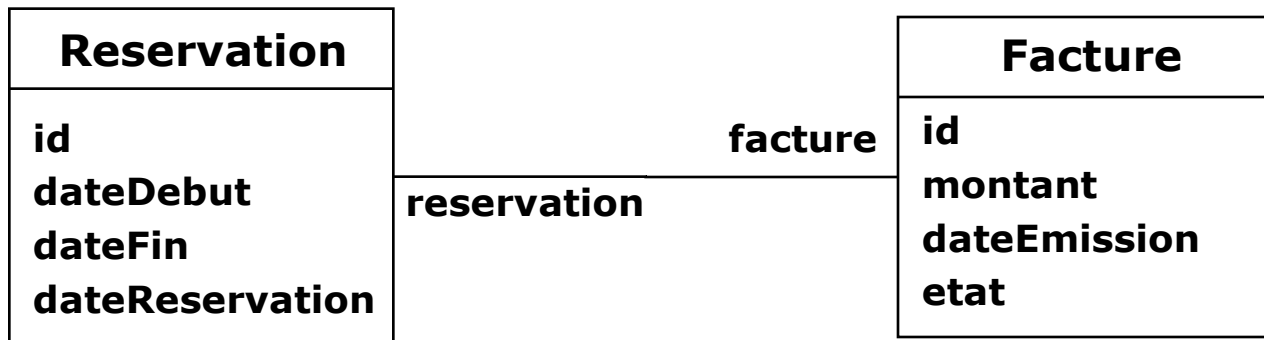
❑ Au chargement, remplacement de la collection déclarée par une collection "interne"

- Par exemple pour Hibernate : PersistentSet, PersistentList...
Implémentent les interfaces standards : java.util.Set, java.util.List...
Package org.hibernate.collection
- Mécanisme de Proxy
A la première utilisation, la collection sous-jacente est chargée



Le lazy loading ne fonctionne pas avec le type "Array"

- ❑ Les relations 1-1 sont implémentées en eager fetching par défaut

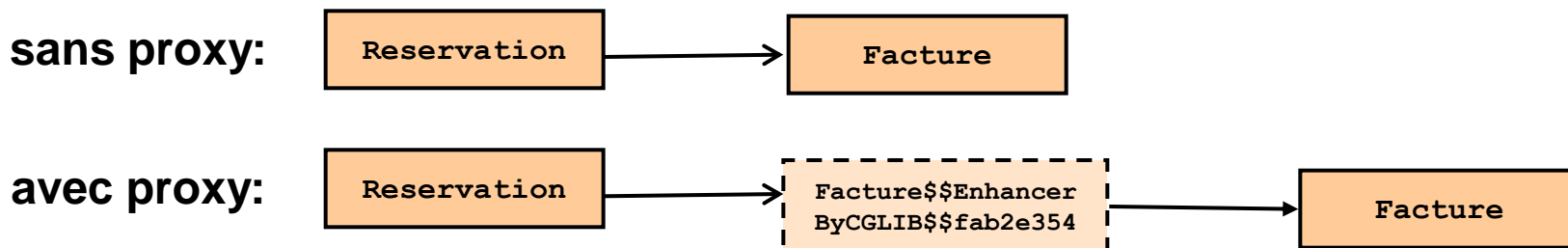


```
Reservation res = em.find(Reservation.class, 501);

Facture fact = res.getFacture();
```

1 requête

- ❑ Possibilité de configurer une relation 1-1 en lazy-loading
 - `@OneToOne (fetch=FetchType.LAZY)`
- ❑ Utilisation d'un proxy sur l'objet référencé
 - Classe créée dynamiquement
 - Bibliothèque cglib
- ❑ Problématique : l'utilisation d'un proxy doit être transparente
 - La classe proxy **hérite** de la classe sous-jacente



```
Reservation res = em.find(Reservation.class, 501);  
  
Facture fact = res.getFacture();
```

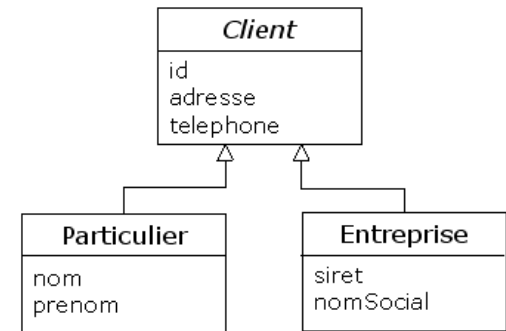
❑ Contraintes liées à l'utilisation d'un proxy (lazy loading)

➤ Attention au polymorphisme

Un proxy sur Client hérite de client

Un proxy sur Client n'hérite pas de Particulier

```
Facture fact = em.find(Facture.class, 501);  
  
Client client = fact.getClient();  
Particulier part = (Particulier) client;
```



ClassCastException !!!

