La correspondance Object-relationnel : Introduction

Hala Skaf-Molli
Hala.Skaf@univ-nantes.fr
http://pagesperso.lina.univ-nantes.fr/~skaf-h

References

- Java Persistence with Hibernate, Christian Bauer, Gavin King, 2007

Persistance des objets

- A la **fin d'une session** d'utilisation d'une application orientée objet **toutes les données des objets existant dans la mémoire vive de l'ordinateur sont perdues**
- **Rendre persistant un objet c'est sauvegarder ses données sur un support non volatile** de telle sorte qu'un objet identique à cet objet pourra être recréé lors d'une session ultérieure

Persistance des objets

- Une application écrite avec un langage objet (JAVA) qui utilise une base de données pour la persistance des données des objets
  - Bases de données orienté objets (SGBDOO)
  - Bases de données objet-relationnelles (SGBDOR)
  - **Bases de données relationnelles (SGBDR)**

Pourquoi pas un SGBD Objet ?

- Principale raison ‘legacy’: de très nombreuses bases relationnelles en fonctionnement
- Manque de normalisation pour les SGBDOO ; trop de solutions propriétaires
- Peu d'informaticiens formés aux SGBDOO

Pourquoi les BD relationnelles ?

- Position dominante
- Une théorie solide et des normes reconnues
- Grande facilité et efficacité pour effectuer des recherches complexes dans des grandes bases de données
- Fonctionnalités de SGBD
  - Cohérence de données: contraintes d'intégrité
  - Gérer les accès concurrents, transactions
  - Partage de données

L'objet et le relationnel sont 2 paradigmes bien différents

- Faire la correspondance entre les données modélisées par un modèle objet et par un modèle relationnel n'est pas simple
- Le modèle relationnel est moins riche que le modèle objet
  - Pas d'héritage,
  - Ni de collections (pas d'attributs multi-valués)

Problèmes du passage Relationnel ⟷ Objet

- Paradigmes *mismatch*
  - Problème de granularité
  - Problème de l'identité
  - Problème d'héritage
  - Problèmes avec les associations

Problème de granularité (1)

- Une classe => une table ??
- Les instances de certaines classes peuvent

être sauvegardées dans la même table qu'une autre classe

- Ces instances sont appelées des objets insérés (embedded) et ne nécessitent pas d'identificateur « clé primaire »

Problème de granularité (2)

Online e-commerce application:

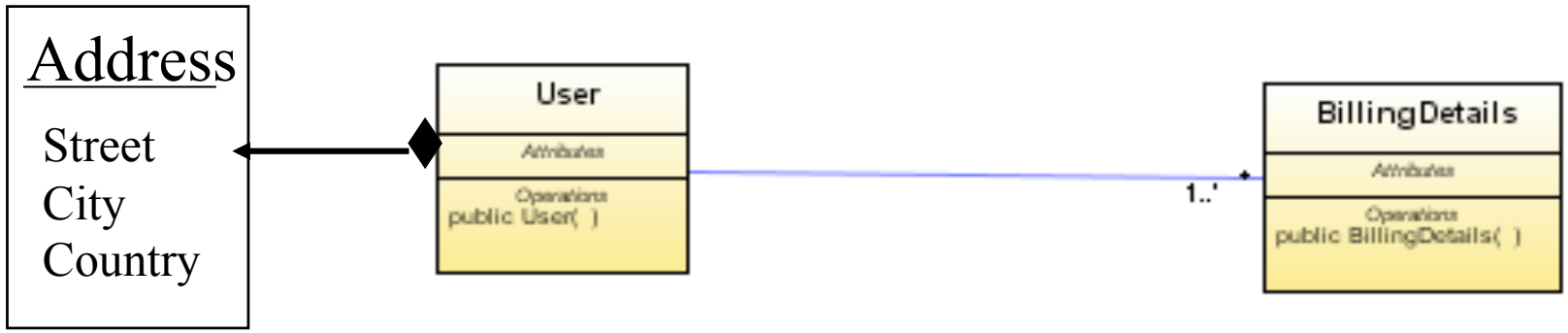

```
public class User {
private String username;
private String name;
private String address;
private set billingDetails;
// Méthodes }
```

```
public class BillingDetails {
private String accountNumber;
private String accountName;
private String accounType;
private User user;
// Méthodes }
```

```
Create table Users (
Username varchar(15) primary key
Name varchar(50) not null,
Address varchar(100)
)
```

```
Create table BillingDetails(
acountNumber varchar(15)  primary key,
accountName varchar(50) not null,
accounType varchar(2) not null,
Username varchar(15) references Users(Username));
```

Problème de granularité (3)

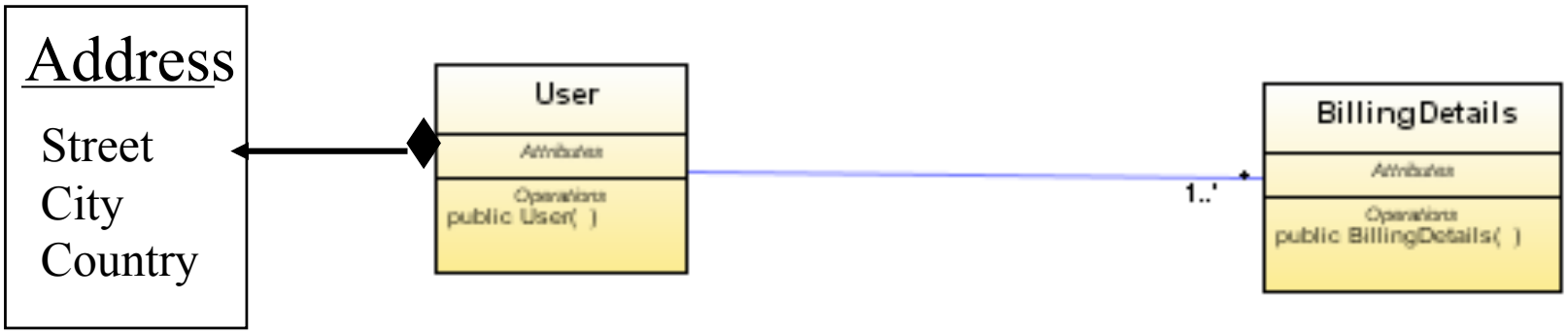


public class Address {
private String street,
private String city,
Private String country
}

public class User {
private String username,
private String name,
Private Address address,
private set billingDetails,..

public class BillingDetails {
private String accountNumber,
private String accountName,
private String accounType
private User user,
// Méthodes }

