Master 2 - Informatique des Organisations Parcours MIAGE
IF (Informatique pour la Finance) en Apprentissage

Persistance des objets et bases de données relationnelles (*Object Relational Mapping*)

Maude Manouvrier

- Partie 1 : Introduction - Problématique générale
- Partie 2 : Non correspondance des modèles
- Partie 3 : Modèle de persistance DAO (*Data Access Object*)
- Partie 4 : *Hibernate* - *Java Persistance API* (JPA)

Bibliographie

- [BK07] ***Java Persistence with Hibernate, Revised Edition of Hibernate in Action***, de Christian Bauer and Gavin King, Manning Publications, 2007 – nouvelle édition en nov. 2015.
- ***Hibernate Recipes 2015 : A Problem-Solution Approach***, de Gary Mak Srinivas Guruzu , et Joseph Ottinger, 2nd Revised edition, Springer-Verlag 2015
- [Pat08] ***Java Persistence et Hibernate,*** d'Anthony Patricio, Eyrolles, 2008
- [BK05] ***Hibernate,*** de Christian Bauer et Gavin King, Campus Press, 2005 – *Traduction de Hibernate in Action* des mêmes auteurs
- [Pat05] ***Hibernate 3.0 : Gestion optimale de la persistance dans les applications Java/J2EE,*** de Anthony Patricio, Eyrolles, 2005
- [Fow03] ***Patterns of Entreprise Application Architecture***, de Martin Fowler, Addison Wesley, 2003
- [Spe03] ***Java Persistence for Relational Databases***, de Richard Sperko, Apress, 2003
- Également utilisé pour préparer ce cours : ***Gestion de la pesistance avec Hibernate*** - Manuel de Cours, de Valtech Training, 2005

Documents en ligne

- **Documentation Hibernate en ligne** : http://hibernate.org/docs
- **Traduction en français de la Documentation Hibernate en ligne** : http://docs.jboss.org/hibernate/core/3.6/reference/fr-FR/html/
- ***Mapping Objects to Relational Databases: O/R Mapping In Detail*, de Scott W. Ambler, 2006**
  http://www.agiledata.org/essays/mappingObjects.html
- [Fus97] ***Foundations of Object-Relational Mapping*, de Mark L. Fussell, 1997**
  http://markfussell.emenar.com/blog/object-relational/

Quelques tutoriaux :

- http://cyrille-herby.developpez.com/tutoriels/java/mapper-sa-base-donnees-avec-pattern-dao/
- http://arodrigues.developpez.com/tutoriels/java/performance/hibernate-performance-part1-strategies-chargement/
- http://bmarchesson.developpez.com/tutoriels/java/hibernate/chargement/
- http://objetdirect.developpez.com/articles/java/hibernate/strategies-heritage/
- http://java.developpez.com/faq/hibernate/
- http://courses.coreservlets.com/Course-Materials/hibernate.html
- http://jmdoudoux.developpez.com/
- http://www.info.univ-angers.fr/~richer/ens/m2cdsii/crs_orm.pdf
- http://orm.bdpedia.fr/bd.html

Partie 1 : Introduction

- **Problématique générale**
- **Définition de la persistance**
- **Définition de l'ORM (*Object/Relational Mapping*)**
- **Architecture multi-couches**
- **Couche de persistance**
- **Solutions ORM**

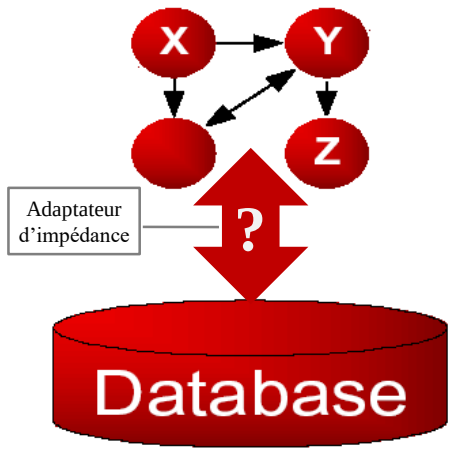
Persistance des objets et bases de données relationnelles

- Majorité de bases de données relationnelles (position dominante sur le marché, théorie solide et normes reconnues)
- Nombreuses applications développées en langage de programmation orienté-objet
- Modélisation UML

⇓

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

Problématique



Propriétés à conserver :

- Objets complexes
- Identification des objets
- Encapsulation
- Classes
- Hiérarchie de classes
- Polymorphisme
- Navigation dans le graphe d'objets
- Cache objet

Propriétés à ajouter :

- Persistance
- Interrogation
- Gestion de la concurrence
- Sécurité et reprise après panne
- Gestion de la mémoire secondaire

Persistance

- Mécanisme permettant à un objet de survivre au processus qui l'a créé [BK05]
- Caractéristiques :
  - Stockage, organisation et récupération des données structurées (tri, agrégation)
  - Concurrence et intégrité des données
  - Partage des données

ORM : *Object/Relational Mapping*

- **Persistance automatisée et transparente d'objets métiers vers une bases de données relationnelles** [BK05]
- Description à l'aide de **méta-données** de la **transformation réversible** entre un modèle relationnel et un modèle de classes [BK05, Pat05]
- Capacité à manipuler des données stockées dans une base de données relationnelles à l'aide d'un langage de programmation orientée-objet
- Techniques de programmation permettant de lier les bases de données relationnelles aux concepts de la programmation OO pour créer une "base de données orientées-objet virtuelle" [Wikipedia]

Architecture multi-couches

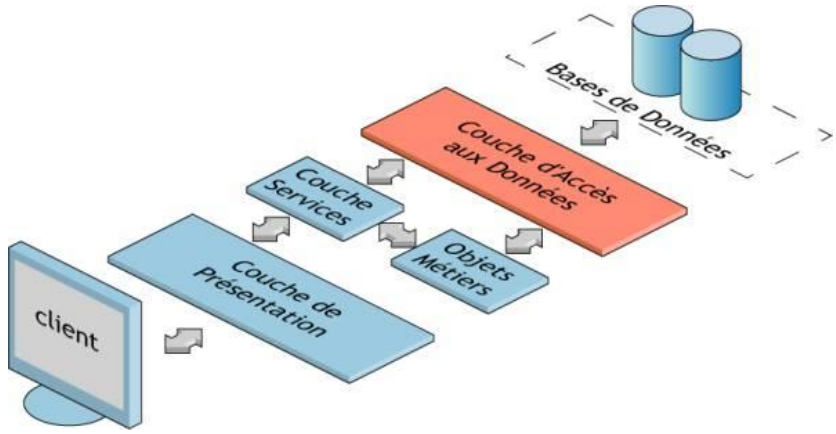


Figure issue de [Ros03]

Architecture multi-couches

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

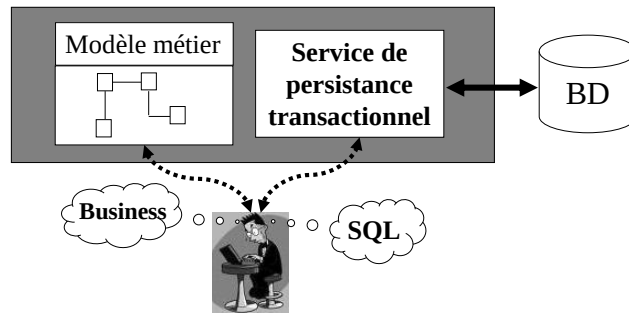
Couche d'accès au données
Couche de persistance

- Prise en charge de toutes les interactions entre l'application et la base de données [Ros03]
- Groupes de classes et de composants chargés du stockage et de la récupération des données [BK05]
- Possibilité de servir de cache pour les objets récupérés dans la base de données pour améliorer les performances [Gri09]
- Inclus un modèle (de métadonnées) des entités du domaine métier

Couche de persistance : à la charge du développeur (1/3)

- Possibilité de programmer manuellement une couche de persistance avec SQL/JDBC (*Java DataBase Connectivity*)
- Possibilité de masquer le JDBC complexe et le SQL non portable à la logique métier par un modèle de conception (ex. *active record*)

Stratégie de persistance non transparente



Couche de persistance : à la charge du développeur (2/3)

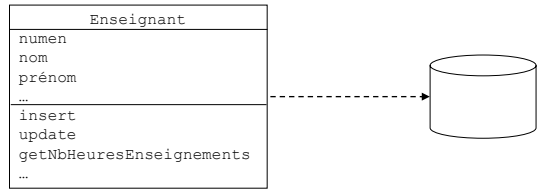
2 niveaux de qualité [Fus97] :

- **Relationnel pur** :
  - Application entièrement conçue autour du modèle relationnel et des opérations relationnelles réalisées en SQL
  - Modèle utilisé dans le cas d'applications simples sans nécessité de réutilisation de code
  - Utilisation d'*Embedded SQL* ou *SQLJ* et de procédures stockées ⇒ décharge d'une partie du travail de la couche métier vers la base de données
  - **Manque de portabilité et de maintenance à long terme**
- **Correspondance objet légère (*Light Object Mapping*)** :
  - Correspondance codée manuellement entre les classes et les relations de la base de données
  - Masquage du SQL/JDBC programmé manuellement par l'utilisation de modèles de conception (*design pattern*) connu – ex. *active record*
  - Utilisée pour des applications ayant un petit nombre d'entités
  - Utilisation de procédures stockées
  - **Couplage trop fort entre les classes métiers et le support de persistance utilisé**

Couche de persistance : à la charge du développeur (3/3)

***Active record* (motif/patron de conception – *design pattern*)** :

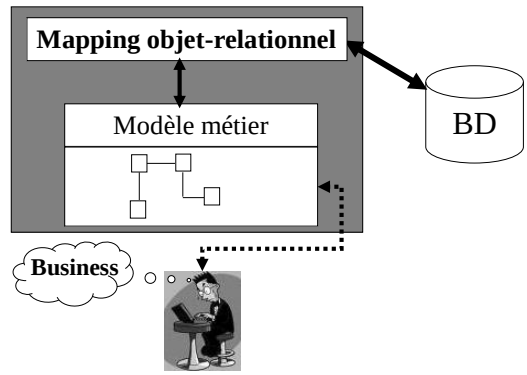
- « *An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data* »
- Partie "Modèle" de l'architecture "Modèle Vue Contrôleur"
- Correspondance de chaque relation de la base avec la définition d'une classe : chaque colonne de la relation = une propriété de la classe
- Correspondance de chaque nuplet de la relation avec une instance de la classe correspondante :
  création d'un nouvel objet ⇒ insertion d'un nouvel nuplet dans la relation
- Méthodes statiques de classes agissant sur l'ensemble des nuplets
- Requêtes CRUD (*Create, Read, Update, Delete*) pouvant être générées automatiquement



Couche de persistance : avec correspondance objet/relationnel (1/3)

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

Stratégie de persistance « transparente »



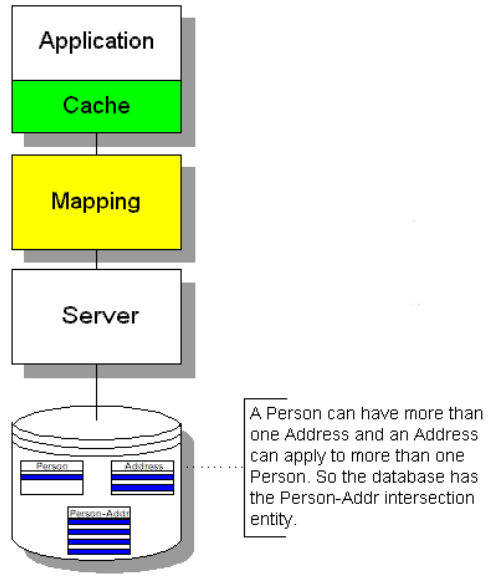
Couche de persistance : avec correspondance objet/relationnel (2/3)

2 niveaux de qualité [Fus97] :

- **Correspondance objet moyenne (*Medium Object Mapping*)** :
  - Application conçue autour d'un modèle objet
  - SQL généré à la compilation par un outil de génération de code ou à l'exécution par le code de l'outil de correspondance (*mapping framework*)
  - Objets mis en cache par la couche de persistance
  - Pour des applications de taille moyenne incluant des transactions complexes
  - *Ex. EJB1.x/2.x Entity Beans*
- **Correspondance objet complète (*Full Object Mapping*)** :
  - Prise en compte de toutes les propriétés objets : composition, héritage, polymorphisme, persistance par accessibilité
  - Persistance *presque* transparente pour le développeur
  - Pas d'héritage d'une classe de base par les classes persistantes, ni d'interface spéciale
  - Stratégie d'extraction efficace (précoce ou tardive) et stratégies de mise en cache implémentées de manière transparente
  - *Ex. EJB 3.0 Persistency*

Couche de persistance : avec correspondance objet/relationnel (3/3)

Persistance *presque* transparente



Repris de http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html

Solutions ORM (1/2)

Normes Java :

- **EJB (*Entreprise Java Beans*)** :
  - Gestion de la persistance par conteneur (CMP- *Container-Managed Persistence* et BMP – *Beans Managed Persistence*)
  - Spécifications EJB3.0 (JSR 220 Mai 2006)
- **JPA (*Java Persistence API*)** : Partie des spécifications EJB 3.0 (JSR 220 en Mai 2006 - JSR 316 en cours) concernant la persistance des composants
- **JDO (*Java Data Object*)** :
  - Spécification de Sun 1999 – JDO 2.0 (JSR243 Mars 2006)
  - Abstraction du support de stockage
  - Implémentation libre : DataNucleus (ex. JPOX) – dernière version 2015

Implémentation de JPA :