- ❑ Par défaut, les composants (embedded) ne sont pas chargés par des proxies (donc mode eager)



```
Personne personne = em.find(Personne.class, 501);  
NomComplet nom = personne.getNomComplet();
```

1 requête

❑ Dans le mapping :

➤ Pour tout type de relation. Ex :

```
@Entity
public class Hotel {
    ...
    /**
     * Les chambres de l'hotel.
     */
    @OneToMany(fetch=FetchType.EAGER)
    private List<Chambre> chambres;
    ...
}
```

❑ Dans une requête JPQL

➤ Utilisation de JOIN FETCH

❑ Résumé, comportement par défaut

- « lazy loading » pour les collections (@*toMany)
- « eager fetching » pour les associations unitaires (@*toOne), les attributs (@Basic)

❑ Comportement adapté pour la plupart des cas

❑ Déclaration des réglages

- Dans le mapping
- Surcharge possible en JPQL

❑ Bonnes pratiques

- Laisser le réglage par défaut dans le mapping
- Utiliser JOIN FETCH dans les requêtes JPQL pour les cas particuliers

❑ L'attribut « cascade » sur les annotations

- permet de propager certaines opérations effectuées sur une entité vers ses entités liées
- est disponible pour toutes les relations
 - many-to-one
 - one-to-one
 - many-to-many
 - one-to-many

❑ Les valeurs possibles sont

- Aucune valeur : aucune opération n'est propagée
- `CascadeType.PERSIST` : propage l'opération `persist()`
- `CascadeType.MERGE` : propage l'opération `merge()`
- `CascadeType.REMOVE` : propage l'opération `remove()`
- `CascadeType.REFRESH` : propage l'opération `refresh()`
- `CascadeType.ALL` : propage toutes les opérations

❑ Il est possible de combiner plusieurs valeurs



L'absence de valeur ou la valeur « all » sont souvent utilisées

❑ Exemple

- Une **Chambre** appartient à un **Hôtel**
- Un **Hôtel** regroupe plusieurs **Chambres**



————— **OneToMany** —————→

←———— **ManyToOne** —————

❑ Mapping

```
@Entity
public class Chambre {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne(targetEntity = Hotel.class)
    private Hotel hotel;
    ...
}
```

Pas d'attribut "cascade"
⇔ "cascade="none"

❑ Utilisation

```
Hotel h1 = new Hotel();
h1.setNom("BeauRegard");
h1.setVille("Lisbonne");
Chambre ch1 = new Chambre();
ch1.setNumero("143");
ch1.setHotel(h1); // direction de la relation : MANY-TO-ONE
em.persist(h1);   // instruction OBLIGATOIRE
em.persist(ch1);
```



L'omission de « em.persist(h1) » provoque une erreur

❑ Mapping

```
@Entity
public class Chambre {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne(targetEntity=Hotel.class, cascade=CascadeType.ALL)
    private Hotel hotel;
    ...
}
```

Utilisation

```
Hotel h1 = new Hotel();
h1.setNom("BeauRegard");
h1.setVille("Lisbonne");
Chambre ch1 = new Chambre();
ch1.setNumero("143");
h1.ajouterChambre(ch1); // direction de la relation : ONE-TO-MANY
em.persist(ch1); // la demande de sauvegarde est propagée à l'hotel
// l'instruction em.persist(h1); est donc maintenant FACULTATIVE
```

□ TP 4 : classe d'association N-N et cascade

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

Java Persistence Query Language

- ❑ Langage de requêtage objet spécifique à JPA
- ❑ Ressemble de très près au langage de requetage Hibernate HQL
- ❑ Ressemble de loin au langage classique SQL
- ❑ Principale différence avec SQL, l'opérateur "."
 - Pas besoin de connaître le nom des tables, ni le nom des colonnes...
 - On se promène le long des relations

Ex : renvoie tous les clients qui ont passé une commande :

```
SELECT o.customer FROM Order o
```

Ex : renvoie tous les téléphones des clients qui ont passé une commande :

```
SELECT  
o.customer.address.homePhoneNumber  
FROM Order o
```


- ❑ **Elaboration de requêtes objets sur les entités**
- ❑ **Utilisation des annotations**
 - @NamedQueries
 - @NamedQuery
- ❑ **Ces annotations permettent de créer une requête**
- ❑ **Attributs**
 - Name
 - Nom unique dans une unité de persistance pour une requête
 - query
 - Requête à effectuer
- ❑ **Si plusieurs requêtes, utilisation de @NamedQueries**

```

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
@NamedQueries({
    @NamedQuery(name = "findAllInventory",
        query = "select object(o) from
            Inventory o"),
    @NamedQuery(name =
        "findInventoryByYear", query=
        "select object(o) from Inventory o
            where o.year=:year"),
    @NamedQuery(name =
        "findInventoryByRegion", query=
        "select object(o) from Inventory o
            where o.region=?1 ") })
public class Inventory implements
    Serializable {

    ...
}