Problème de granularité (4)

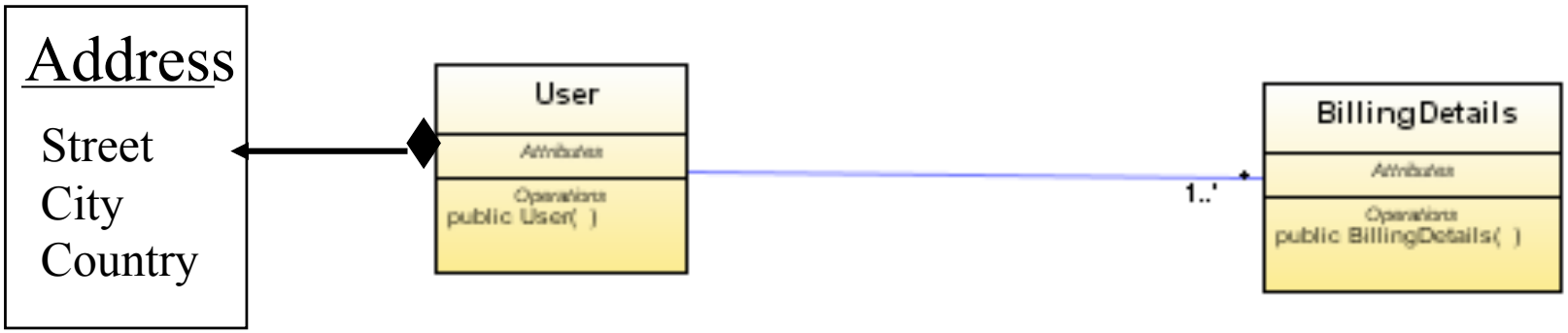


Solution 1:

Create table Users (
Username varchar(15) primary key;
Name varchar(50) not null,
Address_Street varchar(100),
Address_City varchar(30),
Address_Country varchar(30));

Create table BillingDetails (
acountNumber varchar(15) primary key;
accountName varchar(50) not null,
accounType varchar(2) not null,
Username varchar(15) references Users));

Problème de granularité (4)



Solution 2: use user-defined-type of SQL (UDT)

Create type address_Type as object (Street Varchar2(200), City ….)

Create table Users (
Username varchar(15) primary key;
Name varchar(50) not null,
Address adress_Type)

Solution 3: définir une table Address

Problème d'identité (1)

- Dans le monde Objet:
  - Tout objet est identifiable dans les langages objets (adresse de l'emplacement mémoire)
  - 2 objets distincts peuvent avoir exactement les mêmes valeurs pour leurs propriétés
  - En Java, 'sameness':
    - Object identity (a==b a et b référence la même adresse mémoire)
    - Equals (equality by value)
- Dans le monde relationnel
  - Seules les valeurs des colonnes peuvent servir à identifier une ligne
  - Si 2 lignes ont les mêmes valeurs, il est impossible de les différencier
  - Dans les schémas relationnels toute table devrait donc comporter une clé primaire pour identifier une ligne parmi toutes les autres lignes de la même table

Problème d'identité (2)

- Pour effectuer la correspondance objet relationnel, il peut être nécessaire d'ajouter un identificateur à un objet
- Cet identificateur peut être la clé primaire de la ligne de la base de données qui correspond à l'objet

Éviter les identificateurs significatifs

Create table Users (**Username varchar(15) primary key,** …..)

Changer Username, il est référencé dans la table BillingDetails..

Solution: ajouter un identificateur artificiel (Surrogate keys, pas de sens pour l'utilisateur)

Create table Users (

UserID int primary key,

Username varchar(15),

…..)

Create table BillingDetails (

BillingDetailID int primary key,

acountNumber varchar(15) ;

accountName varchar(50) not null,

accounType varchar(2) not null,

UserID bigint references Users));

Les clés **UserID**, **BillingDetailID** peuvent être générés par le système …

Problème d'identité (3)

- Une même ligne de la base de données ne doit pas être représentée par plusieurs objets en mémoire centrale (ou alors le code doit le savoir et en tenir compte)
- Si elle n'est pas gérée, cette situation peut conduire à des duplications de données
  - **`O1`** objet de la classe User avec UserID=1
  - On arrive à trouver le même utilisateur en navigant à partir de la table BillingDetails
  - Une erreur serait de créer en mémoire centrale un autre objet **`O2`** indépendant du premier objet **`O1`** déjà créé

Problème d'identité (4)

- La gestion des identités a un impact sur la gestion de la concurrence et de la cache
  - En mémoire, un objet « cache » répertorie toutes les objets créés et conserve l'identité qu'ils ont dans la base (la clé primaire)
  - Lors d'une navigation ou d'une recherche dans la base, le cache intervient
    - Si l'objet est déjà en mémoire le cache le fournit, sinon, l'objet est créé avec les données récupérées dans la base de données

Problèmes des associations (1)

- Dans le code objet une association peut être représentée par :
  - une variable d'instance représentant « l'autre » objet avec lequel se fait l'association (associations 1:1 ou N:1)
  - une variable d'instance de type collection ou *map* représentant tous les autres objets avec lesquels se fait l'association (associations 1:N ou M:N)
  - une classe « association » (M:N)
- Les associations sont par défaut unidirectionnelle, pour la rendre bidirectionnelle (navigable) il faut la définir dans les deux classes

Problèmes des associations (2)

Unidirectionnelle

```
public class User {
private String username;
private String name;
private String address;

// Méthodes }
```

```
public class BillingDetails {
private String accountNumber;
private String accountName;
private String accounType;
private User user;
// Méthodes }
```

Bidirectionnelle or navigable:

```
public class User {
private String username;
private String name;
private String address;
private set billingDetails;
// Méthodes }
```

```
public class BillingDetails {
private String accountNumber;
private String accountName;
private String accounType;
private User user;
// Méthodes }
```