- **Hibernate** (***JBoss***) : Solution libre faisant partie du serveur d'appli. JBoss – version 5.2 (10/2017) implémentant les spécifications JSR 220 – complète et bien documentée - plugin Eclipse - Gavin King (fondateur) membre de groupes d'expert d'EJB3
- **TopLink** (***Oracle***) : Solution propriétaire utilisée par la serveur d'application d'Oracle - **TopLink Essentials** *:* version libre disponible dans *Netbeans* 5.5 ou le serveur d'application (Java EE 5) *Glassfish* de Sun, intégrée dans le projet *EclipseLink* (version 1 07/2008)
- **EclipseLink** (issu de Toplink) – dernière version 2013

Autres outils :

- **Outils pour d'autres langages :** SQLAlchemy, SQLObject pour Python, nHibernate pour .Net …
- **cf. https://en.wikipedia.org/wiki/Comparison_of_object-relational_mapping_software**

Solutions ORM (2/2)

Objectifs :

Automatiser et faciliter la correspondance entre les données stockées dans des objets et une base de données relationnelles

- Recherche et enregistrement des données associées à un objet dans une base de données
- Détection de la modification d'un objet et enregistrement des mises à jour en optimisant les accès à la base

\+ Moins de code répétitif à écrire : gain de 30 à 40% du nombre de lignes de code

\+ Amélioration de la portabilité du code en cas de changement de SGBD

\+ Développement objet (sans penser en terme relationnel)

\+ Possibilité d'avoir un modèle objet fin (pouvant nécessiter un codage à la main complexe pour la persistance)

\+ Refactorisation *(Refactoring)* du schéma de la base de données ou du modèle objet facilité

\- Pas optimal pour des applications modifiant beaucoup de nuplets à chaque `update` ou comportant essentiellement des requêtes d'agrégation (`group by`) – Ex. OLAP ou data mining

Partie 2 : Non correspondance des modèles objet et relationnel

- **Définition de l'*Impedance mismatch***
- **Exemple simple de correspondance**
- **Identification des objets**
- **Traduction des associations**
- **Traduction de l'héritage**
- **Navigation dans le graphe d'objets**
- **Objet dépendant**

Non correspondance des modèles objet et relationnel

Impedance mismatch

- Entrée : modèle objet
- Sortie : modèle relationnel
- Problèmes de correspondance :
  - Identification des objets
  - Traduction des associations
  - Traduction de l'héritage
  - Navigation entre les objets
  - Objets dépendants

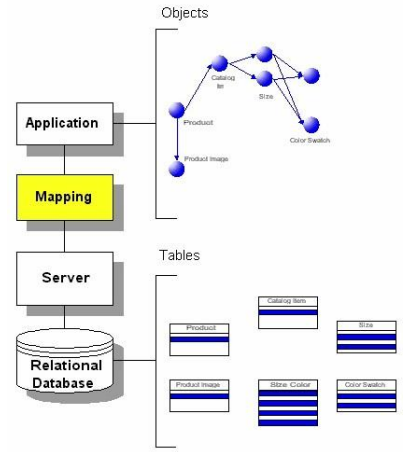


Figure reprise de http://www.service-architecture.com/object-oriented-databases/articles/impedance_mismatch.html

Exemple simple de correspondance

Implémentation **POJO (*Plain Old Java Object*)** de la classe `Departement` :

```
public class Departement implements java.io.Serializable {
  // Fields
  private int departementId;
  private String nomDepartement;
  /** default constructor */
  public Departement() {}
  /** full constructor */
  public Departement(int departementId, String nomDepartement) {
    this.departementId = departementId;
    this.nomDepartement = nomDepartement;
  }
  // Property accessors
  …
}
```

Relation de bases de données `Departement` :

```
CREATE TABLE Departement
(
  departement_id int4 NOT NULL,
  nom_departement varchar(25) NOT NULL
)
```

Clé primaire ??

departement_id int4	nom_departement varchar
1	MIDO
2	LSO
3	MSO
4	LANGUES

1. Identification des objets (1/5)

Dans le monde objet :

- Identification des objets par l'adresse de leur emplacement mémoire
- 2 notions différentes :
  - Identité d'objets : `a==b` (notion définie par la machine virtuelle Java)
  - "Égalité" ou équivalence : méthode `equals()` à implémenter dans la classe

Dans le modèle relationnel :

- Identification des nuplets par la valeur de la clé primaire
- Pas de possibilité d'avoir deux nuplets avec les mêmes valeurs dans une relation

1. Identification des objets (2/5)

En cas d'utilisation d'un SGBD relationnel-objet (ex. PostgreSQL) :

- Ajout automatique d'un attribut OID pour distinguer les nuplets (utilisé comme clé primaire dans les relations système)
- Possibilité de le choisir comme clé primaire dans une relation utilisateur

```
CREATE TABLE departement
(
  departement_id int4 NOT NULL,
  nom_departement varchar(25) NOT NULL,
  CONSTRAINT pk_departement PRIMARY KEY (oid)
)
WITH OIDS;
```

oid	departement_id int4	nom_departement varchar
18494	1	MIDO
18495	2	LSO
18496	3	MSO
18497	4	LANGUES

1. Identification des objets (3/5)

En cas d'utilisation d'un SGBD relationnel pur :

Préférer les clés primaires sans contrepartie dans le monde réel ou clé de substitution (ou artificielle - *surrogate key*)

- Pour éviter les clés composées de plusieurs attributs
- Pour faciliter l'indexation (ex. un entier incrémenté automatiquement)
- Pour faciliter les mises à jour (ne pas avoir à changer toutes les clés étrangères y faisant référence)
- Pour éliminer le lien avec le modèle métier

Penser néanmoins à assurer la cohérence des données en ajoutant une contrainte d'unicité pour les attributs identificateurs métier

1. Identification des objets (4/5)

Objet persistant = représentation en mémoire d'un nuplet

Un même nuplet ne doit pas être représenté par plusieurs objets en mémoire centrale pour une même session de travail

Exemple :
Création en mémoire d'un objet `e1` *de la classe* `Enseignant` *(à l'occasion d'une navigation à partir d'un objet* `Enseignement`*)*
Possibilité de retrouver le même enseignant depuis un autre `Enseignement` *ou depuis un* `Département`
⇒ *Ne pas créer d'objet* `e2` *en mémoire centrale indépendant de* `e1`

⇒ Utilisation du **cache**

- "Index" des objets créés en mémoire
  (avec conservation de l'identité relationnelle – clé primaire)
- Recherche dans le cache avant toute récupération dans la base

1. Identification des objets (5/5)

Différence de granularité entre les deux modèles

- Modèle objet de granularité plus fine :
  - " + de classes que de relations "
  - Instances de plusieurs classes (dépendantes) sauvegardés dans la même relation

Ex. 

⇒ *Pas d'obligation de créer une relation* `Adresse` *dans la base*
Si insertion des attributs de Adresse dans la relation Utilisateur ⇒ des objets `Adresse` *sans identification liée à la base*

- Possibilité d'utiliser les *User Defined Type* de SQL99 – mais problème de compatibilité et de standardisation dans les SGBD

2. Traduction des associations (1/5)

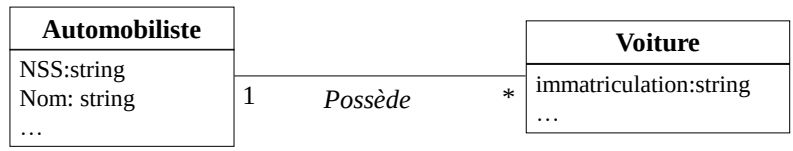
Dans le monde objet :

- Association = ensemble de liens entre objets
- 3 types de représentation :
  - variable d'instance de type objet (multiplicité 1 ou *..1)
  - variable d'instance de type collection d'objets (multiplicité 1..* ou *)
  - Classe-association (multiplicité *)
- Références d'objet par nature unidirectionnelle
- Pas de déduction de la multiplicité à partir de la classe Java (si codage unidirectionnel)

Dans le modèle relationnel

- Représentation des associations par :
  - Une ou plusieurs clés étrangères (multiplicité 1 ou *..1 ou 1..*)
  - Une relation dont la clé primaire est composée de clés étrangères (multiplicité *) : table de liaison ou table d'association
- Bi-direction par jointure entre les relations
- Déduction de la multiplicité par analyse de la définition des clés étrangères

2. Traduction des associations (2/5)



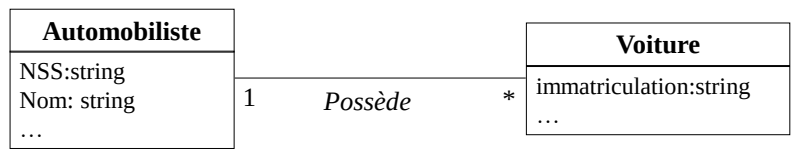
De Automobiliste vers Voiture

```
public class Automobiliste {
  // Fields
  private String NSS;
  private String nom;
  private Collection<Voiture> parc_automobile;
  …
}
```

De Voiture vers Automobiliste

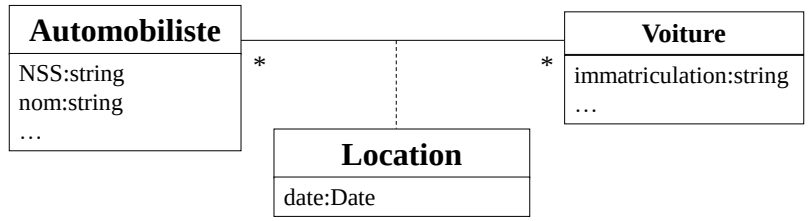
```
public class Voiture {
  // Fields
  private String immatriculation;
  private Automobiliste proprietaire;
  …
}
```


2. Traduction des associations (3/5)




```
CREATE TABLE Automobilsite
( Automobiliste_ID  SERIAL,
  NSS varchar(10) NOT NULL,
  Nom varchar(20) NOT NULL,
  …
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Automobiliste_ID),
) ;

CREATE TABLE Voiture
( Voiture_ID  SERIAL,
  Immatriculation varchar(10) NOT NULL,
  …
  Proprietaire_ID int NOT NULL,
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Voiture_ID),
  CONSTRAINT FK_Automobiliste_Voiture
   FOREIGN KEY (Proprietaire_ID) REFERENCES Automobiliste (Automobiliste_ID)
);
```

2. Traduction des associations (4/5)




```
public class Location {
  private Automobiliste loueur;
  private Voiture vehicule;
  private Date date;
  …
}
```

```
CREATE TABLE Location
( …
 CONSTRAINT PK_Location PRIMARY KEY (Automobiliste_ID,Voiture_ID, Date),
 CONSTRAINT FK_Location_Voiture
   FOREIGN KEY (Voiture_ID) REFERENCES Voiture (Voiture_ID),
 CONSTRAINT FK_Location_Automobiliste
   FOREIGN KEY (Automobiliste_ID) REFERENCES Automobiliste (Automobiliste_ID),
);
```


2. Traduction des associations (5/5)

- Gestion des associations plus complexe en objet qu'en relationnel

  Ex. Modification d'un propriétaire de voiture

  ⇒ *Modification de la valeur de la clé étrangère dans la relation* `Voiture`

  ⇒ *Suppression du véhicule correspondant dans la collection* `parc_automobile` *de l'objet Java* `Automobiliste`

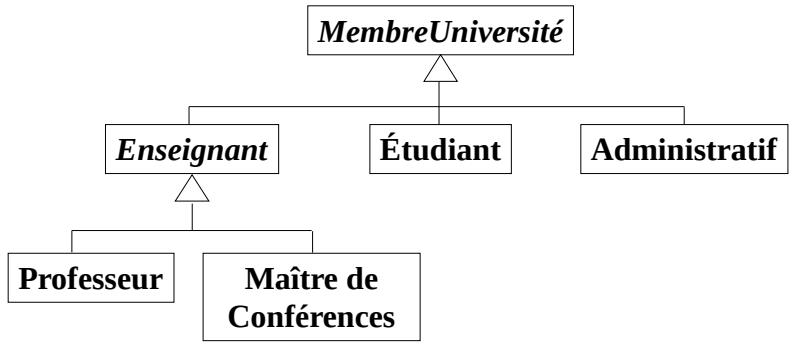
  \+ *mise à jour de l'instance* `proprietaire` *de l'objet* `Voiture`

- Gestion automatique dans EJB 2.x
- Choix de ne rien automatiser dans EJB 3.0

3. Traduction de l'héritage (1/5)

Plusieurs méthodes :

1. Faire correspondre toutes les classes de la hiérarchie à une seule relation de bases de données
2. Représenter chaque classe (abstraite ou concrète) par une relation
3. Représenter chaque classe concrète par une relation



3. Traduction de l'héritage (2/5)

1. Correspondance de toutes les classes de la hiérarchie avec une seule relation de bases de données :

   Ex. une relation `MembreUniversité` *avec les attributs de chaque classe + un attribut* `type`

   \+ Facile à mettre en place

   \+ Possibilité de requêtes et associations polymorphes

   – Obligation de gérer des valeurs NULL pour plusieurs colonnes

   – Pas de possibilité de déclarer une contrainte NOT NULL sur une de ces colonnes même si la contrainte doit être vérifiée

3. Traduction de l'héritage (3/5)

2. Représentation de chaque classe (abstraite ou concrète) par une relation :

   ⇒ Répartition des attributs d'un objet dans plusieurs relations

   Ex. 1 objet `Professeur` *correspond à un nuplet dans la relation* `Enseignant` *et à un nuplet dans la relation* `Professeur`

   ⇒ Préservation de l'identité en donnant la même valeur de clé primaire à chaque nuplet correspondant à l'objet dans les différentes relations

   ⇒ Clés primaires des relations correspondantes aux classes filles = clés étrangères faisant référence à la clé primaire de la relation correspondant à la classe mère

   Ex. la clé primaire de la relation `Professeur` *et aussi une clé étrangère qui fait référence à la clé primaire de la relation* `Enseignant`

3. Traduction de l'héritage (4/5)

2. Représentation de chaque classe (abstraite ou concrète) par une relation (suite) :

   \+ Simple bijection entre les classes et les relations

   \+ Possibilité de faire des requêtes et associations polymorphes

   – Nombreuses jointures à faire en cas de hiérarchie complexe

   Possibilité de limiter certains problèmes en ajoutant des attributs dans les classes mères :

   Ex. un attribut type dans la relation `MembreUniversité` *pour éviter une jointure lors de la requête « quels sont les noms et prénoms des maîtres de conférences ? »*

   – Problème de performances

   \+ Vérification des contraintes d'intégrité

3. Traduction de l'héritage (5/5)

3. Représentation de chaque classe concrète par une relation :
   - Correspondance de chaque classe concrète avec une relation contenant tous les attributs (même les attributs hérités) de la classe
   - En cas de classe concrète avec des classes filles : Clé primaire des relations correspondantes aux classes filles = clés étrangères faisant référence à la clé primaire de la relation correspondant à la classe mère
   + Pas de jointure pour retrouver les informations
   – Problème pour les associations et requêtes polymorphes
   – Redondance d'information
   - Pas de nécessité d'offrir cette solution dans les spec. EJB 3.0

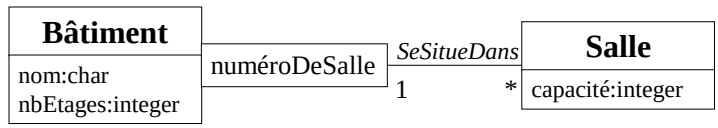
4. Navigation dans le graphe d'objets

Que faire lorsqu'un objet est créé à partir de données récupérées dans la base de données ?