```

```

import java.util.List;
import javax.ejb.Stateless;
import
    javax.persistence.EntityManager;
Import
    javax.persistence.PersistenceConte
        xt;

@Stateless
public class InventoryManagerBean
    implements InventoryManager {
    @PersistenceContext
    EntityManager em;

    public List<Inventory>
        findAllInventory() {
        return
            em.createNamedQuery("findAllI
                nventory").getResultList();
        }
}

```

❑ 2 possibilités

- Indexation
- Nommage des paramètres

Nommage du paramètre

La contrainte de la requête est donc du type

... where year=:year

```
public List<Inventory> findInventoryByYear(Object year) {  
    return  
    em.createNamedQuery("findInventoryByYear").  
        setParameter("year", year).getResultList();  
}
```

```
public List<Inventory> findInventoryByRegion(Object p1) {  
    return em.createNamedQuery("findInventoryByRegion").  
        setParameter(0, p1).getResultList();  
}
```

Utilisation de l'indexation

La contrainte de la requête est donc du type

... where region=?1

- ❑ Il est possible de créer des requêtes à la volée
- ❑ Utilisation de la méthode `createQuery` à partir d'une instance de `PersistenceContext`

```
public class InventoryManagerBean implements InventoryManager {  
    @PersistenceContext  
    EntityManager em;  
  
    ...  
  
    public List<Inventory> findAllInventory() {  
        return em.createQuery("select o from Inventory  
o").getResultList();  
    }  
  
    public int bulkDeleteEmptyInventory() {  
        return em.createQuery("delete from Inventory o where  
o.quantity =0").executeUpdate();  
    }  
}
```

❑ Recherche par clé primaire :

```
/** Find by primary key. */  
public <T> T find(Class<T> entityClass, Object primaryKey);
```

❑ Exécution de requêtes JPQL

```
public List<Account> listAccounts() {  
    Query query = manager.createQuery("SELECT a FROM Account a");  
    return query.getResultList();  
}
```

❑ Requêtes SQL:

```
public Query createNativeQuery(String sqlString, Class resultClass);

public Query createNativeQuery(String sqlString,
    String resultSetMapping);
```

❑ Requêtes nommées:

```
public List<Account> listAccounts() {
    Query query = manager.createNamedQuery("findThem");
    return query.getResultList();
}
```

@Entity

@NamedQuery(name="findThem", queryString="SELECT a FROM Account a")

public class Account implements Serializable {...}

❑ Requêtes paramétrées :

// customers 20-30 named 'Joe', ordered by last name

```
Query q = em.createQuery("select c from Customer c where  
    c.firstName = :fname order by c.lastName");  
  
q.setParameter("fname", "Joe");  
  
q.setFirstResult(20);  
  
q.setMaxResults(10);  
  
List<Customer> customers = (List<Customer>) q.getResultList();
```

// all orders, as a named query

@Entity

@NamedQuery(name="Order:findAllOrders", query="select o from Order o");

public class Order { ... }

Query q = em.createNamedQuery("Order:findAllOrders");

// all people, via a custom SQL statement

```
Query q = em.createNativeQuery("SELECT ID, VERSION, SUBCLASS,  
    FIRSTNAME, LASTNAME FROM PERSON", Person.class);  
List<Person> people = (List<Person>) q.getResultList();
```

// single-result aggregate: average order total price

```
Query q = em.createQuery("select avg(i.price) from Item i");  
Number avgPrice = (Number) q.getSingleResult();
```

❑ Jointures

Ex : Liste toutes les commandes qui ne comprennent pas (LEFT) de produit dont le prix est supérieur à une certaine quantité (et celles qui ne comprennent pas de produits)

// traverse to-many relations

```
Query q = em.createQuery("select o from Order o  
    left join o.items li where li.price > :price");  
q.setParameter("price", 1000);  
List<Order> orders = (List<Order>) q.getResultList();
```

- Requêter **sur plusieurs attributs** renvoie soit un tableau d'Object, soit une collection de tableaux d'Object

```
texte = "select e.nom, e.salaire " +  
        " from Employe as e";
```

```
query = em.createQuery(texte);
```

```
List<Object[]> liste =  
    (List<Object[]>) query.getResultList();
```

```
for (Object[] info : liste) {  
    System.out.println(info[0] + "gagne" + info[1]);  
}
```

❑ Table des compagnies

ID	NAME
1	M*Power Internet Service, Inc.
2	Sun Microsystems
3	Bob's Bait and Tackle

Table D.1 Company

❑ Table des employés

ID	NAME	COMPANY_ID
1	Micah Silverman	1
2	Tes Silverman	1
3	Rima Patel	2

- ❑ Cette requête récupère trois compagnies :

```
SELECT DISTINCT c FROM CompanyOMBid c
```

- ❑ Mais celle-ci uniquement deux :

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN c.employees
```

- ❑ Celle-là : les trois (même si join condition absente)