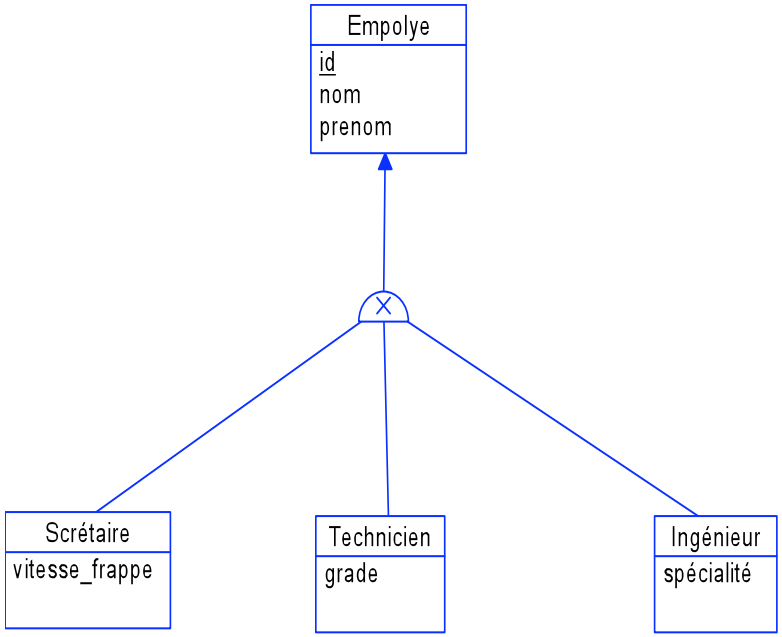
Associations en relationnel

- Association 1-1 (OneToOne)
  - Clé étrangère
- Association 1-n (OneToMany) n-1 (ManyToOne)
  - Clé étrangère
- Association n-m (ManyToMany)
  - Création d'une table pour représenter l’association

L'héritage

- Trois approches:
  - Une table pour chaque sous classe
  - Une table pour chaque classe (is a)
  - Une seule table par hiérarchie de classe
    Discriminateur, valeurs nulles...

Une table pour chaque sous classe



Secrétaire(id, vitesse_frappe, nom, prenom)

Technicien(id, grade, nom, prenom)

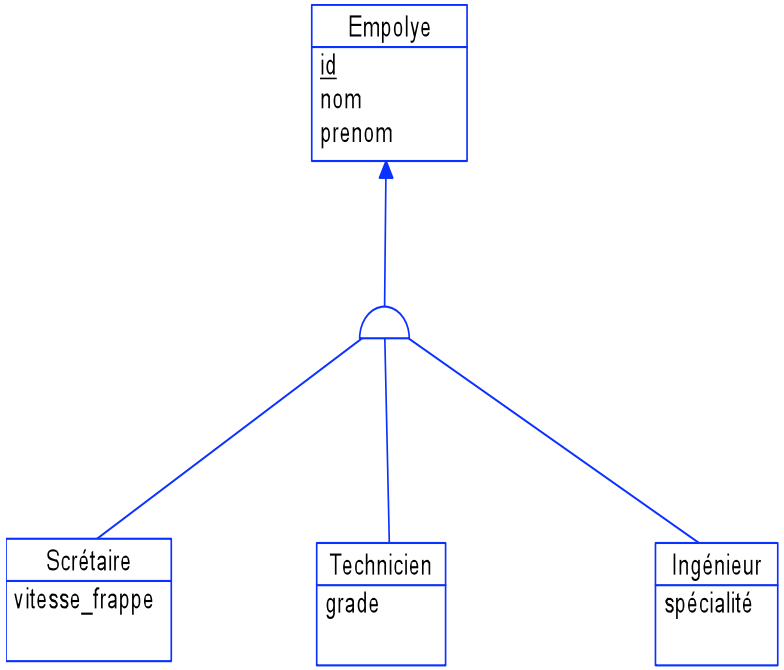
Ingénieur(id, spécialité, nom, prenom)

Problèmes :
Trois reqtes pour trouver Employe !!!

Une table pour chaque classe (is a)

- *Style E/A*: une entité appartient à un réseau de classes liés par *isa* association.
  - créer une relation pour chaque ensemble d'entité
  - les attributs de la relation sont seulement les attributs de ensemble d'entité + clé de la racine..

Exemple



Employe(id, nom, prenom)

Secrétaire(id, vitesse_frappe)

Technicien(id, grade)

Ingénieur(id, spécialité)

Une seule table par hiérarchie de classe

- Utiliser nulls:
  - créer une relation pour la classe racine ou l ’ensemble d’entité racine..
  - les attributs de cette relation sont les attributs de la racine + les attributs de toutes ses sous classes...
  - mettre la valeur nulle pour les attributs n’ayant pas de sens pour une entité donnée...

EmployeHierachie(<u>id</u>, nom, prenom,vitesse_frappe, grade, spécialité)

- La plus efficace mais pas NOT NULL contrainte sur les attributs

Synthèse

- Objet et relationnel sont différents
- Malgré les problèmes le passage Objet relationnel est possible..
- À la main ..
  - Avec JDBC
- Avec Hibernate

À la main ..

