



# PRÉSENTATION D'HIBERNATE



```
graph TD; A[Présentation] --> B[Métier]; B --> C[Persistance]; C --> D[(Base de données)];
```

Présentation

Métier

Persistance

Base de données

- + Framework de persistance Objet focalisé sur les bases de données relationnelles développé par Gavin King en 2001**
  - ▀ Version actuelle 4.1 (au 24 aout 2012)
- + Download accessible depuis <http://www.hibernate.org/>**
- + Hibernate est gratuit et disponible sous Licence LGPL**
- + Hibernate a rejoint le JBoss Group**
  - ▀ Moyens plus importants

# LA COMMUNAUTÉ HIBERNATE


+ URL d'entrée : <http://www.hibernate.org/>

## + Forum utilisateurs




- <http://forum.hibernate.org>
- Pour poser vos questions

## + Mailing List développeurs

- Pour participer aux développements
- List Archive at SourceForge
- GMANE
- [mail-archive.com](http://mail-archive.com)

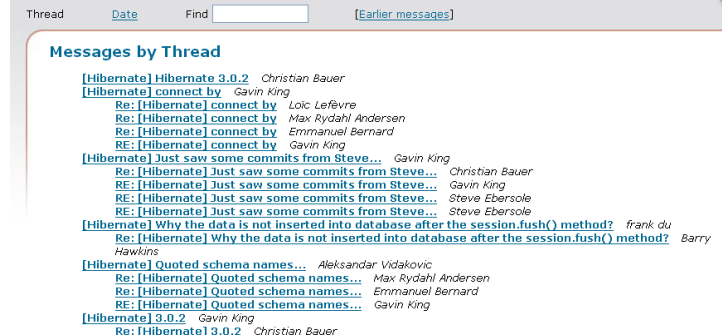


The screenshot shows the top of the Hibernate website. It features the Hibernate logo, a book cover for "The Bible of Hibernate" by Ara Abrahamian, and links to the eBook on Amazon.com and Barnes & Noble. Navigation links include FAQ, Search, Members, and Registration. Below this is the "Hibernate Forums Index" table, which lists various forums with their respective topic and post counts, and the date of the last post.

Forum	Topics	Posts	Last Post
 <b>Hibernate Users</b> General Hibernate usage questions and discussions.	15833	61856	Wed Apr 27, 2005 12:35 pm guilherme_92
 <b>Hibernate3</b> Hibernate3 development and preview questions - EJB3, Annotations, etc.	749	2791	Wed Apr 27, 2005 11:13 am MrLongleg
 <b>Tools</b> Using Hibernate (with) tools: hbm2ddl, hbm2java, Hibern8IDE, Middlegen, XDoclet, AndroMDA.	860	3400	Wed Apr 27, 2005 9:07 am max
 <b>JBoss Application Server &amp; JBoss Cache</b> Installation, integration, and usage of Hibernate with JBoss Application Server and JBoss Cache.	186	725	Tue Apr 26, 2005 5:12 pm fouad
 <b>CaveatEmptor &amp; Hibernate in Action</b> Discussion about the CaveatEmptor demo application ( <a href="http://caveatemptor.hibernate.org/">http://caveatemptor.hibernate.org/</a> ) and the book HIBERNATE IN ACTION, published by Manning Inc. ( <a href="http://www.manning.com/">http://www.manning.com/</a> ).	164	551	Tue Apr 26, 2005 11:19 am dakban
 <b>Administrative News</b> Announcements and Hibernate project news.	74	311	Wed Apr 27, 2005 10:12 am christian

[hibernate-devel](#)

The Mail Archive



The screenshot shows a thread of messages from the hibernate-devel mailing list archive. The messages are listed in a chronological order, with the most recent at the top. Each message entry includes the subject line, the sender's name, and a link to the message. The thread includes discussions about Hibernate 3.0.2, schema names, and session flush methods.

Thread	Date	Find	[Earlier messages]
[Hibernate] Hibernate 3.0.2	Christian Bauer		
[Hibernate] connect by	Gavin King		
Re: [Hibernate] connect by	Loic Lefevre		
Re: [Hibernate] connect by	Max Rydahl Andersen		
Re: [Hibernate] connect by	Emmanuel Bernard		
Re: [Hibernate] connect by	Gavin King		
[Hibernate] Just saw some commits from Steve...	Gavin King		
Re: [Hibernate] Just saw some commits from Steve...	Christian Bauer		
Re: [Hibernate] Just saw some commits from Steve...	Gavin King		
Re: [Hibernate] Just saw some commits from Steve...	Steve Ebersole		
Re: [Hibernate] Just saw some commits from Steve...	Steve Ebersole		
[Hibernate] Why the data is not inserted into database after the session.flush() method?	frank du		
Re: [Hibernate] Why the data is not inserted into database after the session.flush() method?	Barry Hawkins		
[Hibernate] Quoted schema names...	Aleksandar Vidakovic		
Re: [Hibernate] Quoted schema names...	Max Rydahl Andersen		
Re: [Hibernate] Quoted schema names...	Emmanuel Bernard		
Re: [Hibernate] Quoted schema names...	Gavin King		
[Hibernate] 3.0.2	Gavin King		
Re: [Hibernate] 3.0.2	Christian Bauer		

- + Hibernate est une solution open source de type ORM (Object relational mapping)**
- + Hibernate Représente une base de données en objets java et vice versa**
- + Hibernate est implémenté en Java et possède des API Java**
- + Hibernate est compatible avec les standards J2SE et J2EE**
  - ▀ Compatibilité avec tous les serveurs d'application J2EE du marché
    - + Websphere, Weblogic, JBoss, etc..
  - ▀ S'intègre avec JNDI, JDBC, JTA
- + Compatibilité avec la plupart des SGBDR**
  - ▀ Plus de 20 (ex: hypersonic, PostgreSQL, DB2, MySQL, Oracle, ...)

- 
- + Accès plus aisé à la base de données, requêtes simplifiées.**
  - + Tous les objets nécessaires sont montés en mémoire.**
  - + Hibernate est une implémentations de la spécification JPA (Java persistance API)**

---

## + **L'approche JDBC (API, Design Pattern DAO)**

- L'API et les types de drivers JDBC
- Framework iBatis

## + **L'approche composant**

- Open Source : JBoss, JOnAS
- Commerciales : Weblogic (BEA), WAS (IBM), OAS (Oracle)

## + **Les solutions de mapping Objet/Relationnel**

- Open Source : Hibernate, OJB
- Commerciales : TopLink (Oracle), CocoBase

## + **L'approche standard JDO**

- Open Source : JPOX, JORM, TJDO , XORM , RI (Sun)
- Commerciales : Lido (Xcalia), Kodo (SolarMetric), Open Access (Versant)

# LES OUTILS OFFICIELS

---

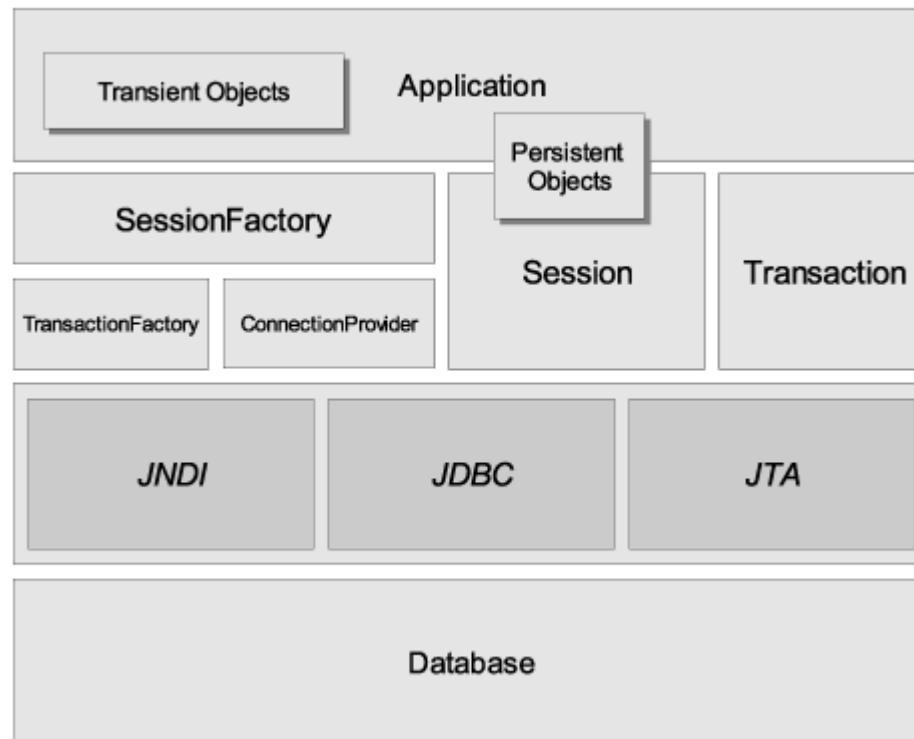
- + **Liste des outils officiels sur <http://tools.hibernate.org/>**
- + **Mapping Editor: éditeur de fichier de mapping XML pour Eclipse**
  - auto-completion pour les attributs XML mais aussi les classes/champs
- + **Console: vues Eclipse permettant la visualisation des entités pour le mapping**
  - Arbre de visualisation des mapping des classes, de leurs attributs et de leurs relations
  - Vue permettant la soumission de requêtes JPQL (Java Persistence query language) et la visualisation des objets résultat (
- + **Development Wizards: différents assistants Eclipse**
  - Génération rapide des fichiers de configuration d'Hibernate (cfg.xml)
  - Génération des entités depuis un schéma SQL existant



# CONFIGURATION D'HIBERNATE



# VUE D'ENSEMBLE



## + **SessionFactory**

- Conserve les mapping des objets et sert de fabrique globale pour les Sessions
- Est threadSafe, instanciée une seule fois au démarrage de l'application

## + **Session**

- Un objet mono-threadé représente une conversation avec la base de données
- Contient un cache de premier niveau des objets persistents

## + **Transaction (optionnel)**

- Une Session peut créer plusieurs Transactions. Une Transaction sert à exécuter des actions de manière atomique

## + **Persistent Objects**

- Objets mono-threadés à vie courte associés à une Session
- Sont en générale des objets de type javaBean

## + **Transient Objects**

- Identiques aux Persistent Objects, mais non-associés à une Session

## + Pour pouvoir fonctionner, Hibernate doit connaître:

- ▀ Le type de la base de données cible
  - le dialecte SQL (ex: MySQL, Oracle, DB2, ...)
  
- ▀ La manière d'obtenir une connexion JDBC
  - + Utilisation d'un pool de connexions interne à Hibernate
  - OU
  - + Utilisation d'un pool de connexions externe à Hibernate
    - DataSource

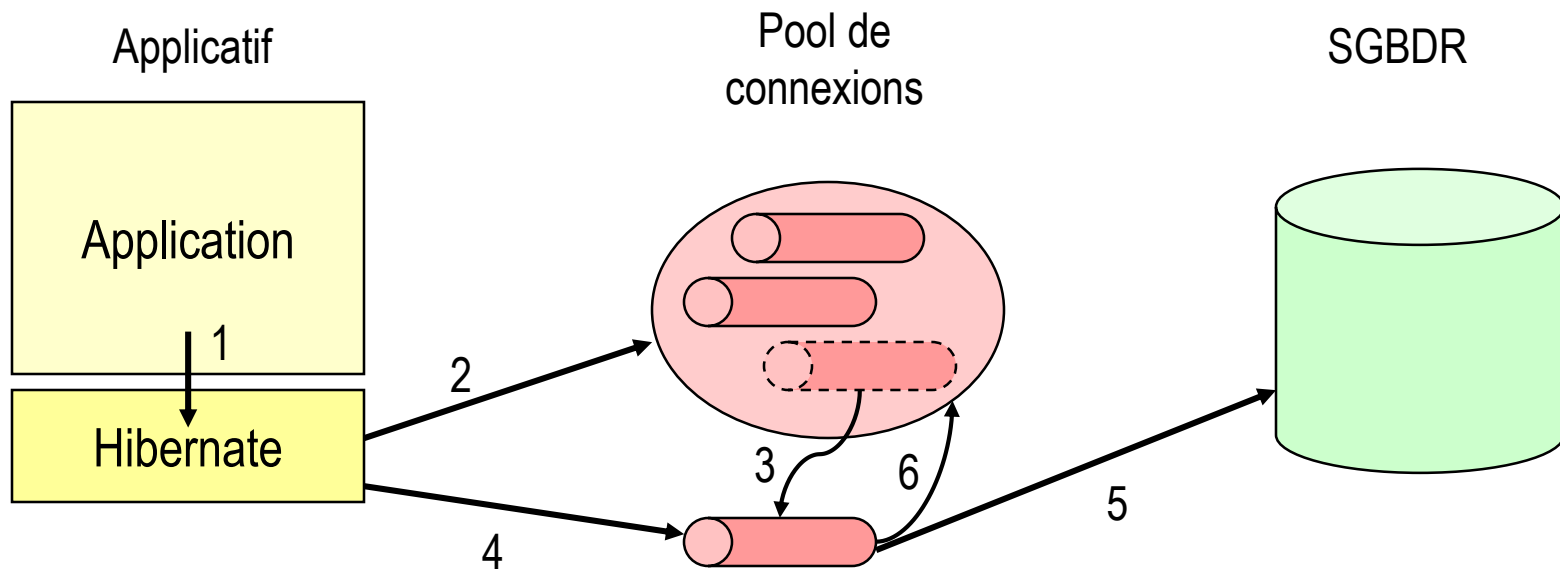
## + 4 méthodes de configuration équivalentes

### + Elles passent par la définition de clés/valeurs

- Directement programmatiquement via un `java.util.Properties`  
OU
- Dans un fichier *hibernate.properties* situé dans le classpath  
OU
- Via des propriétés systèmes (`java -Dproperty=value ...`)  
OU
- Avec des balises `<property>` dans le fichier de mapping  
*hibernate.cfg.xml*

# POOL DE CONNEXIONS

**+ Pour des raisons de performance, les connexions JDBC sont systématiquement « poolées »**



**+ Dans ce cas Hibernate crée et peuple son pool de connexions**

**+ Pour créer ce pool, il faut préciser**

- La taille du pool de connexions
  - Le Driver JDBC
  - L'URL de connexion à la base
  - Le login/password de connexion à la base
- } pour la création des connexions

Propriété	Fonction
<i>hibernate.connection.url</i>	<i>URL JDBC</i>
<i>hibernate.connection.username</i>	<i>Login de l'utilisateur</i>
<i>hibernate.connection.password</i>	<i>Mot de passe de l'utilisateur</i>
<i>hibernate.connection.pool_size</i>	<i>Nombre de connexions poolées</i>
<i>hibernate.connection.driver_class</i>	<i>Driver JDBC</i>

## + Par défaut utilisation d'un pool de connexion rudimentaire

- A ne pas utiliser en production

## + Utilisation possible de

- C3P0 (<http://sourceforge.net/projects/c3p0>)
  - + Propriétés « hibernate.c3p0.\* » → C3P0ConnectionProvider
- Apache DBCP (<http://jakarta.apache.org/commons/dbcp>)
  - + Propriétés « hibernate.dbcp.\* » → DBCPConnectionProvider
- Proxool (<http://proxool.sourceforge.net>)
  - + Propriétés « hibernate.proxool.\* » → ProxoolConnectionProvider



## + Dans ce cas Hibernate ne crée pas de pool de connexion

- Utilisation d'une DataSource qui fournira les connexions

## + Nécessité de préciser

- Le nom JNDI de la DataSource dans l'annuaire JNDI
- Le login/password de connexion à cette DataSource (optionnel)
- L'URL et la fabrique de connexion à l'annuaire (optionnel)

Propriété	Fonction
<i>hibernate.connection.datasource</i>	<i>Nom JNDI de la DataSource</i>
<i>hibernate.connection.username</i>	<i>Login de l'utilisateur (optionnel)</i>
<i>hibernate.connection.password</i>	<i>Mot de passe de l'utilisateur (optionnel)</i>
<i>hibernate.jndi.url</i>	<i>URL du Provider JNDI (optionnel)</i>
<i>hibernate.jndi.class</i>	<i>Classe de l'InitialContextFactory JNDI (optionnel)</i>

# POOL INTERNE VS POOL EXTERNE

---

## + Pool interne

- + Simplicité de mise en œuvre
- Impossibilité de partage avec d'autres applications
- Impossibilité d'utiliser le pool interne par défaut

## + Pool Externe

- + Interface standardisée (DataSource)
- + Partage avec d'autres applications
- + Généralement disponible dans les serveurs d'applications
- Nécessite un annuaire JNDI
- Coût de mise en place

**+ « hibernate.dialect » doit être le nom d'une sous classe de « org.hibernate.dialect.Dialect »**

- ▀ Classe devant correspondre à la base de données choisie
- ▀ ex: org.hibernate.dialect.H2Dialect

**+ De cette classe seront déduits certaines propriétés et comportements**

- ▀ ex: hibernate.jdbc.batch\_size=15

## SQL DIALECTS (2/2)

---

DB2:	org.hibernate.dialect.DB2Dialect
DB2 AS/400:	org.hibernate.dialect.DB2400Dialect
DB2 OS390:	org.hibernate.dialect.DB2390Dialect
PostgreSQL:	org.hibernate.dialect.PostgreSQLDialect
MySQL:	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB:	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM:	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version):	org.hibernate.dialect.OracleDialect
Oracle 9i/10g:	org.hibernate.dialect.Oracle9Dialect
Sybase:	org.hibernate.dialect.SybaseDialect
Sybase Anywhere:	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server:	org.hibernate.dialect.SQLServerDialect
SAP DB:	org.hibernate.dialect.SAPDBDialect
Informix:	org.hibernate.dialect.InformixDialect
HypersonicSQL:	org.hibernate.dialect.HSQLDialect
Ingres:	org.hibernate.dialect.IngresDialect
Progress:	org.hibernate.dialect.ProgressDialect
Mckoi SQL:	org.hibernate.dialect.MckoiDialect
Interbase:	org.hibernate.dialect.InterbaseDialect
Pointbase:	org.hibernate.dialect.PointbaseDialect
FrontBase:	org.hibernate.dialect.FrontbaseDialect
Firebird:	org.hibernate.dialect.FirebirdDialect

## AUTRES PROPRIÉTÉS (1/2)

Propriété	Fonction
<i>hibernate.session_factory_name</i>	<i>Nom JNDI à assigner à la SessionFactory</i>
<i>hibernate.max_fetch_depth</i>	<i>Profondeur des requêtes OUTER JOIN</i>
<i>hibernate.cache.provider_class</i>	<i>Classe du CacheProvider à utiliser</i>
<i>hibernate.show_sql</i>	<i>Afficher toutes les requêtes envoyées</i>
...	...