```
SELECT DISTINCT c FROM CompanyOMBid c LEFT JOIN c.employees
```

- ❑ **Provoque le chargement des entités reliées**

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN FETCH c.employees
```

- ❑ **Prend le devant sur @FetchType.LAZY**

- ❑ **Autre exemple :**

```
SELECT DISTINCT c  
FROM CompanyOMBid c, IN(c.employees) e  
WHERE e.name='Micah Silverman'
```

❑ WHERE et requêtes paramétrées

```
Query q =  
    em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = ?1").  
        setParameter(1, "M*Power Internet Services, Inc.");
```

❑ Autre exemple avec paramètres nommés

```
Query q =  
    em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = :cname").  
        setParameter("cname", "M*Power Internet Services, Inc.");
```

❑ Expressions

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers between 4 AND 5
```

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers IN(2,5)
```

```
SELECT r FROM RoadVehicleSingle r WHERE r.make LIKE 'M%'
```

❑ Le % dans le LIKE = suite de caractères, le _ = un caractère

```
SELECT r FROM RoadVehicleSingle r WHERE r.model IS NOT NULL
```

```
SELECT c FROM CompanyOMBid c WHERE c.employees IS NOT EMPTY
```


❑ MEMBER OF

```
"SELECT e FROM EmployeeOMBid e, CompanyOMBid c  
WHERE e MEMBER OF c.employees"
```

❑ Sous-Requêtes

```
SELECT c FROM CompanyOMBid c WHERE  
(SELECT COUNT(e) FROM c.employees e) = 0
```

```
functions returning strings ::=
    CONCAT(string primary, string primary) |
    SUBSTRING(string_primary,
               simple arithmetic expression,
               simple arithmetic expression) |
    TRIM([[trim_specification] [trim_character] FROM]
          string primary) |
    LOWER(string primary) |
    UPPER(string_primary)
trim specification ::= LEADING | TRAILING | BOTH

functions_returning_numerics ::=
    LENGTH(string primary) |
    LOCATE(string_primary, string_primary[,
          simple_arithmetic_expression])
```

```
functions returning numerics ::=  
  ABS(simple_arithmetic_expression) |  
  SQRT(simple_arithmetic_expression) |  
  MOD(simple_arithmetic_expression, simple_arithmetic_expression) |  
  SIZE(collection_valued_path_expression)
```

// bulk update: give everyone a 10% raise

```
Query q = em.createQuery("update Employee emp  
    set emp.salary = emp.salary * 1.10");  
int updateCount = q.executeUpdate();
```

// bulk delete: get rid of fulfilled orders

```
Query q = em.createQuery("delete from Order o  
    where o.fulfilledDate is not null");  
int deleteCount = q.executeUpdate();
```

// subselects: all orders with an expensive line item

```
Query q = em.createQuery("select o from Order o where exists  
    (select li from o.items li where li.price > 10000)");
```

□ TP3, TP5

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ Les transactions sont déclarées sur un EntityManager
- ❑ Il est possible d'avoir plusieurs transactions dans le même EntityManager

```
EntityManager em = ...

EntityTransaction transaction = null;

transaction = em.getTransaction();
transaction.begin();

    // traitements...
    ...
transaction.commit();

em.close();
```



Garantir la fermeture systématique de l'EntityManager

❑ Utilisation de JTA (Java Transaction API)

- C'est l'API de gestion des transaction standard proposée par Java EE
- Les transactions sont prises en charge par le serveur d'application Java EE

