Hibernate : ORM solution
(In Action, C. bauer & G. King Ed. Manning)

Fabrice Jouanot

Plan du cours

- Introduction, Pourquoi un ORM ?
- Découvrir Hibernate
- Mapping classes – tables
- Gestion des objets persistants
- Querying avec Hibernate
- Transactions et concurrence

Introduction

Pourquoi un ORM ?

Introduction

- Il était une fois SQL
  - Modèle relationnel solide
  - Démocratisé dans tous les SI
  - Outil de définition de schéma
  - Outil d'interrogation
  - Gestion des transactions + concurrences
  - Fiabilité et robustesse
  - Performant et scalable

Introduction

- Et Java fut
  - Java est reconnu comme plateforme multi-tier (J2EE)
  - Langage à objets très utilisé et assez performant
- Lien Java – SQL
  - La BDR = source de persistance
  - Java = application
  - JDBC permet de relier les deux mondes
    - Les requêtes SQL sont écrites (à la main) dans le code
    - Les résultats sont parcourus dans une table "résultat"
    - Les objets d'entreprise sont instanciés à la main
  - Beaucoup de travail fastidieux de bas niveau sans relation direct avec les problèmes métiers !

Cohabitation objet - relationnel

- La persistance des objets restent un problème
  - La sérialisation n'est pas une solution: accès tout ou rien (sauvegarde de l'état du graphe d'objets)
  - Le modèle relationnel ne peut pas capturer directement les paradigmes du modèle à objets
- Le modèle relationnel reste LA solution de persistance, mais comment résoudre
  - Le problème de granularité
  - Le problème d'héritage (sous type)
  - Le problème d'identification
  - Le problème d'associations
  - Le problème de navigation dans un graphe d'objets

Paradigm mismatch

- On peut avoir l'illusion que tout se passe bien: exemple de factures d'un utilisateur

```
public class User {
        private String userName;
        private Set billingDetails;
        …}

Create table USER(
        username varchar(15) not null
primary key,
        …);
```

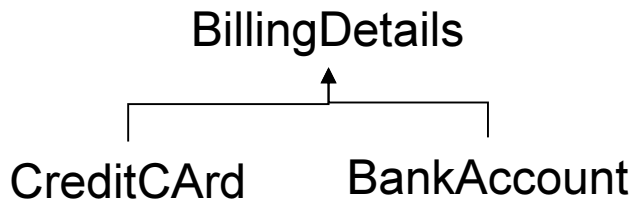
```
public class Billingdetails {
        private String accountNumber;
        private User user;
        …}
Create table billingdetails(
        accountnumber varchar(10) not
null primary key,
        username varchar(15) foreign key
references user(username)
        …);
```

- Les problèmes apparaissent si on introduit un objet Adresse !

Le problème de granularité

- lié à la limite de taille des objets manipulés: exemple de l'adresse (pour un utilisateur)
  - Une classe Adresse en Java qui peut:
    - être décomposée en des éléments à granularité plus fins (numéro, rue, ville, etc.)
    - être référencée dans différentes objets
  - Une table Adresse n'est pas possible (règles de normalisation)
    - utiliser un type utilisateur Adresse (extension SQL)
    - utiliser des attributs associés à l'utilisateur (peu flexible)
- Le problème est facile à résoudre (connaissance de l'application)

Le problème d'héritage



- Java permet de définit aisément cette relation de sous-typage
  - un simple lien d'héritage
  - qui autorise le polymorphisme
- Une BD SQL ne permet pas directement ce type de modélisation
  - pas de sous-type construit par héritage (ou SGBDRO)
  - aucune notion de polymorphisme (selon SGBDRO)

Le problème d'identification

- Le modèle à objet est plus riche en matière d'identité
  - identité d'objets (même emplacement mémoire)
  - égalité d'objets (même objet de surface, opérateur equals())
- Une BD SQL se limite à la définition d'une clé primaire (ou identifiant)
  - le choix d'une clé est difficile (ou OID dans SGBDRO)
  - la définition de clé de remplacement est fréquent (chaîne de caractères = mauvais candidat)

Le problème d'associations

(exemple des factures)

- En java une association est une référence vers un ensemble d'objets
  - Factures et utilisateurs sont stockés indépendamment
  - la relation est explicitement bidirectionnel
- En BD SQL l'association est modélisé par les clés étrangères (sur des clés primaires)
  - Factures et utilisateurs sont stockés dans leur propre table
  - la relation n'est pas bidirectionnel: one-to-many ou one-to-one
  - Une relation bidirectionnelle implique la création d'une table de liaison : artificielle et sans intérêt au niveau métier !

Le problème de navigation

- En java, il est naturel de naviguer dans le graphe d'objets pour construire les objets résultats
  - notation chainée (aUser.getBillingdetails().getAccountNumber())
  - pas de surcoût à la navigation
- En relationnel, la notion de navigation est remplacée par la notion de jointure

```
select *
from user u
left outer join billingdetails bd on bd.user_id=u.user_id
where u.user_id=123
```

  - le passage objet > BD SQL peut générer une requête de jointure pour chaque nœud du graphe (*n+1 selects problem*)

Modèle en couche : solution au surcoût ?