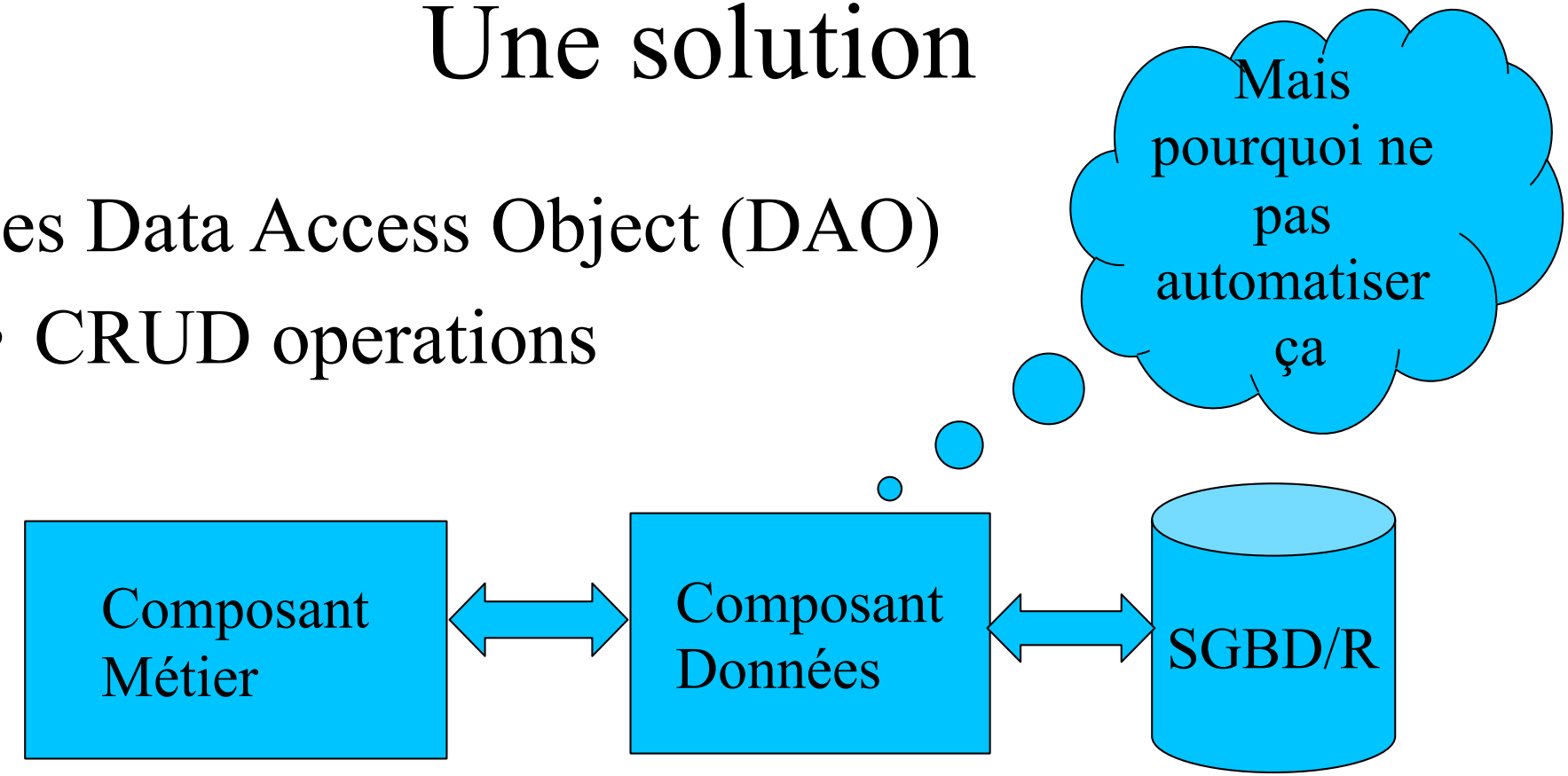
```
public class InsertTable {
  public static void main (String args[]) {
        String url ="jdbc:oracle:thin:@panoramix:1521:depinfo" ;
       Connection con;
       Statement stmt;
    try { // enregistrement du driver
              Class.forName("oracle.jdbc.driver.OracleDriver");
             } catch(ClassNotFoundException e) {
                  System.err.print("ClassNotFoundException");
                  System.err.println(e.getMessage());  }
         try {
           con = DriverManager.getConnection (url, login,
   password);
           stmt = con.createStatement ();
         stmt.executeUpdate("insert into Customer
   values(1,'Titi', 'lala',20)");
           stmt.close();
           con.close();
              } catch (SQLException ex) {
             System.err.println (" SQLException caught: "+
   ex.getMessage());  }}}
```

Les problèmes

- Gestion manuelle de la persistance
- Gestion manuelle des transformations des types
- Gestion des changements
  - Dans les classes
  - Dans le schéma
- Gestion des transactions

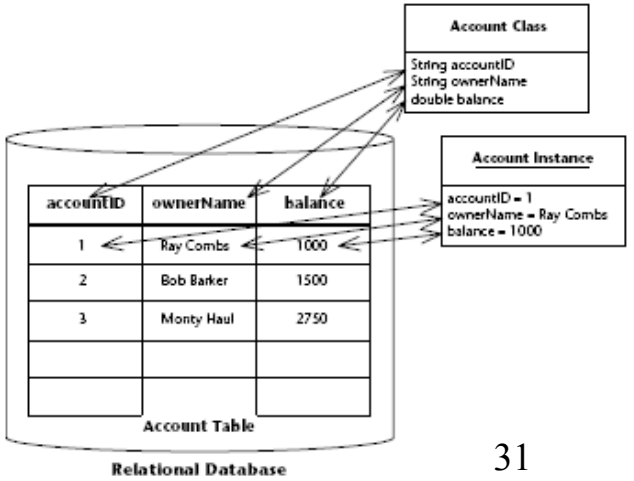
Une solution

- Les Data Access Object (DAO)
  - CRUD operations



Object/Relational Mapping

- Automatiser la transformation Objet/Relationnel
- Utiliser de méta-données pour faire cette transformation
- Plusieurs solutions
  - JPA Java Persistence Specification
  - EJB 3.0 CMP
  - Hibernate
  - TopLink
  - JDO (JSR 12)



Generic ORM problems (1)

1. What do persistent classes look like? How transparent is the persistence tool?
2. How do object identity and equality relate to database (primary key) identity ? How do map instances of particular classes to particular table rows ?
3. How should we map class inheritance hierarchies ?

Generic ORM problems (2)

4. How does the persistence logic interact at runtime with the objects of the business domain ?
5. What is the life cycle of a persistent object ?
6. What facilities are provided for sorting, searching and aggregating ?
7. How do we efficiently retrieve data with associations?

- Transaction an concurrency
- Cache management ..

Why ORM ? (1)