❑ Configuration de l'unité de persistance

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
  <persistence-unit name="jpa">
    <jta-data-source>java:/myDS1</jta-data-source>
    <properties>
      ...
      <property name="jboss.entity.manager.jndi.name" value="java:/EntityManagers/jpa"/>
    ...
    </properties>
  </persistence-unit>
</persistence>
```

persistence.xml

❑ Gestion des transactions avec JTA

- Utilisation de l'API JTA pour une démarcation programmatique des transactions

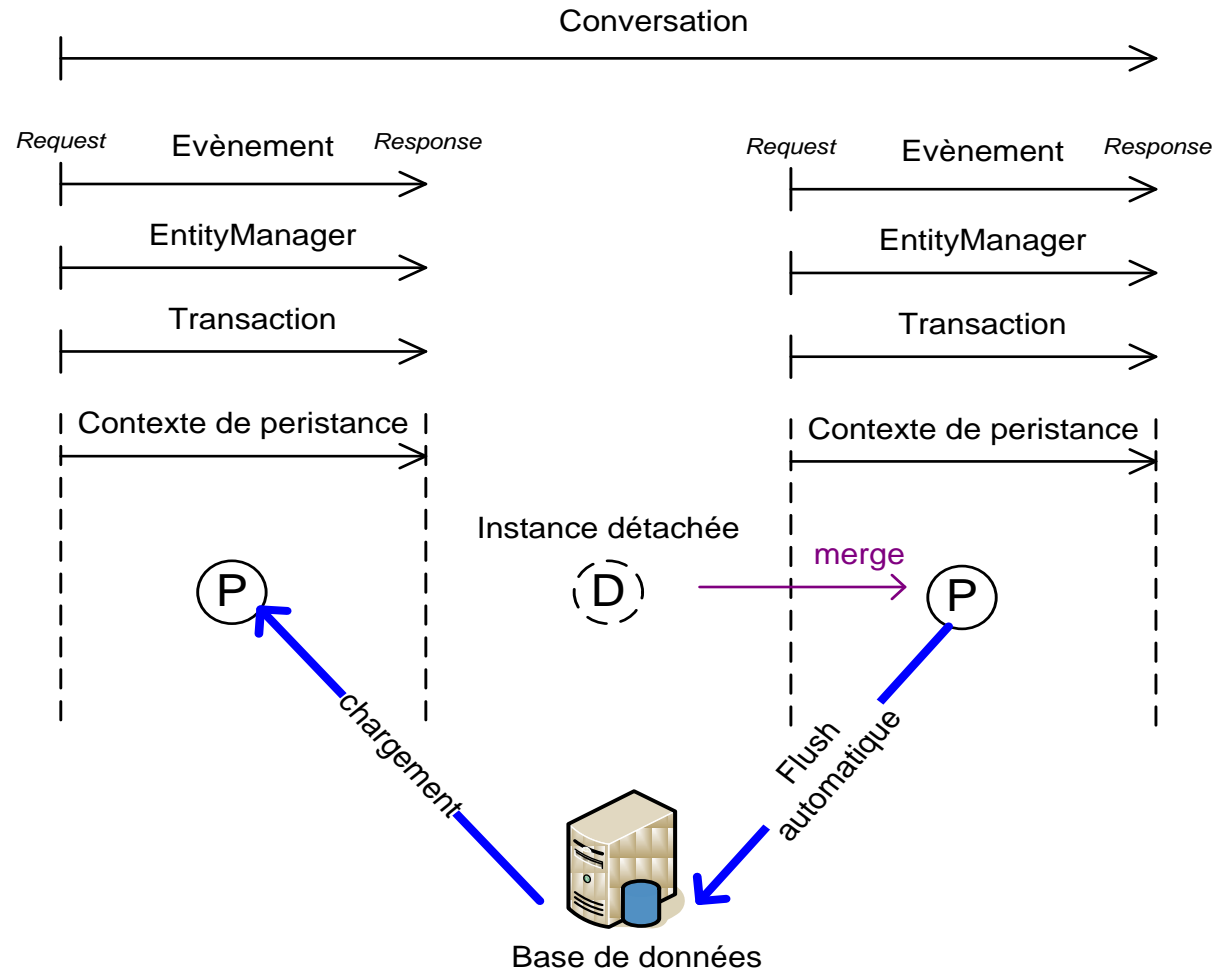
```
InitialContext ctx = new InitialContext();
EntityManager em = (EntityManager)ctx
    .lookup("java:/EntityManagers/jpa");

// Obtenir le gestionnaire de transaction
TransactionManager tm = (TransactionManager)ctx
    .lookup("java:/TransactionManager");

tm.begin();
// traitements
...
tm.commit();
```

- Utilisation de la configuration EJB 3 (annotations ou descripteur XML) pour une démarcation déclarative des transactions

- ❑ Dans une application, une conversation (notion d'« unité de travail ») peut s'étendre sur plusieurs requêtes



❑ Il y a un commit effectif à la fin de chaque requête

- Pas d'atomicité sur l'ensemble de la conversation
- Il n'est pas possible d'avoir de « lecture répétable » sur l'ensemble de la conversation, quelque soit le niveau d'isolation de la transaction de la base de données



Il est possible de solutionner le problème en combinant :

- le mode de flush « manuel » (*), qui permet d'étendre le contexte de persistance sur plusieurs requêtes
- les verrous optimistes
- conserver la transaction (et EM) en session ☹

(*) : spécifique aux différentes implémentations de JPA

❑ Déclaration Java

```
@Entity
public class Client {
    ...
    @Version
    @Column(name = "OBJ_VERSION")
    private int version;
    ...
}
```

- ❑ Les versions sont gérées automatiquement par JPA
- ❑ En cas de conflit, une StaleStateException est lancée
 - Le premier commit s'effectue normalement
 - Lors du 2^{ème} commit, on constate que l'enregistrement a été modifié entre temps. Une exception est lancée.
- ❑ L'utilisation des versions n'augmente pas le nombre de requêtes
 - Exemple de requête générée :

```
update Client set version=2, nom='dupond', prenom='Sam',  
dateNaissance='2006-09-10 09:17:38.0', sexe='m' where  
ID=13 and version=1
```
- Il est possible de forcer l'incrément de la version

```
em.lock(maChambre, LockModeType.WRITE);
```

- ❑ Il est possible d'obtenir un lock sur pessimiste sur une instance