- Récupération immédiate et création des objets associés
  - Risque de récupérer des objets inutiles et mauvaises performances sans raison valable
- Création des objets associés uniquement en cas d'accès par l'application (*lazy loading* ou récupération paresseuse ou à la demande)
  - Problème des « N+1 Selects »
- Choisir le type de récupération en fonction de la requête

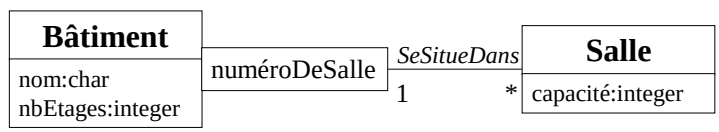
5. Objet dépendant (1/2)

- Objet dont le cycle de vie dépend d'un autre objet (objet propriétaire)

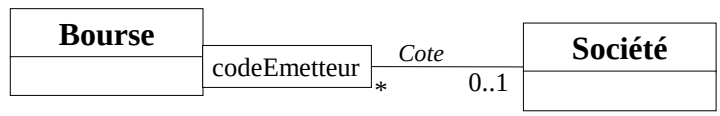


- Clé primaire de la relation correspondant à la classe des objets dépendants constituée de la clé primaire de de la relation correspondant à la classe des objets propriétaires
- Persistance d'un objet dépendant gérée par l'objet propriétaire (ou par le *DAO* de l'objet propriétaire)

5. Objet dépendant (2/2)




```
Batiment(IDBatiment, nom, nbEtages)
Salle(#IDBatiment, numeroSalle, capacité)
```




```
Bourse (IDBourse, …)
Societe(IDSociete, …)
Cotation(#IDBourse, codeEmetteur, #IDSociete)
```

Non correspondance :

- 30% du coût de développement consacré à la mise en correspondance
- Modélisation relationnelle tributaire de la théorie relationnelle
- Modélisation orientée-objet sans définition mathématique rigoureuse ni partie théorique
- Modèles architecturaux ou basés sur les motifs vus comme une solution partielle au problème de non-correspondance : ex. *Entity beans*, *DAO* (*Data Access Object*)
- Réduction du code de correspondance par les outils ORM

Partie 3 : Modèles de persistance
DAO - *Data Access Object*

- **Problématique et définitions**
- **Diagramme de classes du modèle DAO**
- **Diagramme de séquences du modèle DAO**
- **Opérations CRUD**
- **DAO et Exceptions**
- **Utilisation des DAO**
- **DAO et Connexion**
- **DAO et Transaction**
- **DAO et Héritage**
- **Fabrique de DAO**
- ***Présentation du TP JDBC/DAO***

Modèles de persistance : DAO - *Data Access Object* (1/3)

- Problèmes à résoudre :
  - Variation du code de persistance en fonction :
    - Du type de support de persistance (BD relationnelle, BD objet, fichiers, etc.)
    - Des différentes implémentations des fournisseurs de SGBD
  - Difficulté à changer de support de persistance en cas d'imbrication du code de persistance et du code métier
- Solution :
  - Séparation des entrées-sorties des classes métier
  - Utilisation d'un objet particulier - **objet d'accès aux données** - pour abstraire et encapsuler les accès aux données

⇒DAO : Motif de conception – également connu sous le nom de *Data Mapper* (M. Fowler)

Modèles de persistance : DAO - *Data Access Object* (2/3)

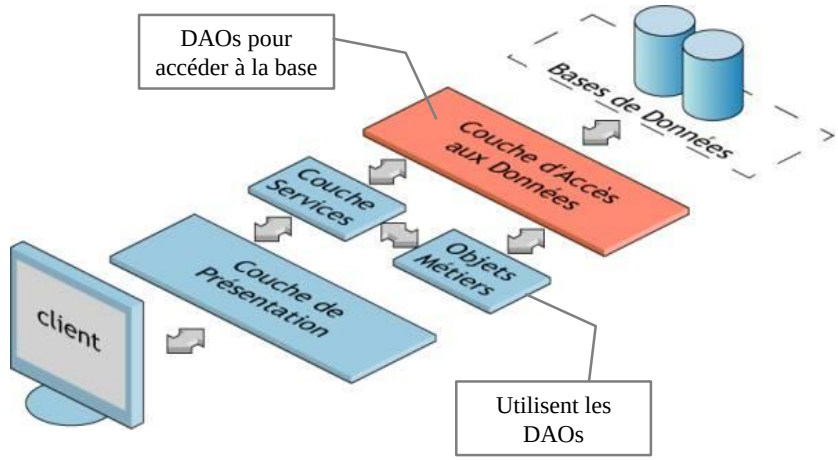


Figure issue de [Ros03] et adaptation de [Gri09]

Modèles de persistance : DAO - *Data Access Object* (3/3)

- Différence entre un DAO et un *active record* :
  « *Les objets manipulant les données n'ont pas accès au code permettant de sauvegarder ces données dans la base* »
- Utilité des DAO :
  - Faciliter la modification du modèle de base de données
  - Factoriser le code d'accès aux données
  - Faciliter l'optimisation des accès à la base en les regroupant au sein d'objets particuliers
- Placement des DAO dans la couche d'accès aux données

Diagramme de classes du modèle de conception DAO (1/2)

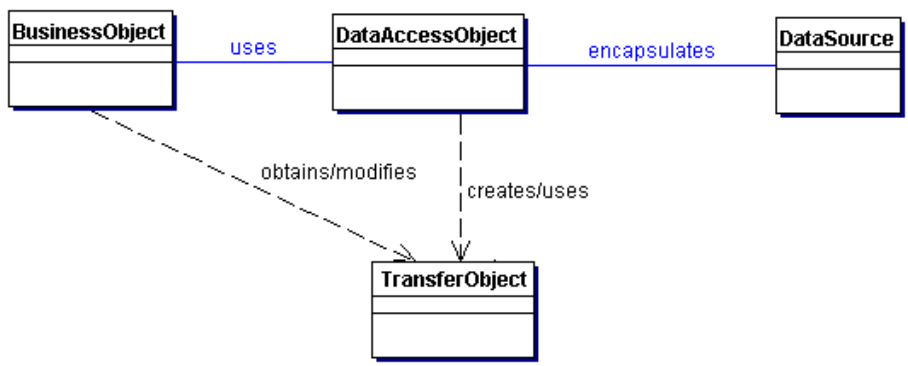


Figure reprise de ***Core J2EE Patterns - Data Access Object 2001-2002 de Sun Microsystems***
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

Diagramme de classes du modèle de conception DAO (2/2)

- **Objet métier (*Business Object*)** :
  - Données vues par le client
  - Objet nécessitant l'accès et le stockage de données depuis le support de persistance
  - Objet pouvant être implémenté par une *EJB session*, EJB entité, or tout autre objet Java en association avec un servlet ou une *helper bean* qui accèdent à la source de données.
- **Objet d'accès aux données (*Data Access Object*)** :
  - Abstraction de l'implémentation de l'accès aux données pour rendre cet accès transparent vis à vis de l'objet métier
  - Délégation des opérations de sauvegarde et de récupération de l'objet métier vers l'objet d'accès aux données
- **Source de données (*Data source*)** : BD relationnelle, BD objet, fichiers, etc.
- **Objet de transfert (*Data Transfert Object - DTO*)** : Objet de transport des données entre la couche d'accès aux données et la couche métier

Diagramme de séquences du modèle DAO

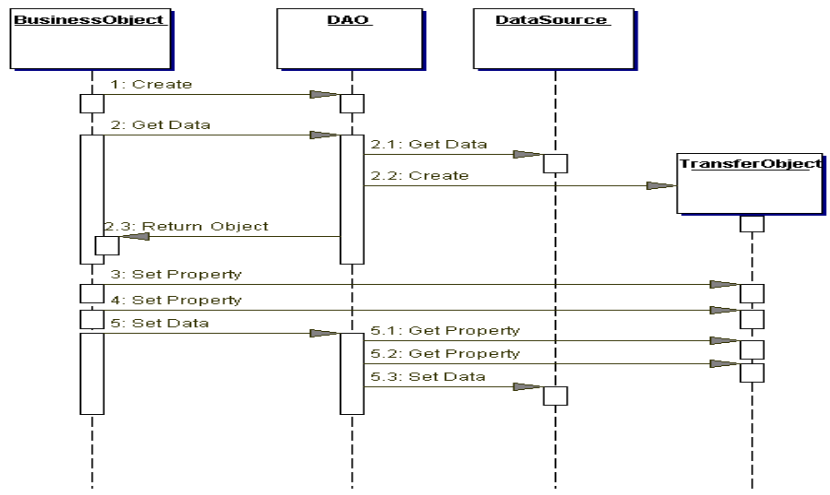


*Figure reprise de **Core J2EE Patterns - Data Access Object 2001-2002 de Sun Microsystems** http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html*

Opérations CRUD (1/5)

Implémentation par les DAO des 4 opérations de base de persistance CRUD :

- ***C****reate* : création d'une nouvelle entité
- ***R****etrieve* : recherche d'une ou plusieurs entités
- ***U****pdate* : modification d'une entité
- ***D****elete* : suppression d'une entité

Plusieurs variantes de signatures pour ces méthodes dans les DAO

Opérations CRUD (2/5)

Create (Persist)

- Paramètres : l'état de la nouvelle entité transmis par
  - Une série de paramètres de type des attributs – solution la plus courante
  - Un objet DTO
  - Un objet métier à rendre persistant
- Type retour :
  - `void`
  - `boolean` - pour indiquer si la création a pu avoir lieu
  - Un identificateur d'entité – utile en cas de génération automatique d'identificateur
  - Un objet métier ou un DTO correspondant à l'entité créée

Opérations CRUD (3/5)

Retrieve (Find)

- Retour d'un objet
  - Paramètre : identificateur de l'entité recherchée
  - Retour : un objet métier ou un DTO contenant les données de l'entité recherchée
- Retour d'une collection d'objets
  - Paramètre : objet ou valeurs « critères de recherche » ou un objet exemple
  - Retour : `ResultSet` ou `RowSet` ou collection – (`Collection`, `Set`, `List` …) d'objets métier ou DTO
- Retour d'une valeur calculée
  - Possibilité de calculer une valeur à partir des objets en mémoire
  - Meilleur choix : interroger directement la base

Opérations CRUD (4/5)

Update

- Diverses variantes de paramètres :
  - Identificateur + valeurs (plusieurs paramètres pour les valeurs ou un DTO)
  - Objet métier correspondant à l'entité à modifier dans la base
- Type retour :
  - `void`
  - `boolean` - pour indiquer si la mise à jour a pu avoir lieu

Opérations CRUD (5/5)

Delete

- Diverses variantes de paramètres :
  - Identificateur de l'entité à supprimer dans la base
  - Objet métier correspondant à l'entité à supprimer dans la base
- Type retour :
  - `void`
  - `boolean` - pour indiquer si la suppression a pu avoir lieu

DAO et Exceptions

- Possibilité de lancer ou attraper des exceptions dans les méthodes des DAO car opérations d'entrées/sorties
- Ne pas lier les exceptions à un type de DAO particulier pour pouvoir facilement en changer

Utilisation des DAO (1/3)

2 stratégies :

- DAO référencé par chaque objet métier pour sa propre persistance
  - Aucune connaissance des DAO par les programmes qui manipulent les objets métiers
  - Nécessité d'une référence vers le DAO utilisé (ex. obtenue par une méthode `static` de la classe DAO)
- DAO directement manipulés par les programmes qui manipulent les objets métier
  - Pas de référence aux DAO par les objets métier
  - Stratégie la plus souvent utilisée
  - Perte de la pureté de la programmation OO

Utilisation des DAO (2/3)

Exemple pour la stratégie 1 :

```
class Departement {
      static private DepartementDAO dao;
      public void devientPersistant() {
            dao = getDAO();
            dao.insertOrUpdate(this);
      }
      private DepartementDAO getDAO() {
            if (dao == null)
                  dao=DepartementDAO.getDAO();
            return dao;
      }
...
}

Departement dept;
...
dept.devientPersistant();
```

Référence au DAO

Possibilité de passer un DTO

Penser à gérer les exceptions

En dehors de la classe métier, le DAO n'est pas visible

Utilisation des DAO (3/3)

Exemple pour la stratégie 2 :

```
DepartementDAO deptDAO = DepartementDAO.getDAO();

int idDepartement = deptDAO.create("1","MIDO",...);

Departement dept = deptDAO.findById(idDepartement);

deptDAO.update(idDepartement, ...);
```

