

Hibernate/JPA

Formateur

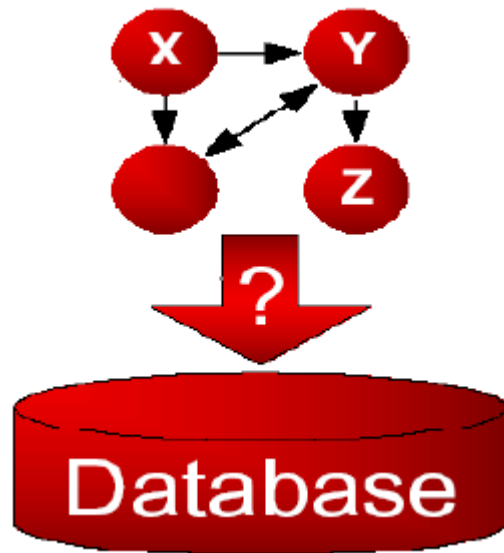


Problématique générale

Comment effectuer la persistance des données d'une application orientée objet dans une base de données relationnelles ?



Problématique



Propriétés à conserver :

- Identification des objets
- Encapsulation
- Classes
- Hiérarchie de classes
- Polymorphisme
- Navigation dans le graphe d'objets
- ...

Propriétés à ajouter :

- Persistance
- Interrogation
- Gestion de la concurrence
- ...

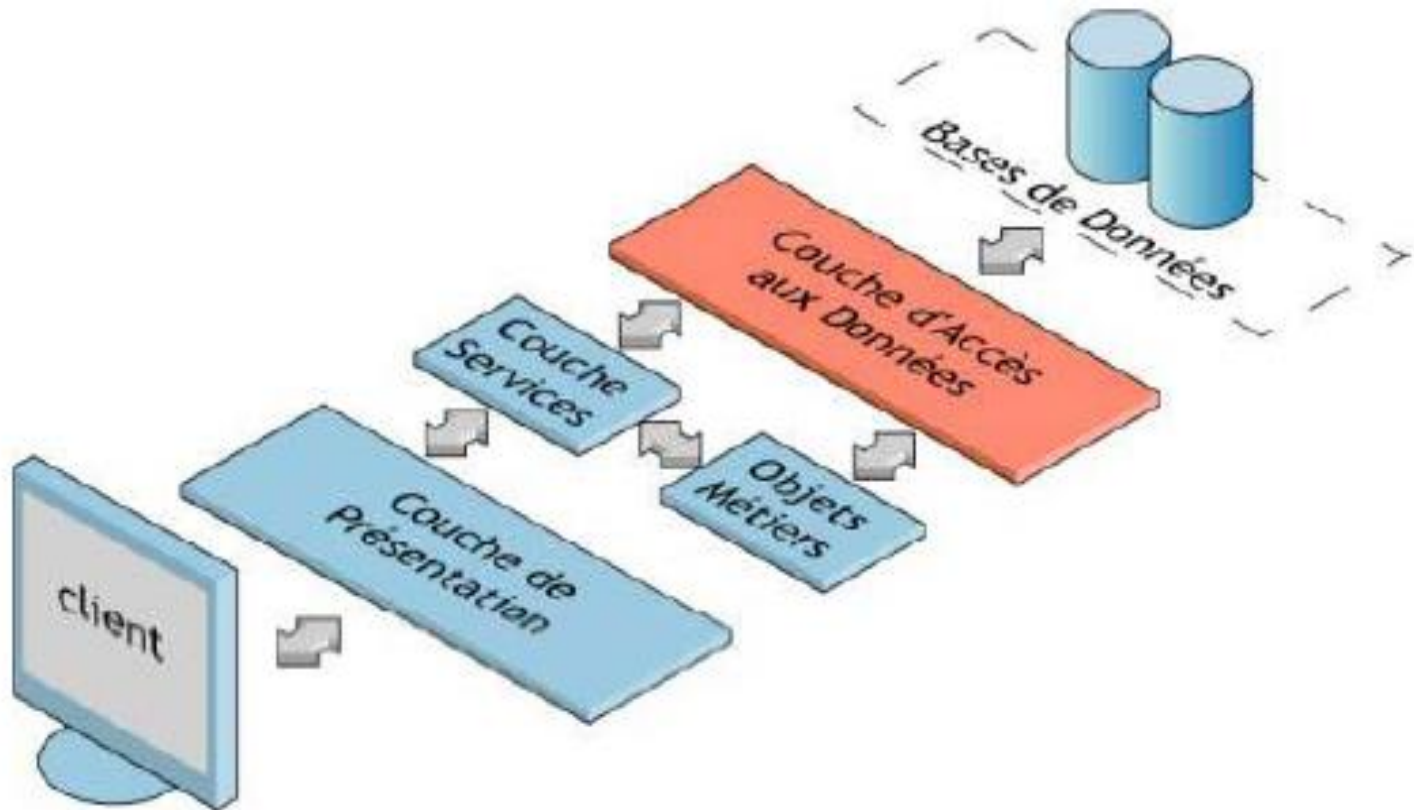


ORM : *Object/Relational Mapping*

- **Persistance automatisée et transparente d'objets métiers vers une bases de données relationnelles.**
- Description à l'aide de **méta-données de la transformation réversible entre un modèle relationnel** et un modèle de classes.
- Capacité à manipuler des données stockées dans une base de données relationnelles à l'aide d'un langage de programmation orientée-objet



Architecture multi-couches



Architecture multi-couches

- **Couche de présentation : logique de l'interface utilisateur**
- **Couche métier : représentation des objets métier – modèle des entités métier**
- **Couche services : traitements représentant les règles métier**



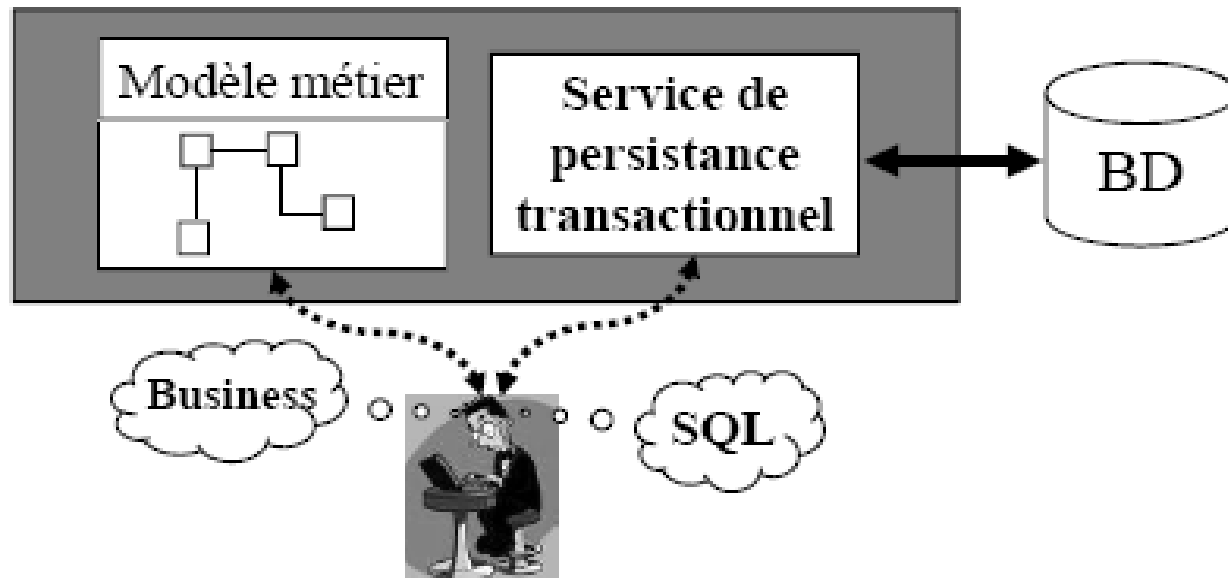
Couche d'accès au données

- Prise en charge de toutes les interactions entre l'application et la base de données
- Groupes de classes et de composants chargés du stockage et de la récupération des données
- Possibilité de servir de cache pour les objets récupérés dans la base de données pour améliorer les performances



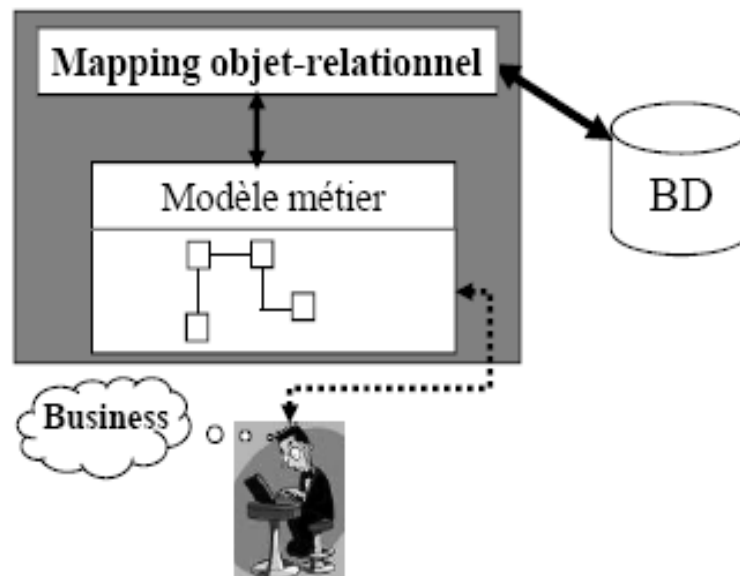
Couche de persistance : à la charge du développeur

- Possibilité de programmer manuellement une couche de persistance avec SQL/JDBC (*Java DataBase Connectivity*)



Couche de persistance : Mapping objet/relationnel

- Utilisation de la couche de persistance comme un service rendant abstraite la représentation relationnelle indispensable au stockage final des objets
- Concentration du développeur sur les problématiques métier



Solutions ORM

Normes Java :

- **EJB (*Enterprise Java Beans*) :**
 - Gestion de la persistance par conteneur (CMP- *Container-Managed Persistence* et BMP – *Beans Managed Persistence*)
- **JDO (*Java Data Object*) :**
 - Spécification de Sun 1999 – JDO 2.0 (JSR243 Mars 2006)
 - Abstraction du support de stockage
 - Implémentation libre : JPOX
- **JPA (*Java Persistence API*) : *Partie des spécifications EJB 3.0 (JSR 220 en Mai)* concernant la persistance des composants**



Solutions ORM

Implémentation de JPA :