Cf. § 3.4. Propriétés de configuration optionnelles

- + Il existe de très nombreuses propriétés, toujours initialisées à des valeurs "correctes" par défaut**
  - ▀ Le manuel de référence d'Hibernate les explique plus en détail
- + Pour la journalisation, Jboss-Logging est utilisé**

# EXEMPLE HIBERNATE.CFG.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="bankonet">

        <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="hibernate.connection.url">jdbc:h2:tcp://localhost/file:C:/bankonetDB/bankonet</property>
        <property name="hibernate.connection.driver_class">org.h2.Driver</property>
        <property name="hibernate.connection.username">sa</property>
        <property name="hibernate.connection.password">manager</property>
        <!-- Enable 2nd level cache -->
        <property name="hibernate.cache.provider_class"> org.hibernate.cache.EhCacheProvider </property>
        <property name="hibernate.cache.use_second_level_cache"> true </property>

        <!-- <property name="hibernate.hbm2ddl.auto">create</property> -->
        <property name="hibernate.show_sql">true</property>
        <mapping class="com.sqli.bankonet.orm.Compte"/>
        <mapping class="com.sqli.bankonet.orm.CompteTransaction"/>
        <mapping class="com.sqli.bankonet.orm.Adresse"/>
        <mapping class="com.sqli.bankonet.orm.Client"/>
    </session-factory>
</hibernate-configuration>
```

# UTILISATION D'HIBERNATE



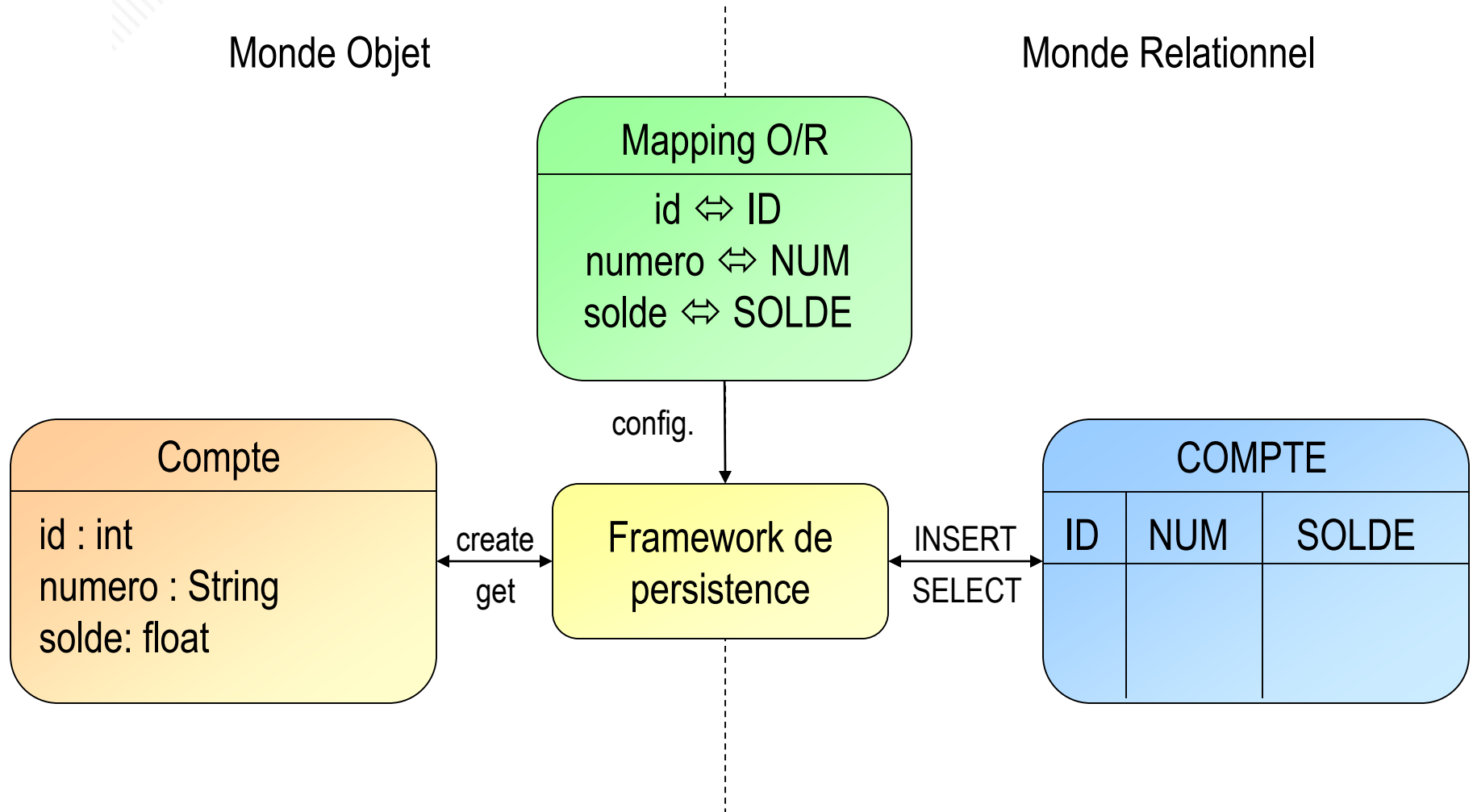


- 
- + Définition des mappings**
  - + Ouverture d'une session Hibernate**
  - + Opérations CRUD (Create Read Update Delete)**
  - + Méthodes update et saveOrUpdate**

# MAPPING OBJET/RELATIONNEL

Monde Objet

Monde Relationnel



# LE MAPPING O/R: UTILITÉ DE L'IDENTITÉ

---

## + L'identité d'un objet est implicite

- ▀ deux objets ayant les mêmes attributs (i.e. le même état) sont tout de même différents

## + L'identité d'un enregistrement est représentée par sa clé primaire

## + Comment maintenir l'identité d'un objet ?

## + Dans la table

- ▀ définir une colonne comme clé primaire (ce qui est usuel)

## + Dans la classe

- ▀ définir un attribut qui identifie de façon unique un objet
- ▀ assurer que le processus de création d'un nouvel objet ne génère pas de doublon de cet attribut

### + Deux approches pour la définition de mapping

- ▀ Utilisation des annotations java 5 (javax.persistence.Id)
- ▀ Utilisation des fichiers de configuration hibernate hbm.xml

### + Un objet java de type POJO mappé vers une table de base de données grâce aux annotations via l'API JPA est nommé Bean entité (Entity Bean)

### + Dans ce cours nous utiliserons l'approche Java 5 ⇔ Les annotations

# CLASSE DE MAPPING (2/2)

## + Exemple :

```
@Entity
@Table(name = "COMPTE")
public class Compte implements Serializable {

    private static final long serialVersionUID = 1L;
    //Id technique
    @Id
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "TYP_COMPTE", nullable = false, length = 2)
    private String typCompte;

    @Column(name = "LIBELLE", nullable = false, length = 50)
    private String libelle;

    @Column(name = "MNT_SOLDE", nullable = false, precision = 10)
    private BigDecimal mntSolde;

    @Column(name = "MNT_DECOUVERT_AUTORISE")
    private Integer decouvertAutorise;

    @Column(name = "TAUX", precision = 4)
    private BigDecimal tauxInteret;

    @Column(name = "MNT_PLAFOND")
    private Integer plafondCredit;

    @OneToOne
    @JoinColumn(name="CLIENT_ID",nullable=false)
    private Client client;

    @OneToMany(mappedBy = "compte", cascade = CascadeType.REMOVE)
    private Set<CompteTransaction> compteTransactions = new HashSet<CompteTransaction>();

    public Compte() {
    }
}
```

# CONTRAINTES SUR LES ENTITIES

---

## + **Doit obligatoirement avoir un constructeur sans argument**

- ▀ Qu'il soit implicite ou explicite \*

## + **Les attributs doivent déclarer un couple getter / setter \*\***

- ▀ Il est fortement conseillé de ne pas les rendre *final*

## + **Les attributs devraient toujours être de type Object**

- ▀ Utilisation des wrappers ⇔ Integer, Double, Float, Long, Character, Boolean, Byte, Short

## + **Les méthodes hashCode et equals devraient être surchargées**

- ▀ hashCode : identifiant servant de clefs dans les structures Map
- ▀ equals : teste l'égalité entre deux objets

## + **Pas indispensable, mais pratique, surcharger toString**

# DÉFINITION DE CLASSES

- + **Toute classe persistante se déclare via l'annotation `@javax.persistence.Entity` & `@javax.persistence.Table`**
- + **Posséder au moins un attribut déclarer comme clé primaire avec l'annotation `@javax.persistence.Id`**
- + **Les 4 annotations les plus importantes sont:**
  - **Entity** : le nom de la classe
  - **Table** : le nom de la table
  - **Id** : la clé primaire de la table
  - **Column** : nom de la colonne

```
@Entity
@Table(name = "COMPTE")
public class Compte implements Serializable {

    private static final long serialVersionUID = 1L;
    //Id technique
    @Id
    @Column(name = "ID", unique = true, nullable = false)
    private int id;
```



Si l'annotation Table n'est pas précisée, sa valeur sera le nom de la classe



# CLASSE DE MAPPING: GÉNÉRATEUR D'IDENTIFIANT

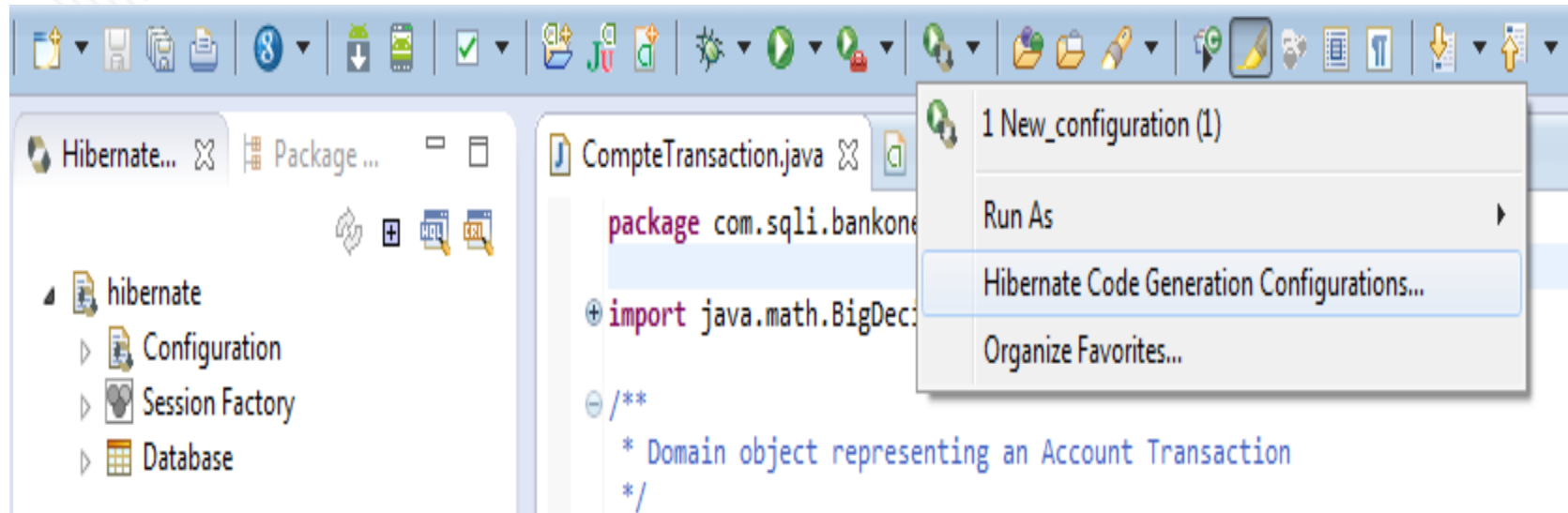
- + Une entité doit déclarer comment sont générés les identifiants
- + Soit utilisation d'une stratégie de génération des identifiants :
  - 4 Strategy de génération : TABLE,SEQUENCE,IDENTITY,AUTO
  - @GeneratedValue indique que la génération est automatique
  - @SequenceGenerator indique le nom de la séquence pour la génération

```
@Entity
@Table(name = "COMPTE_TRANSACTION")
public class CompteTransaction implements Comparable<CompteTransaction> {

    @Id
    @SequenceGenerator(name="SEQ_TRANSACTIONTYPE",sequenceName="SEQ_TRANSACTION_TYPE",initialValue=1)
    @GeneratedValue(strategy=GenerationType.SEQUENCE,generator="SEQ_TRANSACTIONTYPE")
    @Column(name = "COMPTE_TRANSACTION_ID")
    private long compteTransactionId;
```



Utiliser systématiquement *generator* quand cela est possible



Ces outils seront détaillés dans le chapitre "Outils de génération de code"

# APPROCHES DE MAPPING

Bottom-up



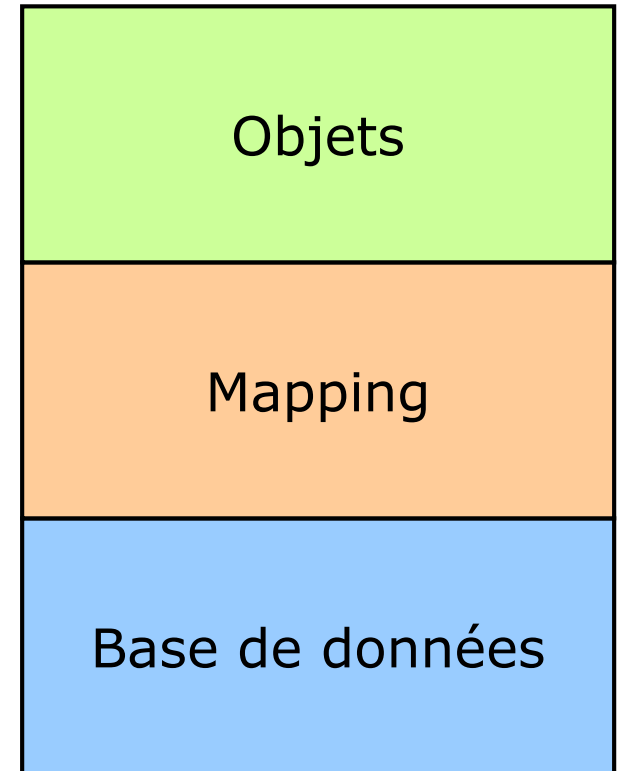
Top-down



Meet in the middle



Le plus fréquent



- 
- + **Définition des mappings**
  - + **Ouverture d'une session Hibernate**
  - + **Opérations CRUD (Create Read Update Delete)**
  - + **Méthodes update et saveOrUpdate**

## CRÉER UNE CONFIGURATION (1/4)

---

- + Un objet **Configuration** représente un ensemble de mappages des classes java d'une application vers une base de données
- + Utilisation de "hibernate.properties" uniquement

```
Configuration cfg = new Configuration();
```

- + Utilisation de "hibernate.properties" et "hibernate.cfg.xml"

```
Configuration cfg = new Configuration().configure();
```

# CRÉER UNE CONFIGURATION (2/4)

## + Exemple

```
public class HibernateUtil {  
  
    public static final ThreadLocal<Session> session = new ThreadLocal<Session>();  
  
    public static final SessionFactory sessionFactory;  
  
    private static ServiceRegistry serviceRegistry;  
  
    static {  
        try {  
            System.out.println(new Date() + " : Initialisation de la session factory hibernate ");  
            // Create the SessionFactory from hibernate.cfg.xml  
            Configuration config = new Configuration().configure();  
  
            serviceRegistry = new ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  
  
            sessionFactory = config.buildSessionFactory(serviceRegistry);  
  
        } catch (Throwable ex) {  
            throw new ExceptionInInitializerError(ex);  
        }  
    }  
  
    public static Session currentSession() throws HibernateException {  
        Session s = (Session) session.get();  
        // Open a new Session, if this thread has none yet  
        if (s == null) {  
            s = sessionFactory.openSession();  
            // Store it in the ThreadLocal variable  
            session.set(s);  
        }  
        return s;  
    }  
  
    public static void closeSession() throws HibernateException {  
        Session s = (Session) session.get();  
        if (s != null)  
            s.close();  
        session.set(null);  
    }  
}
```

# CRÉER UNE CONFIGURATION (3/4)

## + Solution 1: construction déclarative

- Le fichier de configuration d'Hibernate liste les entités de mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="bankonet">

        <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="hibernate.connection.url"> jdbc:h2:tcp://localhost/file:C:/bankonetDB/bankonet</property>
        <property name="hibernate.connection.driver_class">org.h2.Driver</property>
        <property name="hibernate.connection.username">sa</property>
        <property name="hibernate.connection.password">manager</property>

        <!-- Enable 2nd level cache -->
        <property name="hibernate.cache.provider_class"> org.hibernate.cache.EhCacheProvider </property>
        <property name="hibernate.cache.use_second_level_cache"> true </property>

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

        <!-- entities mapping -->
        <mapping class="com.sqli.bankonet.orm.Compte"/>
        <mapping class="com.sqli.bankonet.orm.CompteTransaction"/>
        <mapping class="com.sqli.bankonet.orm.Adresse"/>
        <mapping class="com.sqli.bankonet.orm.Client"/>

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

## + **Solution 2: construction programmatic (moins courante)**

### ▀ 2.1: Utilisation des classes avec des annotations

```
Configuration cfg = new Configuration().configure()  
    .addAnnotatedClass(Compte.class)  
    .addAnnotatedClass(Client.class);
```

### ▀ 2.2 Utilisation des noms des fichiers de mapping

```
Configuration cfg = new Configuration().configure()  
    .addResource("Compte.hbm.xml")  
    .addResource("Client.hbm.xml");
```

- + Hibernate cherchera les fichiers `Compte.hbm.xml` et `Client.hbm.xml` dans le classpath



# CRÉER UNE SESSIONFACTORY

## + L'objet Configuration permet d'instancier une SessionFactory

```
Configuration config = new Configuration().configure();  
  
serviceRegistry = new ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  
  
sessionFactory = config.buildSessionFactory(serviceRegistry);
```

## + Cette SessionFactory peut être partagée par toutes les tâches de l'application



Cette SessionFactory peut être stockée (ex: static, Singleton, JNDI)

# CRÉER UNE SESSION

---

- Une Session représente une « communication » avec la base de données

- Pour ouvrir une session en lecture seule (read-only)

```
Session session = sessions.openSession();
```

- Pour ouvrir une session permettant de lire et d'écrire (read/write)

```
Session session = sessions.openSession();  
Transaction tx = session.beginTransaction();
```

- + **TP01 : Mise en place de l'environnement**
- + **TP02 : Mapping Fichier**
- + **TP03 : Mapping Annotation**

# OPERATION CRUD

---

- + Définition des mappings
- + Ouverture d'une session Hibernate
- + Opérations CRUD (Create Read Update Delete)
- + Méthodes update et saveOrUpdate

## + 4 opérations de base

### + CRUD:

- |                    |             |                                |
|--------------------|-------------|--------------------------------|
| ➤ <b>C</b> reate:  | création    | → INSERT en la base de données |
| ➤ <b>R</b> ead:    | lecture     | → SELECT en base de données    |
| ➤ <b>U</b> ppdate: | mise à jour | → UPDATE en base de données    |
| ➤ <b>D</b> elete:  | suppression | → DELETE en base de données    |



Toutes les opérations de type écriture doivent avoir lieu dans une transaction

- + **La méthode `save(...)` de l'interface `org.hibernate.Session` permet de demander l'enregistrement d'un objet**

```
Hotel h = new Hotel();  
h.setNom("BeauRegard");  
h.setVille("Lisbonne");  
session.save(h);
```

- + **Pour récupérer l'identifiant généré (ex: 13465), 2 méthodes:**
  - ▀ récupérer l'ID en tant que résultat de `save(...)`

```
Long pId = (Long) session.save(h);
```

- ▀ récupérer l'ID depuis l'objet « sauvé »

```
Long pId = h.getId();
```

### + Il existe une forme de `save(...)` prenant deux paramètres

- l'objet à sauver
- l'identifiant de cet objet à sauver (fourni par le programmeur)

```
Hotel h = new Hotel();  
h.setNom("BeauRegard");  
h.setVille("Lisbonne");  
session.save(h, Long.valueOf(13465));
```



A n'utiliser que si l'identifiant est "assigned" (cas rare)

## + Le langage HQL permet de lire des enregistrements selon certains critères

### ▀ Génération de requêtes HQL

```
Query query = session.createQuery(" FROM Hotel");  
List resultList = query.list();  
if (!resultList.isEmpty()){  
    Hotel premierHotel = (Hotel) resultList.get(0);  
}
```

### ▀ Utilisation du namedQuery

```
Query query = session.createNamedQuery("Hotel.readByName");  
List resultList = query.list();  
if (!resultList.isEmpty()){  
    Hotel premierHotel = (Hotel) resultList.get(0);  
}
```



**+ La méthode get(...) (de l'interface org.hibernate.Session), permet la récupération d'un objet par sa classe et son identifiant**

- ▀ renvoie l'objet lu ou « null » si non présent

```
Long pkId = Long.valueOf(13465);  
Hotel h = (Hotel) session.get(Hotel.class, pkId);
```

└ Cast nécessaire

## + Pour modifier un objet il faut

1. Le charger pour l'attacher à la session courante
  - + Récupération de l'objet en lecture/écriture
2. Le modifier

## + Aucune méthode particulière n'est invoquée lors de la modification d'un objet !

```
Hotel h = (Hotel) session.get(Hotel.class, Long.valueOf(13465));  
h.setNom("Beauregard");
```

↑  
Lecture nécessaire



La méthode `update(...)` de `Session` ne sert pas à modifier un objet !

- + **Paradoxe: pour supprimer un objet il faut d'abord le lire**
- + **La méthode delete(...) (de l'interface org.hibernate.Session), permet la demande de suppression d'un objet**

```
Hotel h = (Hotel) session.get(Hotel.class, Long.valueOf(13465));  
session.delete(h);
```



Lecture nécessaire

## + Toute modification doit avoir lieu dans une Transaction Hibernate

### ➤ Unité de travail

```
Session session = sessions.openSession();  
session.beginTransaction();  
// traitements  
...  
Session.getTransaction().commit(); // ou tx.rollback();  
session.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)

## + Exemple d'utilisation

```
* {
    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction(); // debut de la transaction

        session.save(new Hotel(...)); // creation d'un hôtel

        Hotel h1 = session.get(Hotel.class, Long.valueOf(242));
        p1.setVille("Toulouse"); // modification de l'hôtel "242"

        Hotel h2 = session.get(Hotel.class, Long.valueOf(139));
        session.delete(h2); // suppression de l'hôtel "139"

        tx.commit(); // validation de la transaction
    } catch (Exception e) {
        if (tx != null) tx.rollback(); // rollback de la tx
        throw e;
    } finally {
        session.close(); // fermeture de la session
    }
}
```

- 
- + Définition des mappings**
  - + Ouverture d'une session Hibernate**
  - + Opérations CRUD (Create Read Update Delete)**
  - + Méthodes update et saveOrUpdate**

# LA MÉTHODE UPDATE (1/2)

## + Permet de rattacher un objet qui a été modifié en dehors d'une session Hibernate

```
Configuration configuration = new Configuration().configure();
ServiceRegistry reg = new
ServiceRegistryBuilder.ApplySetting(configi.getProperties).buildServiceRegistry();
SessionFactory factory = configuration.buildSessionFactory(reg);
Session session1 = factory.openSession();
session1.beginTransaction();
// création de l'objet
Client client = new Client("Bertrand", "Dupont");
session1.save(client);
Session1.getTransaction().commit();
session1.close();
// L'objet est modifié en dehors de la session
client.setNom("Durand");
Session session2 = factory.openSession();
session2.beginTransaction();
session2.update(client);
Session2.getTransaction().commit();
session2.close();
```

### + Conditions d'appel

- S'il existe un objet persistant avec le même identifiant, une exception `HibernateException` est lancée.
- Si l'objet n'a jamais été persisté (id non rempli), une exception `HibernateException` est lancée



`HibernateException` hérite de `RuntimeException`. Il n'est pas obligatoire de la gérer.