DTO ou objet métier

Nouvelles valeurs pour l'entité Departement

ou bien

```
dept.setNom("Mathématiques,
Informatique, Décision et
Organisation");

deptDAO.update(dept);
```

```
List<Departement> liste = deptDAO.findAll();
```


DAO et Connexion

- Connexion et déconnexion au sein des méthodes des DAO
  - \- Coût élevé sans utilisation de pool de connexions
- Méthodes spécifiques de connexion et de déconnexions par DAO
  - \+ Utilisation d'une même connexion par plusieurs méthodes du DAO

DAO et Transaction

- Le + souvent : début et fin de la transaction gérée au sein de chaque méthode de DAO
- Dans les cas complexes : intervention au sein d'une même transaction de plusieurs méthodes de DAO associés à différents types d'objets métier

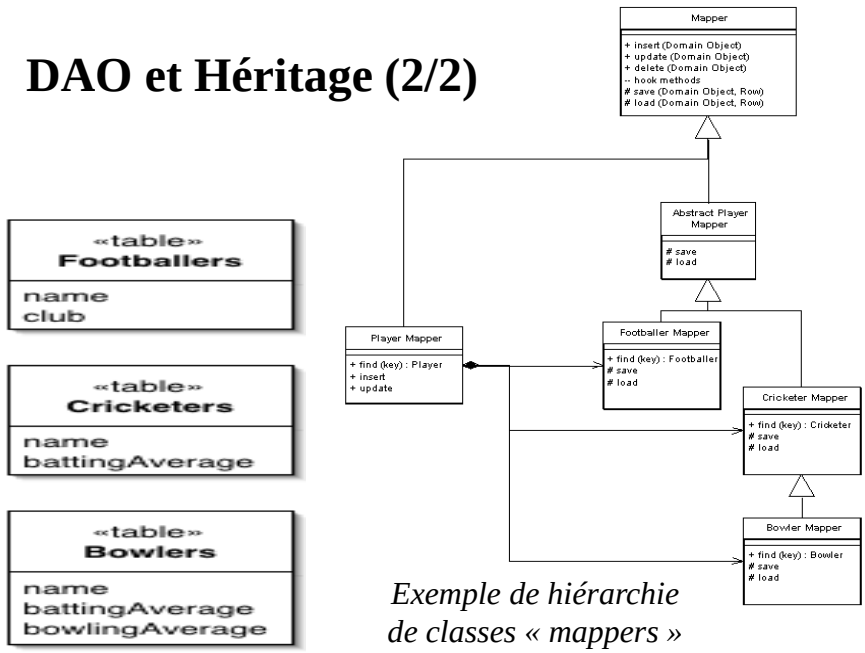
  $\Rightarrow$Gestion des transactions par le client

DAO et Héritage (1/2)

- Problème : rendre persistantes via une classe fille des propriétés `private` de sa classe mère
- Solution : Utilisation d'une hiérarchie parallèle de classes « mappers »

  Transmission au DAO des données nécessaires pour les échanges avec la BD par chaque classe « mapper »

DAO et Héritage (2/2)



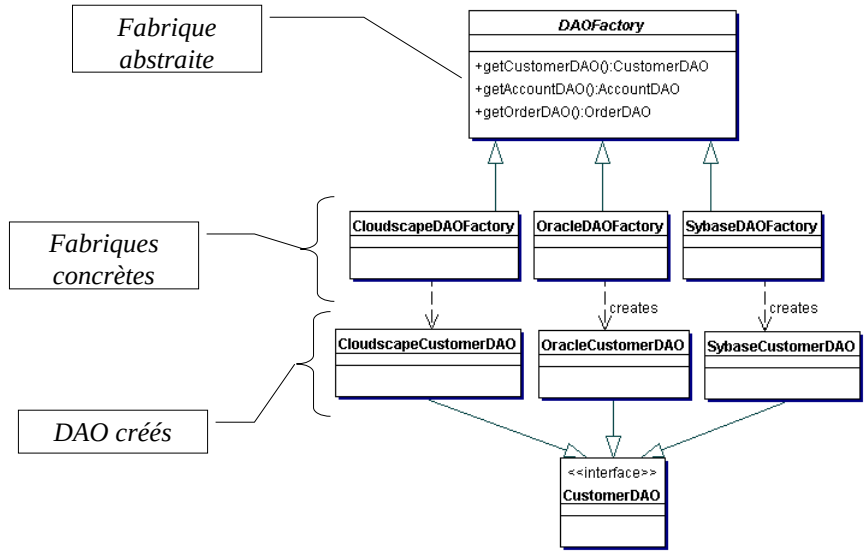
Exemple de hiérarchie de classes « mappers »

Fabrique de DAO (1/10)

DAO Factory :

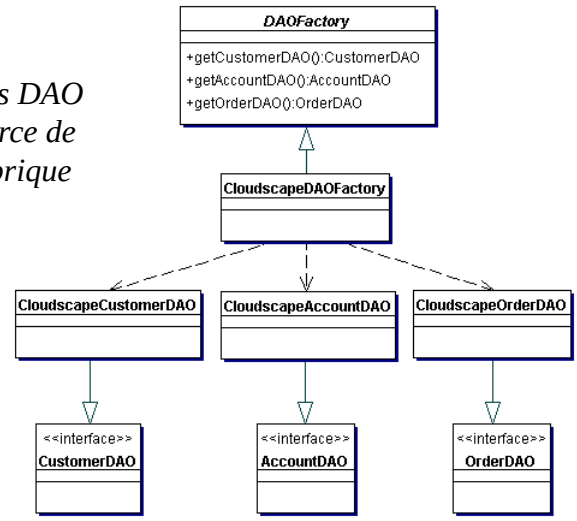
- Utilisation d'une fabrique pour créer/récupérer un DAO et donc cacher le type concret de la classe d'une instance en création
- Fabrique abstraite : pour cacher le type réel d'un ensemble de fabriques concrètes
- Fabriques concrètes : pour fournir tous les DAO (de chaque objet métier) associés à une certaine source de données

Fabrique de DAO (2/10)



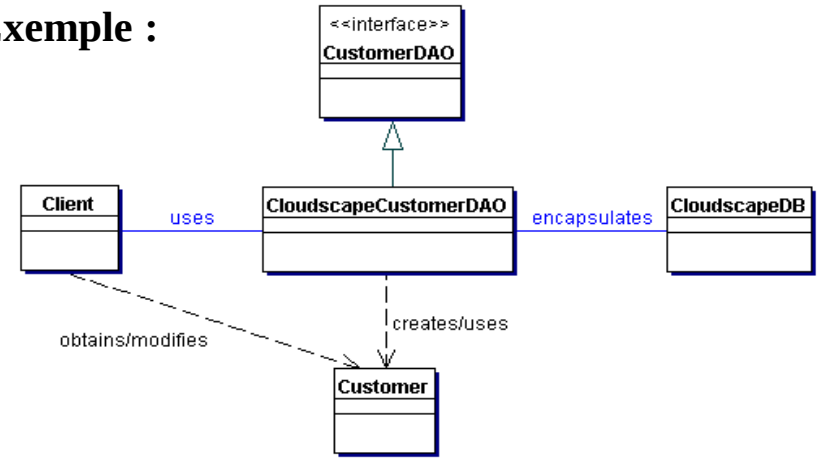
Fabrique de DAO (3/10)

Création de tous les DAO associés à une source de données par la fabrique concrète



Fabrique de DAO (4/10)

Exemple :



Fabrique de DAO (5/10)

```
// Abstract class DAO Factory
public abstract class DAOFactory
 { // List of DAO types supported by the factory
   public static final int CLOUDSCAPE = 1;
   public static final int ORACLE = 2;
   public static final int SYBASE = 3; ...

   // There will be a method for each DAO that can be created.
   // The concrete factories will have to implement these methods.
   public abstract CustomerDAO getCustomerDAO();
   public abstract AccountDAO getAccountDAO();
   public abstract OrderDAO getOrderDAO(); ...

    public static DAOFactory getDAOFactory(int whichFactory) {
        switch (whichFactory) {
                case CLOUDSCAPE: return new CloudscapeDAOFactory();
                case ORACLE : return new OracleDAOFactory();
                case SYBASE : return new SybaseDAOFactory(); ...
                default : return null; }
    }
}
```

Fabrique de DAO (6/10)

```
// Cloudscape concrete DAO Factory implementation
public class CloudscapeDAOFactory extends DAOFactory {

        // method to create Cloudscape connections
        public static Connection createConnection() {
                // Use DRIVER and DBURL to create a connection
                // Recommend connection pool implementation/usage
        }

        public CustomerDAO getCustomerDAO() {
                // CloudscapeCustomerDAO implements CustomerDAO
                return new CloudscapeCustomerDAO();
        }

        public AccountDAO getAccountDAO() {
                // CloudscapeAccountDAO implements AccountDAO
                return new CloudscapeAccountDAO();
        }

        public OrderDAO getOrderDAO() {
                // CloudscapeOrderDAO implements OrderDAO
                return new CloudscapeOrderDAO();
        }

...
}
```


Fabrique de DAO (7/10)

```
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
      public int insertCustomer(...);
      public boolean deleteCustomer(...);
      public Customer findCustomer(...);
      public boolean updateCustomer(...);
      public RowSet selectCustomersRS(...);
      public Collection selectCustomersTO(...); ...
}
```

Fabrique de DAO (8/10)

```
// CloudscapeCustomerDAO implementation of the CustomerDAO interface.
// This class can contain all Cloudscape specific code and SQL statements.
// The client is thus shielded from knowing these implementation details.
public class CloudscapeCustomerDAO implements CustomerDAO {
   public CloudscapeCustomerDAO() { // initialization }
   // The following methods can use CloudscapeDAOFactory.createConnection()
   // to get a connection as required
   public int insertCustomer(...) {
      // Implement insert customer here.
      // Return newly created customer number or a -1 on error
   }
   public boolean deleteCustomer(...) {
    // Implement delete customer here
    // Return true on success, false on failure
   }
   public Customer findCustomer(...) {
     // Implement find a customer here using supplied argument values
     // as search criteria
     // Return a Transfer Object if found,
     // Return null on error or if not found
   }
...
}
```


Fabrique de DAO (9/10)

```
//Customer Transfer Object
public class Customer implements java.io.Serializable {
      // member variables
      int CustomerNumber;
      String name;
      String streetAddress;
      String city;
      ...

      // getter and setter methods
      ...

      ...
}
```

Fabrique de DAO (10/10)

```
// Client code
        // create the required DAO Factory
        DAOFactory cloudscapeFactory =
           DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);
        // Create a DAO
        CustomerDAO custDAO = cloudscapeFactory.getCustomerDAO();
        // create a new customer
        int newCustNo = custDAO.insertCustomer(...);
        // Find a customer object. Get the Transfer Object.
        Customer cust = custDAO.findCustomer(...);
        // modify the values in the Transfer Object.
        cust.setAddress(...);
        cust.setEmail(...);
        // update the customer object using the DAO
        custDAO.updateCustomer(cust);
        // delete a customer object
        custDAO.deleteCustomer(...);
        // select all customers in the same city
        Customer criteria=new Customer();
        Criteria.setCity("New York");
        Collection customersList = custDAO.selectCustomersTO(criteria);
        // returns customersList - collection of Customer Transfer Objects.
        // iterate through this collection to get values. ...
```


TP JBDC/DAO (1/10)

http://www.lamsade.dauphine.fr/~manouvri/HIBERNATE/TP_JDBC/TP_JDBC.html

Ou sur Mycourse M2 MIAGE ID/IF/IF-APP/SITN_2015-2016_Persistance objet-relationnel / Hibernate_Maude Manouvrier

- Données :
  - Une base de données exemple (sous forme de scripts SQL), gérée sous *PostgreSQL*
  - Un programme Java exemple (`TestJDBCPostgresql.java`) contenant les principales commandes JDBC
  - Une classe Java (`ConfigConnection.java`) pour gérer les connexions à la base de données
  - Une classe exemple (type ***Active record*** - `Departement.java`)
  - Un programme (`CreerDepartement.java`) permettant de tester la classe
- Travail à réaliser : Développer 3 DAO

TP JBDC/DAO (2/10)

Connexion à la BD

```
import java.sql.*;
public class TestJDBCPostgresql
{ Connection        db=null;   // Connexion à la BD
  Statement         sql=null;  // Curseur pour la requête
  DatabaseMetaData dbmd; // Méta-données issues du driver

public TestJDBCPostgresql(String argv[])
    throws ClassNotFoundException, SQLException, java.io.IOException
  {
    String database = argv[0];
    String username = argv[1];
    String password = argv[2];

    Class.forName("org.postgresql.Driver"); // Récupération du driver

    db = DriverManager.getConnection("jdbc:postgresql:"+database,
                                     username,
                                     password); // Connexion
```


TP JBDC/DAO (3/10)

Exécution d'une requête de MàJ

```
sql = db.createStatement(); // Création du curseur pour la requête

String sqlText = "CREATE TABLE Note ( Note_ID SERIAL, " +
    "Etudiant_ID integer, " +
    "Inscription_ID integer," +
    "Note real," +
    "CONSTRAINT PK_Notes PRIMARY KEY (Note_ID)," +
    "CONSTRAINT FK_Notes_Etudiant FOREIGN KEY (Etudiant_ID) " +
    " REFERENCES Etudiant (Etudiant_ID)," +
    " CONSTRAINT FK_Notes_Inscription " +
    " FOREIGN KEY (Inscription_ID) " +
    " REFERENCES Inscription (Inscription_ID) " +
    " );";

System.out.println("Executing this command: "+sqlText+"\n");

sql.executeUpdate(sqlText); // Exécution de la requête

db.commit(); // Validation de la requête
```