- 30% du code applicatif
  - gestion SQL/JDBC bas niveau
  - résolution des problèmes de paradigmes
- Modèle de persistance en couche
  - séparation des préoccupations
  - communication top to bottom
  - modèle en 3 couches (BD : en dehors de l'application)
    - Présentation (interface et interaction)
    - Métier (application)
    - Persistance (stockage et gestion des objets persistants)

Différentes approches

- Sérialisation : inadaptée
- SLQ/JDBC : très coûteuse pour mettre en concordance les 2 modèles
- EJB entity beans :
  - peu efficace en pratique
  - mauvaise granularité, pas de polymorphisme, très intrusif
- BDOO : quasi disparu, standard ODMG en déclin
- ORM : Object Relational Mapping
  - mapping transparent entre objets persistants (Java) et un schéma de BDR(O)
  - fournit une multitude de facilité
    - une API pour la persistance,
    - un langage de requête OO (même plusieurs)
    - une définition facile et précise du mapping
    - la prise en compte de l'aspect transactionnel
    - la garantie de performance (partiellement scalable)

Découverte d'Hibernate

Hello World

- Soit la classe suivante qui doit persister:

```
package hello
public class Message {
        private Long id;
        private String text;
        private Message nextMessage;
        private Message() {}
        public Message (String text) { this.text = text; }
        public Long getId() { return id;}
        private void setId(Long id) { this.id= id; }
        public String gettext() { return text; }
        public void setText(String text) { this.text = text; }
        public Message getNextMessage() { return nextMessage; }
        public void setNextMessage(Message nextMessage) {
                this.nextMessage = nextMessage; }
}
```

et son mapping

- Un fichier de mapping permet le lien entre une classe et la BDR (le fichier se nomme Message.hbm.xml)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-maping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd>
<hibernate-mapping>
    <class
            name="hello.Message"
            table="MESSAGES"
            <id name ="id" column="MESSAGE_ID"> <generator class="increment"/></id>
            <property name="text"  column="MESSAGE_TEXT"/>
            <many-to-one name="nextMessage" cascade="all" column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

Squelette de programme

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class FirstExample {
  public static void main(String[] args) {
    Session session = null;
    Configuration cf ;

    try{

// Nos exemples Ici !

    }catch(Exception e){
      System.out.println("catch !:"+e.getStackTrace());
    }
  }
}
```

Premiers pas

- Instanciation habituelle

```
Message message = new Message("Hello World");
System.out.println(message.getText());
```

- Session Hibernate

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
session.save(message);
tx.commit();
session.close();
```

- Type de requête produite

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
    values (1,'Hello World', null);
```

Extraction d'un ensemble d'objets

- Comment récupérer des messages de la BD :

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages = newSession.find("from Message as m order by m.text asc");
System.out.println(messages.size() + " message(s) found:");
for( Iterator iter= messages.iterator(); iter.hasNext();) {
    Message message = (Message) iter.next();
    System.out.println( message.getText());
}
newTransaction.commit();
newSession.close();
```

- Aucune présence de code SQL (qui est généré dynamiquement à l'exécution)

Association entre objet persistant et un nouvel objet (1)

- Hibernate est capable d'automatiser les MAJ à partir de manipulation sur les objets métiers :

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = (Message) session.load(Message.class, new Long(1));
message.setText("Cher Monsieur");
Message nextMessage = new Message("Considérer ma demande
    d'augmentation");
message.setNextMessage( nextMessage);
tx.commit();
session.close();
```

Association entre objet persistant et un nouvel objet (2)

- Génération des requêtes SQL :

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGE m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT,
    NEXT_MESSAGE_ID) values (2, 'Considérer ma demande
    d'augmentation ', null);

update MESSAGES
set MESSAGE_TEXT = 'Cher Monsieur', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Interfaces centrales

- Session
  - interface la plus importante qui définit une session de travail sur les objets persistants
  - créée et détruite sans cesse : coût faible
  - associée à un seul thread à la fois
- SessionFactory
  - une session est obtenue de cette interface
  - une seule interface pour toute l'application, créée à son initialisation
  - Gestion des caches
- Configuration
  - Objet assurant la configuration et le bootstrap Hibernate
  - premier objet créé
- Transaction
  - assure une gestion des transactions indépendantes de l'environnement
  - interface optionnelle
- Requête et critère
  - exécution et optimisation des requêtes
  - gestion des paramètres

Interfaces secondaires

- Callback (option)
  - Observateur des événements sur les objets
  - permet la gestion d'audit
- Types
  - Permet l'association d'un type Hibernate à un type de BD
  - Possibilité de créer des types utilisateurs
- Extension
  - La majorité des fonctionnalités Hibernate est hautement configurable : choix de stratégie
    - PK génération
    - support SQL
    - Cache
    - JDBC, etc.

Configuration de base

- Deux modes
  - Environnement géré: J2EE au dessus de Java.
  - Environnement non géré: Servlet, tomcat, ligne de commande.
- Cours centré sur environnement non géré:
  - Hibernate joue le rôle de client d'un pool de connection JDBC
    - acquérir une nouvelle connexion est couteux
    - Maintenir beaucoup de connexions est couteux
    - Créer des statements est souvent couteux (=>drivers)
  - Définition d'un fichier de propriétés ou d'un fichier de configuration XML

Properties file

- Gestion du POOL JDBC

```
hibernate.connection.driver_class = org.postgresql.Driver
// JDBC driver (dans le CLASSPATH de l'application)
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
// chemin de connection à la BD
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.POstgresSQLDialect
// Dialect SQL pour la BD
hibernate.c3P0.min_size = 5
// minimum de connections à conserver
hibernate.c3P0.max_size = 20
//  nombre limite de connections
hibernate.c3P0.timeout = 300
// avant suppression d'une connection
hibernate.c3P0.max_statements = 50
// maximum de statements cachés
hibernate.c3P0.idle_test_period = 300
```

Démarrage Hibernate

- Méthode
  - Placer hibernate2/3.jar dans le CLASSPATH de l'application
  - Ajouter les dépendances Hibernate dans le CLASSPATH
  - Choisir un pool de connection JDBC supporté par Hibernate => properties file
  - créer une instance de Configuration dans l'application et ajouter le mapping
- Exemple :