# LA MÉTHODE SAVEORUPDATE (1/2)

## + Dans certains cas, on ne sait pas si un objet a déjà été persisté

- ▀ Le code adapté pourrait être long à écrire. Dans le cas le plus simple :

```
Configuration configuration = new Configuration().configure();
SessionFactory factory = configuration.buildSessionFactory();
Session session = factory.openSession();

Transaction transaction = session.beginTransaction();
Client client = new Client("Bertrand", "Dupont");
if (client.getId() == 0) {
    session.save(client);
} else {
    session.update(client);
}
transaction.commit();

session.close();
```

## LA MÉTHODE SAVEORUPDATE (2/2)

---

### + Avec la méthode saveOrUpdate :

```
Configuration configuration = new Configuration().configure();
SessionFactory factory = configuration.buildSessionFactory();
Session session = factory.openSession();

Transaction transaction = session.beginTransaction();
Client client = new Client("Bertrand", "Dupont");
session.saveOrUpdate(client);
transaction.commit();
session.close();
```

---

## + TP 04 : DAO pour nos classes simplifiées

- ▀ insert
- ▀ update
- ▀ delete
- ▀ select(clef primaire)
- ▀ selectAll

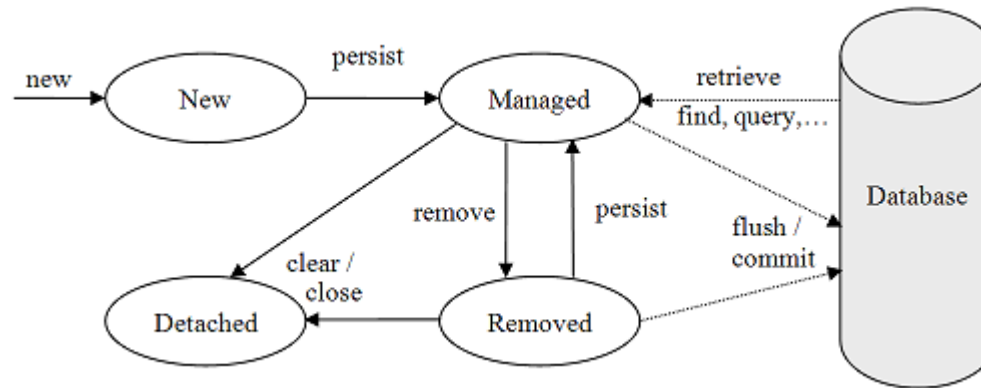
# LES RELATIONS: FONDAMENTAUX



# ENTITÉS ET VALEURS (1/2)

## + 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

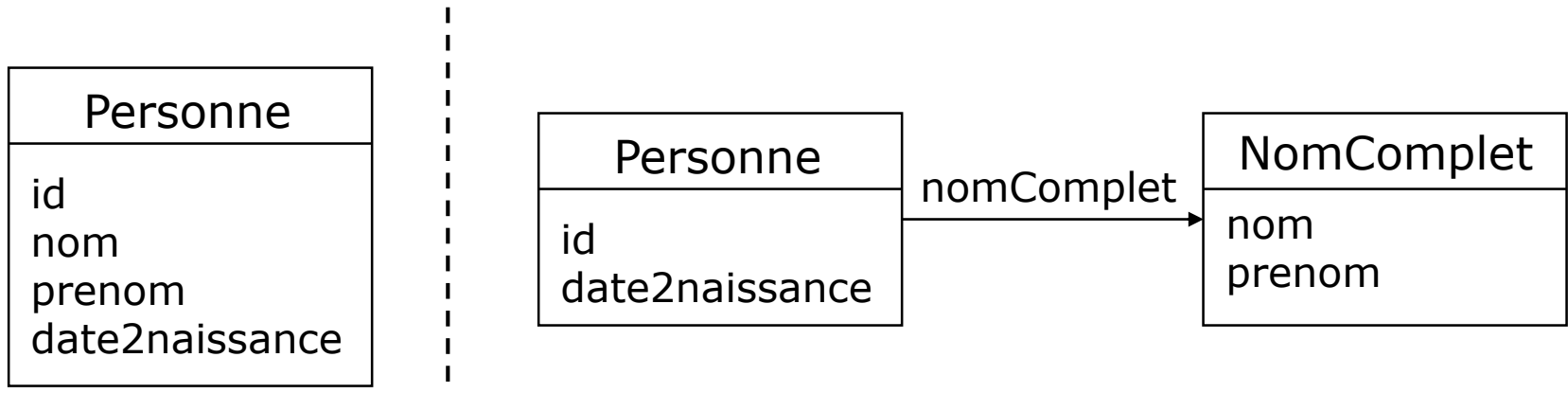
## ENTITIES ET VALEURS (2/2)

### + 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



# MAPPING

- + **Associations 1-n**
- + **Associations n-n**
- + **Associations 1-1**
- + **Composants**
- + **Collections de valeurs**

# MAPPING 1:N PRÉSENTATION

---

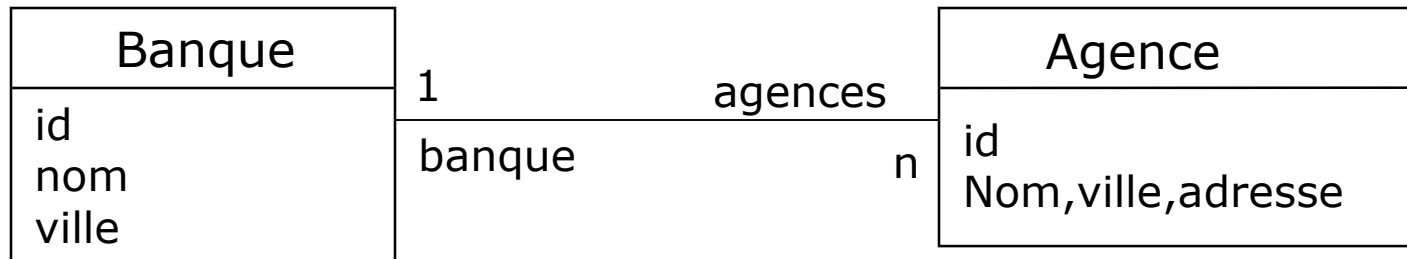
- + Relations la plus souvent utilisée**
- + Lie les tables des deux classes par une clé étrangère**
- + Seule relation correspondant réellement au modèle physique de données (MPD)**
- + Différence entre <one-to-many> et <many-to-one> suivant le sens d'observation de la relation**
- + Possibilité de déclarer cette relation**
  - ▀ Soit unidirectionnelle → <many-to-one> OU <one-to-many>
  - ▀ Soit bidirectionnelle → <many-to-one> ET <one-to-many>  
→ 3 possibilités



# MAPPING 1:N: EXEMPLE

## + Exemple

- Une **Agence** appartient à une banque
- Une **Banque** regroupe plusieurs **Agences**



\_\_\_\_\_ <one-to-many> \_\_\_\_\_→

←\_\_\_\_\_ <many-to-one> \_\_\_\_\_

# 1:N: UNI-DIRECTIONNELLE <MANY-TO-ONE> (1/2)

## + Exemple :

```
public class Agence{
    private int id;
    @ManyToOne
    @JoinColumn(name="banque_id")
    private Banque banque; // reference vers la banque
    ...
    public Banque getBanque() { return banque; }
    public void setBanque(Banque banque) { this.banque = banque; }
}
```

```
public class Banque{
    //pas de reference vers l'agence
    ...
}
```

Si le @JoinColumn n'est pas spécifié , Hibernate générera un foreign key de type :  
Property field + « \_ » + id

# 1:N: UNI-DIRECTIONNELLE <MANY-TO-ONE> (2/2)

## + Mapping

```
@Entity
@Table(name="AGENCE")
public class Agence implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private Long id;

    @Column(name="NOM",length=50,nullable=false)
    private String nom;

    @ManyToOne
    @JoinColumn(name="banque")
    private Banque banque;

    public Agence() {

    }
}
```

## + Utilisation

```
Agence agence = ...;
Banque banque = ...;
agence.setBanque(banque);
```

# 1:N: UNI-DIRECTIONNELLE <ONE-TO-MANY> (1/2)

## + Exemple :

```
public class Banque
{
    private int id;
    @OneToMany
    private Collection<Agence> agences; // référence vers les agences
    ...
    public Banque() { agences= new HashSet<Agence>(); }
    public Collection<Agence> getAgences() { return agences; }
    public void setAgences(Collection<Agence> agences)
        { this.agences = agences; }
}
```

```
public class Agence {
    private int id;
    // pas de référence vers la banque
    ...
}
```



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



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

# 1:N: UNI-DIRECTIONNELLE <ONE-TO-MANY> (2/2)

## + Mapping (2 façons de faire)

```
@Entity
@Table(name="BANQUE")
public class Banque implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private Long id;

    @Column(name="NOM",length=50,nullable=false)
    private String nom;
```

```
@OneToMany(cascade=CascadeType.ALL)
@JoinTable(name="banqueAgence",joinColumns=@JoinColumn(name="agenceID")
    ,inverseJoinColumns=@JoinColumn(name="banqueID"))
private Set<Agence> agences = new HashSet<Agence>();
```

```
@Entity
@Table(name="BANQUE")
public class Banque implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private Long id;

    @Column(name="NOM",length=50,nullable=false)
    private String nom;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="banqueID")
    private Set<Agence> agences = new HashSet<Agence>();
```

```
Banque banque = ...;
Agence agence = ...;
banque.getAgences.add(agence);
```



Sans le @JoinColumn hibernate a besoin d'une table intermédiaire  
(Banque\_adresse)

# 1:N: BI-DIRECTIONNELLE (1/3)

## + Exemple :

```
public class Banque
{
    private int id;
    private Collection<Agence> agences; // référence vers les agences
    ...
    public Banque() { agences= new HashSet<Agence>(); }
    public Collection<Agence> getAgences() { return agences; }
    public void setAgences(Collection<Agence> agences)
    { this.agences = agences; } }
}
```

```
public class Agence{
    private int id;
    private Banque banque; // reference vers la banque
    ...
    public Banque getBanque() { return banque; }
    public void setBanque(Banque banque) { this.banque = banque; }
}
```



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



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

# 1:N: BI-DIRECTIONNELLE (2/3)

## + Mapping

```
@Entity
@Table(name="BANQUE")
public class Banque implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private Long id;

    @Column(name="NOM",length=50,nullable=false)
    private String nom;

    @OneToMany(mappedBy="banque",fetch=FetchType.EAGER) |
    private Set<Agence> agences = new HashSet<Agence>();

    public Banque() { }
```

```
@Entity
@Table(name="AGENCE")
public class Agence implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private Long id;

    @Column(name="NOM",length=50,nullable=false)
    private String nom;

    @ManyToOne
    @JoinColumn(name="banque")
    private Banque banque;

    public Agence() {
    }
```

Pas de table intermédiaire, la clé étrangère est stockée dans la table agence  
La classe banque est la propriétaire de la relation

# 1:N: BI-DIRECTIONNELLE (3/3)

## + Utilisation

```
Banque banque = ...;  
Agence agence = ...;  
  
// première manière  
agence.setBanque(banque);  
  
// deuxième manière  
banque.getAgences().add(agence);  
  
// troisième manière  
agence.setBanque(banque);  
agence.setBanque(banque);
```



Les problèmes de bi-directionnalité seront abordés par la suite



# MAPPING N:N

---

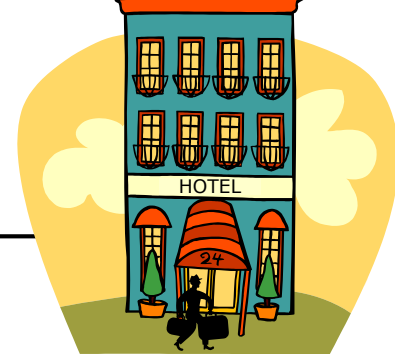
- + Associations 1-n
- + Associations n-n
- + Associations 1-1
- + Composants
- + Collections de valeurs

## MAPPING N:N: PRÉSENTATION

---

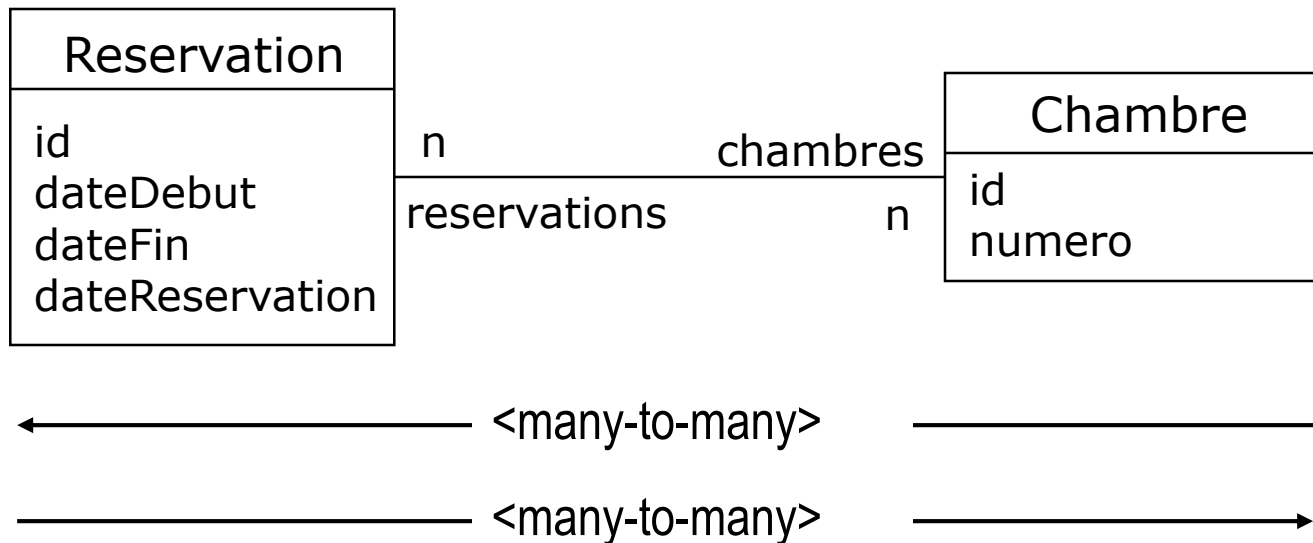
- + Ce type de mapping est moins fréquemment utilisé que le mapping 1:n**
- + Nécessite la création d'une table intermédiaire pour la gestion des associations entre les enregistrements sources et cibles.**
- + Possibilité de déclarer cette relation :**
  - ▀ Soit unidirectionnelle → <many-to-many> d'un des côtés
  - ▀ Soit bidirectionnelle → <many-to-many> des 2 côtés

## MAPPING N:N: EXEMPLE (1/2)



### + Exemple

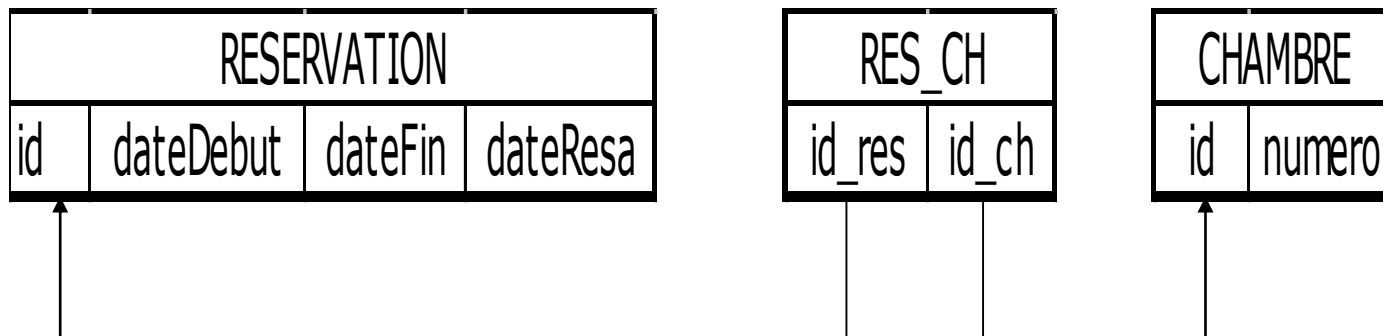
- Une **Reservation** peut concerner plusieurs **Chambre**
- Une **Chambre** peut faire l'objet de plusieurs **Reservation**



## MAPPING N:N: EXEMPLE (2/2)

### + En base de données, utilisation d'une table intermédiaire

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



# MAPPING N:N: UNI-DIRECTIONNELLE

## + Exemple :

```
public class Reservation {  
    private int id;  
    @ManyToMany  
    private Set<Chambre> chambres; // référence vers les chambres  
    ...  
    public Reservation() { chambres = new HashSet(); }  
    public Set getChambres() { return chambres; }  
    public void setChambres(Set<Chambre> 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, Bag, Map



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

# MAPPING N:N

## + Mapping

```
@Entity
@Table(name="COLLABORATEUR")
public class Collaborateur {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID",unique=true,updatable=false)
    private int id;

    @Column(name="NOM",length=50)
    private String nom;

    @Column(name="PRENOM",length=50)
    private String prenom;

    @Column(name="DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;

    @Column(name="SEXE",length=1)
    private char sexe;

    @Column(name="SALAIRE_ANNUEL")
    private BigDecimal salaireAnnuel;

    @ManyToMany(fetch=FetchType.LAZY,cascade=CascadeType.ALL)
    private Set<Responsabilite> responsabilites = new HashSet<Responsabilite>();
}
```

# ASSOCIATION N:N: BI-DIRECTIONNELLE (1/2)

## + Exemple :

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

```
public class Chambre {
    private int id;
    @ManyToMany(mappedBy="reservation")
    private Set<Reservation> reservations; // référence vers les réservations
    ...
    public Chambre() { reservations = new HashSet<Reservation>(); }
    public Set<Reservation> getReservations() { return reservations; }
    public void setReservations(Set<Reservation> res) {
        this.reservations = res; }
}
```



Les Collections doivent être de même type (Set, List, Array, Map)



Ne pas oublier d'initialiser les Collections à « vide » (et pas « null »)

## ASSOCIATION N:N: BI-DIRECTIONNELLE (2/2)

### + Exemple :

```
Reservation rel = ...;  
Chambre ch1 = ...;  
  
// première manière  
rel.getChambres().add(ch1);  
  
// deuxième manière  
ch1.getReservations().add(rel);  
  
// troisième manière  
rel.getChambres().add(ch1);  
ch1.getReservations().add(rel);
```



Les problèmes de bi-directionnalité seront abordés par la suite



# ASSOCIATION N:N: BI-DIRECTIONNELLE

## + Mapping

```
@Entity
@Table(name="COLLABORATEUR")
public class Collaborateur {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID",unique=true,updatable=false)
    private int id;

    @Column(name="NOM",length=50)
    private String nom;

    @Column(name="PRENOM",length=50)
    private String prenom;

    @Column(name="DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;

    @Column(name="SEXE",length=1)
    private char sexe;

    @Column(name="SALAIRE_ANNUEL")
    private BigDecimal salaireAnnuel;

    @ManyToMany(fetch=FetchType.LAZY,cascade=CascadeType.ALL)
    private Set<Responsabilite> responsabilites = new HashSet<Responsabilite>();
}
```

```
@Entity
@Table(name="RESPONSABILITE")
public class Responsabilite {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID",nullable=false,unique=true,updatable=false)
    private Long id;

    @Column(name="LIBELLE",nullable=false,length=50)
    private String libelle;

    @Column(name="DATE_CREATION",nullable=false)
    private Date dateCreation;