```
em.lock (maChambre, LockModeType.READ) ;
```

Apporte une garantie supplémentaire d'isolation

- Aucune autre transaction ne peut poser un lock sur cette instance ni la mettre à jour
- Toute tentative met en attente la transaction à son origine

- ❑ La spécification JPA ne précise pas qu'un lock « read » puisse être posé sans l'activation du versioning

- Certaines implémentations le permettent (ex : Hibernate)

- ❑ La spécification n'impose pas non plus que le lock « read » soit implémenté par un lock pessimiste !

- En pratique c'est pourtant souvent le cas

- Dépend du type de base de données et de l'implémentation de JPA

❑ La problématique

- 1. La donnée D1 est lue par la transaction TxA
- 2. La donnée D1 est lue par la transaction TxB
- 3. La donnée D1 est mise à jour dans la transaction TxA
- 4. La transaction TxA s'achève par un commit
- 5. La donnée D1 est mise à jour dans la transaction TxB
- 6. La transaction TxB s'achève par un commit

❑ Les alternatives

➤ *Last commit wins*

Les deux transactions réalisent un commit avec succès et le second commit écrase les changements du premier. Aucune erreur n'est levée.

→ Mise en œuvre : comportement par défaut de JPA

➤ *First commit wins*

Le commit de TxA s'effectue avec succès mais le commit de TxB lève une erreur. L'erreur est remontée à l'utilisateur.

→ Mise en œuvre : activation du versioning d'entité de JPA

➤ *Merge conflicting updates*

Le commit de TxA s'effectue avec succès mais le commit de TxB lève une erreur

→ Mise en œuvre : activation du versioning d'entité de JPA et développements spécifiques

→ TP7 : transactions

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ **Les caches permettent de diminuer le nombre de requêtes en base de données**
 - Récupération de données : les résultats des précédentes requêtes sont stockées en mémoire côté Java
 - La mise à jour de données est retardée afin de diminuer le nombre de requêtes.

- ❑ **Intégrité des données**
 - Idéalement, les données stockées dans les caches ne sont jamais modifiées par une application concurrente

- ❑ **La gestion des caches est spécifique à chaque implémentation de JPA**



Le cas concret de l'implémentation Hibernate est traité dans la suite

Cache niveau 1

EntityManager

(généralement mono-utilisateur)

Cache niveau 2

Intégration Hibernate

Implémentation

EHCache, OSCache...

(optionnel, multi-utilisateur)

❑ Propre à chaque Session Hibernate (\approx EntityManager)

❑ Toujours activé

- Essentiel au fonctionnement interne d'Hibernate
- Il est impossible de le désactiver
- Il est possible de le vider

```
em.clear()
```

```
((Session)em.getDelegate()).evict(...)
```

①

```
Chambre chambre1 = em.find(Chambre.class, 501);  
Chambre chambre2 = em.find(Chambre.class, 501);
```

1 requête

②

```
Query query = em.createQuery("from Chambre c where c.id like :expr1");  
query.setParameter("expr1", 501);  
Chambre chambre1 = (Chambre) query.getSingleResult();  
Chambre chambre2 = (Chambre) query.getSingleResult();
```

2 requêtes

③

```
Query query = em.createQuery("from Chambre c where c.id like :expr1");  
query.setParameter("expr1", 501);  
Chambre chambre1 = (Chambre) query.getSingleResult();  
Chambre chambre2 = em.find(Chambre.class, 501);
```

1 requête



**Par défaut, les requêtes n'utilisent pas le cache.
Il faut mettre en place un cache de requête (QueryCache)**

①

```
EntityTransaction transaction = em.getTransaction();
transaction.begin();
Chambre chambre1 = em.find(Chambre.class, 501);
chambre1.setNom("nouveauNom");
chambre1.setCouleur("BLEU");
transaction.commit();
```

2 requêtes

②

```
EntityTransaction transaction = em.getTransaction();
transaction.begin();
Chambre chambre1 = em.find(Chambre.class, 501);
chambre1.setNom("nouveauNom");
em.flush();
chambre1.setCouleur("BLEU");
transaction.commit();
```

3 requêtes

①

```
Chambre chambre1 = em.find(Chambre.class, 501);  
Chambre chambre2 = em.find(Chambre.class, 501);  
Session session = (Session)em.getDelegate();  
session.evict(chambre2);
```

1 élément
enlevé

②

```
Chambre chambre1 = em.find(Chambre.class, 501);  
Chambre chambre2 = em.find(Chambre.class, 501);  
em.clear();
```

Le cache
est vidé

③