```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties(System.getProperties());
SessionFactory sessions = cfg.buildSessionFactory();
```

ou

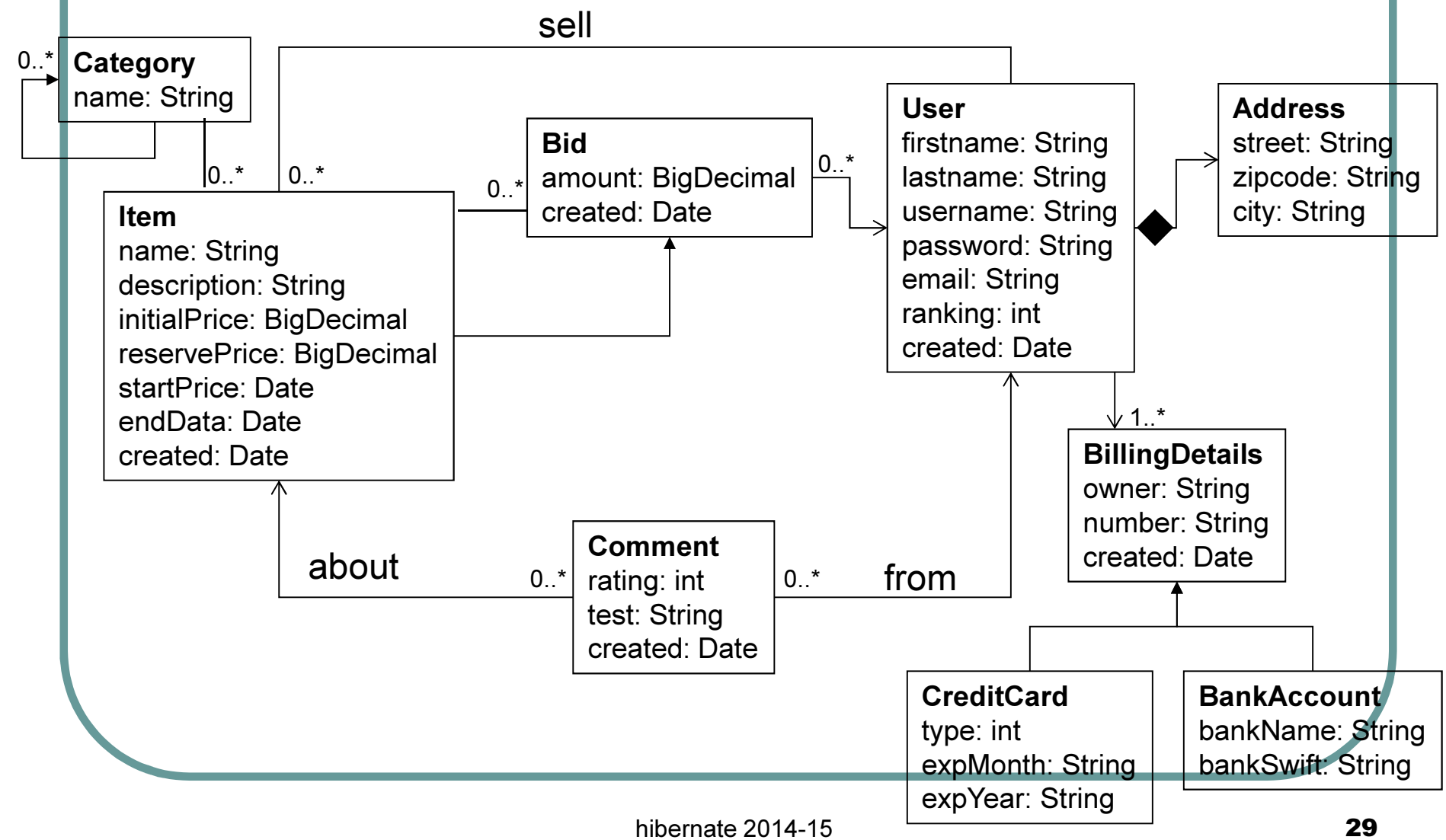
```
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties(System.getProperties())
    .buildSessionFactory();
```

Fichier de configuration xml

- Hibernate peut utiliser un fichier qui centralise tous les paramètres (par défaut *hibernate.cfg.xml*)

```
<hibernate-configuration>
    <session-factory name="java:/hibernate/HibernateFactory">
            <property name="show_sql">true</property>
            <property name="connection.datasource>
                    java:/comp/env/jdbc/AuctionDB
            </property>
            <property name="dialect">
                    net.sf.hibernate.dialect.PostgresSQLDialect
            </property>
            <property name="transaction.manager_lookup_class">
                    net.sf.hibernate.transaction.JBossTransactionManagerLookup
            </property>
            <mapping resource="auction/Item.hbm.xml"/>
            <mapping resource="auction/Category.hbm.xml"/>
            <mapping resource="auction/Bid.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Exemple fil rouge (site d'enchères)



Mapping des classes persistantes

Hibernate = POJO model

- Hibernate repose sur un modèle de programmation POJO (Plain Old Java Objects) issu des Java beans
  - préférer les classes sérialisable
  - utiliser des méthodes d'introspections
  - définir les méthodes métiers

```
public class User implements Serializable {
    private String username;
    private Address address;
    public User(){}
    public String getUsername() { return username;}
    public void setUsername(String username) { this.username = username;}
    public Address getAddress()…
    public void setAddress(Address address)…
    public MonetaryAmount calcShippingCosts(Address fromLocation)…}
```

Les Associations en POJO

- L'objectif est de faire apparaître les associations bidirectionnelles:

```
public class Category implements Serializable{
        private String name;
        private Category parentCategory;
        private Set childCategories = new
HashSet();
        private Set items = new HashSet();
        public String getName()… }
```

```
Public class Item {
        private String name;
        private String description;
        private Set categories = new HashSet();
        …
        public void addCategory(Set categories)
        {       if (category==null) throw …
                category.getItems().add(this);
                categories.add(category); }…
```

Avec l'utilisation qui va avec:

```
Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategory().add(aChild);
```

Ajouter la logique aux classes

- Il faut prévoir d'adapter la logique des méthodes get/set au contenu de la BD:

```
public void setName(String name) {
    StringTokenizer t = new StringTokenizer(name);
    firstname = t.nextToken();
    lastname = t.nextToken(); }
```

si la BD stocke le nom dans 1 ou 2 attribut !