- **Hibernate (JBoss) : Solution libre faisant partie du serveur d'appli. JBoss – version 3.3**
implémentant les spécifications JSR 220 –
complète et bien documentée - plugin Eclipse –
- **TopLink (Oracle) : Solution propriétaire utilisée par la serveur d'application d'Oracle**
- **EclipseLink : Solution libre(2008)**
- **OpenJPA (Apache) ...**



Hibernate : généralités (1/2)

- Version 3.2.x : implémentation du standard de persistance EJB 3.0 *Java Persistence API (JPA)*
- Possibilité d'être utilisé aussi bien:
 - dans un développement client lourd,
 - dans un environnement web léger de type Tomcat
 - dans un environnement J2EE complet
- Code SQL généré à l'exécution via des informations fournies dans un document de correspondance (*mapping*) XML ou des *annotations*



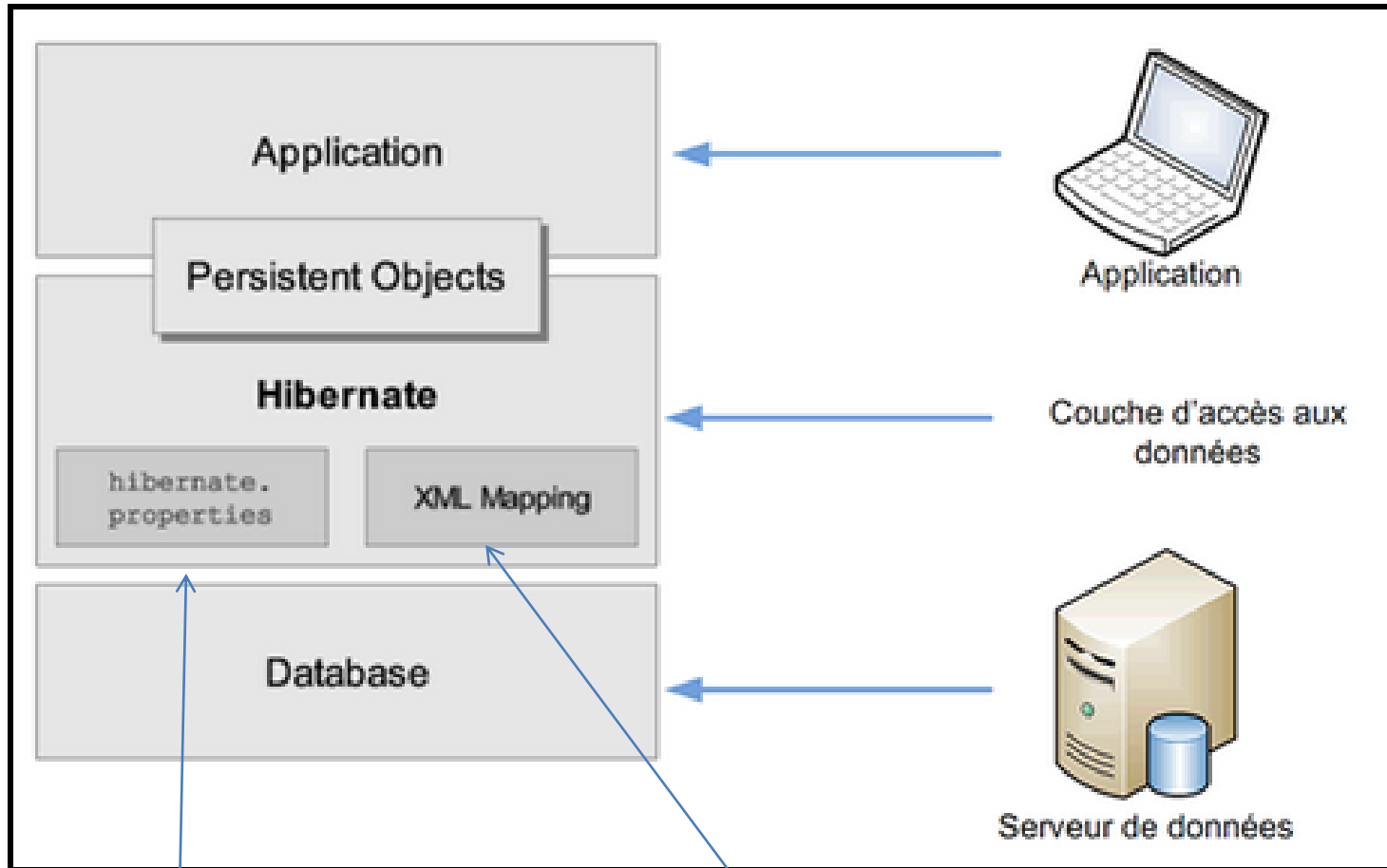
Hibernate : généralités (2/2)

Différents modules :

- ***Hibernate Core*** : API native implémentant les services de base pour la persistance
 - Méta-données au format XML
 - Langage HQL et interface pour écrire des requêtes
- ***Hibernate Annotations*** : Remplacement des fichiers XML par des annotations JDK 5.0 implémentant les annotations du standard JPA + annotations spécifiques à Hibernate
- ***Hibernate Entity Manager*** : Implémentation de la partie des spécifications JPA concernant
 - Les interfaces de programmation,
 - Les règles de cycle de vie des objets persistants
 - Les fonctionnalités d'interrogation
- *Hibernate EntityManager* = wrapper au dessus du noyau Hibernate implémentant une solution complète de persistance JPA



Architecture du noyau Hibernate(1/4)

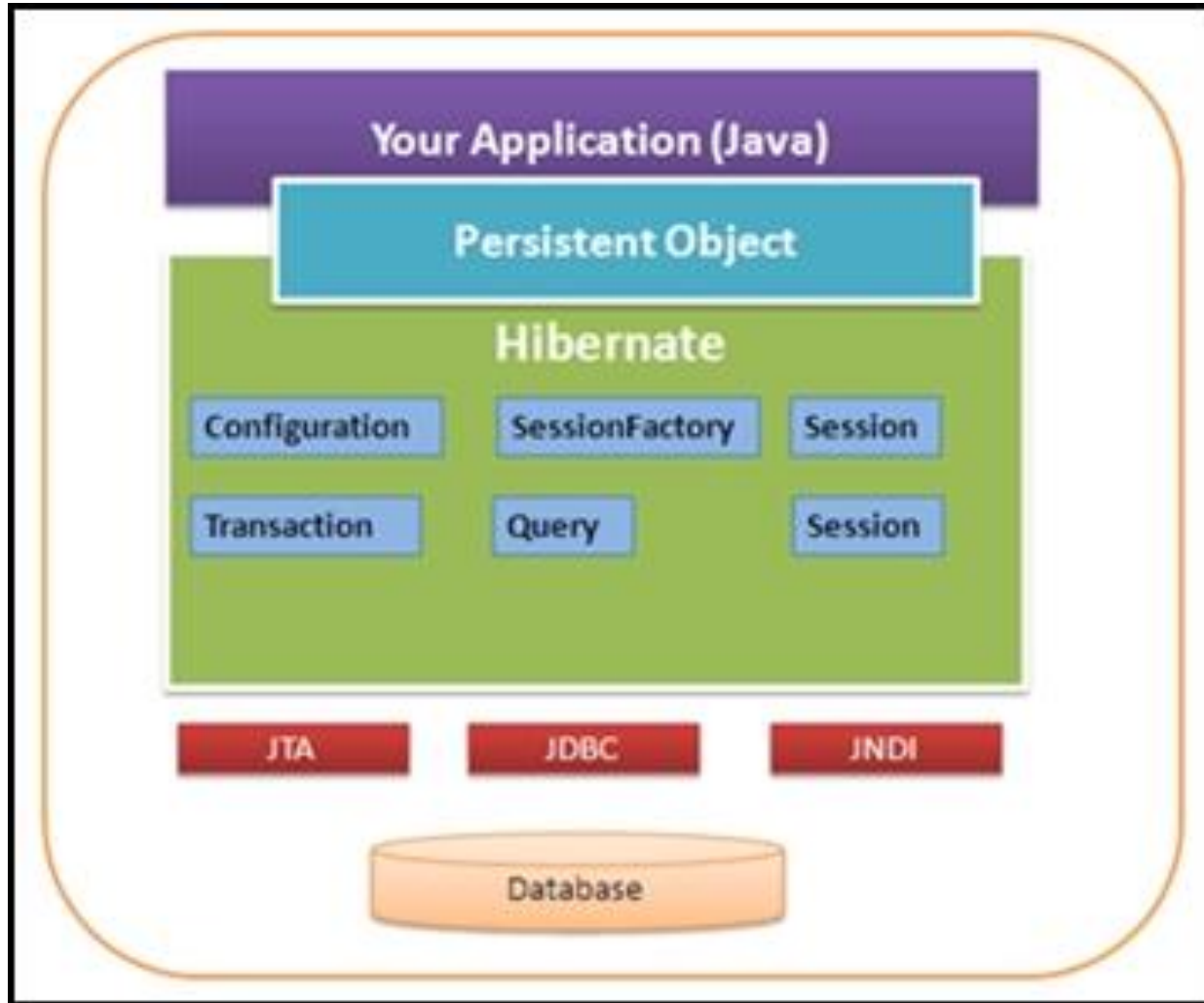


**Fichier de
configuration**

**XML mapping
ou annotations**



Architecture du noyau Hibernate (2/4)



Architecture du noyau Hibernate

(3/4)

SessionFactory (org.hibernate.SessionFactory) :

- Fabrique de Session
- Assure les *correspondances (mappings)* vers une (et une seule) base de données
- Coûteuse à construire car implique l'analyse des fichiers de configuration
- Construite à partir d'un objet Configuration



Architecture du noyau Hibernate (3/4)

Session (`org.hibernate.Session`) :

- Objet *mono-threadé*, à *durée de vie courte*, *représentant une* conversation entre l'application et l'entrepôt de persistance.
- Encapsule une connexion JDBC.
- Contient un cache des objets persistants.