    @ManyToMany(mappedBy="responsabilites")
    private Set<Collaborateur> collaborateurs = new HashSet<Collaborateur>();
}
```

## MAPPING 1:1

---

- + Associations 1-n
- + Associations n-n
- + Associations 1-1
- + Composants
- + Collections de valeurs

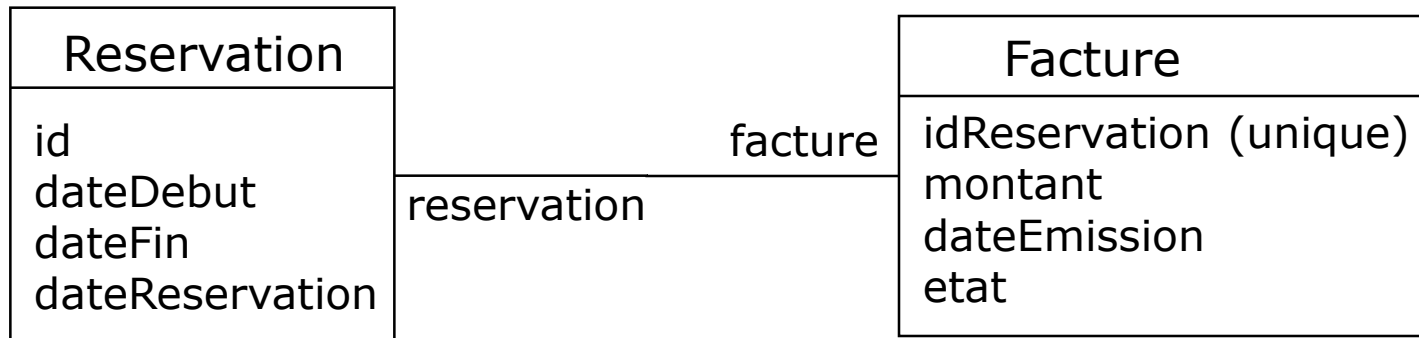
## 1-1 : PRÉSENTATION (1/3)

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

- ▀ Avec contrainte d'unicité

**+ L'une des contreparties peut exister sans l'autre**

- ▀ Il peut y avoir une réservation sans facture
- ▀ Il **ne peut pas** y avoir de facture sans réservation
  - + Facture n'a pas d'identifiant propre



## 1-1 : PRÉSENTATION (2/3)

### + 2 types de relation

#### ➤ <one-to-one>

+ Implique que chaque entité est liée à une et une seule entité

- Correspondant à la table contenant la clé primaire (Réservation)
- Permet de naviguer de reservation vers facture
- Nécessite de surcharger le nom de colonne de la clé étrangère via @JoinColumn

+ Ne peut pas évoluer dans le temps

- Ex : Une facture ne peut pas être attribuée à une autre réservation

#### ➤ <many-to-one> diminuée

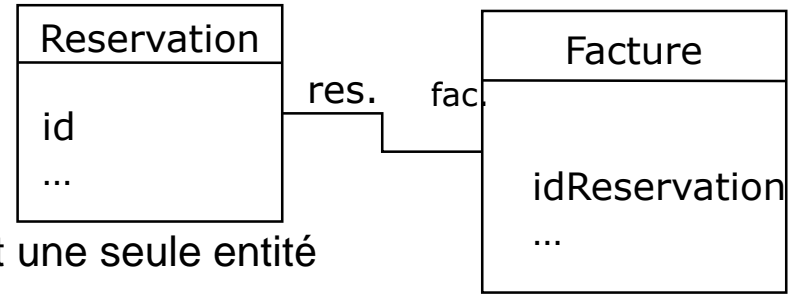
+ Relation many-to-one avec contrainte d'unicité

+ Déclarée du côté "dépendant" (Facture)

- Permet de naviguer de facture vers reservation

+ La relation peut évoluer dans le temps

- Une facture peut être attribuée à une autre réservation



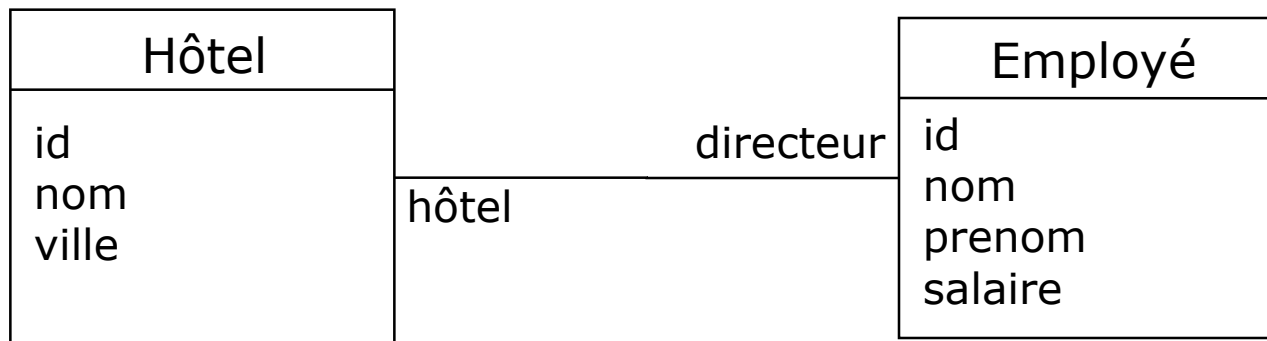
## 1-1 : PRÉSENTATION (3/3)

### + Critères de choix de la relation 1-1 :

- ▀ Quelle table doit porter la clé étrangère ?
- ▀ La relation peut-elle évoluer dans le temps ?

### + Cas pratique : un hôtel et son directeur

- ▀ Un hôtel a un seul directeur
- ▀ Quelle table doit contenir la clé étrangère ?
- ▀ one-to-one ou many-to-one diminuée ?



# 1:1 PAR CLEFS PRIMAIRES: PRÉSENTATION

---

## + Les relations par clefs primaires

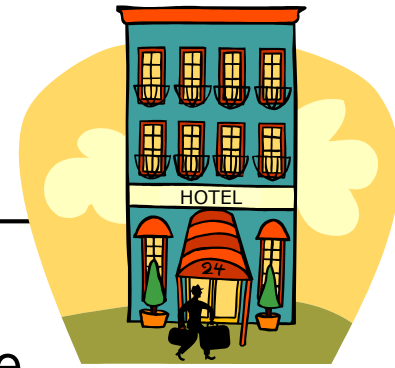
- Les deux Objets liés possèdent le même identifiant
- Le premier objet est créé normalement (génération d'un ID)
- Le second objet utilise l'ID du premier objet comme clé étrangère
- Le premier objet peut exister sans le second, mais le second ne peut exister sans le premier

## + Dans le PMD, ajout d'une foreign Key sur l'ID du second

## + Les identifiants sont constants durant toute la durée de vie des objets (par définition d'un identifiant)

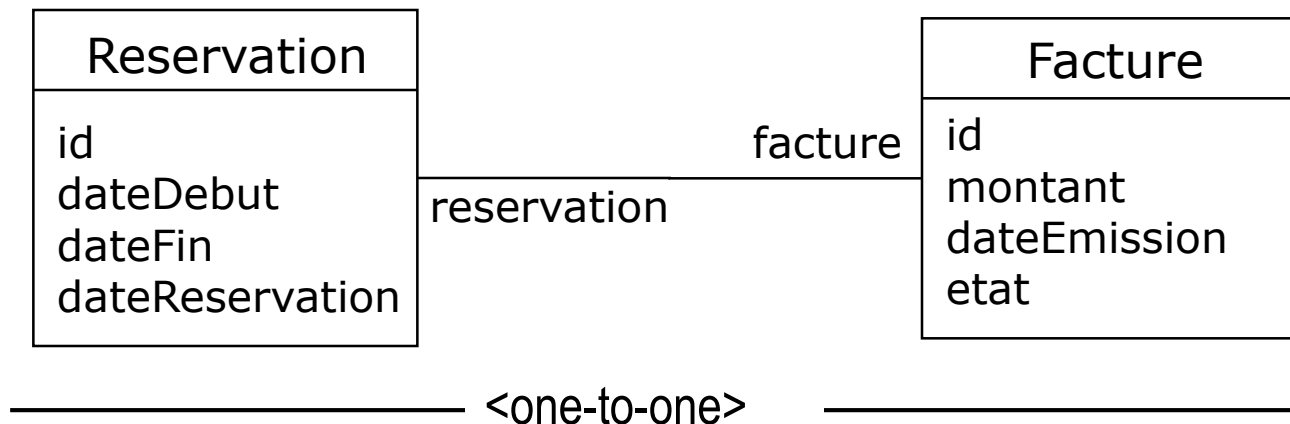
- La relation une fois établie ne peut pas être modifiée
- Suppressions : 2eme sans 1ere → OK / 1ere sans 2eme → KO

## 1:1 PAR CLEFS PRIMAIRES: EXEMPLE (1/3)



### + Exemple

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



## 1:1 PAR CLEFS PRIMAIRES: EXEMPLE (2/3)

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

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



# 1:1 PAR CLEFS PRIMAIRES: EXEMPLE (3/3)

```
@Entity
@Table(name = "CLIENT")
public class Client {

    // Id technique generation AUTO
    @Id
    @Column(name = "ID", unique = true, nullable = false)
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    @Column(name = "NOM", nullable = false, length = 25)
    private String nom;

    @Column(name = "PRENOM", nullable = false, length = 25)
    private String prenom;

    @Column(name = "SEXE", nullable = false, length = 1)
    private char sexe;

    @Column(name = "DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;

    // relation principale personne (one) -> Adresse (one)
    // implémentée par la clé étrangère Personne(adresse_id) ->
    // une Personne doit avoir 1 Adresse (nullable=false)
    // 1 Adresse n'appartient qu'à 1 personne (unique=true)
    @OneToOne
    @JoinColumn(name = "ADRESSE", unique=true, nullable = false)
    // @JoinTable(name="client_adresse",
    // joinColumns=@JoinColumn(name="CLIENT_ID"),
    // inverseJoinColumns=@JoinColumn(name="ADRESSE_ID"))
    private Adresse adresse;
```

```
@Entity
@Table(name = "ADRESSE")
public class Adresse implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    // Id technique generation AUTO
    @Id
    @Column(name = "ID", unique = true, nullable = false)
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    @Column(name = "NUMERO_RUE", nullable=false,length=10)
    private String NumeroRue;

    @Column(name = "RUE", nullable=false,length=50)
    private String rue;

    @Column(name = "CODE_POSTALE", nullable=false,length=5)
    private String codePostal;

    @Column(name = "VILLE", nullable=false,length=20)
    private String ville;
```

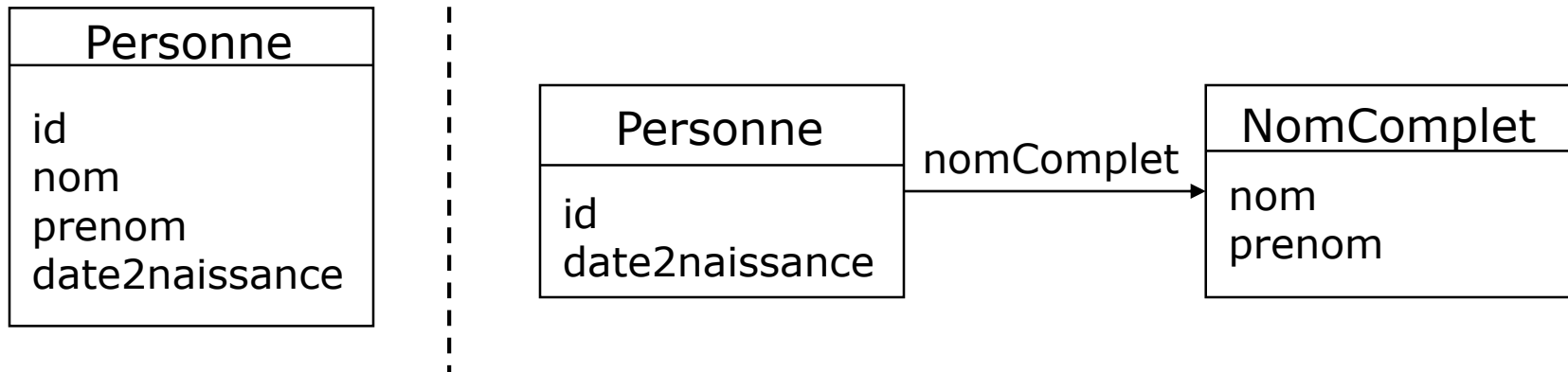
```
Client c1 = ...;
Adresse ad = new Adresse();
c1.setAdresse(ad);
```



Si aucune annotation `@JoinColumn` ou `@PrimaryKeyJoinColumn` n'accompagne un mapping `@oneToOne`, un nom de colonne par défaut est supposé. Ce nom par défaut est formé de la combinaison des noms d'entités sources et du nom de la clé primaire de l'entité cible séparés par le caractère `_`

- 
- + Associations 1-n
  - + Associations n-n
  - + Associations 1-1
  - + Composants
  - + Collections de valeurs

- + **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
- + **Dans un souci de ré-utilisabilité, Hibernate encourage l'utilisation de composants**



## COMPOSANTS (2/4)

```
public class Personne {
    private long id;
    private Date date2naissance;
    @Embedded
    @AttributeOverrides(value={@AttributeOverride(name="nom",column=@Column(name="NOM")),
    @AttributeOverride(name="pernom",column=@Column(name="PRENOM"))})
    private NomCompleet nomCompleet;

    public long getId() {
        return id;
    }

    public NomCompleet getNomCompleet() {
        return nomCompleet;
    }
    public void setNomCompleet(NomCompleet nomCompleet) {
        this.nomCompleet = nomCompleet;
    }
    ...
}
```

## COMPOSANTS (3/4)

```
public class NomComplet {  
    private String nom;  
    private String prenom;  
  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
  
    public String getPrenom() {  
        return prenom;  
    }  
    public void setPrenom(String prenom) {  
        this.prenom = prenom;  
    }  
}}
```

**+ Dans cet exemple, NomComplet sont persistés en tant que composant de personne → persistance dans la même table**

Les composants n'ont par définition pas d'identifiant

## + **Hibernate propose deux façon de gérer les clés primaires**

- L'annotation `@javax.persistence.IdClass`
- L'annotation `@javax.persistence.EmbeddedId`

## + **La classe de la clé primaire doit obligatoirement**

- Être sérialisable
- Posséder un constructeur sans argument
- Fournir une implémentation des méthodes `equals()` et `hashCode()`

# COMPSANTS-CLÉS PRIMAIRES (1/2)

## + Exemple @IdClass

```
@Entity
@Table(name = "PERSONNE")
@IdClass(PersonnePK.class)
public class Personne {

    @Id
    @Column(name = "NOM", nullable = false, length = 25)
    private String nom;

    @Id
    @Column(name = "PRENOM", nullable = false, length = 25)
    private String prenom;

    @Id
    @Column(name = "DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;

    @Column(name = "SEXE", nullable = false, length = 1)
    private char sexe;

    public Personne() {
        super();
    }
}
```

```
public class PersonnePK implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nom;

    private String prenom;

    private Date dateNaissance;

    public PersonnePK() {
        super();
    }
}
```

# COMPOSANTS-CLÉS PRIMAIRES (2/2)

## + Exemple @EmbeddedId

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

    @EmbeddedId
    @AttributeOverrides(value={@AttributeOverride(name="nom",column=@Column(name="NOM")),
                             @AttributeOverride(name="pernom",column=@Column(name="PRENOM")),
                             @AttributeOverride(name="dateNaissance",column=@Column(name="DATE_NAISSANCE"))})

    private PersonnePK pk;

    @Column(name = "SEXE", nullable = false, length = 1)
    private char sexe;
```

```
@Embeddable
public class PersonnePK implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nom;

    private String prenom;

    private Date dateNaissance;

    public PersonnePK() {
        super();
    }
}
```



- 
- + **Associations 1-n**
  - + **Associations n-n**
  - + **Associations 1-1**
  - + **Composants**
  - + **Collections de valeurs**

**+ Une « collection de valeurs » est une collection d'objets représentant la valeur d'un attribut d'une entité**

- UML : notion de composition
- Objet : même cycle de vie

**+ Dans le MPD**

- Une table contenant l'entité
- Une table contenant les valeurs de la collection
- Une FK de la table contenant les valeurs vers l'identifiant de l'entité

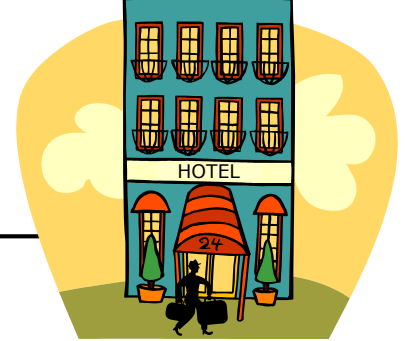
**+ Collections pouvant contenir 2 type de valeurs**

- Des objets simples (ex: String, Long, Date.....)
- Des composants (Objets contenant un ensemble d'objets simples)



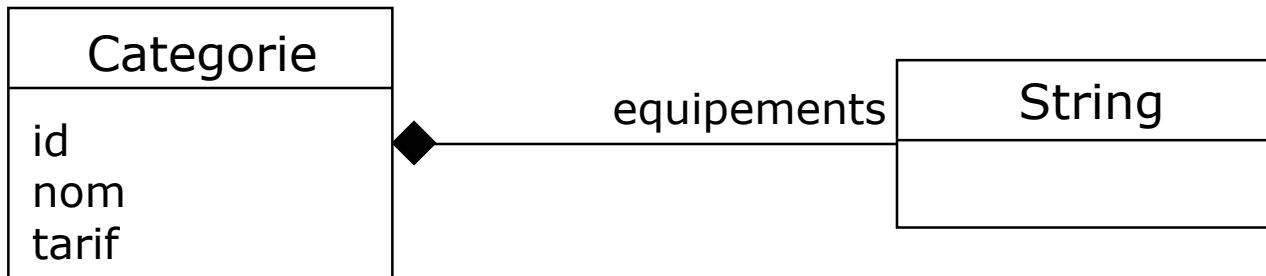
Les objets valeurs n'ont pas d'identifiant

# COLLECTION D'OBJETS SIMPLES: EXEMPLE (1/3)



## + Exemple

- ▶ Chaque **Categorie** possède un ensemble d'**Equipement**



Unidirectionnelle et bidirectionnelle n'ont ici aucun sens !!!