Mapping de la classe Category

```
<hibernate-mapping>
    <classe name="org.hibernate.auction.model.Category"
    table="CATEGORY">
            <id name="id" column="CATEGORY_ID" type="long">
                    <generator class="native"/>
            </id>
            <property name="name" column="NAME" type="string"/>
    </class>
</hibernate-mapping>
```

- Mapping de la classe sur une relation
- Mapping des propriétés sur des attributs
- Mapping de l'identifiant sur la les attributs de la relation

Mapping des propriétés

- Hibernate utilise la réflexion pour déterminer le type d'une propriété:

```
<property name="name" column="NAME" type="string"/>
```

équivalent à

```
<property name="name" column="NAME"/>
```

- Hibernate autorise les propriétés dérivés (ou calculés):

```
<property name="totalIncludingTax"
    formula="TOTAL + TAX_RATE * TOTAL"
    type="big_decimal/>
```

ou encore

```
<property name="averageBidAmount"
    formula="(select AVG(b.AMOUNT) from BID b where b.ITEM_ID=ITEM_ID)"
    type="big_decimal"/>
```

Mapping étendu

- Contrôle des MAJ: le dictionnaire peut autoriser ou non les MAJ

```
<property name="name" column="NAME" type ="string" insert="false" update="true"/>
```

- Déclaration des classes: le Package Java peut être déclarée de 2 manières

```
<hibernate-mapping>
    <class name="org.hibernate.auction.model.Category" table="CATEGORY">…
peut s'écrire
<hibernate-mapping
    package="org.hibernate.auction.model"
    <class name="Category" table="CATEGORY">…
```

- Les informations de mapping peuvent être définies pendant l'exécution !

Gestion des identités

- 3 méthodes pour identifier des objets (comme en BDOO):
  - Identité objet : même emplacement mémoire
  - Egalité d'objet : même valeurs entre objets
  - Identité de BD : même PK
- Implémentation d'une propriété identifiant

```
public class Category {
    private Long id;
    public Long getId() { return this.id;}…}
```

avec son mapping

```
<class name="Category" table="CATEGORY">
    <id name="id" column="CATEGORY_ID" type="long">
        <generator class="native"/>
    </id>
```

Choix du type de PK pour le mapping d'identifiant

- native: utilise le générateur d'identifiant de la BD (identity, sequence, hilo)
- identity: utilise des attributs d'une relation
- sequence: disponible dans Oracle, DB2, Postgres, etc.
- increment: incrémentation de la valeur maximum d'un attribut
- hilo: générateur d'identifiant unique (sur une BD)
- uuid.hex: générateur d'une chaine unique (sur un réseau)

Granularité d'un objet

- Hibernate dispose de 2 grains:
  - l'entité (entity) qui persiste et qui dispose de son propre identifiant BD. Une entité possède un cycle de vie.
  - une valeur (value type) qui ne dispose pas d'identifiant BD et dont le cycle de vie dépend d'une entité.
- Problème entre les classes User et Address : relation *part of*
  - Correct en UML mais Address n'est pas une entité en Hibernate
  - on utilise la notion de composant (component) d'une entité

```
<class name="User" table="USER">
      <id name="id" column="USER_ID" type="long"><generator class="native"/></id>
      <property name="username" column="USERNAME" type="string"/>
      <component name="homeAddress" class="Address">
            <property name="street" type="string" column="HOME_STREET" notnull="true"/>
      …</component>
      <component name="billingAddress" class="Address">
            <property name="street" type="string" column="BILLING_STREET" notnull="true"/>
      …</component>
</class>
```

Mapping pour l'héritage

- 3 stratégies selon les besoins
  - une table par classe concrète: héritage et polymorphisme disparaissent, une classe définie par classe concrète (solution à éviter)
  - une table pour la hiérarchie: mapping avec une seule table dont un attribut pour discriminer les propriétés.
  - une table par sous-classe: chaque table ne contient que les informations spécifiques, le mapping permet d'expliciter le concept de sous-classe.

Une table = la hiérarchie

- Solution la plus performante mais
  - les attributs des sous-classes doivent être Nullable… Pb de cohérence des données
  - nécessité d'un attribut discriminant
- Exemple : Billingdetails avec ses 2 sous classes CreditCard & BankAccount

```
<hibernate-mapping>
    <class name="BillingDetails" table="BILLING_DETAILS" discriminator-value="BD">
            <id name="id" column="BILLING_DETAILS_ID" type="long">
                    <generator class="native"/> </id>
            <discriminator column="BILLING_DETAILS_TYPE" type="string"/>
            <property name="name" column="OWNER" type="string"/> …
            <subclass name="CreditCard" discriminator-value="CC">
                    <property name="type" column="CREDIT_CARD_TYPE"/>
                    …
            </subclass>
    … </class>
</hibernate-mapping>
```

Une table par sous-classe

- Chaque table représentant une sous-classe peut être liée à la table représentant la super classe via PK et FK:

```
<hibernate-maping>
<class name="BillingDetails" table="BILLING_DETAILS">
        <id name="id" column="BILLING_DETAILS_ID" type="long">
                <generator class="native"/> </id>
        <property name="name" column="OWNER" type="string"/> …
        <joined-subclass name="CreditCard" table="CREDIT_CARD">
                <key column="CREDIT_CARD_ID">
                <property name="type" column="TYPE"/>
                …
        </joined-subclass>
    … </class>
</hibernate-maping>
```

Gestion des associations

- Hibernate n'implémente pas de gestionnaire d'associations:
  - les associations sont "naturellement" unidirectionnelles
  - le mapping propose des liens one-to-one, one-to-many, many-to-one et many-to-many
- Reprenons l'exemple d'un objet avec plusieurs enchères:
  - coté classe enchère

```
public class Bid { private Item item; …}
```