Architecture du noyau Hibernate

(4/4)

```
private static Configuration configuration;
private static SessionFactory sessionFactory;
private Session s;
try {
    // étape 1
    configuration = new Configuration();
    // étape 2
    sessionFactory = configuration.configure().buildSessionFactory();
    // étape 3
    s = sessionFactory.openSession();
} catch (Throwable ex) {
    log.error("Building SessionFactory failed.", ex);
    throw new ExceptionInInitializerError(ex);
}
```

*Consultation du fichier
hibernate.cfg.xml présent
dans le classpath de l'application*

*Analyse du fichier
de mapping*

Conseil : utiliser une classe `HibernateUtil` pour factoriser ces étapes



Environnement de travail avec Hibernate (1/8)

L'utilisation atomique de la session Hibernate doit suivre le schéma suivant :

```
Session session = factory.openSession(); ← ❶ la session s'obtient via factory.openSession()  
// ou Session session = HibernateUtil.getSess(repère 1)  
Transaction tx = null ;  
try {  
    tx = session.beginTransaction(); ← ❷ La gestion de transaction (repère 2) est  
    //faire votre travail ← ❸ indispensable  
    ...  
    tx.commit();  
}  
catch (Exception e) {  
    if (tx != null) { ← ❹  
        try {  
            tx.rollback();  
        } catch (HibernateException he) {  
            throw he;  
        }  
    }  
    throw e;  
}  
finally { ← ❺  
    try {  
        session.close();  
    } catch (HibernateException ex) {  
        throw new XXXException(ex); //exception fatale  
    }  
}
```



Environnement de travail avec Hibernate (2/8)

Fichiers nécessaires avec *Hibernate Core* :

- **hibernate.cfg.xml** : fichier de configuration globale contenant
 - Les paramètres de connexion à la base de données (pilote, login, mot de passe, url, etc.)
 - Le dialecte SQL de la base de données
 - La gestion de pool de connexions
 - Le niveau de détails des traces etc.
- Pour chaque classe persistante :
 - **ClassePersistante.java** : Implémentation POJO (*Plain Old Java Objects*) de la classe
 - **ClassePersistante hbm xml** : Fichier XML de correspondance



Environnement de travail avec Hibernate (3/8)

Exemple de fichier de configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>

    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql:///form_hibernate</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>

        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">create</property>

        <mapping class="com.formation.domain.Contact" />

    </session-factory>
</hibernate-configuration>
```



Environnement de travail avec Hibernate (4/8)

Déclaration du type de document utilisé par l'**analyseur syntaxique (parseur) XML pour valider le document de** configuration d'après la DTD de configuration d'Hibernate :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-
    3.0.dtd">
```

Paramètres de configuration nécessaires pour la connexion JDBC :

```
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql:///form_hibernate</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password"></property>
```

Spécification de la variante de SQL générée par Hibernate :

```
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
```

Activation de la génération automatique des schémas de base de

Données :

```
<property name="hbm2ddl.auto">create</property>
```

Fichier de configuration (fichier de *mapping*) des

classes persistentes :

```
<mapping class="com.formation.domain.Contact" />
```



Environnement de travail avec Hibernate (5/8)

Exemple de fichier de correspondance pour la classe `Contact.java`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.formation.domain">
  <class name="Contact" table="CONTACT">
    <id name="id" column="Support_id">
      <generator class="native"/>
    </id>
    <property name="nom" />
    <property name="prenom" />
    <property name="email" />
  </class>
</hibernate-mapping>
```

Déclaration de la DTD

Classe « mappée »

Mapping de l'identifiant

Lab_H_1



Environnement de travail avec Hibernate (6/8)

Fichiers nécessaires pour *Hibernate Annotation* :

- **hibernate.cfg.xml** : fichier de configuration
- **ClassePersistante.java** : POJO avec annotations
- **ClassePersistante.hbm.xml** : optionnel – possibilité de combiner annotations et méta-données XML

Particularité pour la création de la *SessionFactory* :

- **sessionFactory = new AnnotationConfiguration().buildSessionFactory();**
- Et ajout dans le fichier de hibernate.cfg.xml de **<mapping class="Person"/>**



Architecture du noyau Hibernate

(7/8)

- Mapping par annotation – exemple de la classe [Contact.java](#) :

```
@Entity
public class Contact {

    @Id
    @GeneratedValue
    @Column(name="CONTACT_ID")
    private int id;
    private String nom;
    private String prenom;
    private String email;
```

Lab_H_2



Environnement de travail avec JPA

- C'est la boîte noire qui permet de rendre persistants les beans entités.
- Les beans entités étant des objets simples, ils doivent être gérés (managés) par une unité de persistance qui permet d'intégrer l'utilisation de ces beans entités au sein d'applications java EE.
- Cette unité de persistance doit être configurée.
- Une unité de persistance, c'est :
 - Un ensemble de beans entités déclarés
 - Un fournisseur de persistance (Provider)
 - Une source de données (Datasource)



Environnement de travail avec JPA

- **Rôle :**
 - Savoir où et comment stocker les informations.
 - S'assurer de l'unicité des objets de chaque entité persistante.
 - Gérer les objets et leur cycle de vie : c'est le gestionnaire d'entité (EntityManager).
- **Où peut-on trouver une unité de persistance :**
 - Application d'entreprise (EAR).
 - Module Java Bean Entreprise (EJB-JAR).
 - Application Web (WAR).
 - Client d'application entreprise (JAR)
 - Un environnement Java SE compatible.
- **Paramétrage de l'unité de persistance :**
 - Un fichier décrit les éléments de l'unité de persistance.
 - IL DOIT S'APPELER **PERSISTENCE.XML** et se trouver sous **META-INF**, à la racine du projet.
 - Ce fichier est lu par le conteneur d'EJB lors du déploiement de l'application (EAR, EJB-JAR, WAR, ...)
 - Au moment du déploiement le conteneur vérifie la description proposée et contrôle la concordance avec les éléments qu'il connaît



Environnement de travail avec JPA

Fichiers nécessaires pour *Hibernate* *EntityManager* :

```
persistence.xml X
<?xml version="1.0" encoding="UTF-8"?>
<persistance version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="Lab1_3PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.formation.domain.Contact</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://127.0.0.1:3306/form_hibernate" />
      <property name="javax.persistence.jdbc.password" value="" />
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
  </persistence-unit>
</persistance>
```

- **ClassePersistante.java** : POJO avec annotations



Environnement de travail avec JPA

■ Ce dessus :

- ☐ On donne le nom de la persistence unit, qui sera rappelée dans le code
- ☐ On donne le nom du data-source, qui devra être déclaré dans le Serveur d'App
- ☐ on référence un fichier nommé orm.xml, qui pourra contenir les informations de mapping objet/relationnel, dans le cas où les annotations ne sont pas utilisées
- ☐ On donne les éléments de connexion à la base de données
- ☐ On donne le nom du driver de BDD
- ☐ On demande la recreation de la BDD et de ses tables à chaque déploiement

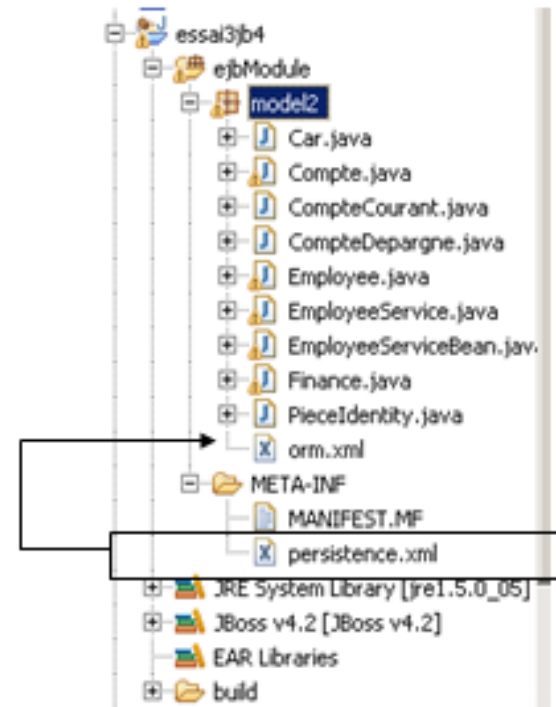
Lab_H_3



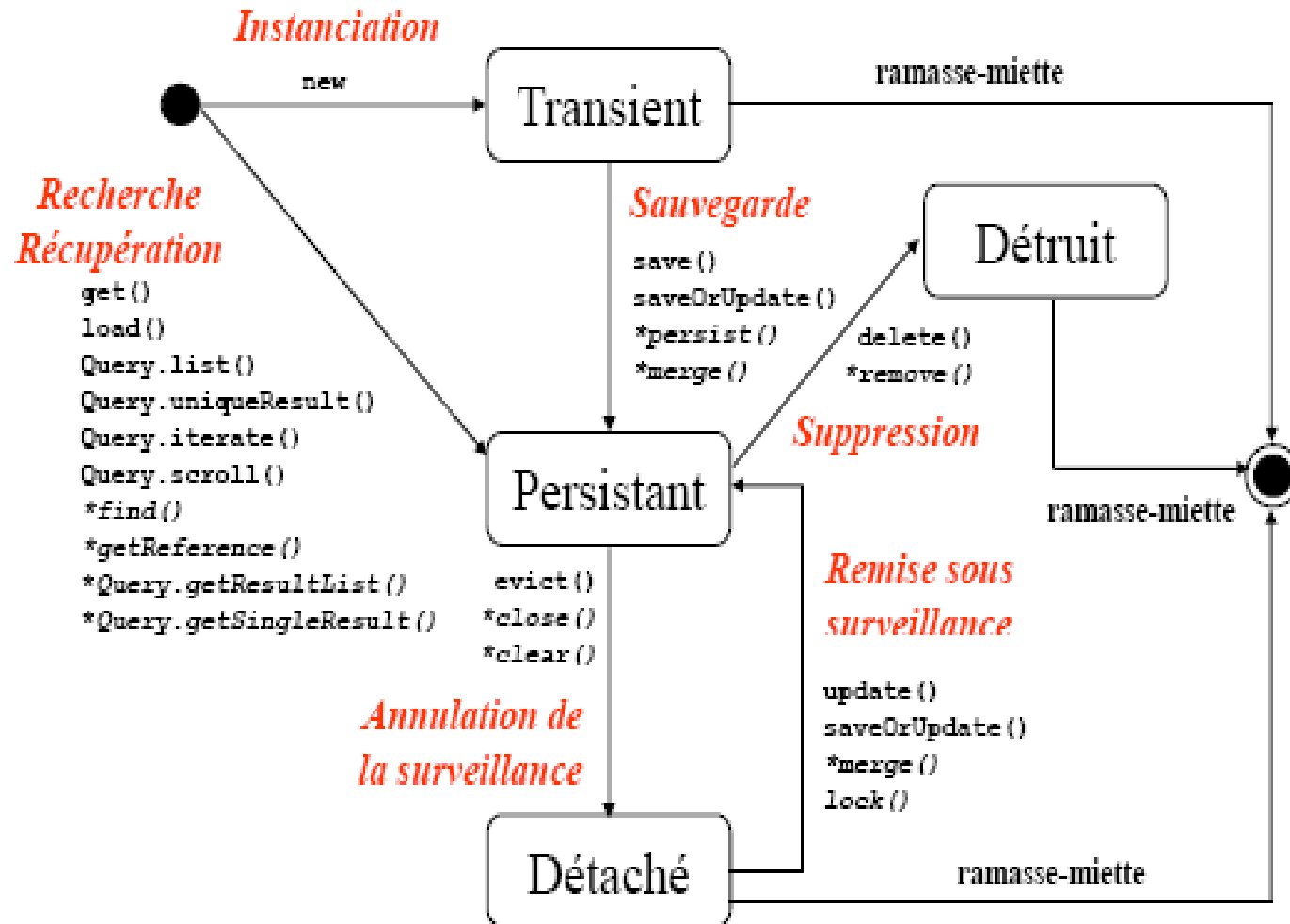
Cas du fichier orm.xml

- JPA permet de déclarer le « binding » dans un fichier à part, nommé **orm.xml**, au lieu de déclarer ces éléments avec des annotations dans les classes entité.
- Ce fichier **orm.xml** doit être pointé depuis le fichier **persistence.xml**
- Exemple de fichier orm.xml et structure typique d'un projet

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
  http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
  <description>pour s'amuser</description>
  <package>model2</package>
  <entity class="model2.Employee" name="Employee">
    <table name="MATABLE"/>
    <attributes>
      <id name="id">
        <column name="macle" length="20"/>
      </id>
      <basic name="name">
        <column name="PRODUCT_NAME" length="100"/>
      </basic>
      <basic name="salary">
        <column length="255"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```



Cycle de vie d'un objet manipulé avec le gestionnaire d'entités



* Méthodes JPA - implémentées dans *Hibernate EntityManager* mais pas dans *Hibernate Core*



Cycle de vie d'un objet manipulé avec le gestionnaire d'entités

- **Passager/Temporaire/Éphémère (*transient*) :**
 - Instance non associée (et n'ayant jamais été associée) à un contexte de persistance
 - Instance sans identité persistante (i.e. valeur de clé primaire)
- **Persistant :**
 - Instance associée à un contexte de persistance (Session)
 - Instance possédant une identité persistante (i.e. valeur de clé primaire) et, peut-être, un enregistrement/nuplet correspondant dans la base.
- **Détaché**
 - Instance ayant été associée au contexte de persistance à présent fermé
 - Instance possédant une identité persistante et peut-être un enregistrement/nuplet correspondant dans la base



Opérations du gestionnaire de persistance

- « Gestionnaire de persistance » = **Session ou EntityManager**
- **Opérations :**
 - Récupérer une instance persistante
 - Rendre une instance persistante
 - Rendre persistantes les modifications apportées à une instance persistante
 - Rendre persistantes les modifications apportées à une instance détachée
 - Ré-attacher une instance détachée
 - Détacher une instance persistante
 - Supprimer une instance persistante
 - Rafraîchir une instance
 - Détecter automatiquement un état



Opérations du gestionnaire de persistance

Récupérer une instance persistante dans *Hibernate Core* :

- **session.load(Class clazz, serializable id)**
⇒ Levée d'une exception s'il n'y a pas de ligne correspondante dans la base de données
- **session.get(Class clazz, serializable id)**
⇒ null si pas de correspondant dans la base
- Possibilité de récupérer une instance aussi par les API **Query, Criteria, SQLQuery**



Opérations du gestionnaire de persistance

Récupérer une instance persistante dans *EntityManager* :

- `entityManager.find(Class clazz, serializable id) ⇔ session.get()`
- `entityManager.getReference(Class clazz, serializable id)`
⇒ Utilisation d'un proxy pour récupérer une référence sur l'objet sans réellement le charger ⇔ `session.load()`



Opérations du gestionnaire de persistance

Rendre une instance persistante :

- **session.save(objet)**
 - Pour rendre persistante un instance temporaire (transiente)
- **session.persist(objet)**
 - En accord avec les spécifications **JPA**
- **session.merge(objet)**
 - Fusionne une instance détachée avec une instance persistante
 - En accord avec les spécifications **JPA**



Opérations du gestionnaire de persistance

Rendre persistante les modifications apportées à d'une instance détachée :

- `session.merge(objet)`
 - En accord avec les spécifications **JPA**
- `session.update(objet)`
- `session.saveOrUpdate(objet)`



Opérations du gestionnaire de persistance

Détacher une instance persistante :

Trois moyens de détacher une instance :

- En fermant la session : **session.close()**
- En vidant la session : **session.clear()**
- En détachant une instance particulière :
session.evict(objet)

Rendre un objet transient :

- Extraction définitive de l'entité correspondante dans la base de données
 - **session.delete(objet)** :



Les relations entre entités

- Nous allons voir ici les différentes sortes de relations qui sont gérées au niveau de JPA :
 - Relation OneToOne
 - Relation OneToMany et ManyToOne
 - Relation ManyToMany
 - Relation d'héritage
 - Relation d'inclusion
- Il s'agit ici, pour les trois premiers cas, de gérer de grappes d'objets.
- Exemple, un employé a plusieurs comptes.

Sur l'entité Employé, la liste (grappe) des comptes va pouvoir être représentée par la propriété relationnelle de la sorte :