## COLLECTION D'OBJETS SIMPLES: EXEMPLE (2/3)

### + Exemple :

```
public class Categorie {  
    private int id;  
    @ElementCollection  
    @CollectionTable(name="EQUIPEMENT_CAT",joinColumns=@JoinColumn(name="CAT_ID")  
    @Column(name="DESCRIPTION")  
    private Set<String> equipements; // référence vers les équipements  
    ...  
    public Categorie() {  
        equipements = new HashSet();  
    }  
    public Set<String> getEquipements() { return equipements; }  
    public void setEquipements(Set<String> eq) { this.equipements = eq; }  
}
```



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



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

---

**+ TP 05 : Reprenez les mapping de vos objets et ajoutez les relations.**

# LES RELATIONS : ASPECTS AVANCÉS



- 
- + **Types de collections**
  - + **Cascade**
  - + **Unidirectionnalité et bidirectionnalité**

**+ 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



Bag dans le mapping → List dans le Java





---

+ **Types de collections**

+ **Cascade**

+ **Unidirectionnalité et bidirectionnalité**

# CASCADE (1/4): PRINCIPE

## + L'attribut <cascade>

- Permet de propager certaines opérations effectuées sur une entité liée
- Est disponible pour toutes les relations
  - + ManyToOne
  - + OneToMany
  - + OneToOne
  - + ManyToMany

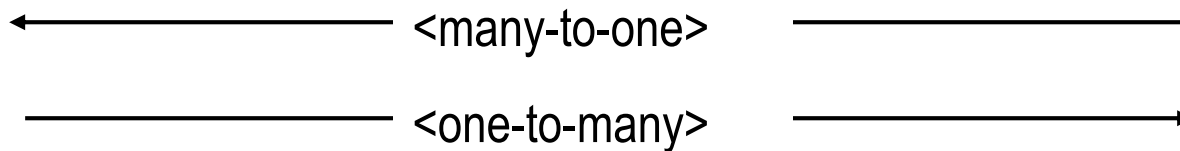
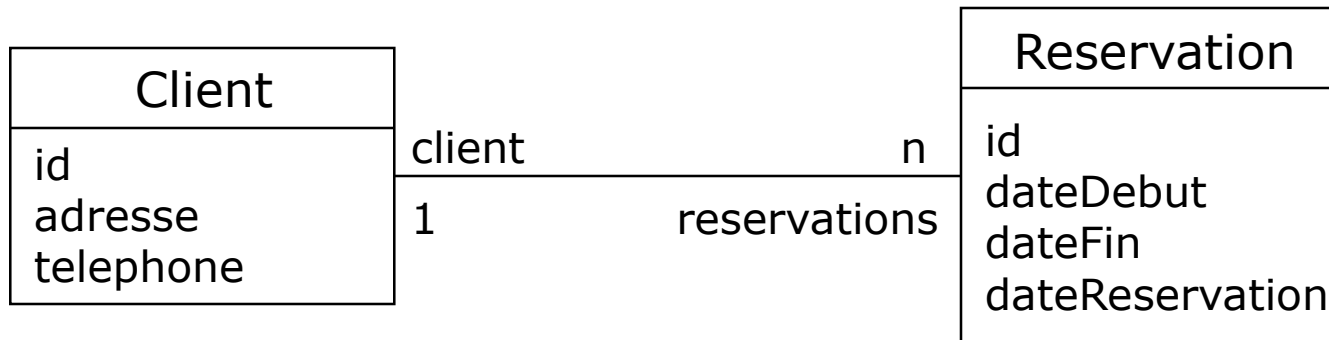
## + Les valeurs possibles sont

- ALL : persist,merge,remove,refresh,detach
- PERSIST : enregistrement de l'objet → enregistrement de la relation
- MERGE : mise à jour des objets liés
- REMOVE : suppression d'un objet → suppression de la relation lié
- REFRESH : ré attache les objets au contexte
- DETACH : Détache les objets en dehors du contexte

## CASCADE (2/4): EXEMPLE

### + Exemple

- Un **Client** peut effectuer plusieurs **Reservation**
- Une **Reservation** est effectuée par un **Client**



# CASCADE (3/4): EXEMPLE

## + Mapping

```
public class Client{
    private int id;
    @OneToMany(mappedBy="client")
    private Set<Reservation> reservations;

    public client(){
        reservations = new HashSet<Reservation>();
    }
}
```

## + Utilisation

```
Client c = new Client();
c.setNom("Dupont");
c.setPrenom("Charlie");
Reservation res= new Reservation();
res.setNum("123544")
c.getReservations.add(res)
ss.save(res);
ss.save(c); // instruction OBLIGATOIRE
```

# CASCADE (4/4): EXEMPLE

## + Mapping

```
public class Client{
    private int id;
    @OneToMany(mappedBy="client", cascade=CascadeType.ALL)
    private Set<Reservation> reservations;

    public client(){
        reservations = new HashSet<Reservation>();
    }
}
```

## + Utilisation

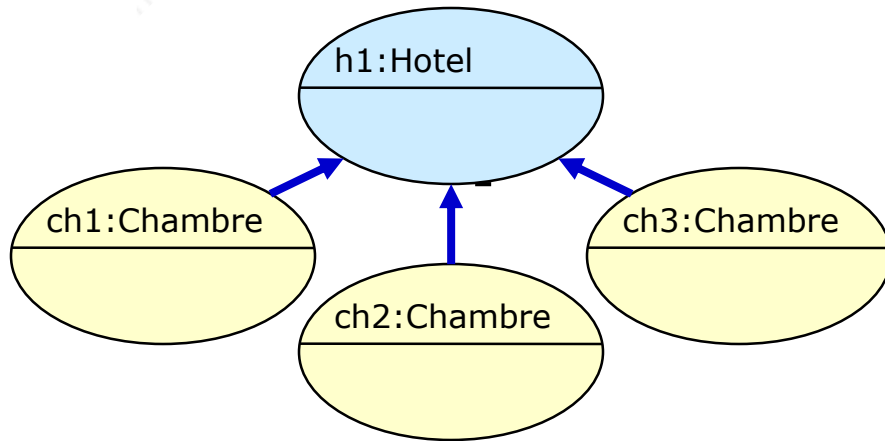
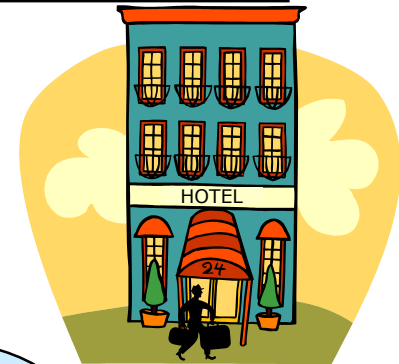
```
Client c = new Client();
c.setNom("Dupont");
c.setPrenom("Charlie");
Reservation res= new Reservation();
res.setNum("123544")
c.getReservations.add(res)

ss.save(c);
```

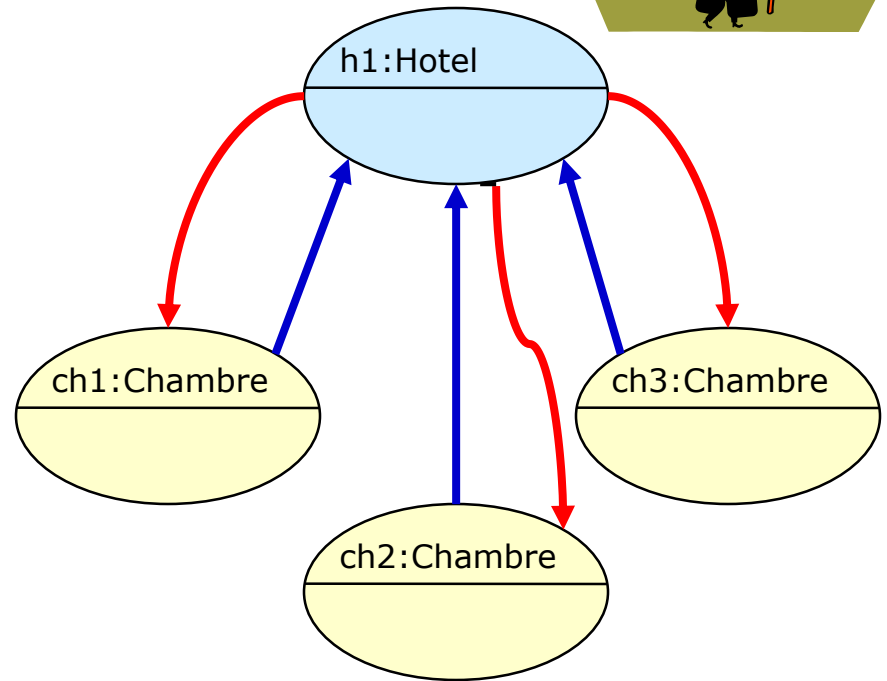
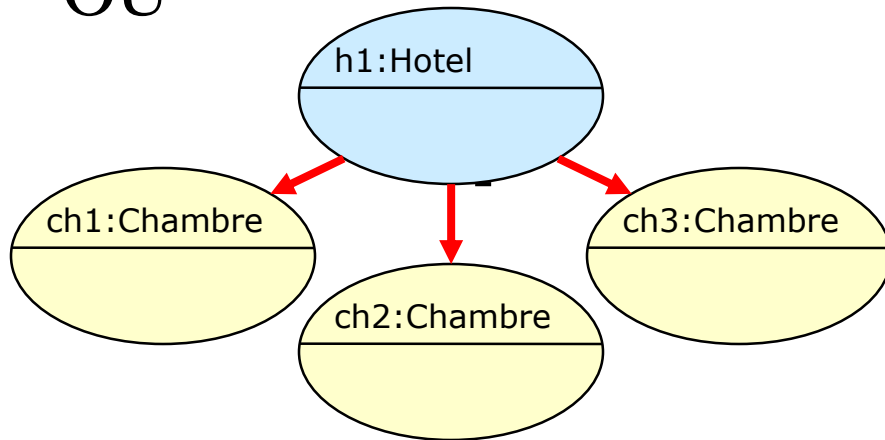
- 
- + Types de collections
  - + Cascade
  - + Unidirectionnalité et bidirectionnalité

# UNIDIRECTIONNELLE / BIDIRECTIONNELLE: QUE CHOISIR ?

## + Exemple de la relation Chambre / Hotel



OU



## + Unidirectionnelle

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

## + Bidirectionnelle

- (+) Possibilité de naviguer dans les deux sens d'une relations
- (-) 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)

# MAINTIEN DE LA BI-DIRECTIONNALITÉ (2/4)

## + 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

### + 2ème 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);
```

## + Attention aux noms des méthodes

- 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(...)

## + Éviter les noms de type :

- créerChambre(...) & supprimerChambre(...)

HÉRITAGE



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

- ▀ Encapsulation
- ▀ Héritage
- ▀ Polymorphisme

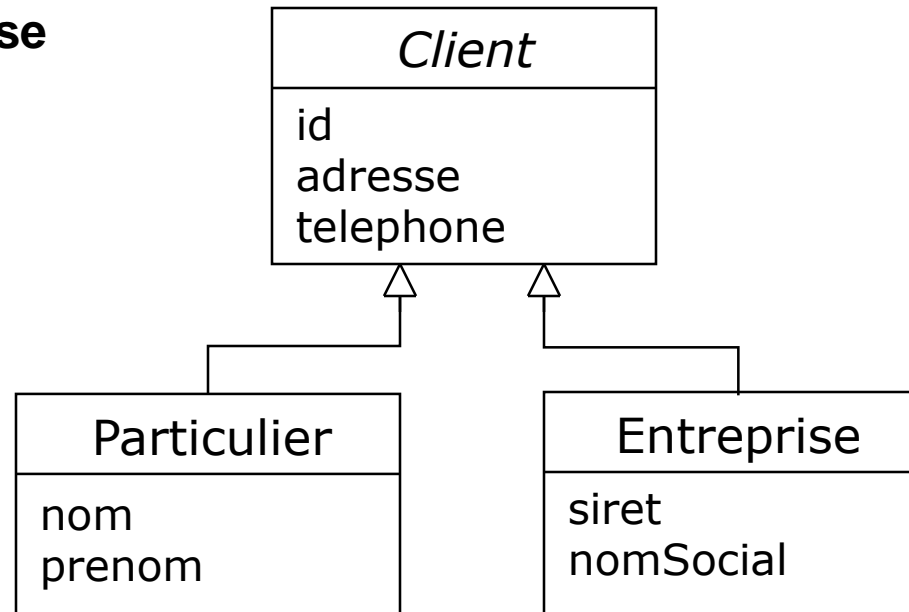
## **+ 3 stratégies permettent le mapping O/R de l'héritage**

- ▀ Chaque stratégie a ses avantages et ses inconvénients

## **+ Hibernate permet l'utilisation des 3 stratégies**

# L'HÉRITAGE: EXEMPLE

- + Un Client peut être
- soit un **Particulier**
  - soit une **Entreprise**



Nous considérerons que la classe Client est abstraite

# 1 TABLE PAR CLASSE FILLE : PRINCIPE

Théorique	++
Technique	--

+ **Appellation: « Table per subclass »**

+ **Idée générale**

- A chaque classe du modèle objet correspond une table
  - + 3 tables dans l'exemple CLIENT-PARTICULIER-ENTREPRISE
- Chaque table contient les attributs de l'objet + l'identifiant
- L'héritage entre classes est modélisé par des clés étrangères

+ **Conceptuellement**

- (+) Solution simple et efficace

+ **Techniquement**

- (-) Jointures lors de chaque requêtes → performance non optimal
- Ne prend pas en charge la stratégie de générateur IDENTITY



# 1 TABLE PAR CLASSE FILLE : EXEMPLE



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

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

# 1 TABLE PAR CLASSE FILLE: EXEMPLE

---

```
public abstract class Client {  
    private int id;  
    private String adresse;  
    private String telephone;  
    ...  
}
```

```
public class Particulier extends Client { // hérite de Client  
    private String nom;  
    private String prenom;  
    ...  
}
```

```
public class Entreprise extends Client { // hérite de Client  
    private String siret;  
    private String nomSocial;  
    ...  
}
```

# 1 TABLE PAR CLASSE FILLE : EXEMPLE (1/2)

## + Mapping

```
@Entity
@Table(name = "CLIENT_INHERITENCE")
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class ClientInheritance extends PersistenceObject{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Column(name="ADRESSE",length=80)
    private String adresse;

    @Column(name="PHONE",length=10)
    private String phone;
```



Si aucun nom de table n'est précisé, utilisation du nom de la classe

# 1 TABLE PAR CLASSE FILLE : EXEMPLE (2/2)

## + Mapping

```
@Entity
@Table(name="PARTICULIER")
@PrimaryKeyJoinColumn(name="ID",referencedColumnName="ID")
public class Particulier extends ClientInheritance {

    /**
     * UID
     */
    private static final long serialVersionUID = 1L;

    @Column(name="NOM")
    private String nom;

    @Column(name="PRENOM")
    private String prenom;

    @Column(name="DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;
}

@Entity
@Table(name="ENTREPRISE")
@PrimaryKeyJoinColumn(name="ID",referencedColumnName="ID")
public class Entreprise extends ClientInheritance {

    /**
     * UID
     */
    private static final long serialVersionUID = 1L;

    @Column(name="NOM_SOCIAL")
    private String nomSocial;

    @Column(name="SIRET")
    private String siret;
}
```

# 1 TABLE PAR CLASSE FILLE: EXEMPLE


## + Utilisation

```
Entreprise en1 = new Entreprise(); // création d'une entreprise
en1.setAdresse("98 route de la Reine");
en1.setTelephone("0141220300");
en1.setNomSocial("Sysdeo");
en1.setSiret("44019981800012");
session.save(en1);

Particulier pa1 = new Particulier(); // création d'un particulier
pa1.setAdresse("14 rue de la Foret");
pa1.setTelephone("0112341234");
pa1.setNom("Durant");
pa1.setPrenom("Cédric");
session.save(pa1);

// recherche de clients
Client c11 = (Client) session.get(Client.class, new Integer(345));
Particulier pa2 = (Particulier) session.get(Client.class, new Integer(114));
Entreprise e2 = (Entreprise) session.get(Entreprise.class, new Integer(76));
```

Recherche sans  
connaissance du  
type réel



# 1 TABLE POUR TOUTE LA HIÉRARCHIE : PRINCIPE

## + Appellation: « Table per class-hierarchy »

Théorique	+-
Technique	--

## + Idée générale

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

## + Conceptuellement

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

## + Techniquement

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

# 1 TABLE POUR TOUTE LA HIÉRARCHIE: EXEMPLE

## + Une seule table dans laquelle sont stockées

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

## + La colonne TYPE contient le discriminateur

P → Particulier      E → Entreprise



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

clé primaire

# 1 TABLE POUR TOUTE LA HIÉRARCHIE: EXEMPLE

---

```
public abstract class Client {  
    private int id;  
    private String adresse;  
    private String telephone;  
    ...  
}
```

```
public class Particulier extends Client { // hérite de Client  
    private String nom;  
    private String prenom;  
    ...  
}
```

```
public class Entreprise extends Client { // hérite de Client  
    private String siret;  
    private String nomSocial;  
    ...  
}
```



# 1 TABLE POUR TOUTE LA HIÉRARCHIE: EXEMPLE (1/2)

## + Mapping

```
@Entity
@Table(name = "CLIENT_INHERITENCE")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE",discriminatorType=DiscriminatorType.STRING)
public abstract class ClientInheritance extends PersistenceObject{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Column(name="ADRESSE",length=80)
    private String adresse;

    @Column(name="PHONE",length=10)
    private String phone;
```

← Nom et type de la  
colonne servant de  
discriminateur



@DiscriminatorColumn doit être déclarée dans la classe supérieure

# 1 TABLE POUR TOUTE LA HIÉRARCHIE EXEMPLE (2/2)

## + Mapping

```
@Entity
@Table(name="ENTREPRISE")
@PrimaryKeyJoinColumn(name="ID",referencedColumnName="ID")
@DiscriminatorValue(value="E")
public class Entreprise extends ClientInheritance {

    /**
     * UID
     */
    private static final long serialVersionUID = 1L;

    @Column(name="NOM_SOCIAL")
    private String nomSocial;

    @Column(name="SIRET")
    private String siret;
```

```
@Entity
@Table(name="PARTICULIER")
@PrimaryKeyJoinColumn(name="ID",referencedColumnName="ID")
@DiscriminatorValue(value="P")
public class Particulier extends ClientInheritance {

    /**
     * UID
     */
    private static final long serialVersionUID = 1L;

    @Column(name="NOM")
    private String nom;

    @Column(name="PRENOM")
    private String prenom;

    @Column(name="DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;
```

# 1 TABLE POUR TOUTE LA HIÉRARCHIE: EXEMPLE

## + Utilisation

```
Entreprise en1 = new Entreprise(); // création d'une entreprise
en1.setAdresse("98 route de la Reine");
en1.setTelephone("0141220300");
en1.setNomSocial("Sysdeo");
en1.setSiret("44019981800012");
session.save(en1);

Particulier pa1 = new Particulier(); // création d'un particulier
pa1.setAdresse("14 rue de la Foret"); pa1.setTelephone("0112341234");
pa1.setNom("Durant");
pa1.setPrenom("Cédric");
session.save(pa1);

// recherche de clients
Client cl1 = (Client) session.get(Client.class, new Integer(345));
Particulier pa2 = (Particulier) session.get(Client.class, new Integer(114));
Entreprise en2 = (Entreprise) session.get(Entreprise.class, new Integer(76));
```

Recherche sans connaissance du type réel



# 1 TABLE PAR CLASSE CONCRÈTE: PRINCIPE

## + 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

# 1 TABLE PAR CLASSE CONCRÈTE : EXEMPLE

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



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

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

clés primaires

# 1 TABLE PAR CLASSE CONCRÈTE

## + Exemple

```
public abstract class Client {  
    private int id;  
    private String adresse;  
    private String telephone;  
    ...  
}
```

```
public class Particulier extends Client { // hérite de Client  
    private String nom;  
    private String prenom;  
    ...  
}
```

```
public class Entreprise extends Client { // hérite de Client  
    private String siret;  
    private String nomSocial;  
    ...  
}
```



# 1 TABLE PAR CLASSE CONCRÈTE: EXEMPLE (1/2)

## + Mapping

```
@Entity
@Table(name = "CLIENT_INHERITENCE")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class ClientInheritance extends PersistenceObject{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Column(name="ADRESSE",length=80)
    private String adresse;

    @Column(name="PHONE",length=10)
    private String phone;
```



Le mapping serait identique si la relation d'héritage était absente

# 1 TABLE PAS CLASSE CONCRÈTE (1/2)

## + mapping

```
@Entity
@Table(name="ENTREPRISE")
public class Entreprise extends ClientInheritance {

    /**
     * UID
     */
    private static final long serialVersionUID = 1L;

    @Column(name="NOM_SOCIAL")
    private String nomSocial;

    @Column(name="SIRET")
    private String siret;
```

```
@Entity
@Table(name="PARTICULIER")
public class Particulier extends ClientInheritance {

    /**
     * UID
     */
    private static final long serialVersionUID = 1L;

    @Column(name="NOM")
    private String nom;

    @Column(name="PRENOM")
    private String prenom;

    @Column(name="DATE_NAISSANCE")
    @Temporal(TemporalType.DATE)
    private Date dateNaissance;
```



# 1 TABLE PAR CLASSE CONCRÈTE : EXEMPLE

## + Utilisation

```
Entreprise en1 = new Entreprise(); // création d'une entreprise
en1.setAdresse("98 route de la Reine");
en1.setTelephone("0141220300");
en1.setNomSocial("SQLI");
en1.setSiret("44019981800012");
session.save(en1);

Particulier pa1 = new Particulier(); // création d'un particulier
pa1.setAdresse("14 rue de la Foret");
pa1.setTelephone("0112341234");
pa1.setNom("Durant");
pa1.setPrenom("Cédric");
session.save(pa1);

// recherche de clients
Particulier pa2 = (Particulier) session.get(Particulier.class, new Integer(43));
Entreprise en2 = (Entreprise) session.get(Entreprise.class, new Integer(43));
```



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

- + Aucune table spécifique pour cette classe**
- + Partage des propriétés communes à travers une classe technique**
  - Mapping des propriétés communes est copié dans chaque classe fille
  - Les propriétés des classes parents non mappées sont ignorées

# HÉRITAGE DE PROPRIÉTÉS DES CLASSES PARENTS (1/2)

## + mapping

```
@MappedSuperclass
public abstract class PersistenceObject implements Serializable {

    /**
     * UID
     */
    public static final long serialVersionUID = 1L;

    @Id
    @SequenceGenerator(name="SEQ_CLIENTIEHERITENCE",sequenceName="SEQ_PERSISTENCTOBJECT",initialValue=1)
    @GeneratedValue(strategy=GenerationType.SEQUENCE,generator="SEQ_CLIENTIEHERITENCE")
    private Long id;

    public Long getId() {
        return id;
    }

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

# HÉRITAGE DE PROPRIÉTÉS DES CLASSES PARENTS (2/2)

## + Mapping

```
@Entity
@Table(name = "CLIENT_INHERITENCE")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class ClientInheritance extends PersistenceObject{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Column(name="ADRESSE",length=80)
    private String adresse;

    @Column(name="PHONE",length=10)
    private String phone;
```