  - et son mapping

```
<class name="Bid" table="BID">
    …
    <many-to-one name="item" column="ITEM_ID" class="Item" not-null="true"/>
</class>
```

Transformer une association unidirectionnelle en bidirectionnelle

- L'objectif est de pouvoir naviguer dans le graphe d'objets:
  - coté classe Item

```
public class Item{ private Set bids = new hashSet();
        …
        public void addBid(Bid bid) {
                bids.setItem(this);
                bids.add(bid); }
}
```

  - et son mapping

```
<class name="Item" table="ITEM">
        …
        <set name="bids">
                <key column="ITEM_ID"/>
                <one-to-many class="Bid"/>
        </set>
</class>
```

- 2 problèmes se posent:
  - double détection des MAJ dans addBid car rien n'explicite l'association bidirectionnelle (2x unidirectionnelles)
  - pas de persistance automatique des enchères: il faut appeler la méthode save() sur l'interface Session

Transformer une association unidirectionnelle en bidirectionnelle

- Le mapping final explicite l'association bidirectionnelle avec persistance auto:

```
<class name="Item" table="ITEM">
    …
    <set name="bids" inverse="true" cascade="save-update">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

- Relation parent/enfants = non indépendance des cycles de vie: "save-update" n'est pas suffisant

```
<class name="Item" table="ITEM">
    …
    <set name="bids" inverse="true" cascade="all-delete-orphan">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

Système de typage Hibernate
"Built-in mapping types"

mapping type	Java Type	SQL type
integer	Int	INTEGER
long	Long	BIGINT
short	Short	SMALLINT
float	Dloat	FLOAT
double	Double	DOUBLE
big_decimal	Java.math.BIgDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte	TINYINT
boolean	boolean	BIT
yes_no	boolean	CHAR(1)('Y' or 'N')
true_false	boolean	CHAR(T)('Y' or 'F')
date	java.util.Date	DATE
time	java.util.Date	TIME
timestamp	java.util.Date	TIMESTAMP
calendar	java.util.calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY (ou BLOB)
text	java.lang.String	CLOB
serializable	java.io.Serializable class	VARBINARY (ou BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Mapping des collections de valeurs

- Nous savons expliciter les associations d'entités… mais pour les valeurs ? => Exemple d'une collection d'images associées aux objets.
- Hibernate propose 4 types de collection: set, bag, list, map

- Collection Set
  - coté Item

```
public class Item{
    private Set images = new hashSet();
    …}
```

  - coté mapping

```
<set name="images" lazy="true" table="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Mapping des collections de valeurs

- Collection Bag : Hibernate ne propose pas directement de liste non ordonnée
  - coté Item: on remplace Set par List (l'ordre n'est pas préservée lors de la persistance avec une sémantique de bag)
  - coté mapping: on reprend la même structure avec une clé supplémentaire (doublon)

```
<idbag name="images" lazy="true" table="ITEM_IMAGE">
    <collection-id type="long" column="ITEM_ID"><generator class="sequence"/>
    <collection-id/>
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

- Collection List : il faut un attribut servant d'index pour repérer la position d'un élément dans la liste (ordre)

```
<list name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

Collections de composants

- Une classe "component" est considérée comme une valeur.
- Exemple d'une classe Image composant de Item, avec les propriétés name, filename, sizeX et sizeY

```
<set name="images" lazy="true" table="ITEM_IMAGE" order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <composite-element class="Image">
        <parent name="item"/>
        <property name="name" column="IMAGE_NAME" not-null="true"/>
        <property ….>
    </composite-element>
</set>
```

(ici le mapping est une association bidirectionnelle)

Association one-to-one

- Comment construire le mapping d'une association 1:1 entre deux entités (ici une classe Adresse et une classe User)
  - définition de la classe

```
<class name="Adress" table="ADRESS">
    <id name="id" column="ADDRESS_ID"><generator class="native"/></id>
    <property name="street"/>…
</class>
```

  - on ajoute un attribut BILLING_ADDRESS_ID dans la table USER

```
<many-to-one name="billlingAddress" class="Address" column="BILLING_ADDRESSE_ID" cascade="all" unique="true"/>
```

  - on ajoute une propriété user dans la classe Adresse pour assurer la bidirection

```
<one-to-one name="user" class="User" property-ref="billingAddress"/>
```

  - Exemple d'utilisation:

```
Address address=new Adress();
address.setStreet("681 rue de la Passerelle");
address.setCity("Grenoble"); address.setZipcode("38000");
Transaction tx=session.beginTransaction();
User user=(User) session.get(user.class, userId);
address.setuser(user);
user.setBillingAddress(adress);
tx.commit();
```

Association many-to-many

- Considérons qu'une catégorie est associée à un ensemble d'objet et les objets sont associés à un ensemble de catégorie.
  - On veut pouvoir faire :

```
Transaction tx=session.begintransaction();
Category cat=(Category) session.get(Category.class, categoryId);
Item item=(Item) session.get(Item.class, itemId);
cat.getItems().add(item);
item.getCategories().add(category);
tx.commit();
```

  - coté Category

```
<class name="Category" table="CATEGORY">…
    <set name="items" table="CATEGORY_ITEM" lazy="true" cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
</class>
```

  - coté Item

```
<class name="Item" table="ITEM"> …
    <set name="categories" table="CATEGORY_ITEM" lazy="true" inverse="true" cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

Mapping & annotations

- Des annotations remplacent les fichiers de mapping
  - Code + mapping au même endroit
  - Plus lisible ou plus brouillon selon les gouts
- Exemple basique

```
@Entity
@Table(name="Category")
public class Category {
      private Long id;
      @Id
      @Column(name="CATEGORY_ID")
      public Long getId() { return this.id;}…}
    … }
```