```
private List<Compte> comptes= new ArrayList<Compte>();
```

+

Get/Set



Déclarer les annotation du bean entité

- Deux type d'annotation existent pour les beans entité :
 - Les annotations liées aux propriétés de l'objet
 - Les annotations liées aux colonnes de la table
- Ces annotations peuvent se placer soit :
 - Directement sur les attributs (type "**FIELD**")
 - ou
 - Sur les accesseurs (méthodes **get**)

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
@Table(name = "NomTable")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
    public Employee() {}
    public Employee(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
@Table(name = "NomTable")
public class Employee {
    private int id;
    private String name;
    public Employee() {}
    public Employee(int id) {
        this.id = id;
    }
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```



Les annotations des propriétés : **@Basic** et **@Transient**

- Par défaut, toutes les propriétés du bean sont persistantes, il existe donc une valeur par défaut :
 - **@Basic** : Qui prend les attributs suivants :
fetch (FetchType.EAGER par défaut)
- Il existe la possibilité de demander à ne pas persister une propriété : **@transient**



Les annotations des propriétés : @Lob

- L'annotation **@Lob** est utile si vous stockez des tableaux de bytes (**byte[]** ou **Byte[]**) pour représenter le contenu d'un fichier.
- L'annotation **@Lob** s'utilise aussi sur des propriétés de type `java.sql.Clob` (Character Large Object) ou `java.sql.Blob` (Binary Large Object).

```
@Lob  
@Basic(fetch=FetchType.LAZY)  
public byte[] getDonnees() { return donnees; }
```

- Ci dessus, le mode paresseux est utilisé. Effectivement, le contenu de cette propriété risque d'être de grande taille.



Les annotations des propriétés : **@Temporal**

- Les types `java.util.Date` ou `java.util.Calendar` utilisés pour définir des propriétés « temporelles » peuvent être paramétrés pour spécifier le format le plus adéquat à sa mise en persistance.
- Ceci peut être précisé grâce à l'annotation **@Temporal** qui prend en paramètre en **TemporalType** (énumération) dont les valeurs sont les suivantes :
 - **DATE** : utilisé pour la date uniquement (`java.sql.Date`),
 - **TIME** : utilisé pour l'heure uniquement (`java.sql.Time`),
 - **TIMESTAMP** : utilisé pour les temps plus précis, date + heure à la seconde près (`java.sql.TimeStamp`).

```
@Temporal(TemporalType.DATE)  
public Calendar getNaissance() { return naissance; }
```



Les annotation des colonnes @Column

Cette annotation peut s'utiliser conjointement avec les précédentes.

- Les attributs de @Column sont :
 - *name* : précise le nom de la colonne liée. Le nom de la propriété est utilisée par défaut.
 - unique : la propriété est-elle une clé unique ou non ?
 - nullable : la colonne accepte-elle des valeurs nulles ou non?
 - insertable : la valeur doit-elle être incluse lors de l'exécution de la requête SQL INSERT? La valeur par défaut est true.
 - updatable : La valeur doit-elle être mise à jour lors de l'exécution de la requête SQL UPDATE. La valeur par défaut est true.



Les annotation des colonnes @Column

- table : précise la table utilisée pour contenir la colonne. La valeur par défaut est la table principale de l'entité. Cet attribut est utilisé lorsqu'un bean entité est mappé sur plusieurs tables.
- length : précise la longueur que la base de données doit associer à un champ texte. La longueur par défaut est 255.
- precision : précise le nombre maximum de chiffres que la colonne peut contenir. La précision par défaut est définie par la base de données.
- scale : précise le nombre fixe de chiffre après le séparateur décimal (en général le point). Cet attribut n'est utilisable que pour des propriétés décimales (float, double, ...). Le nombre de décimales par défaut est définie par la base de données.

```
@Column(updatable = true, name = "prix", nullable = false, precision = 5, scale = 2)  
public double getPrix() { return prix; }
```



Les annotation des colonnes @Id

- Un bean entité doit posséder un attribut dont la valeur est unique, dit clé primaire.
 - Ce champ permet de différencier chaque objet entité des autres.
 - Cette clé primaire doit être définie une seule fois dans toute la hiérarchie du bean entité.
- Il existe la ----- des types composite

```
import javax.persistence.Entity;  
import javax.persistence.Id;  
@Entity  
public class Employee {  
    @Id  
    private int id;  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```



Les annotation des colonnes **@GeneratedValue**

- Associée à l'annotation **@Id**, l'annotation **@GeneratedValue** permet d'indiquer au conteneur d'utiliser la meilleure solution pour la génération de la clé primaire.
 - Il existe quatre stratégies de génération disponibles : **AUTO**, **IDENTITY**, **SEQUENCE** et **TABLE**.
 - Celles-ci sont définies par l'énumération `javax.persistence.GenerationType`



Les annotation des colonnes @GeneratedValue

- **IDENTITY** : indique au provider de persistance d'assigner la valeur de la clé primaire en utilisant la colonne identité de la base de données.