```
EntityTransaction transaction = em.getTransaction();  
transaction.begin();  
Chambre chambre1 = em.find(Chambre.class, 501);  
Session session = (Session)em.getDelegate();  
session.evict(chambre1);  
chambre1.setNom("nouveauNom");  
transaction.commit();
```

Que se
passe-t-il ?

- ❑ **Cache de niveau 1 : transactionnel**
- ❑ **Cache de niveau 2 : JVM ou cluster**
- ❑ **Attention à la concurrence !**
 - Le cache de niveau 2 ne peut fonctionner si d'autres applications modifient la base de données
- ❑ **Hibernate ne propose pas d'implémentations de cache**
 - Branchement de caches dédiés

❑ Hibernate propose des « providers »

- pour se brancher avec des systèmes de cache

❑ Propriété `hibernate.cache.provider_class`

- Indique la classe du provider (classe Hibernate)

❑ Les providers disponibles

- `org.hibernate.cache.NoCacheProvider`
- `org.hibernate.cache.HashtableCacheProvider`
- `org.hibernate.cache.EhCacheProvider`
- `org.hibernate.cache.OSCacheProvider`
- `org.hibernate.cache.SwarmCacheProvider`
- `org.hibernate.cache.TreeCacheProvider`

❑ Exemple avec ehcache

```
<ehcache>
  <cache name="com.hotello.Chambre"
    maxElementsInMemory="10000"
    timeToIdleSeconds="20"
    timeToLiveSeconds="400" />
</ehcache>
```

ehcache.xml

```
...
@Entity
@org.hibernate.annotations.Cache(
  usage = org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE)
public class Chambre {
  ...
}
```

Chambre.java

```
<property name="hibernate.cache.use_second_level_cache" value="true" />
<property name="hibernate.cache.provider_class"
  value="net.sf.ehcache.hibernate.SingletonEhCacheProvider" />
```

Persistence.xml

- ❑ Le cache de requête permet de mémoriser le résultat des précédentes requêtes

```
<ehcache>
  <cache name="com.hotello.Chambre"
    maxElementsInMemory="10000"
    timeToIdleSeconds="20"
    timeToLiveSeconds="400" />
</ehcache>
```

ehcache.xml

```
Query query = em.createQuery("from Chambre c where c.identifiant=:expr1");
query.setParameter("expr1", 1);
query.setHint("org.hibernate.cacheable", true);
```

Appel Java

Activation du cache sur la requête

```
<property name="hibernate.cache.use_second_level_cache" value="true" />
<property name="hibernate.cache.provider_class"
  value="net.sf.ehcache.hibernate.SingletonEhCacheProvider" />
<property name="hibernate.cache.use_query_cache" value="true" />
```

Persistence.xml

❑ Le cache de second niveau est utilisé pour les cas suivants

- Chargement par identifiant
- Chargement lazy d'une entité ou d'une collection

❑ Les résultats de requêtes ne sont pas mis en cache

- Comportement par défaut

❑ Utilisation d'un cache de requêtes

- Activation avec une propriété
- Marquage programmatique de la requête comme « cacheable »
- Utile quand les paramètres des requêtes changent peu

❑ Données candidates

- Qui changent peu
- Données de références (ex. : codes postaux, pays...)
- Pas critiques (ex. : CMS)
- Locales à l'application (non partagées)

❑ Mauvaises données candidates

- Qui changent beaucoup
- Données sensibles (ex. : financières)
- Données partagées avec d'autres applications

- ❑ **Stratégie de concurrence : comment mettre/récupérer les objets du cache ?**
 - Réglage à effectuer
- ❑ **Dépend**
 - Des accès aux données
 - De l'isolation que l'on souhaite
- ❑ **4 niveaux : transactionnel, lecture/écriture, lecture/écriture non-strict, read-only**
 - Correspondent à des niveaux d'isolation transactionnels
- ❑ **Influe aussi sur les performances**
- ❑ **Attribut « usage » dans le mapping**

- ❑ **A utiliser pour les données fortement lues**
 - Ratio (lectures / mises à jour) >>> 1
- ❑ **Simplicité : seulement sur les données en lecture/seule**
- ❑ **Pour les données modifiables, non-sensibles**
 - Mettre des timeout adaptés
- ❑ **Optimisations**
 - Benchmark sans cache niv. 2
 - Benchmarks successifs avec une classe, deux classes... n classes mises en cache

1. Introduction JPA : enjeux de la persistance et de l'ORM
2. Les Entity Beans et le mapping ORM
3. Opération CRUD avec l'EntityManager
4. Relations 1-1, 1-N, N-1, N-M
5. Héritage et polymorphisme
6. Utilisation des collections et des Map
7. Bidirectionnalité des relations
8. Clés primaires composées
9. Etats des objets et contexte de persistance
10. Types de chargement : Lazy et Eager loading
11. Langage de requête JPQL
12. Gestion des transactions avec et sans JTA
13. Gestion du cache Niveau 1 et 2
14. Bonnes pratiques : performances JPA 2.1

- ❑ **Les performances d'une application JPA peuvent être grandement améliorées par :**
 - ❑ un choix adapté du mode de récupération des entités associées
 - ❑ l'utilisation d'opérations de modifications en volume, sans création d'entités
 - ❑ une bonne utilisation du cache de 2^{ème} niveau

- ▶ Utiliser la bonne stratégie de chargement (LAZY ou EAGER) en rapport à la taille des données et l'utilisation qui en est faite : pour les petits volumes de données, le mode EAGER a toujours du sens
- La pagination des résultats de requêtes est aussi une fonction important de JPA (pour éviter les problèmes de mémoire quand le volume de données retournées est grand)
- JPA fournit 2 méthodes pour contrôler la pagination:
`setMaxResults(int maxResult)` et `setFirstResult(int startPosition)` sur l'objet Query

- ❑ Activer le cache des Entity est important pour améliorer les performances
- ❑ Activer également le cache au niveau requête : en ajoutant des « hints » dans les requêtes nommées comme ceci :