- Exemple de gestion des associations

```
@Entity
@Table (name="ITEM")
public class Item{ private Set bids = new hashSet();
      …
      @OneToMany(mappedBy="ITEM_ID", inverse="true",cascade="all-delete-orphan")
      @OrderBy("created")
      public List<bid> getBids { return bids; }      …}
```

Gestion du Polymorphisme

- Supposons que nous ayons le mapping suivant pour BillingDetails:

```
<many-to-one name="user" class="User" column="USER_ID"/>
```

- Et celui-ci pour Users:

```
<set name="billingDetails" lazy="true" cascade="save-update" inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

- Le polymorphisme est immédiat sous Hibernate (pour des collections comme pour des associations simple) si des classes sont déclarées avec <subclass> ou <joined-subclass>:

```
CreditCard cc=new CreditCard();
cc.setNumber(ccNumber); cc.setType(ccType); cc.setexpirydate(ccExpiryDate);
Session session=getSessionFactory.openSession();
Transaction tx=session.beginTransaction();
user user=(User) session.get(User.class,uid);
user.addBillingDetails(cc);
tx.commit();
session.close();
```

Gestion du polymorphisme

- Itération sur l'ensemble des paiements d'un utilisateur:

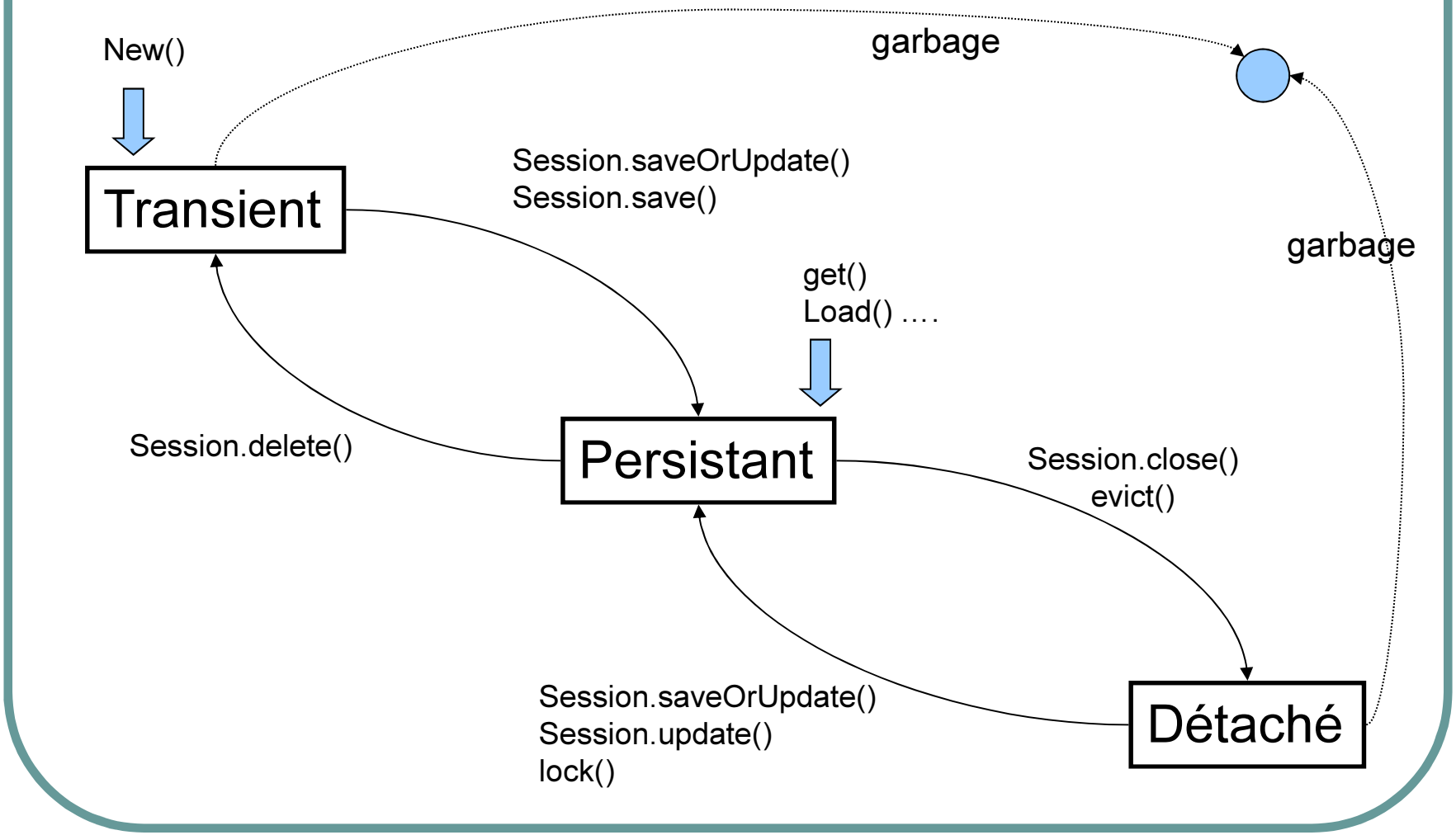
```
Session session=getSessionFactory.openSession();
Transaction tx=session.beginTransaction();
user user=(User) session.get(User.class,uid);
Iterator iter=user.getBillingDetails().iterator();
while (iter.hasnext()){
    BillingDetails bd=(BillingDetails) iter.next();
    bd.pay(ccPaymentAmount);
tx.commit();
session.close();
```

Travailler avec des classes persistantes

Cycle de vie des objets

- Objets transients
  - tous les objets Hibernate ne sont pas persistants
  - un objet transient n'est pas attaché à une BD et ne participe pas aux transactions
  - un objet transient peut devenir persistants
- Objets persistants
  - un objet persistant possède un identifiant lié à une clé primaire de la BD
  - les objets extraits à l'aide d'une requête hibernate sont persistants
  - Hibernate détecte les objets persistants à mettre à jour : transparent pour le développeur.
- Objets détachés
  - un objet persistant devient détaché après fermeture d'une session
  - un objet détaché peut redevenir persistant

Cycle de vie des objets



Portée des objets persistants

- Deux objets utilisant la même clé sont identiques (au niveau référence mémoire)