Sous MySQL, la clé primaire auto-générée est marquée avec **AUTO_INCREMENT**

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public int getId() { return id; }
public void setId(int id) { this.id = id; }
```

- **SEQUENCE** : Oblige le fournisseur de persistance à utiliser une séquence de BDD. Celle-ci peut être déclarée au niveau de la classe grâce à l'annotation @SequenceGenerator et ses attributs :

```
@Entity
@SequenceGenerator(name = "SEQ_USER",
sequenceName = "SEQ_USER")
public class Utilisateur implements Serializable {
    private int id;
    @Id
    @GeneratedValue(strategy =
    GenerationType.SEQUENCE, generator =
    "SEQ_USER")
```



Les annotation des colonnes @GeneratedValue

- TABLE : Indique au fournisseur de persistance d'utiliser une table annexe pour générer les clés primaires numériques.
- Attributs de l'annotation @TableGenerator associés :
 - *name* : nom de la table annexe.
 - *table* : nom de la table dans la base de données.
 - *pkColumnName* : nom de la colonne qui contient le compteur de clé primaire.
 - *pkColumnValue* : spécifie la colonne de la clé primaire liée.
 - *allocationSize* : définit le nombre d'incrémentations effectuées lorsque le fournisseur demande à la table une nouvelle valeur. Cela permet au fournisseur d'utiliser un système de cache afin de ne pas demander une nouvelle valeur à chaque demande d'un nouvel id

```
@Entity
@TableGenerator(
    name="CONTACT_GEN",
    table="GENERATOR_TABLE",
    pkColumnName="lacle",
    valueColumnName="hi",
    pkColumnValue="id",
    allocationSize=25
)
public class Utilisateur implements Serializable {
    private int id;
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "CONTACT_GEN")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
```



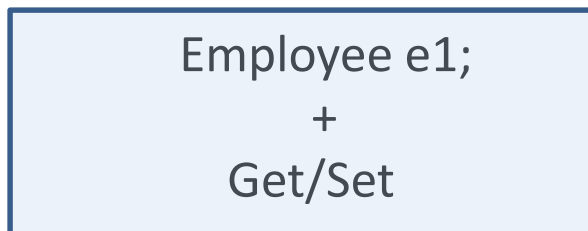
Méthodes callbacks

- Chaque étape du cycle de vie d'un bean entité peut être tracée avec des méthodes forcément signées de la sorte :
 - `public void avantpersistance(){...;}`
 - Les annotations suivantes permettent une trace du cycle de vie :
 - **@PrePersist, @PreRemove, @PostPersist, @PostRemove, @PreUpdate, @PostUpdate, @PostLoad**
 - Les méthodes annotées avec **@PrePersist ou @PreRemove** :
 - sont invoquées sur un bean entité avant l'exécution des méthodes `persist()` et `remove()`
 - Les méthodes annotées avec **@PostPersist ou @PostRemove**
 - sont invoquées sur un bean entité après l'exécution des méthodes `persist()` et `remove()`.
 - Les méthodes annotées avec **@PreUpdate ou @PostUpdate**
 - sont invoquées sur un bean entité respectivement avant et après la mise à jour de la base de données.
 - La méthode annotée avec **@PostLoad**
 - est invoquée après le chargement, depuis la base de données, du bean entité dans le contexte de persistance



Les relations entre entités

- Le représentation de la grappe d'objets va pouvoir se faire avec les types suivants :
 - `java.util.Collection`, `Set`, `List` et `Map`
- On comprend, dans l'exemple précédent, que l'entité Employé a connaissance, maîtrise sa liste de compte.
- Mais un compte doit-il avoir conscience de l'Employé auquel il appartient ?
 - Si oui, alors on parle de relation bidirectionnelle et on ajoute, sur Compte :



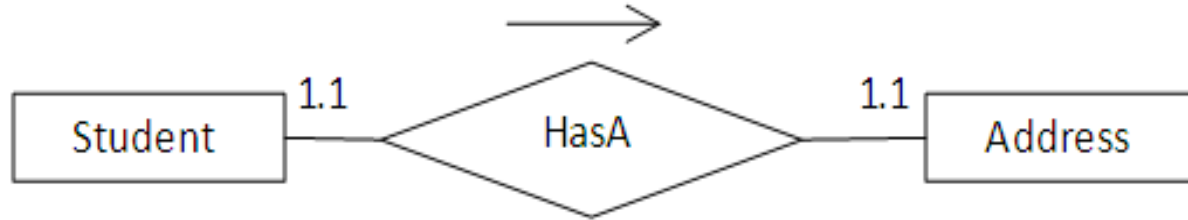
Relation OneToOne

- Utilisée pour lier deux entités uniques indissociables.
- Pour associer deux entités avec ce type de relation, il faut utiliser l'annotation **@OneToOne**
- Liste des attributs de l'annotation :
 - **cascade** : spécifie les opérations à effectuer en cascade
 - **mappedBy** : spécifie le champ propriétaire de la relation dans le cas d'une relation bidirectionnelle.
- **Important** : Cette relation peut être créée de deux façons dans la base de données :
 - Même valeur pour les clés primaires des deux entités.
 - On utilise alors l'annotation **@PrimaryKeyJoinColumn**
 - Clé primaire et étrangère
 - On utilise alors d'annotation **@JoinColumn**



Relation OneToOne

Exemple:



Relation OneToOne avec Foreign Key

Ci-dessous, exemple avec `@JoinColumn`

```
@Entity
public class Student implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private long studentId;
    private String studentName;

    @OneToOne
    @JoinColumn(name="ADR_ID")
    private Address studentAddress;
```

```
@Entity
public class Address implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private long addressId;
    private String street;
```



Comment créer ces entités

OneToOne ?

- Il est nécessaire de créer deux objets entités séparément
- Pour ces deux objets, vous devez définir la clé primaire qui doit être identique
- Le plus important encore, c'est que vous devez rendre persistant chacun de ces beans entités

```
tx.begin();

Address address1 = new Address("OMR Road", "Chennai", "TN", "600097");
Student student1 = new Student("Eswar", address1);
em.persist(student1);
em.persist(address1);

tx.commit();
```



Modification et suppression des entités OneToOne

- Suppression

```
Student employee = em.find(Student.class, 11);  
em.remove(employee.getStudentAddress());  
em.remove(employee);
```

- Modification
 - Idem les deux cotés

EXERCICE Lab_H_4_OneToOneUni



Relation OneToOne avec clé primaire et étrangère : @JoinColumn

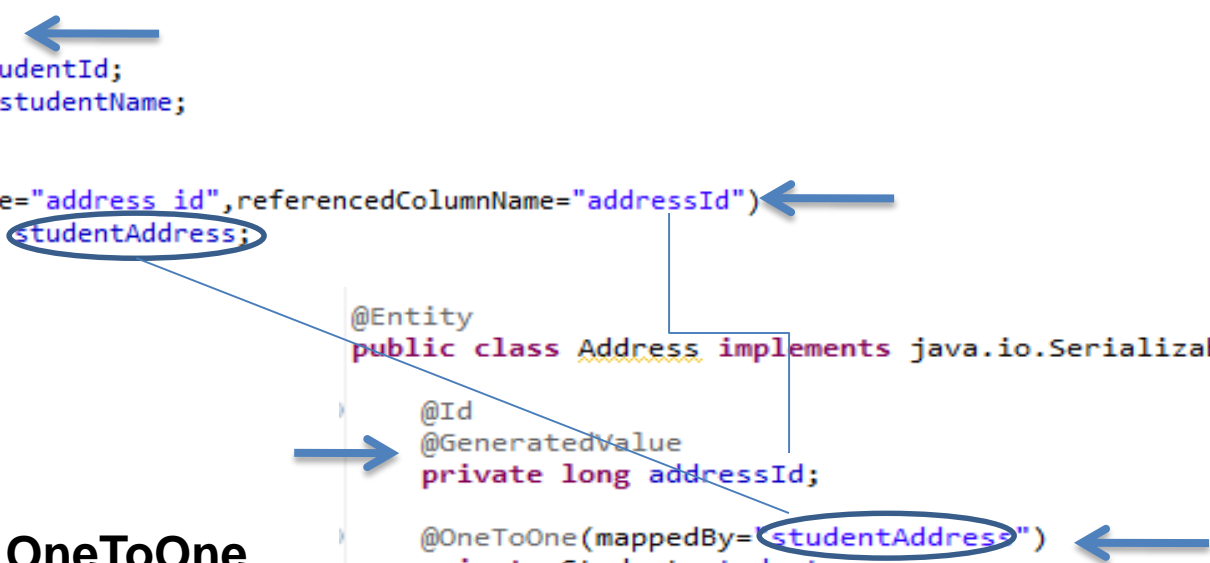
- La deuxième solution consiste à utiliser une clé étrangère d'un côté de la relation.
- L'annotation à utiliser est @JoinColumn. Elle permet de paramétrer la colonne de jointure à utiliser.
- Voir les modifications ci deesous, où l'on passe également à la biderectionnalité

```
@Entity
public class Student implements java.io.Serializable {
    @Id
    @GeneratedValue
    private long studentId;
    private String studentName;

    @OneToOne
    @JoinColumn(name="address_id",referencedColumnName="addressId")
    private Address studentAddress;
}

@Entity
public class Address implements java.io.Serializable {
    @Id
    @GeneratedValue
    private long addressId;

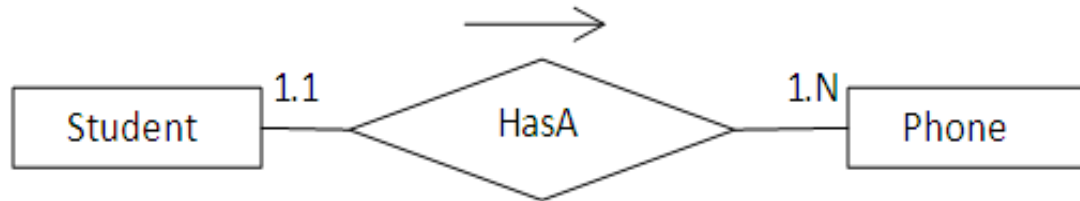
    @OneToOne(mappedBy="studentAddress")
    private Student student;
}
```



Lab_H_5_OneToOne
BI



Cas de la relation OneToMany unidirectionnelle



Dans ce cas, le côté « n » n'a pas la conscience de l'existence du côté one

```
@Entity
public class Student implements java.io.Serializable {

    @Id
    @GeneratedValue
    private long studentId;
    private String studentName;

    @OneToMany
    @JoinTable(name = "STUD_PHONE",
        joinColumns = @JoinColumn(name = "STUD_ID"),
        inverseJoinColumns = @JoinColumn(name = "PHONE_ID"))
    private List<Phone> studentPhoneNumbers;
```

Nom de la table de jointure

Nom Colonne pour StudentId

Nom Colonne pour PhoneId

Optionnel



Traitement de la collection

- Exemple de ce que pourrait être un attachement des Phones à Student

```
List<Phone> phoneNumbers = new ArrayList<Phone>();
Phone phone1=new Phone("house","32354353");
Phone phone2=new Phone("mobile","9889343423");
phoneNumbers.add(phone1);
phoneNumbers.add(phone2);
em.persist(phone1);
em.persist(phone2);
Student student = new Student("Eswar", phoneNumbers);
em.persist(student);
```

EXERCICE OneToMany_1

- Récupération des Phones de Student

Il devient important ici de prendre en compte la valeur du « **lazy loading** ».



Importance ici du « **lazy loading** » ou « **eager loading** »

- La valeur « lazy » signifie que le chargement de la collection n'aura lieu qu'à une demande expresse du client, ceci à des fins d'optimisation.
- La valeur « **eager** » entraîne un chargement immédiat.
- Deux façons de faire:
 1. La valeur « **eager** » est positionnée.
 2. Si la valeur « **lazy** » avait été positionnée, on aurait pu forcer la récupération de la sorte :

```
Student student=em.find(Student.class,1L);  
student.getStudentPhoneNumbers().size();
```

EXERCICE OneToMany_1 (suite)



Cas de la relation OneToMany bi-directionnelle

- Dans le cas d'une relation bidirectionnelle, l'autre côté doit utiliser l'annotation `@ManyToOne`.
 - Si tel est le cas, le côté « n » pourra toujours obtenir son propriétaire.