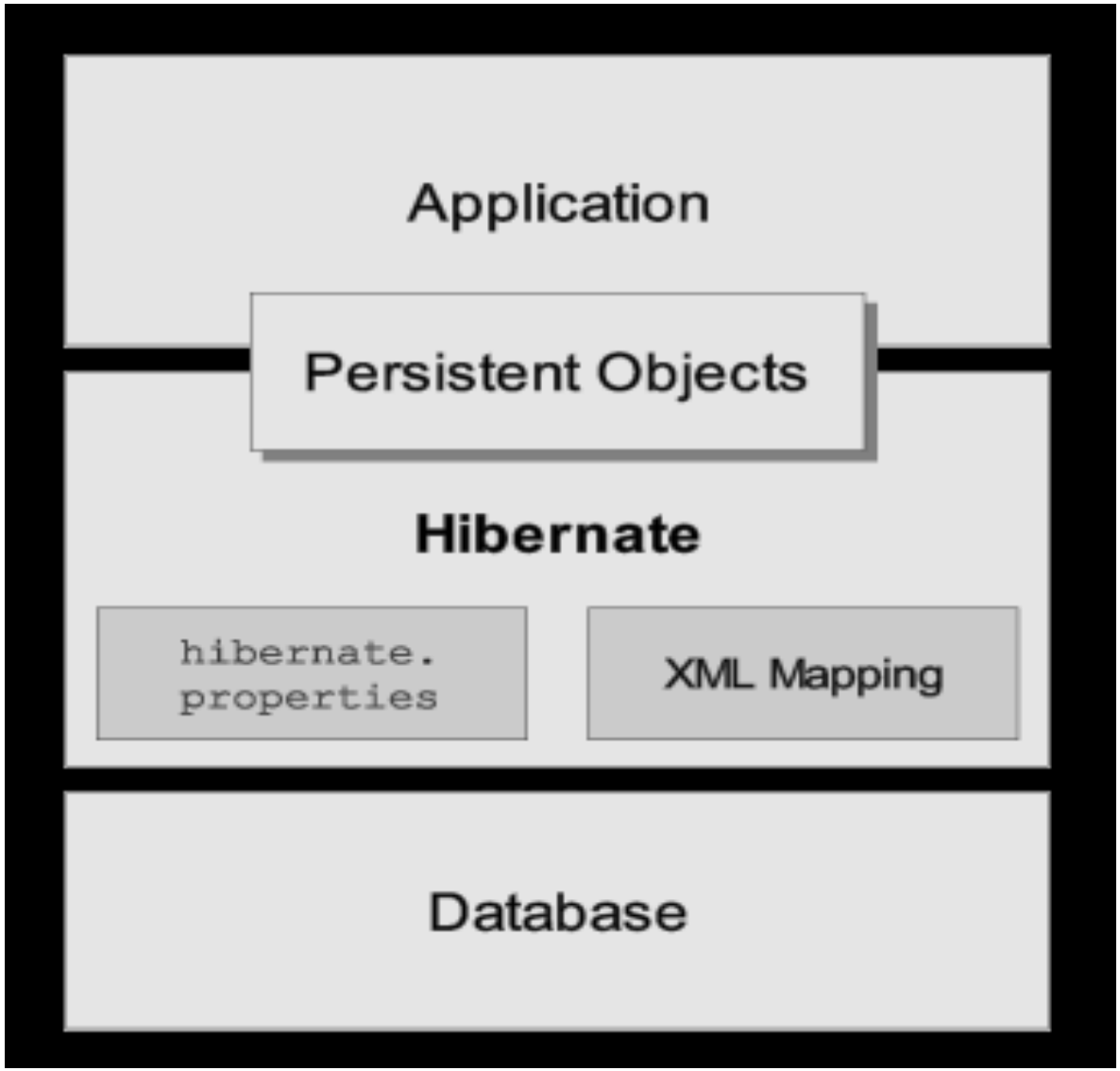
- Productivity
  - Reduce the time of development
- Maintainability
  - Fewer lines of codes (LOC): system more understandable, easier to refactor, ORM insulating object model and relational model from minor changes to the other

Why ORM ? (2)

- Performance
  - More optimization than hand-coded persistence that are used in all applications
- Vendor independence
  - ORM abstracts your application away from the underling SQL database and SQL dialect.

Hibernate

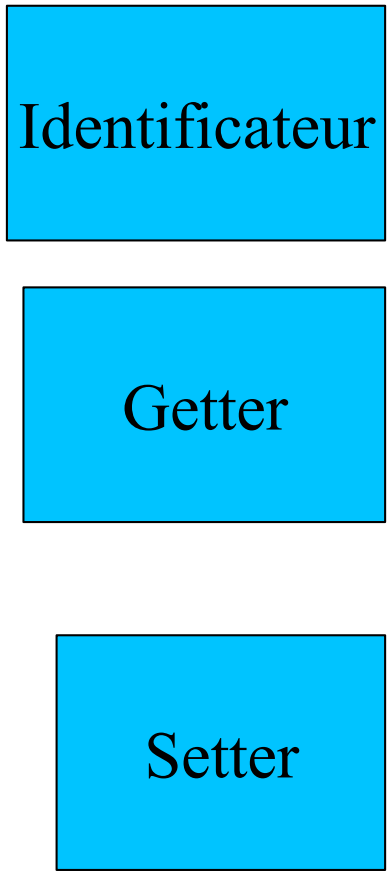
- Gérer la communication avec le SGBD
  - Stockage
  - Chargement
  - Mise à jour
  - CRUD opérations
- Gère la synchronisation des objets avec la base de données



Source: Hibernate doc

Entités

- Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécifications de JPA (Java Persistence API).
- Annotation: @Entity
- Importer:
  - **Javax.Persistence.Entity** ;(idem pour toutes les annotations)




```
public class Sample {
int id;
public int getId() {
return this.id;
}
public void setId(int id){
this.id =id;
}
public String toString() {
return this.getClass().getName()+":"+id;
}
```

Entity Class

Classe d'objets

pouvant être rendu persistant

Déclaration d'entité

Clé primaire

Méthode métier

```
import javax.persistence.*;
@Entity
public class Sample {
int id;
@Id
public int getId() {
return this.id;
}
public void setId(int id){
this.id =id;
}
public String toString() {

returnthis.getClass().getName()+":"+id;
}
```