```
Session session1=sessions.OpenSession();
Transaction tx1=session1.beginTransaction();
Object a=session1.load(Category.class,new Long(1234));
Object b=session1.load(Category.class,new Long(1234));
if (a==b) {System.out.println("a et b identiques.");}
tx1.commit(); session1.close();
```

- Hibernate ne garantit pas l'identité hors d'une session

```
Session session2=sessions.OpenSession();
Transaction tx2=session2.beginTransaction();
Object b2=session2.load(Category.class,new Long(1234));
if (a!=b2) {System.out.println("a et b différents.");}
System.out.println( a.getId().equals(b2.getId()) );
tx2.commit(); session2.close();
```

Implémenter equals() et hashCode()

- Hibernate utilise ces 2 méthodes pour identifier les duplications d'éléments
  - java.long.Object par défaut marche tant que les objets sont de mêmes natures.
  - sinon il faut les redéfinir
- 2 méthodes de redéfinitions
  - par comparaison de valeurs

```
public class User {…
    public boolean equals(Object other) {
            if (this==other) return true;
            if (!(other instanceof User)) return false;
            final User that=(User) other;
            if (!this.getUsername().equals(that.getUsername()) return false;
            if (!this.getPassword().equals(that.getPassword()) return false;
            return true; }
    public int hashCode() { int result=14;
            result=29*result+getUsername.hashCode();
            result=29*result+getPassword.hashCode();
            return true; }
```

  - basé sur l'égalité métier des clés

Utilisation du gestionnaire de persistance

- Faire persister un objet

```
User user=new User();
user.getName().setFirstName("John");
user.getName().setLastName("Doe");
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
session.save(user);
tx.commit(); session.close();
```

  - Il est possible de modifier user après save(): l'état de user sera modifié au commit() car il est devenu persistant.

- Mettre à jour un objet détaché

```
user.setPassword("secret");
Session session2=sessions.OpenSession();
Transaction tx=session2.beginTransaction();
session2.lock(user,LockMode.NONE);
session2.update(user);
user.setUsername("johnny");
tx.commit(); session2.close();
```

Utilisation du gestionnaire de persistance

- Récupérer un objet persistant

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
User user=(User) session.get(User.class, new Long(1234));
tx.commit(); session.close();
```

- Mettre à jour un objet persistant

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
User user=(User) session.get(User.class, new Long(1234));
usert.setPassword("secret");
tx.commit(); session.close();
```

Utilisation du gestionnaire de persistance

- Faire d'un objet persistant, un objet transient

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
User user=(User) session.get(User.class, new Long(1234));
session.delete(user);
tx.commit(); session.close();
```

- Faire d'un objet détaché, un objet transient

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
session.delete(user);
tx.commit(); session.close();
```

Persistance transitive

- Un objet est persistant transitivement si il est accessible via le graphe d'objet par un objet persistant
  - par défaut Hibernate n'assure pas cette persistance: il faut l'expliciter dans le mapping, **cascade=**
    - "none" : ignore les associations
    - "save-update" : considère les associations lors d'un commit ou d'un save/update pour sauver de nouveaux transients (ou pour faire persister des détachés)
    - "delete" : considère les associations lors de la suppression d'un objet persistant
    - "all" : idem "save-update"+"delete"
    - "all-delete-orphan" : idem "all" + suppression des persistants ne faisant plus partie de l'association
    - "delete-orphan" : suppression des persistants hors association

Exemple de persistance transitive

- Gestion des enchères

```
<class name="Category" table="CATEGORY">
    …
    <property name="name" column="CATEGORY_NAME"/>
    <many-to-one name="parentCategory" class="Category"
            column="PARENT_CATEGORY_ID" cascade="none"/>
    <set name="childCategories" table="CATEGORY" cascade="save-update"
            inverse="true">
            <key column="PARENT_CATEGORY_ID"/>
            <one-to-many class="Category"/>
    </set>
</class>
```

Exemple de persistance transitive

- Persistance automatique d'une sous-catégorie:

```
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
Category computer=(Category) session.get(Category.class,computerID);
Category laptops=new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
tx.commit(); session.close();
```

- Persistance automatique d'une sous-catégorie définie hors session:

```
Category computer= … détaché d'une session précédente
Category laptops=new Category("Laptops");
computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
Session session=sessions.OpenSession();
Transaction tx=session.beginTransaction();
session.save(laptops);
tx.commit(); session.close();
```

Comment récupérer des objets

- par l'identifiant

User user = (User) session.get(User.class, userId); *(peut renvoyer NULL)*

ou

User user = (User) session.load(User.class, userId); *(ne renvoie jamais NULL => exception)*

- par requête HQL (uniquement interrogation)

```
Query q = session.createQuery("from User u where u.firstname= :fname");
q.setString("fname","Max");
List result = q.llist();
```

- par critères

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Expression.like("firstname","Max");
List result = criteria.list();
```

Comment récupérer des objets

- par l'example (QBE)

```
User exampleUser = new User();
exampleUser.setFirstname("Max");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(exampleUser));
List result = criteria.list();
```

- Fetching
  - Le fait de compléter un objet avec des informations complémentaire (un utilisateur avec ses adresses, un objet avec ses enchères, etc.) => jointure externe en SQL
  - Hibernate gère plusieurs stratégies de Fetching
    - Immediate : récupère tout le sous-graphe d'objets
    - **Lazy : récupère les éléments du graphe en fonction des accès**
    - Eager : les objets associés à charger sont explicités
  - Hibernate utilise un proxy/cache pour réaliser ces stratégies.