TP JBDC/DAO (4/10)

Exécution d'une requête d'interrogation

```
sqlText="SELECT capacite FROM Salle " +
                          "WHERE Batiment ='B' " +
                             "AND Numero_Salle='020' FOR UPDATE";
System.out.println("Executing this command : "+sqlText+"\n");

ResultSet rset = sql.executeQuery(sqlText);

// Pour afficher le résultat de la requête
while (rset.next()) {
   System.out.println("Capacité de la salle B020 : "
                      + rset.getInt(1) + "\n");
}
```


TP JBDC/DAO (5/10)

Exécution d'une requête de MàJ paramétrable

```
System.out.println("\n\nNow demostrating a prepared statement...");
sqlText = "INSERT INTO Salle VALUES (?,?,?)";
System.out.println("The Statement looks like this: "+sqlText+"\n");
System.out.println("Looping several times filling in the fields...\n");
PreparedStatement ps = db.prepareStatement(sqlText);

// Exécution d'une requête paramétrée
String [] NumBatiment = {"A", "B", "C", "P", "D"};
String [] NumSalle = {"208", "026", "405", "340", "120"};
int lenNB = NumBatiment.length;
for (int i=0, c=30 ; (i<lenNB) && (c<35) ;c++,i++)
{ System.out.println(i+" " + NumBatiment[i]+ " " + NumSalle[i]+ "...\n");
   ps.setString(1,NumBatiment[i]); //Affectation de la colonne 1(Batiment)
   ps.setString(2,NumSalle[i]);; // Affectation de la colonne 2(Numero_Salle)
   ps.setInt(3,c); //Affectation de la colonne 3(Capacite)

   ps.executeUpdate();
}
db.commit();
ps.close();
```

TP JBDC/DAO (6/10)

Classe de connexion

```
public class ConfigConnection {
  public static Connection getConnection(String nomFichierProp)
      throws IOException, ClassNotFoundException, SQLException {
    Properties props = new Properties();
    URL urlFichierProp = ConfigConnection.class.getResource(nomFichierProp);
    BufferedInputStream bis = null;
    try {
      bis = new BufferedInputStream(urlFichierProp.openStream());
      props.load(bis);
      String driver = props.getProperty("driver");
      String url = props.getProperty("url");
      String utilisateur = props.getProperty("utilisateur");
      String mdp = props.getProperty("mdp");
      Class.forName(driver);
      return DriverManager.getConnection(url, utilisateur, mdp);
    }
    finally {
      if (bis != null) {
        bis.close();
      }
    }
  }
```


TP JBDC/DAO (7/10)

Utilisation de la classe de Connexion

```
String username = argv[0];
String password = argv[1];
String fichierProp = argv[2];

System.out.println("Username=" +username + " Passwd="+password+"
   fichierProp="+fichierProp+"\n");

db = ConfigConnection.getConnection(fichierProp,username,password);
```

TP JBDC/DAO (8/10)

Exemple de classe type Active Record

```
public class Departement {
    private int id;
    private String nom;
    private boolean _builtFromDB;
    private static String _query = "SELECT * FROM Departement";

 public Departement() {
    _builtFromDB=false;
    }

public Departement(ResultSet rs) throws SQLException {
        id = rs.getInt("Departement_ID");
        nom = rs.getString("Nom_Departement");
        _builtFromDB = true;
    }
```


TP JBDC/DAO (8bis/10)

Exemple de classe type Active Record

```
private String _update() {
            return "UPDATE Departement SET Nom_Departement='" + nom +
             "' WHERE Departement_ID=" + id;
    }

private String _insert() {
            return "INSERT INTO Departement"
            + " VALUES(nextval('departement_departement_id_seq'), '" +
            nom + "')";
    }

private String _delete() {
        return "DELETE FROM Departement"
        + " WHERE Departement_ID = " + id ;
    }
```

TP JBDC/DAO (9/10)

Exemple de classe type Active Record (suite)

```
public void save(Connection cx) throws SQLException {
        Statement s = cx.createStatement();
        if(_builtFromDB) {
        System.out.println("Executing this command: "+_update()+"\n");
            s.executeUpdate(_update());
        }

        else {
        System.out.println("Executing this command: "+_insert()+"\n");
        s.executeUpdate(_insert());
        _builtFromDB=true;

    // Pour récupérer la valeur de clé artificielle
        s.executeUpdate(_insert(), Statement.RETURN_GENERATED_KEYS);
        ResultSet r = s.getGeneratedKeys();
        while(r.next())
            id = r.getInt(1);


        }
      }
    }
```


TP JBDC/DAO (10/10)

Exemple de classe type Active Record (suite)

```
public void delete(Connection cx) throws SQLException {
    Statement s = cx.createStatement();
        if(_builtFromDB) {
        System.out.println("Executing this command: "+_delete()+"\n");
            s.executeUpdate(_delete());
        }
        else System.out.println("Objet non persistant!");
    }
```

Exemple d'utilisation de la classe :

```
// Insertion d'un nouveau département
     Departement d = new Departement("DEP");
     d.save(_cx); // _cx paramètre de Connexion
     _cx.commit();
     System.out.println("Département créé et persistant : " + d.toString());
```

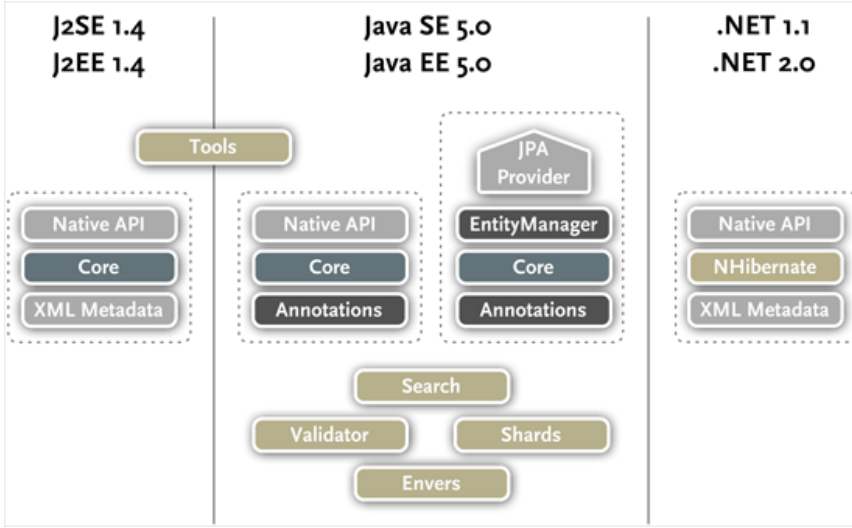
Partie 4 : Hibernate / JPA

- **Présentation générale**
- **Architecture du noyau Hibernate**
- **Environnement de travail**
- **Classes persistantes dans Hibernate**
- **Opérations du gestionnaire de persistance**
- **Méta-données : fichiers de *mapping/annotations* des classes métier**
- **Récupération d'instances persistantes**
- **Requêtes**
- **Transactions**

Hibernate : généralités (1/4)

- Outil ORM ou Cadre *(Framework)* de persistance libre (*open source)* gérant la persistance des objets Java/JEE en base de données relationnelle [Wikipédia, Pat05]
- Version ORM 5,2 : implémentation du standard de persistance EJB 3.0 *Java Persistence API* (JPA)
- Possibilité d'être utilisé aussi bien dans un développement client lourd, que dans un environnement web léger de type Apache Tomcat ou dans un environnement J2EE complet [Wikipédia]
- Code SQL généré à l'exécution via des informations fournies dans un document de correspondance (*mapping*) XML ou des annotations

Hibernate : généralités (2/4)



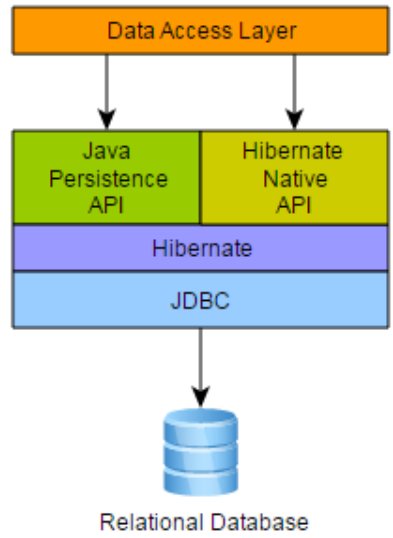
Hibernate : généralités (3/4)

Différents modules :

- ***Hibernate Core*** : API native implémentant les services de base pour la persistance
  - Méta-données au format XML (+ annotations depuis la version 3.6)
  - Langage HQL et interface pour écrire des requêtes
- ***Hibernate Annotations*** (inclus dans *Hibernate Core* 3.6) : Remplacement des fichiers XML par des annotations JDK 5.0 implémentant les annotations du standard JPA + annotations spécifiques à Hibernate
- ***Hibernate Entity Manager*** : Implémentation de la partie des spécifications JPA concernant
  - Les interfaces de programmation,
  - Les règles de cycle de vie des objets persistants
  - Les fonctionnalités d'interrogation

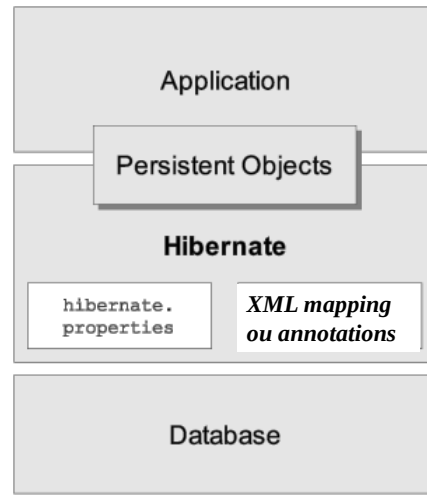
  Hibernate Entity Manager = *wrapper* au dessus du noyau Hibernate implémentant une solution complète de persistance JPA (cf. documentation Hibernate)

Hibernate : généralités (4/4)

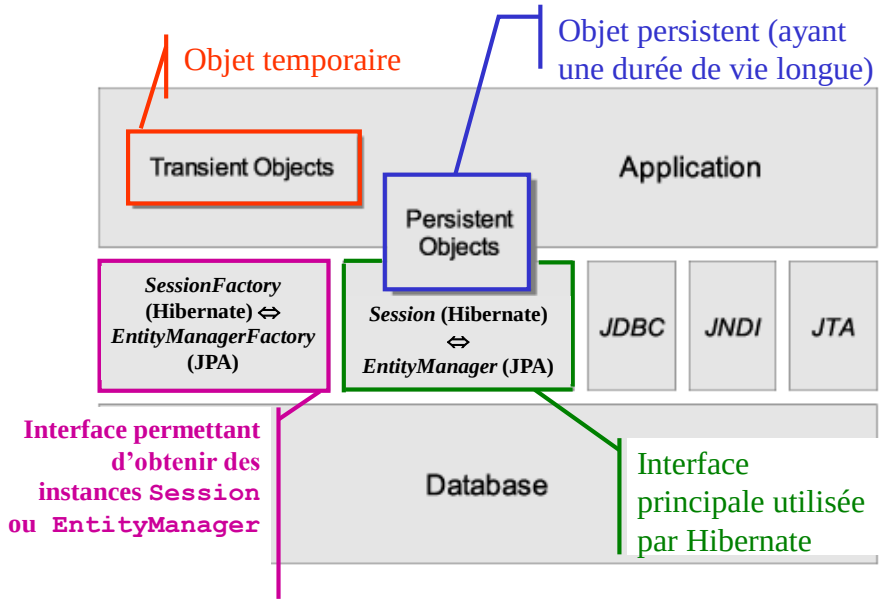


Architecture du noyau Hibernate (1/13)

- `hibernate.properties` :
  Fichier de configuration
  - Version XML : **`hibernate.cfg.xml`** permettant un paramétrage plus fin
  - Configuration par programmation
- *XML mapping* ou annotations :
  - Méta-données (paramètres) décrites sous la forme de fichiers de correspondance XML ou sous forme d'annotation
  - Utilisées pour mettre en correspondance les classes Java et le modèle relationnel



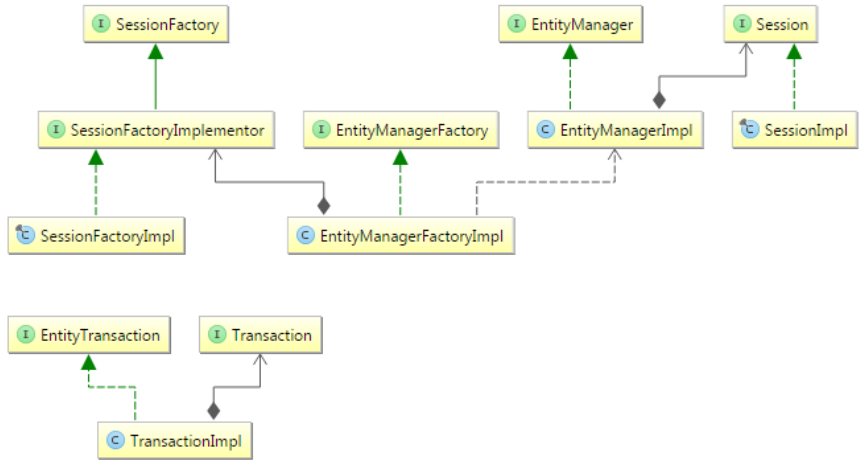
Architecture du noyau Hibernate (2/13)



Architecture du noyau Hibernate (3/13)

- **`SessionFactory`** ou **`EntityManagerFactory`** :
  - Cache immuable (*threadsafe*) des correspondances (*mappings*) vers une (et une seule) base de données
  - Coûteuse à construire car implique l'analyse des fichiers de configuration
  - Pouvant contenir un cache optionnel de données (de second niveau) réutilisable entre les différentes transactions
  - Construite à partir d'un objet (`Ejb3`)`Configuration`
- **`Session`** ou **`EntityManager`** :
  - Objet *mono-threadé*, à durée de vie courte, représentant une conversation entre l'application et l'entrepôt de persistance
  - Encapsule une connexion JDBC
  - Contient un cache (de premier niveau et obligatoire) des objets persistants