```
@Entity
public class Student implements java.io.Serializable {

    @Id
    @GeneratedValue
    private long studentId;
    private String studentName;

    @OneToMany(mappedBy="student")
    private List<Phone> studentPhoneNumbers;

    ...
}
```

```
@Entity
public class Phone implements java.io.Serializable {

    ...

    @ManyToOne
    private Student student;

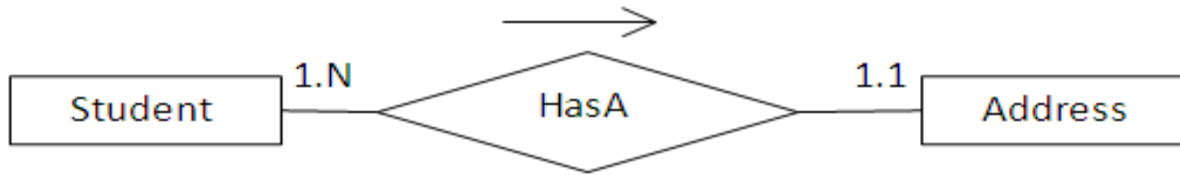
    public Student getStudent() {
        return student;
    }

    public void setStudent(Student student) {
        this.student = student;
    }
}
```

EXERCICE OneToMany_2



Cas de la relation ManyToOne unidirectionnelle



- EXERCICE ManyToOne



Cas des relations ManyToMany

- Ce type de relation peut être utilisé pour lier deux entités entre-elles.
 - *Par exemple, entre des articles et des catégories.*
 - *Un article peut être associé à plusieurs catégories (cardinalité n) et une catégorie peut regrouper plusieurs articles (cardinalité m).*
- *Pour cela, il suffit d'utiliser des propriétés de type collection de chaque côté de la relation (si celle-ci est bidirectionnelle) et de les annoter **@ManyToMany** (de chaque côté).*
- *La façon de procéder est totalement identique aux cas précédents de **OneToMany**.*
- *En terme relationnel, cette relation impose l'utilisation d'une table d'association qui se met automatiquement en place*
 - *Vous avez la possibilité de choisir vos noms de colonnes au travers de l'annotation **@JoinTable***



Cas des relations ManyToMany

Exemple ManyToMany bidirectionnel

```
@Entity  
public class Employee implements  
Serializable {
```

```
    @ManyToMany  
    (cascade=CascadeType.ALL,  
     fetch=FetchType.EAGER)  
    private List<Car>cars= new  
    ArrayList<Car>();  
}
```

```
@Entity
```

```
public class Car {
```

```
...
```

```
    @ManyToMany(cascade=CascadeType.  
    ALL ,fetch=FetchType.LAZY)  
    private List<Employee>employee= new  
    ArrayList<Employee>();
```



Les opérations en cascade

- Les annotations **@OneToOne**, **@OneToMany**, **@ManyToOne** et **@ManyToMany** possèdent toutes l'attribut *cascade*.
 - Celui-ci spécifie les opérations à effectuer en cascade.
- La cascade signifie qu'une opération appliquée à une entité se répercute (se propage) sur les autres entités qui sont en relation avec elle.
- Par exemple, lorsqu'un utilisateur est supprimé, son compte peut être automatiquement supprimé également.



Les opérations en cascade

- Il existe quatre opérations possibles sur les entités : **ajout, modification, suppression, rechargement.**
- Ces opérations sont regroupées dans l'énumération CascadeType :
 - **CascadeType.PERSIST :**
 - automatise l'enregistrement des entités liées à l'association marquée lors de l'enregistrement de l'entité propriétaire - méthode persist().
 - **CascadeType.MERGE :**
 - automatise l'enregistrement des modifications des entités liées à l'association marquée lors de l'enregistrement des modifications de l'entité propriétaire - méthode merge().
 - **CascadeType.REMOVE :**
 - automatise la suppression des entités liées à l'association marquée lors de la suppression de l'entité propriétaire - méthode remove().
 - **CascadeType.REFRESH :**
 - automatise le rechargement (côté base de données) des entités liées à l'association marquée, lors du rechargement de l'entité propriétaire - méthode refresh().
 - **CascadeType.ALL :**
 - cumule les quatre types de cascade.
- **CascadeType.REMOVE** ne peut être appliqué qu'aux associations Un à Un ou Un à Plusieurs.



Les opérations en cascade

- En clair, par exemple, cela signifie
 - que vous n'êtes plus obligé de gérer les opérations d'enregistrement (persist) des objets membres déclarés en cascade persist ou all
 - Exemple :
 - Si l'on ajoute le cascading sur la relation @OneToMany ci-dessous, alors, page suivante

```
@Entity
public class Employee implements Serializable {
    @Id
    private int id;
    @OneToMany
(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    private Collection<Compte> comptes= new
    ArrayList<Compte>();

    public Collection<Compte> getComptes() {
        return comptes;
    }
    public void setComptes(Collection<Compte> comptes) {
        this.comptes = comptes;
    }
}
```

```
@Entity
public class Compte {
    @Id
    @GeneratedValue
    private int id;

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
}
```



Les opérations en cascade