Interrogation avancée

Exécution d'une requête

- L'interface standard est HQL via createQuery()

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

- équivalente à la requête SQL:

```
Query sqlQuery = session.createSQLQuery (
    "select {u.*} from USERS {u}", "u", User.class) ;
```

- qui peut s'écrire sous forme de critère

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

Exploitation des résultats

- Pagination des résultats:

```
Query query = session.createQuery("from User u order by u.name asc");
query.setFirstResult(0);
query.setMaxResults(10);
```

- Récupération d'une liste

```
List result = query.list();
```

- Récupération d'un élément

```
Bid maxBid = (Bid) session.createQuery("from Bid b order by b.amount desc")
    .setMaxResults(1).uniqueResult();

Bid bid = (Bid) session.createCriteria(Bid.class)
    .add(Expression.eq("id",id).uniqueResult();
```

Paramétrage des requêtes

- On évitera de construire une règle sous la forme d'une String prêt à l'emploi : on utilise l'instanciation de paramètres
  - en nommant les paramètres

```
String query="from Item item where item.description like :searchString;
List result = session.createQuery(query)
    .setString("searchString",searchString).list();
ou  .setParameter("searchString",searchString,Hibernate.STRING).list();
```

  - en positionnant les paramètres

```
String query = "from Item item where item.description like ? and item.date > ?";
List result = session.createQuery(query).setString(0,searchString).setDate(1,minDate)
    .list();
```

Requête de base

from Bid ⇔ Session.createCriteria(Bid.class)
est traduit en
select B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED from BID B

- Utilisation d'Alias

from Bid as bid (ou Bid bid)

- requête Polymorphique

from BillingDetails
et pour avoir les objets concrets des sous-classes :
from CreditCard

- Filtrage : via la clause WHERE
  - opérateur de comparaison classique
  - opérateur arithmétique
  - String matching LIKE
  - opérateur logique
  - opérateur IS NULL

from User u where u.email is not null
⇔
session.createCriteria(User.class).add(Expression.isNotNull("email")).list();

Jointure

- 4 manières d'exprimer des jointures
  - jointure ordinaire dans la clause from
  - Fetch jointure
  - Theta-style
  - implicite
- Jointure en mode Fetch (jointure externe optimisée)
  - mode HQL

```
from Item item
left join fetch item.bids where item.description like "%gc%"
```

  - mode critère

```
session.createCrieria(Item.class).setFetchMode("bids",FetchMode.EAGER)
    .add(Expression.like("description","gc",MatchMode.ANYWHERE)).list();
```

  - traduction SQL

```
select I.DESCRIPTION, I.CREATED, I.SUCCESSFUL_BID, B.BID_ID,
    B.AMOUNT,B.ITEM_ID,B.CREATED
from ITEM I left outer join BID B on I.ITEM_ID=B.ITEM_ID
where I.DESCRIPTION like '%gc%'
```

Jointure ordinaire

- Une jointure classique utilise l'opérateur join et les Alias:

```
from Item item join item.bids bid
where item.description like '%gc%' and bid.amount>100
```

- A la différence du Fetch mode, l'association n'est pas reformatée : le résultat est un tableau de paires (Item, Bid)

```
Query q= session.createQuery("from Item item join item.bids bid");
Iterator pairs=q.list().iterator();
while (pairs.hasNext() ) {
      Object[] pair=(Object[]) pairs.next();
      Item item=(Item)pair[0]; Bid bid=(Bid)pair[1];
}
```

Jointure implicite

- Permet une interrogation à la manière OQL (BDOO)
  - pour interroger des objets composants
  - pour naviguer dans des associations implicites

```
from User u  where u.address.city = 'Grenoble'
```

⇔

```
session.creatCriteria(User.class)
    .add(Expression.eq("address.city","Grenoble"));
```

```
from Bid bid where bid.item.category.name like 'Laptop%'
```

Jointure Theta-Style

- Utile pour réaliser une jointure avec un critère ne faisant pas intervenir des attributs liés par une association

```
from User user, LogRecord log
where user.username = log.username
```

qui s'utilise via un Iterator

```
Iterator i= session.createQuery("from User user, LogRecord log
where user.username = log.username").list().iterator();
while (i.hasNext() ) {
    Object[] pair=(Object[]) i.next();
    User user=(User)pair[0]; LogRecord log=(LogRecord)pair[1];
}
```

Agrégation

- Hibernate permet d'écrire via HQL des projections, des Group By Having, des fonctions d'agrégations, la suppression de doublon (distinct)

```
Select p.LASTNAME, count(A)
        From Person p join Addresse a
        group by p.LASTNAME
        having count(a)>10
```

Notion de sous-requêtes

- Hibernate est l'un des rares ORM a proposé des sous requêtes dans les clauses FROM et WHERE

```
from User u where 10<(
    select count(i) from u.items i where i.successfulBid is not null )

from Bid bid where bid.amount +1 >= (
    select Max(b.amount) from Bid b )
```

- Opérateurs ANY, ALL, SOME, IN disponibles.

Gestion des transactions

Définition d'une transaction

- Une transaction est définie entre les appels beginTransaction() et commit()

```
Session session=sessions.OpenSession();
Transaction tx=null;
try {
      tx=session.beginTransaction();
      concludeAuction();
      tx.commit();
} catch (Exception e) {
      if (tx!=null) {
            try { tx.rollback(); }
            { catch (HibernateException he {…}
      } throw e;
} finally {
      try { session.close(); }
      catch (HibernateException he) { throw he;}
}
```

Niveau d'isolation

- Par défaut celui de JDBC (soit read committed soit repeatable read)
- Option de configuration (donc pour tout le pool de connections):

  Hibernate.connection.isolation = x

- Où x peut être :
  - 1 - Read uncommitted
  - 2 - Read commited
  - 3 - Repeatable read
  - 8 - Serializable