## TABLE SECONDAIRE

---

- + Une seule classe**
- + Mapping dans plusieurs tables secondaires en base de données**
- + Utilisation d'une requête de jointure**

# MAPPING TABLE SECONDAIRE

## + Mapping

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables(value={@SecondaryTable(name="EMPLOYEE_ADRESSE",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="EMP_ID",referencedColumnName="EMP_ID"))})
public class Employee {

    @Id
    @Column(name="EMP_ID")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="NOM")
    private String nom;

    @Column(name="PRENOM")
    private String prenom;

    @Column(name="DATE_NAISSANCE")
    private Date dateNaissance;

    @Column(name="ADRESSE",table="EMPLOYEE_ADRESSE")
    private String adresse;

    @Column(name="CODE_POSTALE",table="EMPLOYEE_ADRESSE")
    private Integer codePostal;

    @Column(name="VILLE",table="EMPLOYEE_ADRESSE")
    private String ville;
```

## + Les éléments à prendre en compte

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

### + Recommendations

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



### **+ Table per hierarchy pour les problèmes simples**

- Facile à mettre en œuvre, bonnes performances

### **+ Pour des cas plus complexes : table per subclass**

- Selon les performances JOINED

### **+ Penser / reconcevoir sans héritage**

- Utiliser la délégation
- Penser aux composants

---

**+ TP 06 : Ajouter l'héritage à vos classes.**

REQUÊTAGE



# PRÉSENTATION

---

- + **Présentation**
- + **Utilisation courante**
- + **Polymorphisme**
- + **Restriction**
- + **Jointures**

**+ Hibernate dispose de 3 techniques de requêtage**

**+ JPA-QL / HQL(Java Persistence Query Language /Hibernate Query language)**

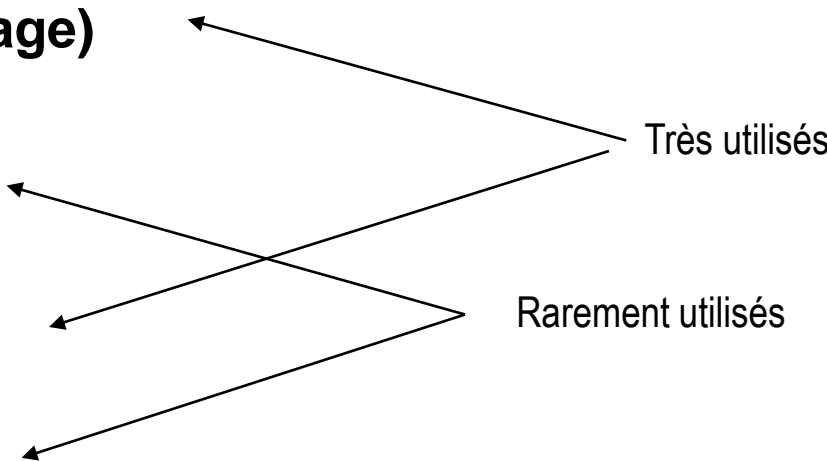
**+ SQL Direct(Native Query)**

**+ API Criteria**

- QBC(Query by Criteria)
- QBE(Query by Exemple)

Très utilisés

Rarement utilisés



# SÉLECTION: EXEMPLE

## + But

- ▀ Réussir à sélectionner les hôtels dont le nom est 'Lyon-Centre'



Hotel
id nom ville

## LES 3 TYPES D'APPROCHES

---

### + HQL : approche langage de requête « objet »

```
session.createQuery("from Hotel h where h.nom like 'Lyon-Centre").list();
```

**Ou**

```
Session session = session.getNamedQuery("hotel.readByNom").list();  
Session.setParameter("name", "Lyon-Centre);
```

### + CriteriaAPI : approche modélisation objet d'une requête

```
session.createCriteria(Hotel.class).add(Restrictions.like("nom", "Lyon-Centre")).list();
```

### + SQL Direct (Native Query) : approche pur SQL

```
session.createSQLQuery("SELECT {h.*} from HOTEL h WHERE NOM like 'Lyon-Centre'")  
.addEntity("h", Hotel.class).list();
```

## + Syntaxe inspirée du SQL, adaptation Objet

- Accepte les noms de classes et attributs
- Ne supporte pas les noms de tables et de colonnes

```
"from Hotel h where h.nom like 'Lyon-Centre'"
```

## + Analyse syntaxique complète

- La requête est portée par une chaîne de caractères
- Les éventuelles erreurs de syntaxes ne peuvent pas être découvertes en phase de **compilation**
- Pas d'assistance pour le **refactoring** (renommage de classe...)

## + Langage concis

- Mais il sera vu par la suite que la syntaxe HQL s'alourdit avec les paramètres nommés



## + Syntaxe orientée Java

```
Criteria c = session.createCriteria(Hotel.class);  
c.add(Restrictions.like("nom", "Lyon-Centre"));  
List resultList = c.list();
```

## + Précompilation partielle

- Une partie des erreurs de syntaxe peut être vue en phase de compilation
- Refactoring facilité
  - + En cas de renommage de classe
  - + Pas pour les renommages d'attributs (méthodes get/set)

## + Limites sur certains types de requête

- Pas adapté pour les requêtes imbriquées (utiliser HQL)
- Requêtes utilisant des expressions arithmétiques (utiliser HQL)
- ...

## + De moins en moins nécessaire

- ▀ HQL et Criteria sont maintenant très complets

## + Utilisé pour profiter d'optimisations spécifiques à un SGBD

- ▀ Pour le cas où Hibernate ne prend pas en compte nativement ces optimisations
  - + Ex : 'connect by prior' dans Oracle

- 
- + **Présentation**
  - + **Utilisation courante**
  - + **Polymorphisme**
  - + **Restrictions**
  - + **Jointures**

## L'API DE SÉLECTION (1/2)

**+ Des interfaces permettent de manipuler les requêtes de sélection**

**+ Pour l'API HQL → l'interface Query**

```
Query hqlQuery = session.createQuery("from Hotel");
```

**OU**

```
Query hqlQuery = session.getNamedQuery("hotel.readByName");
```

**+ Pour Direct SQL → l'interface SQLQuery (étend Query)**

```
SQLQuery sqlQuery = session.createSQLQuery("select {h.*} from Hotel h");  
sqlQuery.addEntity("h", Hotel.class);
```

**+ Pour l'API Criteria → l'interface Criteria**

```
Criteria crit = session.createCriteria(Hotel.class);
```

### + Ajout de critères

```
Query query = session.createQuery("from Hotel h order by h.nom asc");  
query.setMaxResults(10);
```

```
Criteria crit = session.createCriteria(Hotel.class)  
crit.add( Order.asc("nom") );  
crit.setFirstResult(40);  
crit.setMaxResults(20);  
List results = crit.list();
```

### + Création et exécution de requête

```
List result = session.createQuery("from Hotel").list();
```

```
List results = session.createCriteria(Hotel.class)  
.add( Order.asc("nom")).setFirstResult(40).setMaxResults(20).list();
```

# BONNE UTILISATION DE HQL

## + Tentant mais à proscrire !

```
String nom = "Lyon-Centre";  
String queryString = "from Hotel h where h.nom like '" + nom + "'";  
List result = session.createQuery(queryString).list();
```

## + Bonne solution

```
@Entity(name="HOTEL")  
@NamedQueries(value={@NamedQuery(name="hotel.readByName",query=" FROM Hotel h where h.nom=:nom")})  
public class Hotel {  
    @Column(name="NOM")  
    private String nom;
```

```
Query query = session.getNamedQuery("hotel.readByName");  
query.setParameter("nom", "Lyon-centre");  
List result = query.list();
```

## + 2 approches possibles

### ▀ Par position

+ Premier index : 0 (contrairement à JDBC)

```
String queryStr = "from Hotel h where h.nom like ? ";  
List result = session.createQuery(queryStr).setString(0,"Lyon-Centre").list();
```

### ▀ Par paramètre nommé

```
String queryStr = "from Hotel h where h.nom like :nom";  
List result = session.createQuery(queryStr).setString("nom","Lyon-Centre").list();
```

## + En Criteria, le parameter binding est implicite

```
Criteria criteria = session.createCriteria(Hotel.class);  
criteria.add(Restrictions.eq("nom", "Lyon-Centre"));  
criteria.add(Restrictions.eq("ville", "Lyon"));  
List list1 = criteria.list();
```



## + Named Query

- Les requêtes sont déclarées au niveau de chaque classe déclarant l'annotation @Entity
- Chaque requête est englobée dans @NamedQuery
- Uniquement pour HQL

## + Utilisation

- dans le code Java

```
session.getNamedQuery("hotel.readByName").setString("nomHotel",  
nomHotel).list()
```

- mapping

```
@Entity(name="HOTEL")  
@NamedQueries(value={@NamedQuery(name="hotel.readByName",query=" FROM Hotel h where h.nom=:nom")})  
public class Hotel {  
  
    @Column(name="NOM")  
    private String nom;
```

## MAXRESULTS (1/3)

### + Il est possible de borner le nombre maximum des résultats d'une requête

- Ex : pour préparer l'affichage d'un tableau avec des boutons précédent / suivant

Number	Operation date	Description	Amount
1	2004-04-10 00:00:00.0	A.T.M.	55.7
1	2004-04-07 00:00:00.0	A.T.M.	12.2
1	2004-04-05 00:00:00.0	Cool disk store	98.23
1	2004-04-02 00:00:00.0	Coolest stuff	67.15
1	2004-04-02 00:00:00.0	Video game paradise	34.6
1	2003-05-02 00:00:00.0	FNAC	74.0
1	2003-05-02 00:00:00.0	Pizza Cesar	125.0
1	2003-05-02 00:00:00.0	FNAC	74.0
1	2003-05-02 00:00:00.0	Pizza Cesar	125.0
1	2003-05-02 00:00:00.0	Transfer to account 20040429SA000002	200.0
1	2003-05-01 00:00:00.0	Le notre	145.0
1	2003-05-01 00:00:00.0	Maxims	12.0
1	2003-05-01 00:00:00.0	Pomme de pain	77.0
1	2003-05-01 00:00:00.0	A.T.M.	45.0
1	2003-05-01 00:00:00.0	Transfer to account 20040429SA000002	12.0

<< < [2 / 5] > >>

### + En HQL

- Pour récupérer les 20 premiers résultats :

```
Query query = session.createQuery("from Client c");  
query.setMaxResults(20);  
query.setFirstResult(0);  
List list = query.list();
```

- Et les 20 suivants :

```
Query query = session.createQuery("from Client c");  
query.setMaxResults(20);  
query.setFirstResult(20);  
List list = query.list();
```

### + Avec l'API Criteria

```
Criteria criteria = session.createCriteria(Client.class);  
criteria.setFirstResult(20);  
criteria.setMaxResults(10);  
List list = criteria.list();
```

### + Requête SQL générée

- ▀ HQL et Criteria génèrent la même requête

```
select ... limit 20, 10
```

*Exemple de génération avec MySQL*

## + Possibilité de mettre à jour la base de données

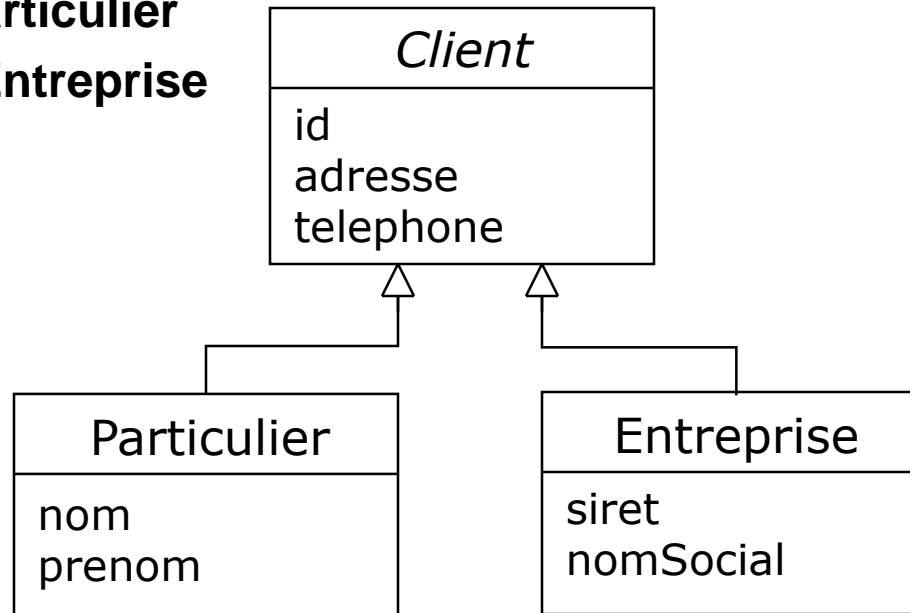
- ▀ Nécessité de déclarer une transaction
- ▀ Utilisation de la méthode **executeUpdate()**

```
Session session = BankonetTools.getSession();  
Transaction transaction = session.beginTransaction();  
Query query = session.createQuery("delete from Hotel h where h.nom= :nom");  
query.setString("nom", "Lyon-Centre");  
query.executeUpdate();  
transaction.commit();  
session.close();
```

- 
- + **Présentation**
  - + **Utilisation courante**
  - + **Polymorphisme**
  - + **Restrictions**
  - + **Jointures**

# LES REQUÊTES POLYMORPHES (1/3)

- + **Sélection des instances d'une classe et de ses sous-classes**
- + **Un Client peut être**
  - soit un **Particulier**
  - soit une **Entreprise**



## LES REQUÊTES POLYMORPHES (2/3)

### + Requête renvoyant toutes les instances Particulier et Entreprise

- ▀ Si le client est mappé

```
from Client
```

### + Pour renvoyer uniquement les particuliers

```
from Particulier
```



Peu importe la stratégie d'héritage utilisée (1 table, 2 tables ou 3 tables)



## + Requête par interfaces implémentées

- ▀ Nécessité de préciser les noms de package
- ▀ Si **BusinessObject** est une interface :

```
from com.bankonet.model.BusinessObject
```

*(renvoie les instances de toutes les classes mappées implémentant BusinessObject)*

- 
- + **Présentation**
  - + **Utilisation courante**
  - + **Polymorphisme**
  - + **Restrictions**
  - + **Jointures**

# LES RESTRICTIONS

- + Dans la pratique, il est rare de récupérer toutes les instances d'une classe
- + En général, application de contraintes sur les champs des objets

```
from Hotel h where h.nom = 'Lyon-Centre'
```

## + Avec l'API Criteria

```
Criteria crit = session.createCriteria(Hotel.class);  
crit.add(Restrictions.eq("nom", "Lyon-Centre"));  
Hotel hotel = (Hotel) crit.uniqueResult();
```

Permet de  
récupérer un  
résultat  
unique

## + Les deux APIs vont générer le même code SQL

```
select H.ID, H.NOM, H.VILLE  
from HOTEL H where H.NOM = 'Lyon-Centre'
```

# LES OPÉRATEURS DE COMPARAISON (1/2)

## + HQL supporte les mêmes opérateurs que le langage SQL

▀ =, <>, <, >, >=, <=, between, not between, in, not in

```
from Categorie cat where cat.tarif between 65 and 100
```

```
from Categorie cat where cat.tarif > 85
```

```
from Categorie cat where cat.nom in ("Standard", "Suite")
```

## + Avec l'API Criteria

```
session.createCriteria(Categorie.class)
    .add(Restrictions.between("tarif", new Float(65), new Float(100)))
    .list();
```

# LES OPÉRATEURS DE COMPARAISON (2/2)

## + Opérateur Null

- ▀ Cette requête récupère les hôtels qui n'ont pas de nom
  - + Attention, une chaîne de caractères vide n'est pas nulle !

```
from Hotel h where h.nom is null
```

- ▀ Avec l'API Criteria

```
session.createCriteria(Categorie.class)
    .add(Restrictions.isNull("nom")).list();
```

## + HQL supporte les expressions arithmétiques

- ▀ (pas supportée par la Criteria API)

```
from Hotel h where cat.tarif * ((100 - 90) / 100) < 90.0
```

# LES OPÉRATIONS SUR LES STRINGS (1/2)

## + Opérateur like, avec caractère de remplacement identique au langage SQL

```
from Hotel h where h.nom like "L%"
```

```
from Hotel h where h.nom not like "%t%"
```

## + Avec l'API Criteria

```
session.createCriteria(Hotel.class)
    .add(Restrictions.like("nom", "L", MatchMode.START))
    .list();
```

## + ou

```
session.createCriteria(Hotel.class)
    .add(Restrictions.like("nom", "L%"))
    .list();
```

## LES OPÉRATIONS SUR LES STRINGS (2/2)

### + Appel de fonctions SQL (si supportées par la base)

```
from Hotel h where upper(h.nom) = 'LYON-CENTRE'
```

### + Non supportée par l'API Criteria

- ▀ mais possibilité d'effectuer des recherches non case-sensitive

```
session.createCriteria(Hotel.class)
    .add(Restrictions.eq("nom", "Lyon-Centre").ignoreCase()))
    .list();
```

### + Concaténation de chaîne de caractères

```
from Hotel h
where (h.nom || '.' || h.ville) like 'Lyon-Centre.Lyon'
```

# LES OPÉRATEURS LOGIQUES

## + Combinaison d'opérateurs logiques et de parenthèses, pour regrouper les expressions

```
from Categorie cat
where ( cat.nom like "S%" and cat.tarif < 85.50 )
or cat.nom in ("Standard", "Suite")
```

## + Avec l'API Criteria

```
session.createCriteria(Categorie.class)
.add(Restrictions.or(
    Restrictions.and(
        Restrictions.like("nom", "S%"),
        Restrictions.lt("tarif", new Float(85.50)),
        Restrictions.in("nom", new String[]{"Standard", "Suite"}));
```



## + Tri ordonné

```
from Categorie cat order by cat.tarif
```

## + Tri ordonné sur plusieurs champs avec critères asc, desc

```
from Categorie cat order by cat.nom asc, cat.tarif desc
```

## + Avec l'API Criteria

```
List results = session.createCriteria(Categorie.class)
    .addOrder( Order.asc("nom") )
    .addOrder( Order.desc("tarif") )
    .list();
```

- 
- + Présentation**
  - + Utilisation courante**
  - + Polymorphisme**
  - + Restrictions**
  - + Jointures**

**+ Il est très souvent nécessaire de pouvoir poser des conditions sur des objets liés à l'objet que l'on cherche réellement**

**+ Hibernate permet de réaliser des jointures**

- ▀ De manière explicite en précisant la relation à suivre
  - + Sans toutefois devoir écrire les critères de jointure
- ▀ De manière implicite (dans certain cas)
  - + Sans devoir exprimer aucune jointure

# REQUÊTAGE SUR LES ASSOCIATIONS: EXEMPLE

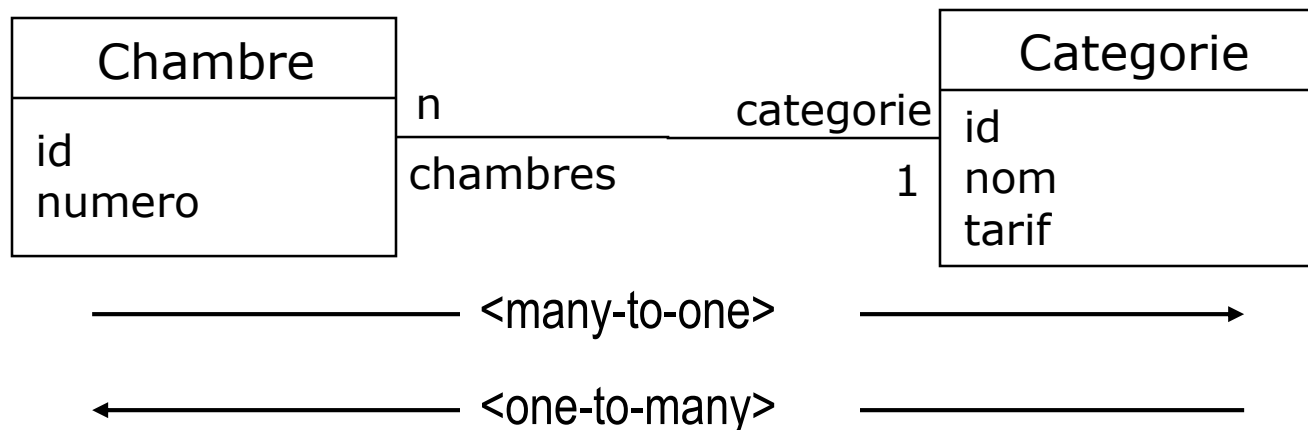
## + Exemple d'objets

- Une **Chambre** a une **Categorie**
- Une **Catégorie** regroupe plusieurs **Chambres**



## + Exemple de requêtes

- Les **Chambres** ayant une **Categorie** dont le tarif est < 85 euros
- Les **Categories** des **Chambres** dont le numéro commence par 1



**+ Les Chambres ayant une Categorie dont le tarif est < 85 euros**

**+ Avec HQL**