```
Employee emp = new Employee();  
emp.setId(id);  
emp.setName(name);  
emp.setSalary(salary);  
Compte c1 = new Compte();  
c1.setBanque("soc gen");  
    c1.setType_compte(1);  
    // em.persist(c1);  
    Compte c2 = new Compte();  
c2.setBanque("cred lyon");  
    c2.setType_compte(2);  
    // em.persist(c2);  
emp.getComptes().add(c1);  
emp.getComptes().add(c2);
```

**Fonctionne,
mais inutile**



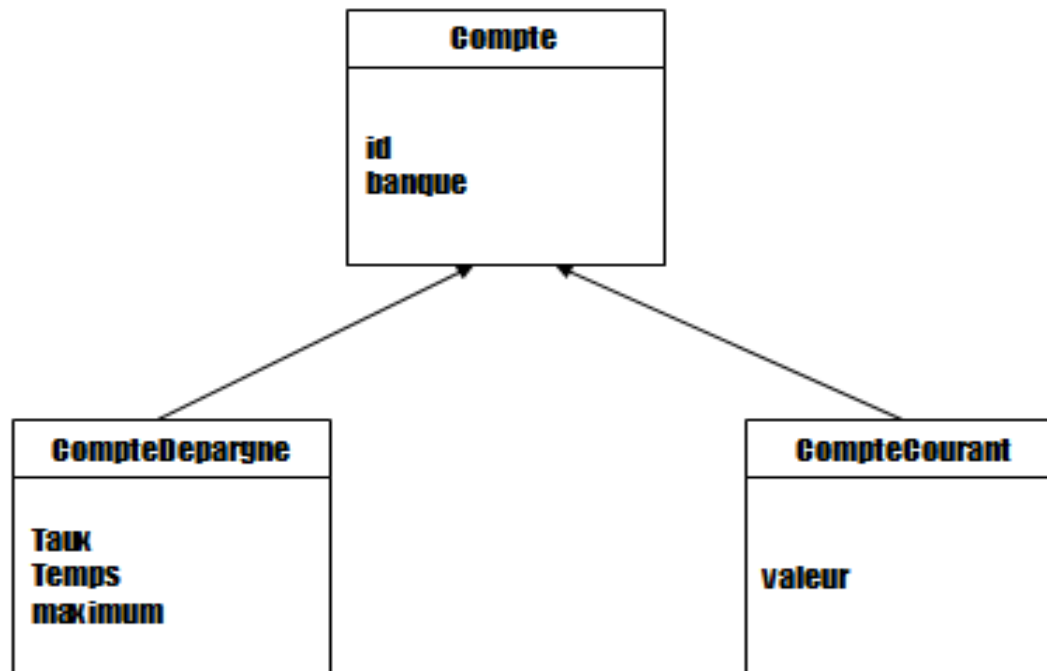
L'héritage

- Cette notion est la même que pour la programmation objet.
- Par contre, la représentation relationnelle d'un héritage d'objets est tout à fait particulier au niveau des tables de la base de données.
- Nous pouvons avoir :
 - Une seule table unique pour toute la hiérarchie des classes.
 - Une table pour chaque classe concrète.
 - Une séparation des attributs spécifiques de la classe fille par rapport à ceux de la classe parente. Il existe ainsi, une table pour chaque classe fille, plus une table pour la classe parente.



Exemple d'héritage

- **Important** : il n'y aura qu'une seule clé primaire pour l'ensemble de la hiérarchie.
 - Le plus simple est de la positionner dans la classe mère.
 - Exemple :



Cas de l'héritage avec table unique

- Dans cette stratégie, toutes les classes de la hiérarchie sont mappées dans une même et unique table.
- Le type d'héritage est spécifié au niveau du bean entité ancêtre à l'aide de l'annotation **@Inheritance**.
 - Le seul attribut de cette l'annotation **@Inheritance** est strategy.
 - Pour définir un héritage utilisant la stratégie "Table unique", c'est la valeur

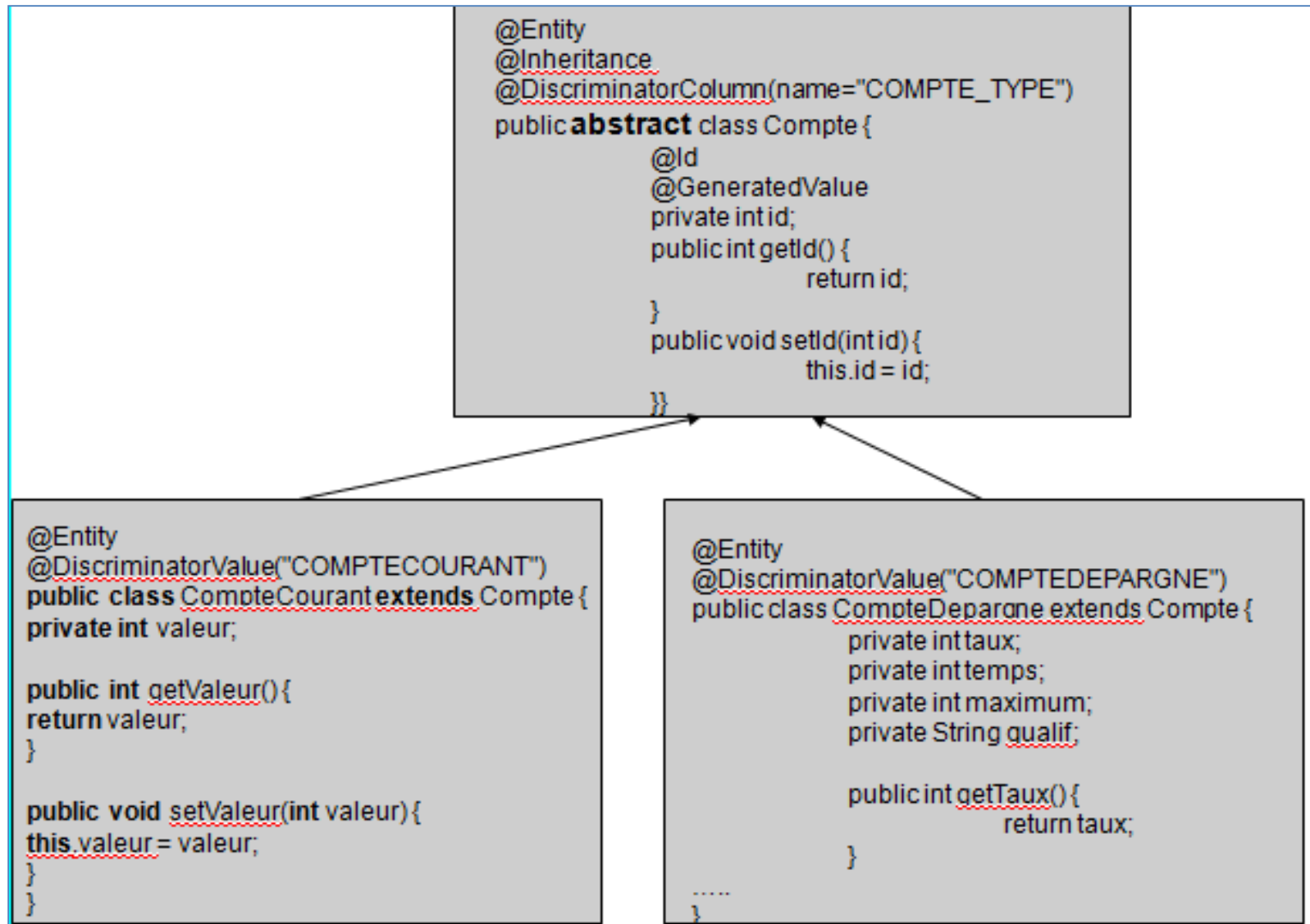


Cas de l'héritage avec table unique

- Cette stratégie requiert une colonne permettant de différencier la classe fille qui va être réellement utilisée dans la hiérarchie pour chacune des lignes présentes dans la table
 - Cette colonne particulière se nomme **discriminateur**.
 - Pour l'identifier, vous devez utiliser l'annotation **@DiscriminatorColumn** afin de préciser tous les détails la concernant.
 - Les attributs de cette annotation sont alors :
 - **name** : nom de la colonne de discrimination qui sera utile pour indiquer le nom de la classe utilisée dans la hiérarchie.
 - **discriminatorType** : classe du discriminateur à utiliser via l'énumération `DiscriminatorType`. Celle-ci possède les attributs suivants :
 - » **CHAR**
 - » **INTEGER**
 - » **STRING**
 - **length** : taille de la colonne pour les discriminateurs à base de chaîne de caractères.
 - **columnDefinition** : fragment SQL à utiliser pour la déclaration de la colonne (utilisé lors de la génération des tables par le conteneur).
- Les classes filles concrètes peuvent préciser la valeur discriminatoire grâce à l'annotation **@DiscriminatorValue**.



Cas de l'héritage avec table unique exemple



Cas de l'héritage avec table par classe concrète

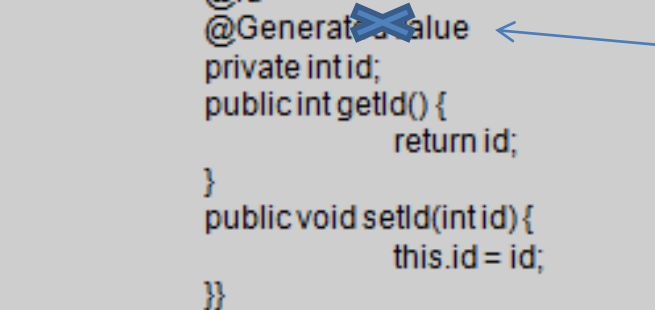
- La seconde stratégie est une table par classe concrète. Dans ce cas, chaque classe représentant le bean entité choisi est liée à sa propre table.
- Cela signifie que toutes les propriétés de la classe (avec celles récupérées par héritage) sont incluses dans la table liée à cette entité
- Pour spécifier cette stratégie, il faut juste choisir **InheritanceType.TABLE_PER_CLASS** dans l'annotation **@Inheritance**.
- Les classes filles héritent simplement sans avoir besoin de préciser des spécifications supplémentaires autres que **@Entity**.



Cas de l'héritage avec table par classe concrète

EXERCICE HERITAGE_2

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Compte {
    @Id
    @GeneratedValue
    private int id;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```



```
@Entity
public class CompteCourant extends Compte {
    private int valeur;

    public int getValeur() {
        return valeur;
    }

    public void setValeur(int valeur) {
        this.valeur = valeur;
    }
}
```

```
@Entity
public class CompteDeparone extends Compte {
    private int taux;
    private int temps;
    private int maximum;
    private String qualif;

    public int getTaux() {
        return taux;
    }
}
.....
}
```



Cas de l'héritage avec table jointe

- Dans cette dernière stratégie, la classe ancêtre des entités est représentée par une table.
- Chaque classe fille est liée à sa propre table séparée contenant uniquement ses propriétés spécifiques.
- La liaison entre les tables filles et la table parente se fait via les clés primaires. En effet, la clé primaire de la classe fille est liée à celle de la classe parente.
- L'héritage est ici déclaré avec la stratégie **InheritanceType.JOINED** dans l'annotation **@Inheritance**.
- Encore une fois, les classes filles n'ont pas besoin d'autre annotation que **@Entity**



Cas de l'héritage avec table jointe

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Compte {
    @Id
    @GeneratedValue
    private int id;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

```
@Entity
public class CompteCourant extends Compte {
    private int valeur;

    public int getValeur() {
        return valeur;
    }

    public void setValeur(int valeur) {
        this.valeur = valeur;
    }
}
```

```
@Entity
public class CompteDepargne extends Compte {
    private int taux;
    private int temps;
    private int maximum;
    private String qualif;

    public int getTaux() {
        return taux;
    }
    .....
}
```



Avantages et inconvénients des trois stratégies

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
Avantages	Aucune jointure, donc très performant	Performant en insertion	Intégration des données proche du modèle objet
Inconvénients.	Organisation des données non optimale	Polymorphisme lourd à gérer	Utilisation intensive des jointures, donc baisse de performance



Les objets incorporés

- On appelle ces objets des **embedded objects** (objets incorporés).
- Lorsqu'une classe est prévue pour être incorporée dans une autre, il faut préciser cette particularité au moyen de l'annotation **@Embeddable**.
- Lorsqu'une classe utilise un objet incorporé, elle doit également préciser cette agrégation au moyen d'une annotation spécifique **@Embedded**



Les objets incorporés : exemple

```
@Embeddable
public class Finance {
    private int type_compte;
    private String banque;
    public int getType_compte() {
        return type_compte;
    }
    public void setType_compte(int type_compte) {
        this.type_compte = type_compte;
    }
    public String getBanque() {
        return banque;
    }
    public void setBanque(String banque) {
        this.banque = banque;
    }
}
```

```
public abstract class Compte {
    @Id
    @GeneratedValue
    private int id;

    @Embedded
    private Finance fi;
    public int getId() {
        return id;
    }
    public Finance getFi() {
        return fi;
    }
    public void setFi(Finance fi) {
        this.fi = fi;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```



JPQL/HQL



Hibernate Query Language (HQL)

- **Hibernate fournit un langage d'interrogation extrêmement puissant qui ressemble au SQL.**
- **HQL est totalement orienté objet, comprenant des notions d'héritage, de polymorphisme et d'association.**
- **Sensible à la casse**
- **La clause from, avec un alias s, renvoie toutes les instances de Support**

from Support s



Hibernate Query Language (HQL)

Associations et jointures

- Exemple :
- **from** Proposition proposition **left outer join** proposition.plans **as** plans
- Supporte :
 - inner join (jointure fermée)
 - left outer join (jointure ouverte par la gauche)
 - right outer join (jointure ouverte par la droite)



Hibernate Query Language (HQL)

select

- La clause **select** sélectionne les objets et propriétés qui doivent être retournés dans le résultat de la requête
- **select** proposition.budget **from** Proposition **as** proposition
- Remarque: on peut ne pas mettre le mot clé **as**



Hibernate Query Language (HQL)

Fonctions d'agrégation

- **select avg(proposition.budget),
sum(proposition.budget),
max(proposition.budget),
count(proposition)
from Proposition proposition**



Hibernate Query Language (HQL)

Requêtes polymorphiques

- **from** Support as s
 - Retourne non seulement les instances de Support, mais aussi celles des sous classes comme Plan.
- Les requêtes Hibernate peuvent nommer n'importe quelle classe ou interface Java dans la clause from.
- La requête retournera les instances de toutes les classes persistantes qui étendent cette classe ou implémente cette interface.
- La requête suivante retournera tous les objets persistants :
`from java.lang.Object o`



Hibernate Query Language (HQL)

where

- La clause where vous permet de réduire la liste des instances retournées

from Support s **where** s.nom="Hibernate"



Hibernate Query Language (HQL)

Expressions

- opérateurs mathématiques +, -, *, /
- opérateur de comparaison binaire =, >=, <=, <>, !=, like
- opérateurs logiques and, or, not
- Parenthèses (), indiquant un regroupement
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of
- ...