Architecture du noyau Hibernate (4/13)



Architecture du noyau Hibernate (5/13)

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

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

Analyse du fichier de mapping

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

Architecture du noyau Hibernate (6/13)

Exemple de fichier de configuration `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>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">org.postgresql.Driver
     </property>
    <property name="hibernate.connection.password">passwd</property>
    <property name="hibernate.connection.url">jdbc:postgresql:BDTest2</property>
    <property name="hibernate.connection.username">login</property>
   <!-- SQL dialect -->
   <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
   <!-- Enable Hibernate's automatic session context management -->
   <property name="current_session_context_class">thread</property>
   <!-- Disable the second-level cache  -->
   <property  name="cache.provider_class">org.hibernate.cache.NoCacheProvider
    </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 resource="events/Event.hbm.xml"/>
   <mapping resource="events/Person.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```


Architecture du noyau Hibernate (7/13)

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

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

- Paramètres de configuration nécessaires pour la connexion JDBC :

```
<property
  name="hibernate.connection.driver_class">org.postgresql.Driver
 </property>
<property name="hibernate.connection.password">passwd</property>
<property name="hibernate.connection.url">jdbc:postgresql:BDTest2
 </property>
<property name="hibernate.connection.username">login
 </property>
```

Architecture du noyau Hibernate (8/13)

- Spécification de la variante de SQL générée par Hibernate :

```
<!-- SQL dialect -->
   <property
     name="dialect">org.hibernate.dialect.PostgreSQLDialect
   </property>
```

- Activation de la génération automatique des schémas de base de données - directement dans la base de données :

```
<!-- Drop and re-create the database schema on startup -->
   <property name="hbm2ddl.auto">create</property>
```

- Fichier de configuration (fichier de *mapping*) des classes persistantes :

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

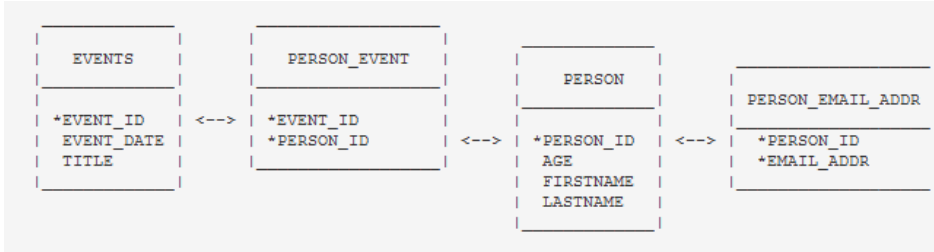

Architecture du noyau Hibernate (9/13)

Possibilité de transmettre les paramètres de configuration à l'exécution

```
Configuration cfg = new Configuration()
    .addClass(Person.class)
    .addClass(Event.class)
    .setProperty(Environment.HBM2DDL_AUTO, "create");

cfg.setProperty("hibernate.show_sql", "true");

cfg.setProperty("hibernate.connection.driver_class", "org.postgresql.Driver")
    .setProperty("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect")
    .setProperty("hibernate.connection.url", "jdbc:postgresql:BDTest1")
    .setProperty("hibernate.connection.username", "login")
    .setProperty("hibernate.connection.password", "passwd")
    .setProperty("hibernate.order_updates", "true");

factory = cfg.buildSessionFactory();
```

Exemple : Tutorial Hibernate

Modèle relationnel :



Modèle objet :

```
public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;
    private Set events ;
    private Set emailAddresses ;
    …
}
```

```
public class Event {

    private Long id;
    private String title;
    private Date date;
    private Set participants;
    …
}
```


Architecture du noyau Hibernate (10/13)

Exemple de fichier de correspondance pour la classe `Person`

`Person.hbm.xml` :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
          <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
        <set name="events" table="PERSON_EVENT">
          <key column="PERSON_ID"/>
          <many-to-many column="EVENT_ID" class="events.Event"/>
        </set>
        <set name="emailAddresses" table="PERSON_EMAIL_ADDR">
          <key column="PERSON_ID"/>
          <element type="string" column="EMAIL_ADDR"/>
        </set>
    </class>
</hibernate-mapping>
```

Déclaration de la DTD

Déclaration du mapping

Classe `event.Person` « mappée » en relation `PERSON`

Mapping de l'identifiant

Mapping de collection

Attribut de jointure entre PERSON et PERSON_EVENT

Pour préciser qu'il y a une « table association »

Pour définir une collection de valeurs

Architecture du noyau Hibernate (11/13)

Mapping par annotation – exemple de la classe `Person` :

```
@Entity
@Table (name="PERSON")
public class Person {
    private Long id;
    private int age;
    private String firstname;
    private String lastname;
    public Person() {}

    // Accessor methods
    @Id@GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="PERSON_ID")
    public Long getId() { return this.id; }

    private Set events = new HashSet();

    @ManyToMany
    @JoinTable( name="PERSON_EVENT",
                joinColumns=@JoinColumn(name="PERSON_ID"),
                inverseJoinColumns=@JoinColumn(name="EVENT_ID") )
    protected Set getEvents() { return events;}

    private Set emailAddresses = new HashSet();

    @CollectionOfElements
    @JoinTable( table=@Table(name="PERSON_EMAIL_ADDR"),
                joinColumns = @JoinColumn(name="PERSON_ID") )
    public Set getEmailAddresses() { return emailAddresses; }
}
```

Classe `event.Person` « mappée » en relation `PERSON`

Attention avant la version 3.6 : il faut utiliser Hibernate Annotation !!

Mapping de l'identifiant

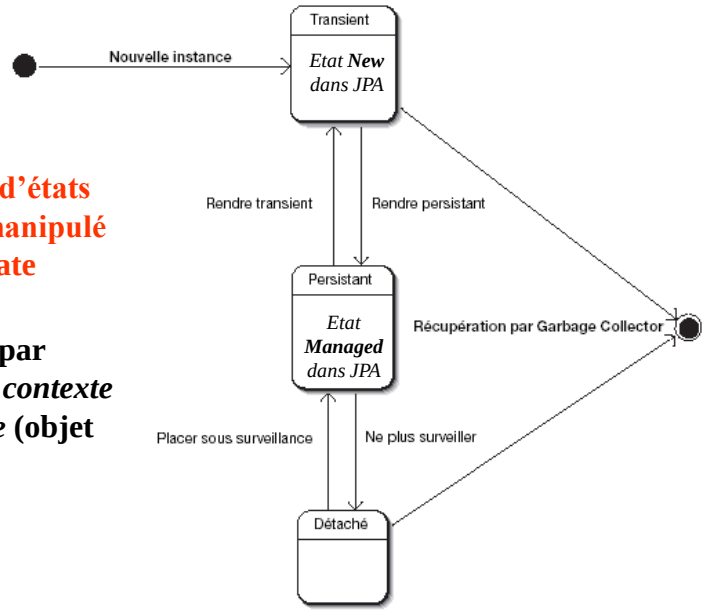
Mapping de collection d'objets

Mapping de collection de valeurs

Architecture du noyau Hibernate (12/13)

Diagramme d'états d'un objet manipulé avec Hibernate

États définis par rapport à un *contexte de persistance* (objet `Session`)



Architecture du noyau Hibernate (13/13)

- **Passager/Temporaire/Éphémère (*transient*) :**
  - Instance non associée (et n'ayant jamais été associée) à un contexte de persistance
  - Instance sans identité persistante (i.e. valeur de clé primaire)
- **Persistant :**
  - Instance associée à un contexte de persistance (`Session`)
  - Instance possèdant une identité persistante (i.e. valeur de clé primaire) et, peut-être, un enregistrement/nuplet correspondant dans la base
  - Pour un contexte de persistance particulier, *garantie* par Hibernate de l'équivalence entre l'identité persistante et l'identité Java (i.e. emplacement mémoire de l'objet)
- **Détaché**
  - Instance ayant été associée au contexte de persistance à présent fermé ou instance ayant été sérialisée vers un autre processus
  - Instance possèdant une identité persistante et peut-être un enregistrement/nuplet correspondant dans la base
  - Aucune garantie par Hibernate sur la relation entre l'identité persistante et l'identité Java

Environnement de travail avec Hibernate (1/6)

Bibliothèques *Hibernate Core* (en plus de **`hibernate3.jar`**) :

- **`Antlr-xxx.jar`**
  ANTLR (*Another Tool for Language Recognition*) - outil permettant de développer un langage maison avec une grammaire capable de le reconnaître. Utilisé pour créer langage HQL (*Hibernate Query Language*) - Indispensable à l'exécution
- **`commons-collections-xxx.jar`**
  Bibliothèques du projet *Apache Jakarta Commons* pour manipuler les collections - Indispensable à l'exécution
- **`Dom4j-xxx.jar`**
  API Open Source Java permettant de travailler avec XML, XPath et XSLT - Analyseur Syntaxique de configuration XML et de mapping - Indispensable à l'exécution
- **`javassist-xxx.GA.jar`**
  API de manipulation de *bytecode* (fichier `.class`), dans un contexte de réflexion, c-à-d de modification du contenu d'une classe en phase d'exécution - Indispensable à l'exécution
- **`jta.jar`**
  API JTA standard (interfaces Java standards entre un gestionnaire de transaction et les différentes parties impliquées dans un système de transactions distribuées) – Requis pour les applications s'exécutant en dehors d'un serveur d'application - Indispensable à l'exécution
- **`slf4j-api-xxx.jar`** (avant `log4J`)
  (*Simple Logging Facade for Java*) Abstraction d'un système de login permettant de s'abstraire de l'implémentation utilisée - Indispensable à l'exécution
  Pour pouvoir configurer le *logging* : `slf4j-api.jar`

Environnement de travail avec Hibernate (2/6)

Fichiers nécessaires avec *Hibernate Core* :

- **`hibernate.cfg.xml`** : fichier de configuration globale contenant
  - Les paramètres de connexion à la base de données (pilote, login, mot de passe, url, etc.)
  - Le dialecte SQL de la base de données
  - La gestion de pool de connexions
  - Le niveau de détails des traces etc.
- Pour chaque classe persistante :
  - **`ClassePersistante.java`** : Implémentation POJO (*Plain Old Java Objects*) de la classe
  - **`ClassePersistante.hbm.xml`** : Fichier XML de correspondance (*mapping*)
  - **`ClassePersistanteHome.java`** ou **`ClassePersistanteDAO.java`** : Implémentation du DAO (*Data Access Object*) pour l'isolation avec la couche de persistance – Optionnel
  - **`ClassePersistante.sql`** : Code SQL de création de la ou les relations correspondantes – Optionnel – pouvant être généré par Hibernate

Environnement de travail avec Hibernate (3/6)

- Bibliothèques *Hibernate Annotation* (avant la version 3.5) :
  - **`hibernate-annotations.jar`** Annotations propres à Hibernate
  - **`lib/ejb3-persistence.jar`** Annotations du standard EJB3

  } **`hibernate3.jar`** *depuis la version 3.5*
- Fichiers nécessaires pour *Hibernate Annotation* :
  - **`hibernate.cfg.xml`** : fichier de configuration
  - **`ClassePersistante.java`** : POJO avec annotations
  - **`ClassePersistante.hbm.xml`** : optionnel – possibilité de combiner annotations et méta-données XML
- Particularité pour la création de la *SessionFactory* :

```
sessionFactory = new AnnotationConfiguration().buildSessionFactory();
```

Et ajout dans le fichier de `hibernate.cfg.xml` de `<mapping class="Person"/>`

ou

```
sessionFactory = new AnnotationConfiguration()
                 .addPackage("domain") // le nom complet du package
                 .addAnnotatedClass(Person.class)
```

Environnement de travail avec Hibernate (4/6)

- Bibliothèques *Hibernate EntityManager* (avant la version 3.5) :
  - **hibernate-commons-annotations.jar**
  - **hibernate-entitymanager.jar**
- Bibliothèques *Hibernate EntityManager* (depuis version 3.5) :
  hibernate-jpa-xx-api-xx.Final.jar
- Fichiers nécessaires pour *Hibernate EntityManager* :
  - **persistence.xml** : fichier de configuration situé dans le répertoire META-INF

```
<persistence … >
   <persistence-unit name="manager1" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <class>org.acme.Person</class>
      <class>org.acme.Event</class>
      <properties>
         <property name="hibernate.dialect"
                   value="org.hibernate.dialect.PostgreSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto"
                   value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

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

Environnement de travail avec Hibernate (5/6)

- Propriété pour se connecter :

```
<property name="javax.persistence.jdbc.driver"
          value="org.postgresql.jdbcDriver"/>
<property name="javax.persistence.jdbc.user"
          value=""/>
<property name="javax.persistence.jdbc.password"
          value=""/>
<property name="javax.persistence.jdbc.url"
          value="jdbc:postgresql:BDTest1"/>
```

- Possibilité d'utiliser un fichier `hibernate.cfg.xml` :

```
<property name="hibernate.ejb.cfgfile"
          value="/org/hibernate/ejb/test/hibernate.cfg.xml"/>
```

- Replacement des propriétés `hibernate.cfg.xml` par celles du fichier persistence.xml

Environnement de travail avec Hibernate (6/6)

- Pour créer le gestionnaire de persistance :

```
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("manager1");
```

- Possibilité de fournir des propriétés au gestionnaire de persistance à l'exécution :

```
Map<String, Object> configOverrides =
            new HashMap<String, Object>();
configOverrides.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory programmaticEmf =
    Persistence.createEntityManagerFactory("manager1",
  configOverrides);
```