```
FROM Chambre ch WHERE ch.categorie.tarif < 85
```

**+ Avec l'API Criteria**

```
session.createCriteria(Chambre.class)
.add( Restrictions.lt ("categorie.tarif", new Float(85))).list();
```

# JOINTURE EXPLICITE

## + Se caractérise par l'utilisation de 'join'

```
SELECT ch,cat FROM Chambre ch JOIN ch.categorie cat WHERE cat.tarif < 85
```

## + Certaines fonctionnalités nécessitent une jointure explicite

- Renvoi de couples d'objets
  - + Ex : une chambre avec la catégorie correspondante
- Initialisation immédiate
  - + JOIN FETCH
  - + EX: charger un hôtel avec sa collection de chambres
- Jointure ouverte (droite ou gauche)

## + Exemple de récupération de couples d'objets

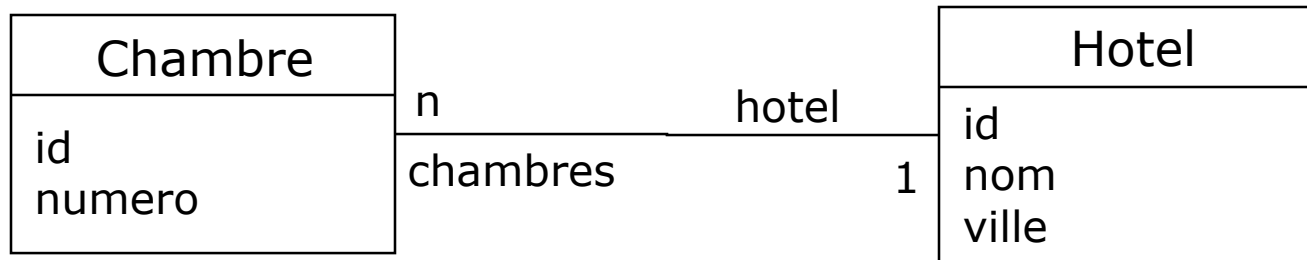
### ▀ N-tuples

```
List result
= session.createQuery("FROM Hotel h JOIN h.chambres ch WHERE
    ch.numero like '1%'").list();

for (Iterator ite=result.iterator(); ite.hasNext(); ) {
    Object[] hotelEtChambre = (Object[]) ite.next();
    Hotel hotel = (Hotel) hotelEtChambre[0];
    Chambre chambre = (Chambre) hotelEtChambre[1];
}
```

### + Hibernate utilise le lazy-loading par défaut

- ▀ Les objets associés sont chargés au dernier moment
  - + lors de la récupération d'un hôtel, les chambres associées ne sont pas chargées.
  - + Elles seront chargées au premier appel de **monHotel.getChambres()**





### + JOIN FETCH

- ▶ Permet de spécifier qu'une relation doit être immédiatement chargée

```
Query query = session.createQuery(  
    "from Hotel h join fetch h.chambres");
```

- ▶ Par défaut les résultats sont renvoyés sous forme de n-tuples
  - + Ex : couples Hôtel-Chambre
- ▶ Il est possible de renvoyer uniquement les objets "parents de l'association"
  - + Ex : uniquement les hôtels
  - + Utilisation de **group by**

```
Query query = session.createQuery(  
    "from Hotel h join fetch h.chambres group by h");
```

# JOINTURE OUVERTE

## + Jointure fermée (standard)

- Les hôtels et leurs chambres

```
FROM Hotel h JOIN h.chambres ch
```

## + Jointure ouverte à gauche

- Les hôtels et leurs chambres + Les hôtels qui n'ont pas de chambre

```
FROM Hotel h LEFT JOIN h.chambres ch
```

## + Jointure ouverte à droite

- Les hôtels et leurs chambres + les chambres qui n'ont pas d'hôtel

```
FROM Hotel h RIGHT JOIN h.chambres ch
```



Comme pour le JOIN FETCH, les résultats viennent sous forme de n-tuples, mais il est possible d'appliquer une clause 'group by'

## + Chaque ligne de résultat peut contenir

### ➤ 1 entité

```
from Categorie as cat where cat.nom = ?
```

### ➤ N entités

```
from Categorie as cat join cat.chambres as ch
```

### ➤ Un ensemble de données scalaires

```
select cat.tarif, ch.numero from Categorie cat join cat.chambres ch
```

### ➤ Un mélange de données scalaires et d'entités

```
select ch, cat.tarif from Chambre ch join ch.categorie cat
```

TP

+ TP 07

# ASPECTS AVANCÉS



- 
- + Cycle de vie et état des objets**
  - + Transactions**
  - + Les caches**
  - + Lazy loading**
  - + Intégration dans une application Web**
  - + Best practices**

- + 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

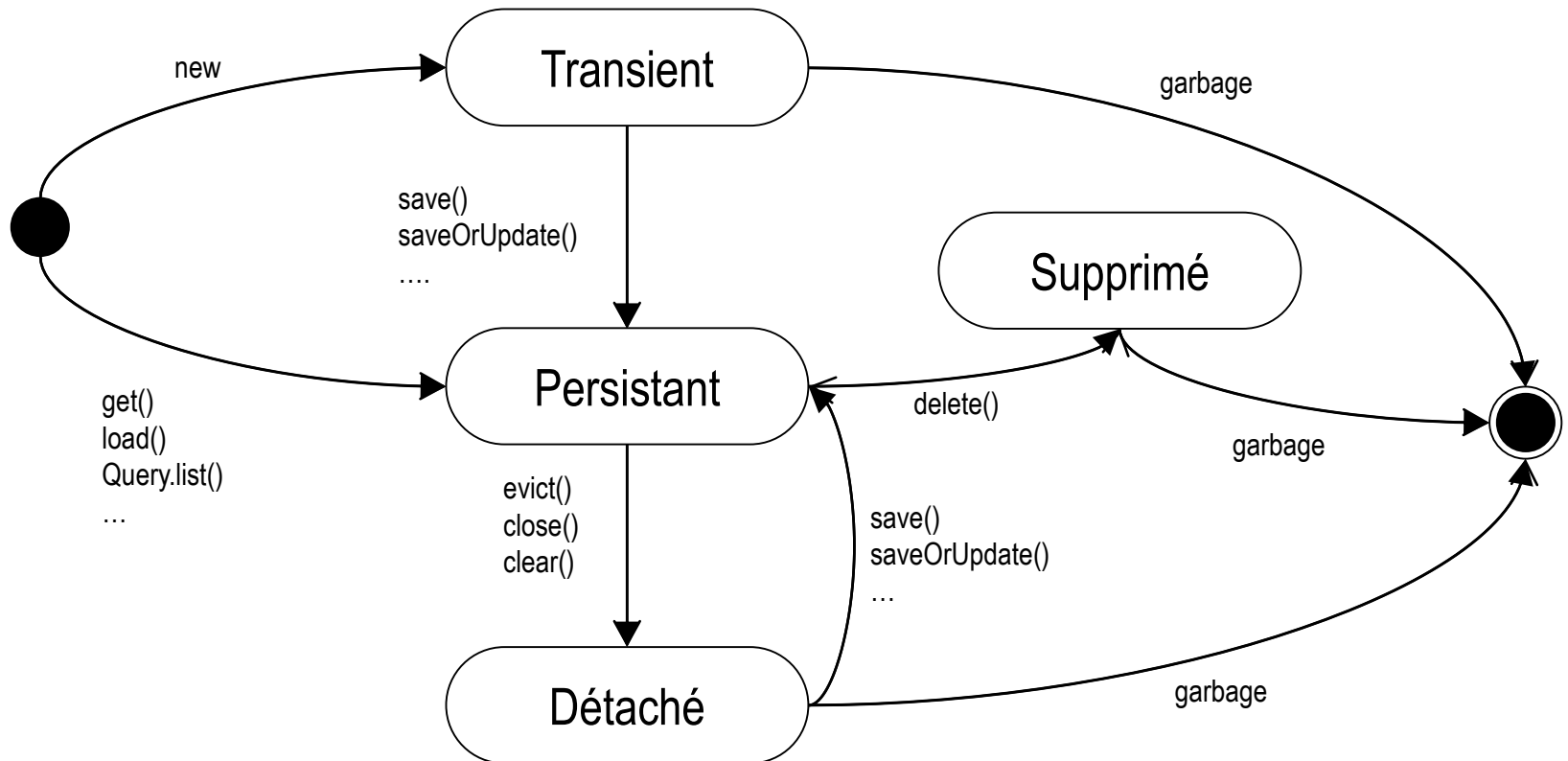
+ **org.hibernate.Session API représente le contexte de persistance des objets**

+ **Hibernate définit 4 états pour les objets**

- **New ou Transient** : l'objet juste instancié , et pas attaché au contexte de persistance, pas de représentation correspondante en base de données
- **Managed ou Persistent** : a une identité , rattaché à un contexte de persistance
- **Removed** : a une identité, attaché au contexte de persistance, programmé pour être supprimé en base
- **Detached** : a une identité mais n'est plus associé au contexte de persistance, le contexte de persistance a été fermée



# ETATS DES OBJETS



## + Garantie d'Hibernate :

- ▀ Objets dont l'état managed/persistent : synchronisés avec le SGBD
- ▀ Objets détachés : aucune garantie

## + Les objets détachés peuvent être

- ▀ Ré-attachés « SaveOrUpdate »
- ▀ Fusionnés (« merge »)

## + La gestion des états est faite via le contexte de persistance

- ▀ Session Hibernate

- + La session Hibernate « contient » le contexte de persistance**
- + Equivalent à un cache d'entités totalement gérées**
- + Le contexte de persistance**
  - Est propre à une unité de travail ( $\approx$  transaction)
  - Vérifie si les objets persistants ont été modifiés
    - + « Automatic dirty checking »
  - Effectue les requêtes « en fond »
    - + « transactionnal write-behind »
  - Cache de premier niveau
  - 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 conflit 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 de `Session`

- + `save ()` : attachement, rend persistant, programmé pour un **INSERT**
- + `saveOrUpdate()` : attachement, crée ou mis à jour (**INSERT** ou **update**)
- + `get ()` : attachement, récupération dans la session ou dans le **SGBD**
- + `update ()` : attachement, la session va « suivre » l'objet pour savoir si un **UPDATE** est nécessaire
- + `delete ()` : 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 `session.flush()`

**+ Réglages possible, avec `session.setFlushMode()`**

- AUTO : cf. ci-dessus
- COMMIT : seulement lors du commit de la transaction
- MANUAL : appel explicite seulement

## + 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



- 
- + **Cycle de vie et état des objets**
  - + **Transactions**
  - + **Les caches**
  - + **Lazy loading**
  - + **Intégration dans une application Web**
  - + **Best practices**

## **+ En JDBC, les accès en base sont effectués en auto-commit par défaut**

- Transactions simplistes
- Il est possible de désactiver le mode auto-commit
  - + `connection.setAutoCommit(false)`

## **+ Hibernate efface ce comportement**

- Autocommit désactivé lors de l'ouverture explicite d'une transaction
- Toute transaction qui ne se termine pas par un "commit" passe en mode "rollback"

# LES TRANSACTIONS HIBERNATE (1/3)

- + Les transactions sont déclarées sur une session
- + Il est possible d'avoir plusieurs transactions dans une même session

```
Session session = ...;
Transaction transaction = null;
try {
    transaction = session.beginTransaction();

    // do some work

    ...
    transaction.commit();
} catch (Exception e) {
    if (transaction != null)
        transaction.rollback();
    throw e;
} finally {
    session.close();
}
```

## LES TRANSACTIONS HIBERNATE (2/3)

---

- + Hibernate propose plusieurs politiques de gestion des transactions**

- ▀ JDBC (défaut)
- ▀ JTA

- + L'API Transaction de Hibernate encapsule ces notions**

- ▀ Pas de dépendance directe

- + Configurable avec la propriété :**

`hibernate.transaction.factory_class`

- + JTA sert pour les environnements managés (ex. : CMT)**

## + Points importants

- Implémenter le pattern DAO
- Les DAO ont une dépendance vers Hibernate
- Le contexte (Session Hibernate) n'est pas géré par le DAO
  - + « session-per-request » et « une session – une transaction BD »
- Les DAO ne gèrent
  - + Ni l'ouverture/fermeture de la session
  - + Ni les transactions

## + Comment faire ?

- Solution « maison »
- Support d'autres frameworks (ex. : Spring)

### **+ L'approche optimiste est préconisée**

- plus performante
- Approche par défaut
- Lock-optimiste avec résolution par versions

### **+ Hibernate propose un mécanisme de réconciliation en cas de conflit entre deux transactions**

- Système de gestion des versions

### + Déclaration xml

```
<hibernate-mapping package="com.bankonet.model">
<class name="Chambre" table="Chambre">
  <id name="id" column="ID" type="int">
    <generator class="native" />
  </id>
  <version name="version" column="version" type="integer"/>
  <property name="nom" column="nom" type="string" />
  ...
</class>
</hibernate-mapping>
```

### + Déclaration Java

```
public class Client {  
    ...  
    @Version private int version;  
    public int getVersion() {return version;}  
    public void setVersion(int version) {this.version = version;}  
}
```



### + En cas de conflit, une StaleStateException est lancée

- ▀ Le premier commit s'effectue normalement
- ▀ Lors du 2ème 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 par Hibernate :

```
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
```

# APPROCHE PESSIMISTE

---

## + Il est possible d'obtenir un lock sur :

- ▀ Toute une requête

```
myQuery.setLockMode(..., LockMode.READ);
```

- ▀ Une instance

```
session.get(Chambre.class, new Integer(5), LockMode.PESSIMISTIC_WRITE);
```

- ▀ Toutes les instances d'une classe

```
session.lock(maChambre, LockMode.PESSIMISTIC_WRITE);
```

## + Requête générée : **SELECT FOR UPDATE...**

## + Transactions justifiant plusieurs aller-retour client-serveur

- Pour une application Web, la session hibernate est généralement stockée dans la session http
- La session hibernate ne doit pas être fermée
  - + Problème : si la session n'est pas fermée, la connexion reste monopolisée
  - + Utilisation de `session.disconnect()` pour rendre la connexion
  - + Utilisation de `session.reconnect()` pour raccrocher une connexion à la session

**+ Hibernate peut être utilisé conjointement avec un gestionnaire de transactions**

- ▀ Spring
- ▀ EJB 3.0

**+ Ils permettent la gestion des transactions en dehors du code Java**

- ▀ Annotations (depuis Java 5) ou paramétrage xml
- ▀ Dans ce cas, les instructions `session.beginTransaction()` et `tx.commit()` **ne sont plus nécessaires**

## TRANSACTIONS DÉCLARATIVES (2/2)

---

### + Exemple :

- ▀ Déclaration d'une transaction avec les EJB 3.0 :

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Ticket createHotel(Hotel hotel) {    ...    }
```

- ▀ Déclaration d'une transaction avec Spring

```
@Transactional(propagation=Propagation.REQUIRED)
public Ticket createHotel(Hotel hotel) {    ...    }
```

- ▀ REQUIRED : indique au conteneur qu'il doit créer une nouvelle transaction (si elle n'existe pas encore)

- 
- + **Cycle de vie et état des objets**
  - + **Transactions**
  - + **Les caches**
  - + **Lazy loading**
  - + **Intégration dans une application Web**
  - + **Best practices**

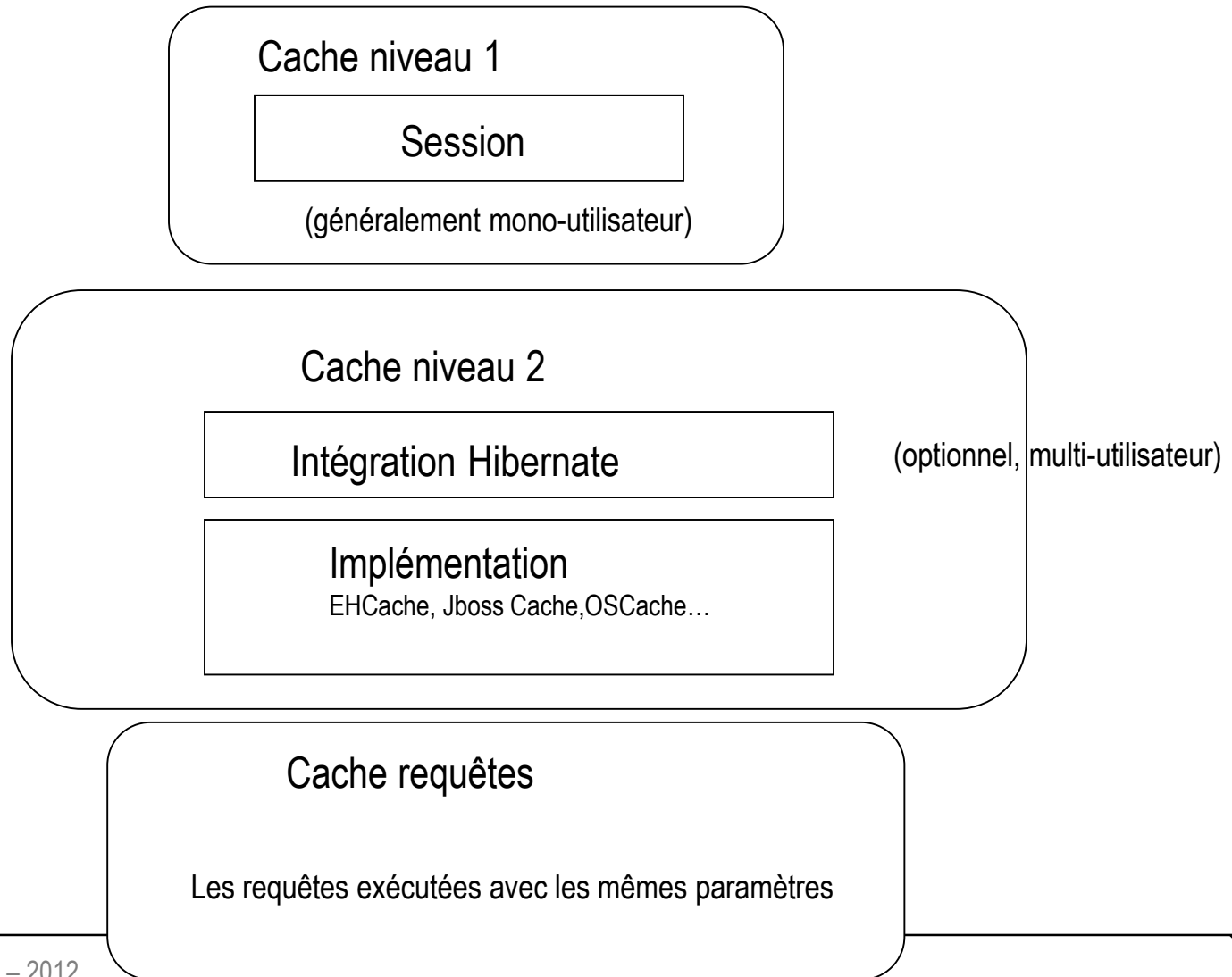
## + Les caches permettent de diminuer le nombre de requêtes en base de données

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

## + Intégrité des données

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

# PRÉSENTATION (2/2)





## + **Propre à chaque session Hibernate**

## + **Toujours activé (par défaut)**

- Essentiel au fonctionnement interne d'Hibernate
- Il est impossible de le désactiver
- Il est possible de le vider
  - + `session.clear(...)`, `session.evict(...)`

# RÉCUPÉRATION DE DONNÉES

- ① 

```
Chambre chambre1 = (Chambre) session.get(Chambre.class, new Integer(501));  
Chambre chambre2 = (Chambre) session.get(Chambre.class, new Integer(501));
```

 1 requête
- ② 

```
Query query = session.createQuery("from Chambre c where c.id like :expr1");  
query.setInteger("expr1", 501);  
Chambre chambre1 = (Chambre) query.uniqueResult();  
Chambre chambre2 = (Chambre) query.uniqueResult();
```

 2 requêtes
- ③ 

```
Query query = session.createQuery("from Chambre c where c.id like :expr1");  
query.setInteger("expr1", 501);  
Chambre chambre1 = (Chambre) query.uniqueResult();  
Chambre chambre2 = (Chambre) session.get(Chambre.class, new Integer(501));
```

 1 requête



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

1

```
Transaction transaction = session.beginTransaction();
Chambre chambrel = (Chambre) session.get(Chambre.class, new Integer(501));
chambrel.setNom("nouveauNom");
chambrel.setCouleur("BLEU");
transaction.commit();
```

2 requêtes

2

```
Transaction transaction = session.beginTransaction();
Chambre chambrel = (Chambre) session.get(Chambre.class, new Integer(501));
chambrel.setNom("nouveauNom");
session.flush();
chambrel.setCouleur("BLEU");
transaction.commit();
```

3 requêtes

# ENLEVER UN OU PLUSIEURS ÉLÉMENT(S) DU CACHE

1

```
Chambre chambre1 = (Chambre) session.get(Chambre.class, new Integer(501));  
Chambre chambre2 = (Chambre) session.get(Chambre.class, new Integer(501));  
session.evict(chambre2);
```

1 élément  
enlevé

2

```
Chambre chambre1 = (Chambre) session.get(Chambre.class, new Integer(501));  
Chambre chambre2 = (Chambre) session.get(Chambre.class, new Integer(501));  
session.clear();
```

Le cache  
est vidé

3

```
Transaction transaction = session.beginTransaction();  
Chambre chambre1 = (Chambre) session.get(Chambre.class, new Integer(501));  
session.evict(chambre1);  
chambre1.setNom("nouveauNom");  
transaction.commit();
```

Que se  
passe-t-il ?