```
hints={@QueryHint(name="eclipselink.query-results-cache", value="true") }
```

- ❑ Utiliser les interactions par batch JDBC en envoyant des groupes de requêtes insert/update/delete.

Activation dans le persistence.xml avec les propriétés suivantes:

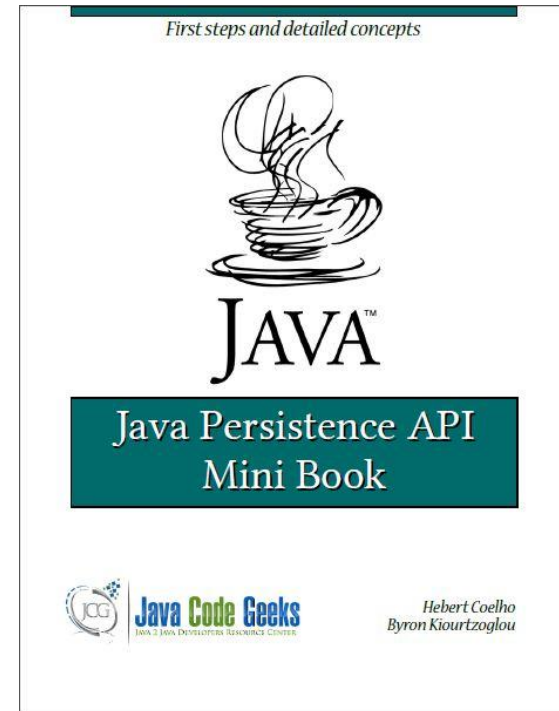
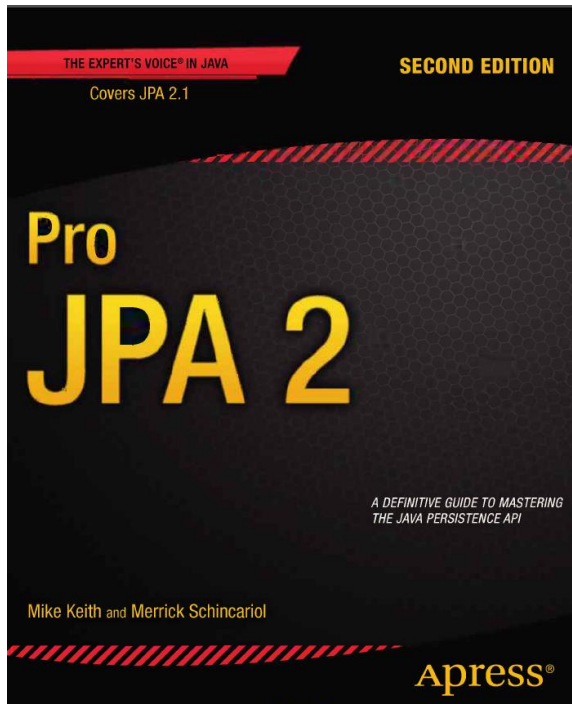
```
"eclipselink.jdbc.batch-writing"="JDBC"
```

```
"eclipselink.jdbc.batch-writing.size"="2000"
```

- ❑ Utiliser les Entity « read-only » quand c'est possible (quand elle ne seront pas modifiées par l'application). Ex. cas de templates d'email

→ Annotation `@ReadOnly` au niveau de l'Entity

❑ Bibliographie



- ▶ **Bonne référence en ligne sur JPA2 :**
 - ▶ <http://www.objectdb.com/api/java/jpa> (en particulier la section JPA2 annotations)
- ▶ **Javadoc Java EE 6 packages : javax.persistence :**
 - ▶ <http://docs.oracle.com/javaee/6/api/index.html?javax/persistence/package-summary.html>

❑ Deux petites choses avant de partir ☺ :

→ **Evaluation du module**

- Prenez le temps d'évaluer ce module en remplissant le formulaire :
- <http://goo.gl/forms/Xg6V36uWi6>

→ **QCM**

- Retournez vos claviers et répondez aux questions du QCM
- Correction en séance

MERCI !

Fin Cours – JPA 2