- Utilisation de **`Ejb3Configuration`**, **`EntityManagerFactory`** et **`EntityManager`** (à la place de **`Configuration`** , **`SessionFactory`** et **`Session`**) et possibilité de configurer la fabrique à l'exécution :

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
cfg.addProperties( properties )
   .addAnnotatedClass( MyAnnotatedClass.class )
   .addClass( NonAnnotatedClass.class )
  .addResource( "mypath/MyOtherCLass.hbm.xml )
  .configure("/mypath/hibernate.cfg.xml")
  .buildEntityManagerFactory();
```


Classes persistantes dans Hibernate (1/6)

- Classes persistantes :
  - implémentation des entités métiers
  - sous la forme de POJO
- Pas d'implémentation de certaines interfaces d'héritage de classes particulières
- 5 règles à respecter :
  - Implémenter un constructeur sans paramètre (pouvant être privé mais de préférence accessible par le paquetage)
  - Fournir une propriété d'identifiant (optionnel mais fortement recommandé)
  - Favoriser les classes non finales (semi-optionnel)
  - Implémenter des modificateurs (mutateurs - *setter*) et accesseurs (*getter*) pour chaque champ persistant
  - Implémenter de Collections avec les classes du cadre d'application (*framework*) **`Collections`** d'Hibernate (**`Set`**, **`List`**, **`Map`** etc.)

Classes persistantes dans Hibernate (2/6)

```
/*** Departement generated by hbm2java*/
public class Departement implements java.io.Serializable {
  // Fields
  private int departementId;
  private String nomDepartement;
  private Set<Enseignement> enseignements = new HashSet<Enseignement>(0);
  private Set<Enseignant> enseignants = new HashSet<Enseignant>(0);
  // Constructors
  /** default constructor */
  public Departement() {}
  // Property accessors
  public int getDepartementId() {
  return this.departementId;
  }
  public void setDepartementId(int departementId) {
  this.departementId = departementId;
  }
  public String getNomDepartement() {
  return this.nomDepartement;
  }

...
}
```


Classes persistantes dans Hibernate (3/6)

- Garantie par Hibernate de l'équivalence de l'identité persistante (ou de base de données) et de l'identité Java mais uniquement à l'intérieur de la portée d'une session particulière
- Méthodes **`equals()`** et **`hashCode()`** à surcharger en particulier si :
  - Instances de classes persistantes gérées dans un `Set` (manière recommandée pour représenter des associations pluri-valuées)
  - Utilisation du réattachement d'instances détachées (*on y reviendra …*)
  - Utilisation d'identificateur composite (*on y reviendra …*)

Classes persistantes dans Hibernate (4/6)

Règles pour surcharger `equals()` et `hashCode()`:

- http://docs.jboss.org/hibernate/stable/core.old/reference/fr/html/persistent-classes-equalshashcode.html
- Recommandation Hibernate : implémenter **`equals()`** et **`hashCode()`** en utilisant *l'égalité par clé métier*
- Exemple de surcharge de **`equals()`** :

```
public boolean equals(Object other) {
   // Test d'égalité de référence en mémoire
   if (this == other) return true;
   //Vérification de la classe dont other est instance
   if ( !(other instanceof MyClass) ) return false;
   // Test de comparaison des propriétés appartenant
   // à la clé métier
   final MyClass obj = (MyClass) other;
   if ( !obj.getProp1().equals( getProp1() ) )
     return false;
   ...
return true; }
```

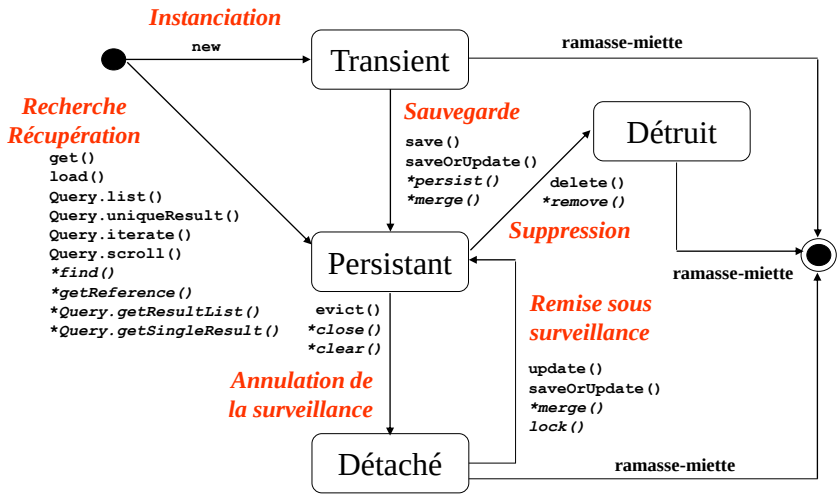

Classes persistantes dans Hibernate (5/6)

- Pour manipuler des objets persistants :
  1. Ouverture d'une `Session` Hibernate
  2. [Débuter une transaction] – fortement conseillé
  3. Appliquer les opérations de `Session` pour interagir avec l'environnement de persistance
  4. [Valider (`commit()`) la transaction en cours]
  5. Synchroniser avec la base de données (*flush*) et fermer la session

Annuler (`rollback()`) la transaction et fermer la `Session` en cas d'exception soulevée par les méthodes de `Session`

Classes persistantes dans Hibernate (6/6)

Diagramme d'états des objets d'une classe persistante



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

Contexte de persistance (1/4)

`Session` ou **`EntityManager`** associée à un **contexte de persistance :**

- Vérification des modifications et synchronisation avec la base de données (*automatic dirty checking*)
- Utilisation du contexte de persistance comme cache de 1er niveau
- Garantie de l'identité Java
- Extension du contexte de persistance à une conversation (unité de travail longue)

Contexte de persistance (2/4)

***Automatic dirty checking* :**

- Fermeture de l'unité de travail ⇒ synchronisation avec la base des entités mises à jour (*dirty*)
- Possibilité de synchronisation avec la base avant l'exécution d'une requête
- Propagation des modifications dans la base aussi tard que possible et maintien des verrous de base de données le moins longtemps possible
- Possibilité de ne propager que les mises à jour des propriétés réellement modifiées

Contexte de persistance (3/4)

Utilisation du cache :

- Pour vérifier les mises à jour
- Pour améliorer les performances en diminuant l'interaction avec la base de données
- Pour garantir l'isolation (*repeatable read*)
- Pour éviter le dépassement de capacité de la pile (*stack overflow*) en cas de navigation dans le graphe d'objets
- Pour éviter les conflits de représentation (pas deux objets correspondants à la même information en base de données)

Possibilité de gérer le cache (détachement d'objets et nettoyage du contexte de persistance)

Contexte de persistance (4/4)

Conversation ou transaction applicative :

- Ensemble d'opérations courtes réalisées par interaction entre l'utilisateur et l'application (ex. application Web)
- Implémentation :
  - En détachant les entités
    - Pas de garantie de l'identité des objets (d'où importance de la redéfinition des méthodes `equals` et `hashCode`)
    - Gestion manuelle du ré-attachement ou de la fusion des entités au contexte de persistance (attention pas de gestion du ré-attachement dans JPA)
  - En étendant le contexte de persistance
    - Pas de fermeture du contexte de persistance pendant l'interaction avec l'utilisateur mais déconnexion de la base
    - Pas de possibilité de valider les changements avant la fin

Opérations du gestionnaire de persistance (1/15)

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

Opérations du gestionnaire de persistance (2/15)

Récupérer une instance persistante dans *Hibernate Core* :

- **`session.load(Class clazz, serializable id)`**
  ⇒ Levée d'une exception irrécupérable s'il n'y a pas de ligne correspondante dans la base de données
- **`session.get(Class clazz, serializable id)`**
  ⇒ `null` si pas de nuplet correspondant dans la base
- Pour ces deux méthodes :
  - Génération d'un ordre SQL un `SELECT` sur la base pour récupérer l'entité à partir de son identificateur
  - Possibilité de charger dans une instance nouvellement créée ou dans une instance existante
  - Possibilité de transformer le `SELECT` en `SELECT ... FOR UPDATE` en utilisant `LockMode.UPGRADE` en 3ème paramètre (*on y reviendra …*)
- Possibilité de récupérer une instance aussi par les API `Query`, `Criteria`, SQLQuery (*on y reviendra …*)

`load` versus `get` (1/5)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(2));
session.getTransaction().commit();
```

⇒Aucune requête générée et pas de levée d'exception si le nuplet correspondant n'existe pas

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
session.getTransaction().commit();
```

⇒ **Hibernate: select person0_.PERSON_ID as PERSON1_2_0_, person0_.age as age2_0_, person0_.firstname as firstname2_0_, person0_.lastname as lastname2_0_ from PERSON person0_ where person0_.PERSON_ID=?**

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(2));
System.out.println(aPerson1.getFirstname());
session.getTransaction().commit();
```

⇒ **Hibernate: select person0_.PERSON_ID as PERSON1_2_0_, person0_.age as age2_0_, person0_.firstname as firstname2_0_, person0_.lastname as lastname2_0_ from PERSON person0_ where person0_.PERSON_ID=?**

load versus get (2/5)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(1));
if (aPerson1!=null) System.out.println(aPerson1.getFirstname());
session.getTransaction().commit();
```

⇒**Hibernate: select person0_.PERSON_ID as PERSON1_2_0_, person0_.age as age2_0_, person0_.firstname as firstname2_0_, person0_.lastname as lastname2_0_ from PERSON person0_ where person0_.PERSON_ID=?**

Exception in thread "main" org.hibernate.ObjectNotFoundException: No row with the given identifier exists: [events.Person#1]

```
Session session =HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.get(Person.class, new Long(1));
if (aPerson1!=null) System.out.println(aPerson1.getFirstname());
session.getTransaction().commit();
```

⇒**Hibernate: select person0_.PERSON_ID as PERSON1_2_0_, person0_.age as age2_0_, person0_.firstname as firstname2_0_, person0_.lastname as lastname2_0_ from PERSON person0_ where person0_.PERSON_ID=?**

Mais pas d'exception!! (sauf si on n'oublie de tester aPerson1)

121

load versus get (3/5)

```
public class Person { …
        public void addToEvent(Event event) {
                this.getEvents().add(event);
                event.getParticipants().add(this);
        }
…
}
…
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(2));
Event anEvent1 = (Event) session.load(Event.class, new Long(8));
aPerson1.addToEvent(anEvent1); // Méthode métier au-dessus
session.getTransaction().commit();
```

Mise à jour des deux cotés de l'association

⇒ **select … from PERSON person0_ where person0_.PERSON_ID=?**

⇒ **select ... from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?**

⇒ ***select … from EVENTS event0_ where event0_.EVENT_ID=?*** **(sauf si association déjà existante ou si get et après la 1ère requête)**

⇒ **select … from PERSON_EVENT participan0_ inner join PERSON person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where participan0_.EVENT_ID=?**

⇒ ***insert into PERSON_EVENT (PERSON_ID, EVENT_ID) values (?, ?)***

122

load versus get (4/5)

```
// Si on a changé la stratégie par défaut en chargement agressif
// lazy="false"  - on y reviendra!!
Person aPerson1 = (Person) session.get(Person.class, new Long(1));
Event anEvent1 = (Event) session.load(Event.class, new Long(6));
Person1.addToEvent(anEvent1);
```

⇒ select … from PERSON person0_ where person0_.PERSON_ID=?

⇒ select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?

⇒ *select … from PERSON_EVENT participan0_ inner join PERSON person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where participan0_.EVENT_ID=?*
(si collection events non vide pour récupérer la collection participants correspondante de chaque élément de events et autant de requêtes que d'éléments dans events)

⇒ *select … from EVENTS event0_ where event0_.EVENT_ID=?*
(sauf si association déjà existante)

⇒ select …_ from PERSON_EVENT participan0_ inner join PERSON person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where participan0_.EVENT_ID=?

⇒ *select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?*
(si collection participants non vide pour récupérer la collection events correspondante de chaque élément de participants et autant de requêtes que d'éléments dans participants)

⇒ *select … from PERSON_EVENT participan0_ inner join PERSON person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where participan0_.EVENT_ID=?*
(pour récupérer les events de chaque participant et autant de requêtes que d'éléments dans events)

⇒ *insert into PERSON_EVENT (PERSON_ID, EVENT_ID) values (?, ?)*

123

load versus get (5/5)

```
// Si chargement à la demande par défaut
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.get(Person.class, new Long(1));
Event aEvent1 = (Event) session.load(Event.class, new Long(6));
// Si association gérée de manière unidirectionnelle
aPerson1.getEvents().add(aEvent1);
session.getTransaction().commit();
```

⇒ select … from PERSON person0_ where person0_.PERSON_ID=?

⇒ select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?

⇒ *select … from EVENTS event0_ where event0_.EVENT_ID=?*
(si equals() et hashCode() redéfinies – sinon utilisation du proxy!!)

⇒ *insert into PERSON_EVENT (PERSON_ID, EVENT_ID) values (?, ?)*

124

Opérations du gestionnaire de persistance (3/15)

Récupérer une instance persistante dans *EntityManager* :

- **`entitymanager.find(Class clazz, serializable id)`**
  ⇔ `session.get()`

```
Person aPerson1 = (Person) em.find(Person.class, new Long(2));
⇒ select … from PERSON person0_ where person0_.PERSON_ID=?
```

- **`entitymanager.getReference(Class clazz, serializable id)`**
  ⇒ Utilisation d'un proxy pour récupérer une référence sur l'objet sans réellement le charger (utile en particulier pour attacher un enfant à un parent sans charger le parent)
  ⇔ `session.load()`