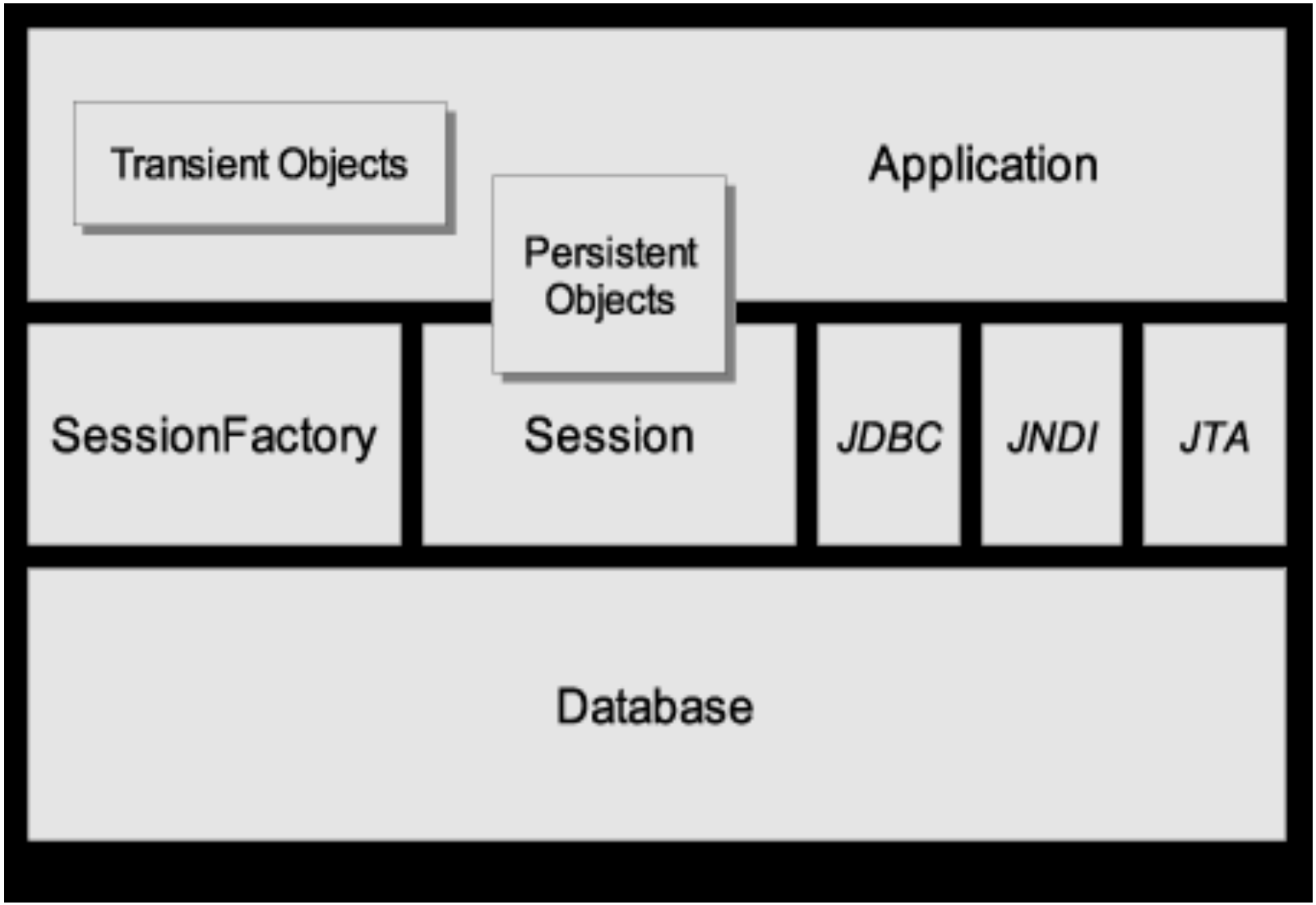
La plomberie

- La persistence est gérée par une “**une session**”
- Créé à partir de
  - Une SessionFactory
  - Un fichier de configuration hibernate.cfg.xml

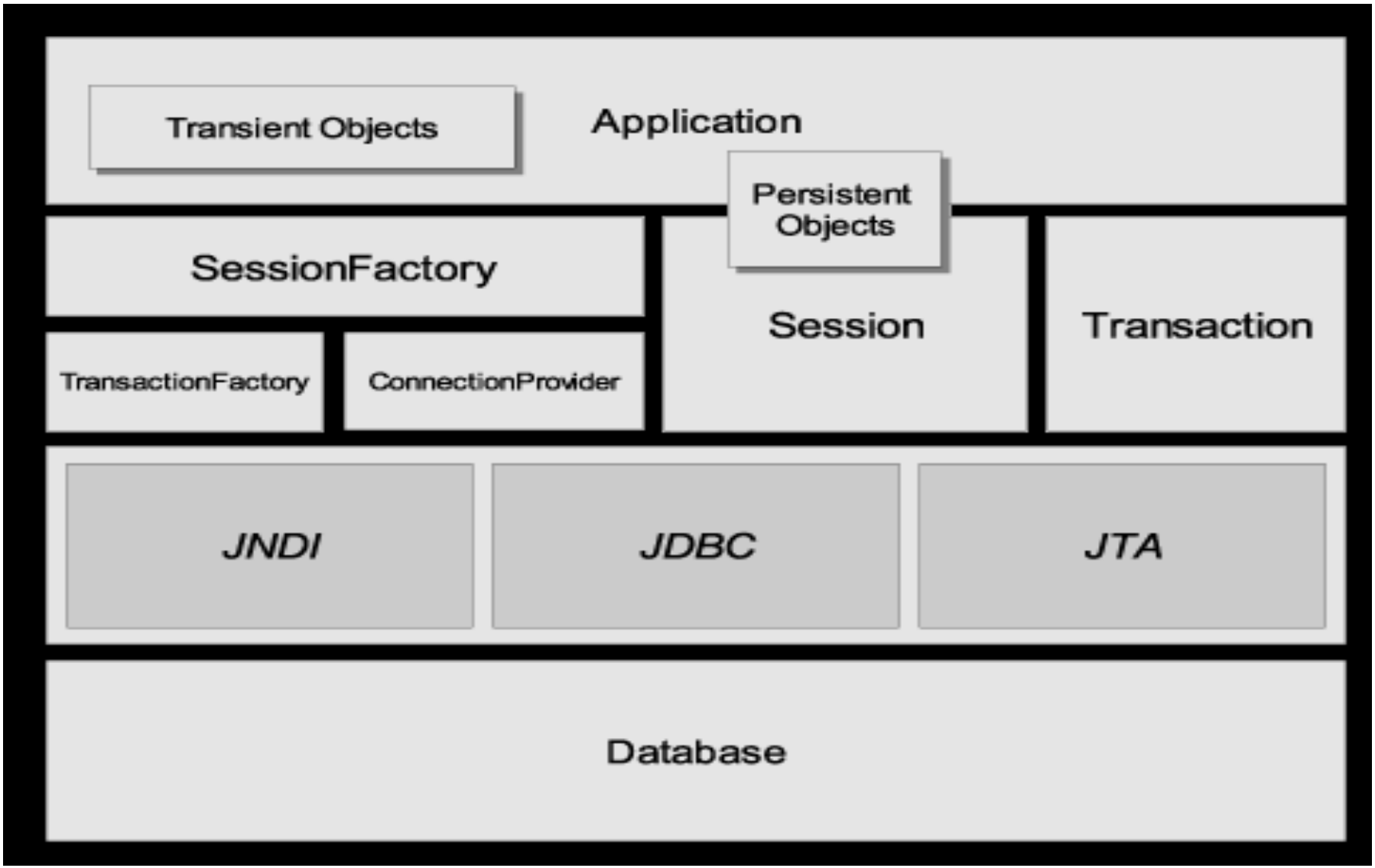
hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="hala">
        <property
name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver </property>
        <property name="hibernate.connection.url">
            jdbc:hsqldb:.</property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect</property>
   <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

<!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>
        <mapping class="Sample" />
    </session-factory>
</hibernate-configuration>
```



Source: doc Hibernate



Source: doc Hibernate

Session Factory