Hibernate Query Language (HQL)

order by

- La liste retournée par la requête peut être triée par n'importe quelle propriété de la classe ou du composant retourné :

from Proposition proposition **order by**
proposition.contexte



Hibernate Query Language (HQL)

group by

- Si la requête retourne des valeurs agrégées, celles ci peuvent être groupées par propriété ou composant :
- **select** proposition.budget,
count(proposition)
from Proposition as proposition
group by proposition.contexte



Hibernate Query Language (HQL)

Sous-requêtes

- Pour les bases de données le supportant, Hibernate supporte les sous requêtes dans les requêtes.
- Une sous requête doit être entre parenthèses
- **from** Proposition importante **where** importante.budget > (select



Hibernate Query Language (HQL)

Exécuter des requêtes

```
Transaction tx = session.beginTransaction();
Query q = session.createQuery("from Support");
for (Iterator<Support> i = q.list().iterator() ; i.hasNext() ; )
{
    Support s = i.next();
    System.out.println(s.getId() + " " + s.getNom());
}
tx.commit();
```



Hibernate Query Language (HQL)

Lier des paramètres

```
Transaction tx = session.beginTransaction();
Query q = session.createQuery("from Support s where s.nom=:n");
q.setString("n", "UML");
for (Iterator<Support> i = q.iterate();i.hasNext() ; ) {
    Support s = i.next();
    System.out.println(s.getId() + " " +s.getNom());
}
tx.commit();
```

EXERCICE HQL



Hibernate Query Language (HQL)

Externaliser des requêtes nommées (xml)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.entity">
  <class name="Insurance" table="insurance">
    <id name="lngInsuranceId" type="long" column="ID">
      <generator class="increment" />
    </id>

    <property name="insuranceName">
      <column name="INSURANCE_NAME" />
    </property>
    <property name="investissementAmount">
      <column name="INVESTED_AMOUNT" />
    </property>
    <property name="investissementDate">
      <column name="INVESTEMENT_DATE" />
    </property>
  </class>

  <sql-query name="mySqlQuery">
    <return-scalar column="id" type="long" />
    <return-scalar column="name" type="string" />
    <return-scalar column="amount" type="int" />

    SELECT
    i.ID AS id,
    i.INSURANCE_NAME AS name,
    i.INVESTED_AMOUNT AS amount
    FROM Insurance i WHERE i.INSURANCE_NAME LIKE 'Life%'
  </sql-query>
</hibernate-mapping>
```



Hibernate Query Language (HQL)

- Externaliser des requêtes nommées

Accès par programmation

```
Query query = session.getNamedQuery("mySqlQuery");  
List list = query.list();  
for (Iterator it = list.iterator(); it.hasNext();) {  
    Object [] ins = (Object []) it.next();  
    System.out.println("Insurance Id: " + ins[0]);  
    System.out.println("Insurance Name: " + ins[1]);  
    System.out.println("Insurance Amount: " + ins[2]);  
}
```



Hibernate Query Language (HQL)

- On peut créer des requêtes SQL dans Hibernate via la session Ou les externaliser

```
Transaction tx = session.beginTransaction();
SQLQuery q = session.createSQLQuery("SELECT {s.nom} FROM support AS {s}");
for (Iterator<String> i = q.list().iterator() ; i.hasNext() ; ) {
    String nom = i.next();
    System.out.println( nom );
}
tx.commit();
```



CRITERIA

- Haut niveau
- API de recherche orientée objet
- Intuitive & simple
- Requête dépendant des actions utilisateurs
- Nombreuses fonctionnalités (restrictions, tris, filtres, ...)



CRITERIA

OPERATEURS

- Opérations logiques : **and, not, or**
- Opérations ensemblistes : **between, in, conjunction, disjunction**
- Opérations de comparaisons : **=, <=, >=, <, >, !=**
- Comparaisons de chaînes de caractères : **like, ilike**



CRITERIA

EXAMPLE

- Criteria criteria =
sess.createCriteria(Author.class)
.add(Restrictions.like("lastname", "Toto"));
List results = criteria.list();
- Criteria criteria =
sess.createCriteria(Author.class)
.add(Restrictions.ilike("lastname", "TOTO"));
List results = criteria.list();



CRITERIA

TRIS & FILTRES

- Trier les résultats de façon ascendante ou descendante
- Limiter le nombre de résultats retournés 90

- Par exemple :

Criteria criteria =

```
session.createCriteria(Author.class)
```

```
    .add(Restrictions.between("weight", 55, 80))
```

```
    .addOrder(Order.asc("weight"))
```

```
    .addOrder(Order.desc("lastname"));
```

```
List results = criteria.list();
```



PRÉCISIONS SUR BATCH_SIZE

```
// set the JDBC batch size (it is fine somewhere between 20-50)  
hibernate.jdbc.batch_size 30
```

```
Transaction T = session.beginTransaction();  
for (int i = 0; i < 200000; ++i) {  
    Record r = new Record(...);  
    session.save(record);  
    if ((i % 30) == 0) {  
        // 30, same as the JDBC batch size  
        // flush a batch and release memory  
        session.flush();  
        session.clear();  
    }  
}  
T.commit();
```



JPQL Le requêtage

- La base :

SELECT e FROM Employee e

- La différence avec SQL est que l'on a remplacé le nom de table par le nom de l'entité (Employee)
- On utilise aussi un alias, e, pour retourner les données
- On retourne donc ici un ou n entités Employee.
 - Attention à la casse sur le nom des entités
- Dans ce dernier cas, le résultat est un ou n types String retournés.
 - **SELECT e.name from Employee e**
- Si maintenant, on demande :
 - **SELECT e.pi from Employee e**
 - » On ramene, dans ce dernier cas, des types pi, à savoir Pièce d'identité



JPQL Le requêtage

- Comme SQL, JPQL supporte la clause **WHERE**.
- A ce titre, tous les opérateurs connus de SQL sont disponibles :
 - **IN, LIKE, BETWEEN**
- Mais également les fonctions comme :
 - SUBSTRING, LENGTH
- Exemple :
 - "SELECT e FROM Employee e WHERE e.salary>'2000' AND e.name IN('Paul','belat')"



Les queries agrégées

- Cinq fonction d'agrégation :
 - **AVG, COUNT, MIN, MAX, SUM**
- La syntaxe est très proche du SQL également.
- Exemple:
 - Retourne les employés et, pour chaque employé, le nombre de comptes ce nombre est égal ou supérieur à 2 :
- « **SELECT** e, **COUNT(c)** **FROM** Employee e **JOIN** e.comptes c **GROUP BY** e **HAVING COUNT(c) >=2** »



Les requêtes paramétrées

- Il existe deux façons de passer des paramètres aux requêtes :
 - Tout d'abord en passant un point d'interrogation suivi par le numéro du paramètre :
 - ```
SELECT e FROM Employee e WHERE e.salary>?1 AND e.name IN(?2,?3)"
```

```
.setParameter(1,new Long(500))
```

```
.setParameter(2, "Louis")
```

```
.setParameter(3,"pierre");
```
  - L'autre possibilité consiste à passer des paramètres nommés :
    - ```
SELECT e FROM Employee e WHERE e.salary>:sal AND e.name IN(:nom1,:nom2)"
```

```
.setParameter("nom1"," Louis");
```

```
.setParameter("sal",500L);
```

```
.setParameter("nom2","pierre");
```



Les queries, ou requêtes

- Il existe deux approches pour créer des queries :
 - Soit dynamiquement
 - Etant des string, on peut les créer à la volée
 - Soit statiquement, et définies dans les métadonnées de la persistence unit (via les annotations ou XML)
 - Dans ce cas, elles sont référencées par nom
 - Les « **@NamedQueries** »



Exemple de requêtes dynamiques ne ramenant qu'une valeur

- La constitution d'une requête de query ramenant une seule valeur pourra se faire comme suit :
 - Sachant que la fonction COUNT(e) ramène un type Long :

```
@Stateless (name="EmployeeServiceBean2")
@Remote (EmployeeService.class)
public class EmployeeServiceBean implements EmployeeService {

    @PersistenceContext(unitName="EmployeeServiceUnit")
    protected EntityManager em;

    public long nombreemployees(){
        return (Long) em.createQuery("SELECT COUNT(e) FROM Employee e").getSingleResult();
    }
}
```

- Et son exploitation côté client(par ex une servlet) de la sorte :

```
EmployeeService service;
service = (EmployeeService) context.lookup("EmployeeServiceBean2");
if (action.equalsIgnoreCase("nombre")){
    long i = service.nombreemployees();
    System.out.println("nombre d'employés : "+ i);
}
```



Exemple de requêtes dynamiques ramenant une collection

Requête ramenant une collection d'entités :

```
@Stateless (name="EmployeeServiceBean2")
@Remote (EmployeeService.class)
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeServiceUnit")
    protected EntityManager em;

    public Collection<Employee> findAllEmployees(){
        Query query = em.createQuery("SELECT e FROM Employee e WHERE e.salary>'2000' AND e.name
        IN('facon','pierre')");
        return (Collection<Employee>) query.getResultList();
    }
}
```

Traitement du résultat de la requête :

```
EmployeeService service;
service = (EmployeeService) context.lookup("EmployeeServiceBean2");
if (action.equalsIgnoreCase (« collection »)) {
    Collection<Employee> emps = service.findAllEmployees();
    if (emps.isEmpty()){
        System.out.println("Pas d'employee");
    } else {
        System.out.println("Employee trouvé : </br>");
        for (Employee emp : emps) {
            System.out.println(emp + "<br/>");
        }
    }
}
```



Les requêtes nommées

- Les requêtes nommées, utilisées à des fins de performance, peuvent être définies à l'aide de l'annotation

```
@NamedQuery(name=« nom_de_la_requete »,query=« requête »)
```

Ou

```
@NamedQueries(  
    { @NamedQuery(name=« n1 »,query=q1 »),  
      NamedQuery(name=« n2 »,query=q2») }  
)
```

- Ces requêtes ne sont pas modifiables au run-time

EXERCICE NamedQUERY



Création et utilisation de requêtes nommées

■ Le bean entité avec requête nommée

```
@Entity
@Table (name="Employee")
@NamedQuery(name="salaires",
    query="SELECT e " +
        "FROM Employee e " +
        "WHERE e.name like :nom")
public class Employee implements Serializable {
    @Id
    private int id;
    private String name;
    private long salary;
    ...
}
```

■ La servlet utilisatrice

```
if (action.equalsIgnoreCase("named")) {
    Collection<Employee> emps = service.namedQuery();
    if (emps.isEmpty()) {
        System.out.println("Pas d'employee named query");
    } else {
        System.out.println("Employee trouvé via named query : </br>");
        for (Employee emp : emps) {
            System.out.println(emp + "<br/>");
        }
    }
}
```

■ Le bean service

```
@Stateless (name="EmployeeServiceBean2")
@Remote (EmployeeService.class)
@Transactional(TransactionAttributeType.REQUIRED)
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeServiceUnit")
    protected EntityManager em;

    public Collection<Employee> namedQuery() {
        return (Collection<Employee>)em.createNamedQuery("salaires").setParameter
            ("nom", "fac%").getResultList();
    }
}
```