### **+ Cache de niveau 1 : transactionnel lié à la session hibernate**

- ▀ Les objets cachés ne sont visibles que pour une seule transaction
- ▀ Ne peut pas être désactivé

### **+ Cache de niveau 2 : JVM ou cluster lié à la session factory**

### **+ Cache de requêtes : QueryCache**

### **+ 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.region.factory_class`  
    ➤ Indique la classe du provider (classe Hibernate)

**+ Les providers disponibles**

- HashTable
- EhCache
- OsCache
- SwarmCache
- Jboss cache 1.x
- JbossCache 2.x

# LE CACHE DE NIVEAU 2 – EHCACHE (1/2)

## Implémentation par défaut : ehcache

### Exemple :

*ehcache.xml*

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

*Chambre.hbm.xml*

```
<hibernate-mapping package="com.hotello">
  <class name="Chambre" table="Chambre"
    dynamic-update=false
    dynamic-insert=false>

    <cache usage="read-only"/>
    ...
  </class>
</hibernate-mapping>
```

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Hotel implements Serializable {
```

```
<property name="hibernate.cache.region.factory_class">
  org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>

<property name="hibernate.cache.use_second_level_cache">true</property>
```

*hibernate.cfg.xml*

## LE CACHE DE NIVEAU 2 – EHCACHE (2/2)

### + Le cache de requête permet de mémoriser le résultat des précédentes requêtes

#### ➤ Criteria ou HQL

```
<ehcache>
  <cache name="com.hotello.Chambre"
    maxEntriesLocalMap="1000"
    eternal="false"
    overFlowToDisk="true"
    timeToLiveSeconds="400" />

  <cache name="com.hotello.Hotel.chambres"
    maxEntriesLocalMap="10000"
    eternal="false"
    overFlowToDisk="true"
    timeToLiveSeconds="400" />

  <cache name="query.readChambreByName"
    maxEntriesLocalMap="5"
    eternal="false"
    overFlowToDisk="true"
    timeToLiveSeconds="400" />
</ehcache>
```

```
Query query = session.createQuery("from
Chambre c where c.identifiant=:expr1");
query.setInteger("expr1", 1);
query.setCacheable(true);
Query.setCacheRegion("chambre")
```

*Appel Java*

*ehcache.xml*

```
<property name="hibernate.cache.region.factory_class">
org.hibernate.cache.ehcache.EhCacheRegionFactory /property>

<property name="hibernate.cache.use_second_level_cache">true</property>
```



### **+ 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

# LE CACHE DE NIVEAU 2 – QUELS TYPES DE DONNÉES ?

---

## + 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

# LE CACHE DE NIVEAU 2 – MAPPING

## + Directement dans le mapping de la classe

- ▀ Balise <cache> dans <table> ou dans <set> (ou <list>, ...)

```
<class name="eg.Cat" .... >
  <cache usage="read-write"/>
  ....
  <set name="kittens" ... >
    <cache usage="read-write"/>
    ....
  </set>
</class>
```

## + Ou dans hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    ...
    <class-cache usage="read-only" class="eg.Cat"/>
    <collection-cache usage="read-only" collection="eg.Cat.kittens" />
    ...
  </session-factory>
</hibernate-configuration>
```

## + Déclaration de la session factory

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  <property name="annotatedClasses">
    <list>
      <value>com.sqli.bankonet.domaine.Compte</value>
    </list>
  </property>

  <property name="dataSource" ref="dataSource" />
  <property name="hibernateProperties" ref="hibernateProperties" />
</bean>
```

### + Propriétés

```
<bean id="hibernateProperties"
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="properties">
    <props>
      ...
      <prop key="hibernate.cache.region.factory_class">
        org.hibernate.cache.ehcache.EhCacheRegionFactory
      </prop>
      <prop key="hibernate.cache.use_second_level_cache">true</prop>
      <prop key="hibernate.cache.use_query_cache">true</prop>
      ...
    </props>
  </property>
</bean>
```

# LE CACHE DE NIVEAU 2 – CONCURRENCE

---

- + 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**



Les implémentations de cache ne supportent pas tous les niveaux !

## LE CACHE DE NIVEAU 2 – BONNES PRATIQUES

---

### **+ 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**

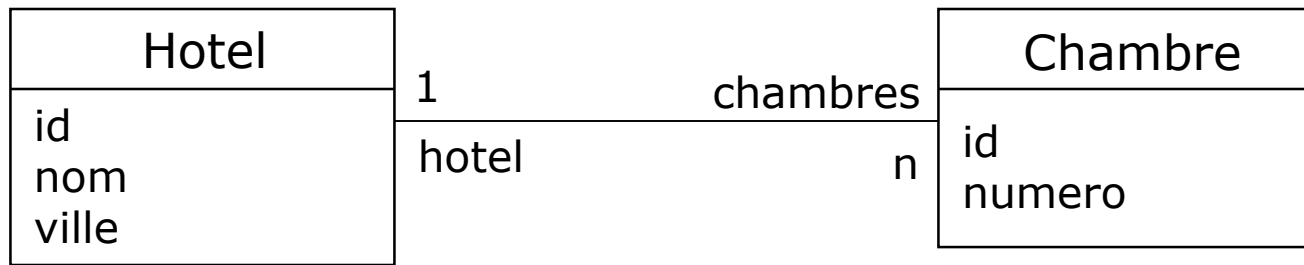
- ▀ Benchmark sans cache niv. 2
- ▀ Benchmarks successifs avec une classe, deux classes... n classes mises en cache

- 
- + **Cycle de vie et état des objets**
  - + **Transactions**
  - + **Les caches**
  - + **Lazy loading**
  - + **Intégration dans une application Web**
  - + **Best practices**



## RELATIONS 1-N ET N-N (1/2)

- + Par défaut, ManyToMany et OneToMany en lazy-loading
- + Par default OneToOne et ManyToOne en Eager-loading



```
Hotel hotel = (Hotel) session.get(Hotel.class, new Integer(501));  
  
Set chambres = hotel.getChambres();  
for (Iterator ite=result.iterator(); ite.hasNext(); ) {  
    Chambre c = (Chambre) ite.next();  
}
```

} 1 requête

} 1 requête

### + Au chargement, Hibernate remplace la collection déclarée par une collection 'interne'

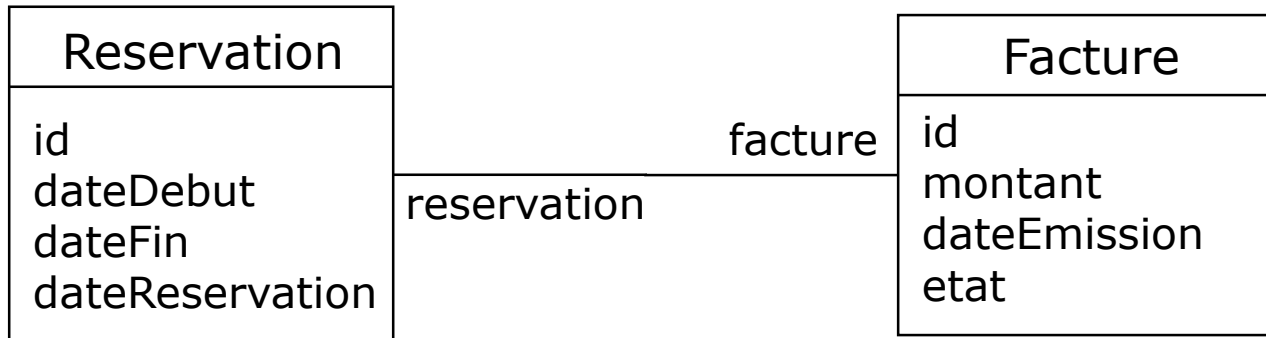
- PersistentSet, PersistentList...
  - + Implémentent les interfaces standards : java.util.Set, java.util.List...
  - + Package org.hibernate.collection
- Mécanisme de Proxy
  - + A la première utilisation, la collection sous-jacente est chargée



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

## RELATION 1-1 ET N-1 (1/3)

**+ Les relations 1-1 ainsi que n-1 sont implémentées en Eager loading par défaut**



```
Reservation res = (Reservation) session.get(Reservation.class, new Integer(501));
```

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

} 1 requête

} 1 requête

## RELATION 1-1 (2/3)

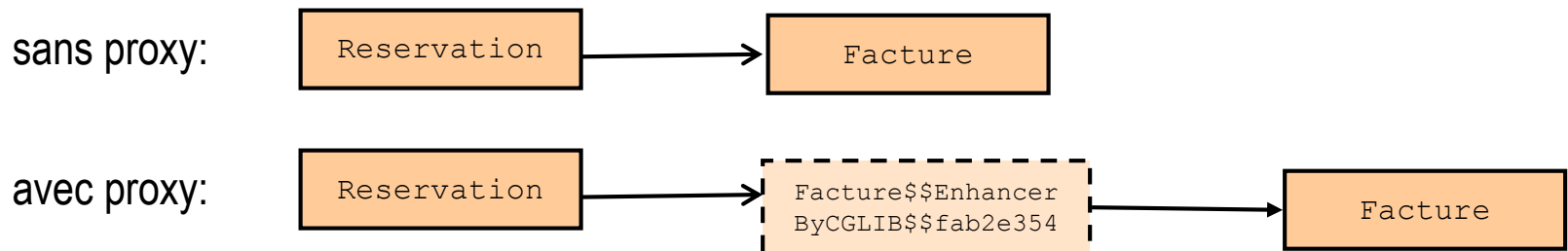
### + Utilisation d'un proxy sur l'objet référencé

- ▀ Classe créée dynamiquement

+ Bibliothèque cglib

### + Problématique : l'utilisation d'un proxy doit être transparente

- ▀ La classe proxy **hérite** de la classe sous-jacente

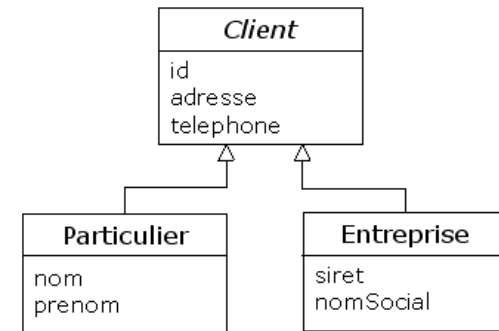


```
Reservation res = (Reservation) session.get(Reservation.class, new Integer(501));  
Facture fact = res.getFacture();
```

## RELATION 1-1 (3/3)

### + Contraintes liées à l'utilisation d'un proxy

- ▀ Attention au polymorphisme
  - + Un proxy sur Client hérite de client
  - + Un proxy sur Client n'hérite pas de Particulier



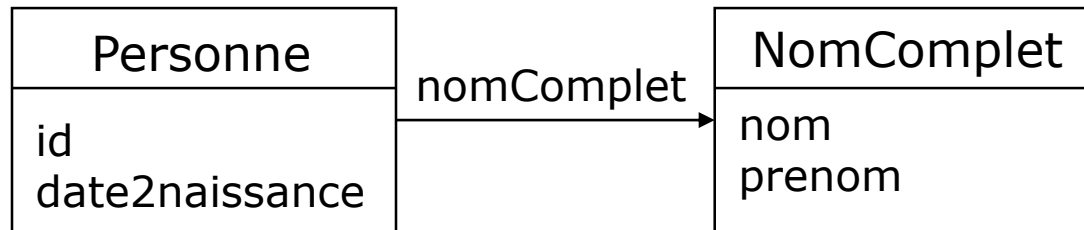
```
Facture fact = (Facture) session.get(Facture.class,
new Integer(501));
```

```
Client client = fact.getClient();
Particulier part = (Particulier) client;
```

← ClassCastException !!!

# CHARGEMENT DE COMPOSANTS

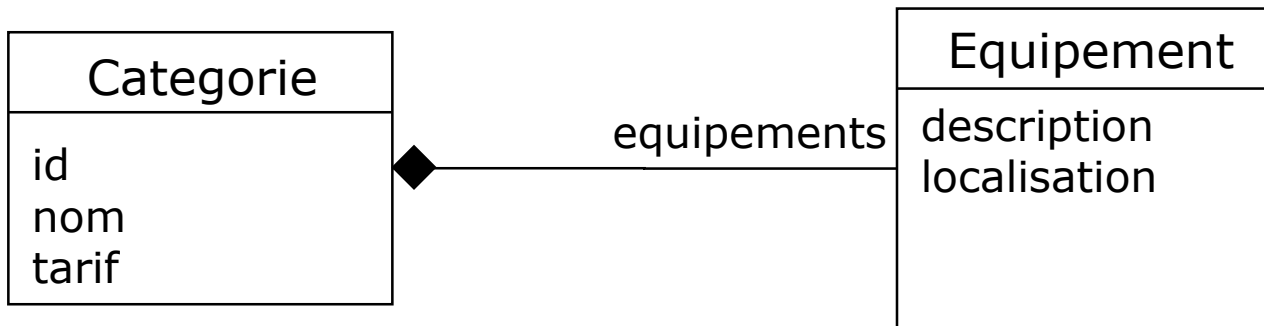
**+ Par défaut, les composants ne sont pas chargés par des proxies**



```
Personne personne = (Personne) session.get(Personne.class, new Integer(501));
NomComplet nom = personne.getNomComplet();
```

1 requête

**+ Par défaut, les collections de valeurs sont chargées par des proxies**



```
Categorie categorie = (Categorie) session.get(Categorie.class,
new Integer(501));
Set equipements = categorie.getEquipement();
for (Iterator ite=equipements.iterator(); ite.hasNext(); ) {
    Equipement e = (Equipement) ite.next();
}
```

2 requêtes

## FORCER LE CHARGEMENT (1/2)

### + Problématique : charger le contenu d'une collection avant la fermeture de la session

- Appel à `Hibernate.initialize(...)`
- S'applique aux relations 1-1, 1-n, n-n

```
Hotel hotel = (Hotel) session.get(Hotel.class, new Integer(501));  
Hibernate.initialize(hotel.getChambres());
```

} 2 requêtes

```
Reservation res = (Reservation) session.get(Reservation.class, new  
Integer(501));  
Hibernate.initialize(res.getFacture());
```

} 2 requêtes



## FORCER LE CHARGEMENT (2/2)

### + Dans le mapping :

- ▀ Pour tout type de relation. Ex :

```
<class name="Hotel">
  ...
  <set name="chambres" lazy="false">
    ...
  </set>
</class>
```

```
public class Hotel{
  ...
  @OneToMany(fetch=FetchType.LAZY)
  private Set chambre;

  ...
}
```

### + Dans une requête

- ▀ Utilisation de JOIN FETCH

```
Query query = session.createQuery("from Hotel h join fetch h.chambres");
```

## + Résumé, comportement par défaut

- ▀ « lazy select fetching » pour les collections
- ▀ « lazy proxy fetching » pour les associations unitaires

## + Comportement adapté pour la plupart des cas

## + Réglages possibles avec Hibernate :

- ▀ Quand
- ▀ Comment (SQL utilisé)

## + Déclaration des réglages

- ▀ Dans le mapping
- ▀ Surcharge possible (HQL ou Criteria)

## + **Join fetching : même SELECT, avec OUTER JOIN**

- ▀ Récupération de l'instance associé ou la collection dans le même select

## + **Select fetching : un 2<sup>ème</sup> SELECT**

- ▀ Pour les entités ou les collections

## + **Subselect fetching : une requête imbriquée**

- ▀ Pour les collections

## + **Batch fetching : récupération dans un SELECT, en précisant une liste de PK ou de FK**

# STRATÉGIES DE RÉCUPÉRATION - QUAND

---

- + **Immediate fetching** : association récupérée immédiatement lors du chargement du père
- + **Lazy collection fetching** : collection récupérée quand on y accède
  - ▀ Comportement par défaut
- + **Extra-lazy collection fetching** : récupération individuelle des éléments de la collection
  - ▀ Pour les très grandes collections
- + **Proxy fetching** : association récupérée quand on accède à autre chose que son ID
- + **No-proxy fetching** : association récupérée quand on y accède
- + **Lazy attribute fetching** : attribut récupéré quand on y accède

## + Réglages possibles dans le mapping :

```
<set name="permissions" lazy="false" fetch="join" >
  <key column="userId"/>
  <one-to-many class="Permission"/>
</set>

<many-to-one name="mother" class="Cat" lazy="false" fetch="join" />
```

## + Le réglage du fetch dans le mapping influe sur :

- ▀ get et load
- ▀ Lors de la navigation
- ▀ Requête Criteria

## + Dans une requête

### ▀ Utilisation de JOIN FETCH

```
Query query = session.createQuery("from Hotel h join fetch h.chambres");
```

## + Avec Criteria

```
List users = session.createCriteria(User.class)
    .setFetchMode("permissions", FetchMode.JOIN)
    .list();
```

- + **Laisser le comportement par défaut dans le mapping**
- + **Utiliser le Join Fetching**
  - ▀ HQL ou Criteria
- + **Eviter Hibernate.initialize() dans la mesure du possible**
- + **load et get utilise le mapping**
  - ▀ Que faire si on veut les associations chargées ?
  - ▀ Solution : utiliser Criteria

```
User user = (User) session.createCriteria(User.class)
    .setFetchMode("permissions", FetchMode.JOIN)
    .add( Restrictions.idEq(userId) )
    .uniqueResult();
```

- 
- + Cycle de vie et état des objets**
  - + Transactions**
  - + Les caches**
  - + Lazy loading**
  - + Intégration dans une application Web**
  - + Best practices**



## + Dans une application Web :

- La configuration Hibernate et la SessionFactory doivent être chargées une seule fois
  - + Généralement au démarrage de l'application
- Il est courant d'automatiser les ouvertures / fermetures de session

## + Démarche usuelle :

- A chaque début de requête utilisateur, une session est ouverte
  - + Éventuellement : ouverture de transaction
- A chaque fin de requête, la session est fermée
  - + Éventuellement : commit de transaction

- 
- + Transactions**
  - + Les caches**
  - + Lazy loading**
  - + Intégration dans une application Web**
  - + Best practices**

## BEST PRACTICES (1/4)

---

- + Utiliser autant que possible des classes de faible granularité**
  - ▀ Si la table contient beaucoup de colonnes, utiliser les composants
  - ▀ Le modèle Objet doit toujours rester intuitif
  
- + Privilégier les identifiants de synthèse sur les classes persistantes**
  - ▀ Identifiant métier différent de l'identifiant technique
  
- + Privilégies les namedQuery**
  - ▀ HQL ou Criteria

## BEST PRACTICES (2/4)

---

- + En HQL, utiliser systématiquement des variables "bindées"**
  - Gain de performances important
- + Eviter de gérer les connexions JDBC soi-même**
- + N'effectuer des requêtes natives à la main qu'en dernier recours**
  - Syntaxe spécifique au SGBD

- + Réfléchir à l'utilisation de `saveOrUpdate()`**
- + Considérer les exceptions comme 'irrécupérables' par défaut.**
  - Stratégie usuelle : gérer les exceptions au point le plus haut
- + Utiliser le lazy loading pour les associations d'entités**
  - Relations 1-n, n-n, 1-1

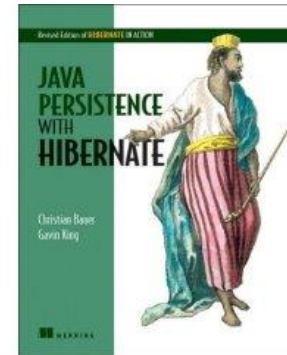
- + Introduire un degré d'abstraction entre le code métier et Hibernate**
- + Implémenter equals() et hashCode() avec des identifiants métiers**
  - ▀ Indispensable pour l'utilisation de collections n'acceptant pas les doublons (HashSet, TreeSet...)
- + Bien réfléchir au type d'association entre entités**

CONCLUSION



## + Hibernate in Action (Manning)

- Gavin King et Christian Bauer
- Écrit par les créateurs d'Hibernate
- Disponible en anglais uniquement



## + Java persistence with hibernate (Eyrolles)

- Anthony Patricio
- Disponible en français