```
SessionFactory sf=new
    AnnotationConfiguration().configure().buildSessio
    nFactory();
```

- Créer un objet SessionFactory à partir du fichier de configuration hibernate.cfg.xml
- SessionFactory un singleton, thread-safe

- Session s = sf.openSession();

```
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
public class Main {
        public static void main(String[] args) {
        SessionFactory sf = new AnnotationConfiguration().configure().buildSessionFactory();
                Session s = sf.openSession();
                s.beginTransaction();
                Sample s1 = new Sample(); s1.setId(1);
                Sample s2 = new Sample(); s2.setId(2);
                s.save(s1);
                s.save(s2);
                s.getTransaction().commit();
                s.close();
```

// Read

1. Par une requête HQL

s = sf.openSession();

List<Sample> result = s.createQuery("from Sample").list();

2. Par la méthode **Load (rechreche par clé)**

Sample p = (Sample)s.load(Sample.class,1);

SessionFactory (org.hibernate.SessionFactory)

SessionFactory is a thread safe (immutable) cache of compiled mappings for a single database. Usually an application has a single SessionFactory. Threads servicing client requests obtain Sessions from the factory. The behavior of a SessionFactory is controlled by properties supplied at configuration time.

Session (org.hibernate.Session)

A session is a connected object representing a conversation between the application and the database. It wraps a JDBC connection and can be used to maintain the transactions. It also holds a cache of persistent objects,used when navigating the objects or looking up objects by identifier.

Transaction (org.hibernate.Transaction)

Transactions are used to denote an atomic unit of work that can be committed (saved) or rolled back together. A transaction can be started by calling session.beginTransaction() which actually uses the transaction mechanism of underlying JDBC connection,JTA or CORBA.A Session might span several Transactions in some cases.

Conditions pour les classes entités

Conditions pour les classes entités

- Un attribut qui représente la clé primaire dans la base (@Id)
- Un constructeur sans paramètre protected ou public
- Aucune méthode ou champ ne doit être final
- Une classe entité ne doit pas être une classe interne
- Une entité peut être une classe abstraite mais elle ne peut être une interface

Convention de nommage JavaBean

- Un JavaBean possède des propriétés.
- Une propriété est représentée par 2 accesseurs
- (« getter» et «setter») qui doivent suivre la convention de nommage suivante :
  - si prop est le nom de la propriété, le getter doit être getProp (ou isProp si la propriété est de type boolean) et le setter setProp
- Souvent une propriété correspond à une variable d'instance

2 Types d'accès

- Le fournisseur de persistance (Hibernate,Topink) accédera à la valeur d'une variable d'instance
  - soit en accédant directement à la variable d'instance (par introspection)
  - soit en passant par ses accesseurs (getter ou setter)
- Le type d'accès est déterminé par l'emplacement des annotations (associées aux variables d'instance ou aux getter)

Entity Class

Classe d'objets

pouvant être rendu persistant

Clé Primaire Sur la variable d'instance

```
import javax.persistence.*;
@Entity
public class Sample {
@Id
int id;
public int getId() {
return this.id;
}
public void setId(int id){
this.id =id;
}
public String toString() {

returnthis.getClass().getName()+":"+id;
}
```

Entity Class

Classe d'objets

pouvant être rendu persistant

Clé Primaire sur le getter

```
import javax.persistence.*;
@Entity
public class Sample {
int id;
@Id
public int getId() {
return this.id;
}
public void setId(int id){
this.id =id;
}
public String toString() {

returnthis.getClass().getName()+":"+id;
}
```

Choix du type d'accès

- Le choix doit être le même pour toutes les classes d'une hiérarchie d'héritage
  (interdit de mélanger les 2 façons)
- En programmation objet il est conseillé d'utiliser plutôt les accesseurs que les accès directs aux champs (meilleur contrôle des valeurs)

Vocabulaire JPA

- Un champ désigne une variable d'instance
- JPA parle de propriété lorsque l'accès

se fait en passant par les accesseurs

(getter ou setter)

- Lorsque le type d'accès est indifférent,

JPA parle d'attribut

Les tables

- Dans les cas simples, une table correspond à une classe (entité)
- Le nom de la table est le nom de la classe
- Les noms des colonnes correspondent aux noms des attributs persistants

Changement du nom de table

- Pour donner à la table un autre nom que le non de la classe, il faut ajouter une annotation @Table

Chgt du nom

```java
import javax.persistence.*;
@Entity
@Table(name="Client")

public class Customer {
    public int id;
    public  String CustName;

    // getter
    @Id

    public int getId() {
        return this.id;
    }
    // setter
    public void setId(int id){
        this.id =id;
    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return this.getClass().getName()+":"+id;
    }
```

Nom de colonne

- Pour donner à une colonne de la table un autre nom que le nom de l'attribut correspondant, il faut ajouter une annotation @Column
- Cette annotation peut aussi comporter des contraintes (voir doc)

```java
public class Customer {
    public int id;
    public  String CustName;
//  public Address address;

    // getter
    @Id

    public int getId() {
        return this.id;
    }
    // setter
    public void setId(int id){
        this.id =id;
    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return this.getClass().getName()+":"+id;
    }
@Column(name="nom",length=200,nullable=false)
    public String getCustName() {
        return CustName;
    }
    public void setCustName(String custName) {
        CustName = custName;
```

Attributs persistants

- Par défaut, tous les attributs d'une entité sont persistants
- Seuls les attributs dont la variable est «transient » ou qui sont annotés par ***@*Transient** ne sont pas persistants

```java
import javax.persistence.*;
@Entity
public class Customer {
    public int id;
    public  String CustName;

    // getter
    @Id
    public int getId() {
        return this.id;
    }
    // setter
    public void setId(int id){
        this.id =id;
    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return this.getClass().getName()+":"+id;
    }

    @Transient
    public String getCustName() {
        return CustName;
    }
    public void setCustName(String custName) {
        CustName = custName;
    }
```

Clé primaire

- Un attribut Id correspond à la clé primaire dans la table associée
- Le type de la clé primaire (ou des champs d'une clé primaire composée) doit être un des types suivants :
  - type primitif Java
  - classe qui enveloppe un type primitif
  - java.lang.String
  - java.util.Date, java.sql.Date
- Ne pas utiliser les types numériques non entier

Génération automatique de clé

- Si la clé est de type numérique entier, l'annotation **@GeneratedValue** indique que la clé primaire sera générée automatiquement par le SGBD
- Cette annotation peut avoir un attribut **strategy** qui indique comment la clé sera générée (il prend ses valeurs dans l'énumération **GeneratorType**)

```java
/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub

    SessionFactory sf = new AnnotationConfiguration().

    Session s = sf.openSession();
    s.beginTransaction();
    Customer s1 = new Customer();
    // s1.setId(1);
    s1.setCustName("Hala1");
    Customer s2 = new Customer();
    // s2.setId(2);
    s2.setCustName("Hala2");
    s.save(s1);
    s.save(s2);
```

```java
@Entity
@Table(name="Client")

public class Customer {
    public int id;
    public  String CustName;
//  public Address address;

    // getter
    @Id
    @GeneratedValue
    public int getId() {
        return this.id;
    }
    // setter
    public void setId(int id){
        this.id =id;
    }
    @Override
```

Types de génération

- AUTO : le type de génération est choisi par le fournisseur de persistance, selon le SGBD (séquence, table,…) ; valeur par défaut
- SEQUENCE : une séquence est utilisée
- IDENTITY : une colonne de type IDENTITY est utilisée
- TABLE : une table qui contient la prochaine valeur de l'identificateur est utilisée

```java
@Entity
@Table(name="Client")

public class Customer {
    public int id;
    public  String CustName;
//  public Address address;

    // getter
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() {
        return this.id;
    }
    // setter
    public void setId(int id){
        this.id =id;
    }
    @Override
```

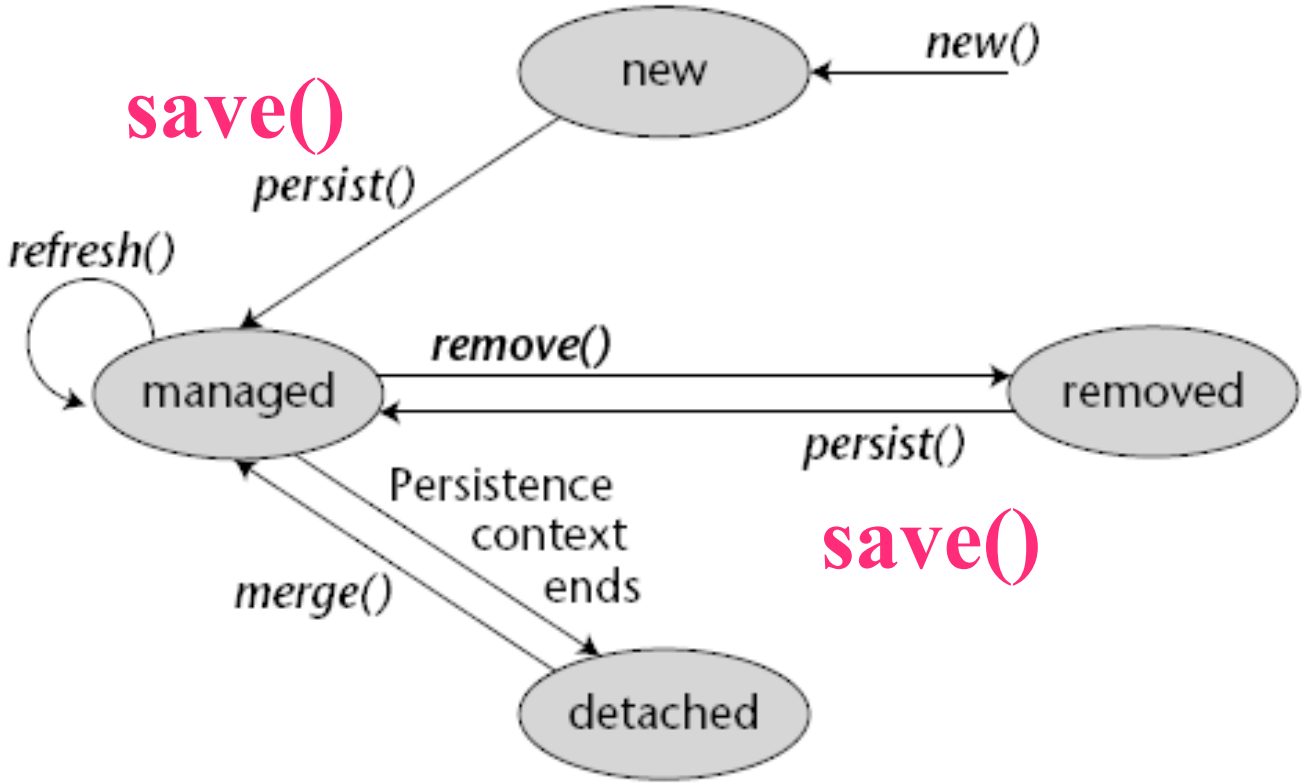
Gestionnaire de persistance

- Gestionnaires des entités (GE) en EJB 3.0, **Session** en Hibernate.
- Il fournit les méthodes pour gérer le cycle de vie d'une entité
  - Rendre persistance (save, merge)
  - Supprimer (delete)
  - Chercher (load, query)
  - Update (flush)
- Principe de base: la persistance des entités n'est pas transparente
- Une instance devient persistante (save ou merge)

Contexte de persistance

- Les entités gérées par un gestionnaires de persistance forment un contexte de persistance.
  - « cache »
- Quand une entité est incluse dans un contexte de persistance (save ou merge), l'état de l'entité est automatiquement sauvegardé dans la base au moment du **commit** de la transaction.
- Propriété importante: dans un contexte de persistance il n'existe pas 2 entités différentes qui représentent des données identiques dans la base

Cycle de vie d'une entité



Cycle de vie d'une instance d'entité

- Nouvelle (new) : elle est créée mais pas associée à un contexte de persistance
- Gérée par un gestionnaire de persistance ;
  - elle a une identité dans la base de données
  - un objet peut devenir géré
    - par la méthode save,
    - merge d'une entité détachée ;
    - provenir d'une requête faite par un gestionnaire d'entités ou d'une navigation à partir d'un objet géré

Cycle de vie d'une instance d'entité

- Détachée : elle a une identité dans la base mais elle n'est plus associée à un contexte de persistance. Une entité peut, par exemple, devenir détachée
  - si le contexte de persistance est vidé ou
  - si elle est envoyée dans une autre JVM par RMI
- Supprimée : elle a une identité dans la base ; elle est associée à un contexte de persistance et ce contexte doit la supprimer de la base de données (passe dans cet état par la méthode remove)

La suite

- Associations
- Héritage
- Requêtes