```
Person aPerson1 = (Person) em.getReference(Person.class, new Long(2));
```

⇒ **Aucune requête générée sauf si accès aux attributs de l'objet**

Opérations du gestionnaire de persistance (4/15)

Rendre une instance persistante :

- **`session.save(objet)`**
  - Pour rendre persistante un instance temporaire (transiante)
  - Génération d'une commande `INSERT` uniquement exécutée au moment du lancement de la méthode `session.commit()`
  - Génération de l'identificateur de l'instance (de type `Serializable` et retourné par la méthode) sauf si de type `assigned` ou composite
- **`session.persist(objet)`**
  ou **`entityManager.persist(objet)`**
  En accord avec les spécifications EJB 3.0
- **`session.merge(objet)`**
  ou **`entityManager.merge(objet)`**
  - Fusionne une instance détachée avec une instante persistante (existante ou chargée depuis la base)
  - Effectue un `SELECT` avant pour déterminer s'il faut faire `INSERT` ou `UPDATE` ou rien (JPA)

On reviendra sur ces méthodes pour la persistance des graphes d'objets …

save/persist

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();

Event theEvent = new Event();
theEvent.setTitle(title);
theEvent.setDate(theDate);

session.save(theEvent); // Même chose avec session.persist(theEvent);
                        // ou avec entityManager.persist(theEvent);

session.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**

car dans les méta-données XML :

```
<id name="id" column="EVENT_ID">
        <generator class="native"/>
 </id>
```

ou dans les annotations :

```
@Id @GeneratedValue
@Column(name="EVENT_ID")
 public Long getId() {
        return id;
 }
```

On y reviendra!!

⇒ **insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)**

127

merge

```
// Après le code du transparent précédent
theEvent.setTitle("Test2"); // Mise à jour d'une instance détachée
Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
Event theEvent2 = (Event)session2.merge(theEvent);
// TheEvent est toujours détaché, theEvent2 est persistant
session2.getTransaction().commit();
```

Suite au code du transparent précédent :

⇒ select nextval ('hibernate_sequence')

⇒ insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)

Suite au merge :

⇒ **select … from EVENTS event0_ where event0_.EVENT_ID=?**

⇒ **update EVENTS set EVENT_DATE=?, title=? where EVENT_ID=?**

ou

insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)

si pas d'utilisation de la session dans le transparent précédent

128

Opérations du gestionnaire de persistance (5/15)

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

⇒ Pas de méthode particulière (*automatic dirty checking*)

- Tout modification d'une *instance persistante transactionnelle* (objet chargé, sauvegardé, créé ou requêté par la `Session`) est rendu persistant par la méthode `flush()`
- Surveillance (*dirty checking*) de toutes les instances persistantes par la session
- Instance persistante modifiée = instance sale (*dirty*)
- Synchronisation avec la base définitive une fois la transaction sous-jacente validée

automatic dirty checking

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Person aPerson1 = (Person) session.load(Person.class, new Long(2));
aPerson1.setAge(32);
session.getTransaction().commit();
```

⇒ **`select … PERSON person0_ where person0_.PERSON_ID=?`**

⇒ **`update PERSON set age=?, firstname=?, lastname=? where PERSON_ID=?`**

Il faut appeler `save` ou `persist` pour rendre un objet transient persistant (pas de persistance automatique)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Event theEvent = new Event();
theEvent.setTitle("Test3");
theEvent.setDate(new Date());
session.getTransaction().commit();
```

⇒ **aucune requête générée**

flush

```
Event theEvent = new Event();
theEvent.setTitle("Test10");
theEvent.setDate(new Date());
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
session.save(theEvent);
theEvent.setTitle("Test11");
session.flush();
List result = session.createQuery("from Person").list();
session.getTransaction().commit(); // crée un flush automatique
```

Sans le `flush()` :

⇒ **`select nextval ('hibernate_sequence')`**
⇒ **`select … from PERSON`**
⇒ **`insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)`**
⇒ **`update EVENTS set EVENT_DATE=?, title=? where EVENT_ID=?`**

Avec le `flush()` ou si la requête est (`"from Event"`) :

⇒ **`select nextval ('hibernate_sequence')`**
⇒ **`insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)`**
⇒ **`update EVENTS set EVENT_DATE=?, title=? where EVENT_ID=?`**
⇒ **`select … from PERSON ou select … from EVENTS`**

131

Opérations du gestionnaire de persistance (6/15)

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

- Pas de surveillance possible des instances détachées ⇒ nécessité de ré-attacher les instances en rendant persistant les modifications apportées
- **`session.merge(objet)`**
  `ou entityManager.merge(objet)`
  - Effectue un `SELECT` avant l'`UPDATE` pour récupérer les données dans la base et les fusionner avec les modifications apportées
  - Retourne l'instance persistante correspondante
  - En accord avec les spécifications EJB 3.0 – mais plusieurs variantes
  - Ne se préoccupe pas de l'état du contexte de persistance
- **`session.update(objet)`**
  - Force la mise à jour (`UPDATE`) de l'objet dans la base
  - Lève une exception si une instance de même identificateur existe dans le contexte de persistance

merge versus update (1/2)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());
session.persist(theEvent);
session.getTransaction().commit();
theEvent.setTitle("Event2"); //theEvent est détaché
Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
session2.update(theEvent); // theEvent redevient persistant
session2.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**
⇒ **insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)**
⇒ **update EVENTS set EVENT_DATE=?, title=? where EVENT_ID=?**

⇒ **contrairement au `merge`, pas de select avant le `update` et objet `theEvent` ré-attaché**

133

merge versus update (2/2)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());
session.persist(theEvent);
session.getTransaction().commit();
theEvent.setTitle("Event2"); //theEvent est détaché
Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
// Récupération du précédent objet via une requête
Event anEvent3 = (Event) session2.createCriteria(Event.class)
                         .add( Restrictions.eq("id", theEvent.getId()) )
                         .uniqueResult();
session2.update(theEvent); // theEvent redevient persistant
session2.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**
⇒ **insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)**
⇒ **select … from EVENTS this_ where this_.EVENT_ID=?**

⇒ **Exception in thread "main" org.hibernate.NonUniqueObjectException: a different object with the same identifier value was already associated with the session**

134

Opérations du gestionnaire de persistance (7/15)

Détecter automatiquement un état :

- Soit pour sauvegarder une instance temporaire en générant un nouvel identifiant
- soit pour mettre à jour/réattacher les instances détachées associées à l'identifiant courant
- **`session.saveOrUpdate(objet)`**

En général : utilisation des méthodes `update()`, `saveOrUpdate()` ou `merge()` dans le cas d'utilisation d'instance d'une session à une autre

saveOrUpdate

```
Event theEvent = new Event();
theEvent.setTitle("Event34");
theEvent.setDate(new Date());

Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
Event anEvent1 = (Event) session2.load(Event.class, new Long(6));
anEvent1.setTitle("Test");
session2.saveOrUpdate(theEvent);
session2.saveOrUpdate(anEvent1);
session2.getTransaction().commit();
```

⇒ **select … from EVENTS event0_ where event0_.EVENT_ID=?**
⇒ **select nextval ('hibernate_sequence')**
⇒ **insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)**
⇒ **update EVENTS set EVENT_DATE=?, title=? where EVENT_ID=?**

Si on ajoute après le 1er `saveOrUpdate()` :
`Event anEvent1 = (Event) session2.get(Event.class, theEvent.getId());`
⇒ Exception car une instance existe déjà du même id dans le contexte de persistance (mais attention pas si on utilise uniquement un proxy c-à-d pas avec `load` sans accéder aux propriétés de l'objet ex. sans `anEvent1.setTitle("Test")`)

Opérations du gestionnaire de persistance (8/15)

Utilisation de `update()` ou `saveOrUpdate()`

Dans le cas d'un scénario du type du scénario suivant **:**

1. Chargement d'un objet dans une première session
2. Objet passé à la couche utilisateur
3. Modifications effectuées sur l'objet
4. Objet retourné à la couche logique métier
5. Persistance de ces modifications réalisées en appelant `update()` dans une seconde session

Opérations du gestionnaire de persistance (9/15)

Fonctionnement de `saveOrUpdate()` :

- Si objet déjà persistant et non mis à jour dans la session courante ⇒ ne fait rien
- Si un autre objet associé à la session avec le même identifiant ⇒ lève un exception
- Si objet sans propriété d'identifiant ⇒ **`save()`**
- Si identifiant de l'objet avec une valeur assignée à un objet nouvellement instancié ⇒ **`save()`**
- Si objet versionné (par **`<version>`** ou **`<timestamp>`**) et si propriété de version de même valeur que celle assignée à un objet nouvellement instancié ⇒ **`save()`**
- Sinon ⇒ **`update()`**

Opérations du gestionnaire de persistance (10/15)

Fonctionnement de `merge()` :

- Si instance persistante avec le même identifiant couramment associée à la session ⇒ copie de l'état de l'objet donné dans l'instance persistante
- Si pas d'instance persistante associée à cette session ⇒ chargement à partir de la base de données ou création d'une nouvelle instance persistante
- Instance persistante correspondante retournée

Mais instance passée en paramètre toujours détachée et donc non associée à la session

Opérations du gestionnaire de persistance (11/15)

Ré-attacher une instance détachée :

⇒ **`Session.lock(objet, LockMode)`**

- Diffère de `merge()` par la non prise en compte des modifications effectuées avant le ré-attachement de l'instance
- Permet d'effectuer une vérification de version pour les objets versionnés et de verrouiller l'enregistrement en base de données
- **`LockMode`** = type de verrouillage
  - `NONE` : Aucun verrou et pas `SELECT` pour vérifier la version
  - `READ` : Aucun verrou mais `SELECT` pour vérifier la version

Si supporté par le SGBD

- `UPGRADE` : `SELECT … FOR UPDATE` ⇒ gestion d'une file d'attente en cas d'accès concurrent avant la fin de la transaction
- `UPGRADE_NOWAIT` : `SELECT … FOR UPDATE NOWAIT` ⇒ Exception soulevée en en cas d'accès concurrent avant la fin de la transaction

lock (1/2)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());
session.persist(theEvent); // ou session.save(theEvent);
session.getTransaction().commit();

theEvent.setTitle("Test3"); // theEvent est détaché

Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
session2.buildLockRequest(new LockOptions(LockMode.READ)).lock(theEvent);
//session2.lock(theEvent, LockMode.READ); // Deprecated API
session2.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**
⇒ **insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)**
⇒ **select EVENT_ID from EVENTS where EVENT_ID =? sauf si LockMode.NONE**

141

lock (2/2)

entityManager.lock() modifie uniquement le mode de verrouillage de l'entité mais ne l'a ré-attache pas ⇒ exception en cas d'entité détachée

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();
em.getTransaction().begin();
Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());
em.persist(theEvent);
em.getTransaction().commit();

theEvent.setTitle("Test3"); // theEvent est détaché

EntityManager em2 =
   HibernateUtil.getEntityManagerFactory().createEntityManager();
em2.getTransaction().begin();
LockModeType l = LockModeType.valueOf("OPTIMISTIC"); // ~ READ
em2.lock(theEvent,l);
em2.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**
⇒ **insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)**
⇒ **Exception in thread "main" java.lang.IllegalArgumentException: entity not in the persistence context**

142

Opérations du gestionnaire de persistance (12/15)

Détacher une instance persistante :

- Plus (pas) de surveillance de l'instance par le gestionnaire d'entités :

  ⇒ Plus (pas) aucune modification rendue persistante de manière transparente

  ⇒ Erreur en cas de contact avec un proxy

- Trois moyens de détacher une instance :
  - En fermant la session :

    `session/entityManager.close()`
  - En vidant la session :
    `session/entityManager.clear()`
  - En détachant une instance particulière :
    `session.evict(objet)` ou
    `entityManager.detach(object)`

close (1/4)

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();

session.beginTransaction();

Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());

session.persist(theEvent); // session.save(theEvent);

session.close();

session.getTransaction().commit();


⇒ select nextval ('hibernate_sequence')
⇒ Exception in thread "main" org.hibernate.SessionException:
    Session is closed!
```

close (2/4)

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();

em.getTransaction().begin();

Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());

em.persist(theEvent);

em.close(); // Après le persist()

em.getTransaction().commit();


⇒ select nextval ('hibernate_sequence')
⇒ insert into EVENT (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
⇒ Exception in thread "main" java.lang.IllegalStateException:
    EntityManager is closed
```

145

close (3/4)

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();

em.getTransaction().begin();

Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());

em.close(); // Avant le persist()

em.persist(theEvent);

em.getTransaction().commit();


⇒ Exception in thread "main" java.lang.IllegalStateException:
    EntityManager is closed
```

146

close (4/4)

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();

try {

    em.getTransaction().begin();

    // do some work
    // ...

    em.commit();

}

catch (RuntimeException e) {

    if (em.getTransaction() != null) em.getTransaction().rollback();
    throw e; // or display error message

}

finally {
    em.close();
}
```

147

clear

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();

em.getTransaction().begin();

Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());

em.persist(theEvent);

em.clear();

em.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**
⇒ **Pas d'insertion mais valeur de l'id initialisée !!**

Même comportement avec **session.clear()**

148

session.evict()/entityManager.detach()

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();

em.getTransaction().begin();

Event theEvent = new Event();
theEvent.setTitle("Event1");
theEvent.setDate(new Date());

em.persist(theEvent);

em.detach(theEvent);

em.getTransaction().commit();
```

⇒ **select nextval ('hibernate_sequence')**
⇒ **Pas d'insertion mais valeur de l'id initialisée !!**

Même comportement avec **session.evict(object)**

149

Opérations du gestionnaire de persistance (13/15)

Rendre un objet transient :

⇒ Extraction définitive de l'entité correspondante dans la base de données

- **session.delete(objet)** ou **entityManager.remove(object)** :

  Enregistrement correspondant plus(pas) présent dans la base

  - Instance toujours présente dans la JVM tant que l'objet est référencé – instance transiente
  - Objet ramassé par la ramasse miette dès qu'il n'est plus référencé

On reviendra sur cette méthode en cas de graphe d'objets …

session.delete()/ entityManager.remove()

```
EntityManager em =
  HibernateUtil.getEntityManagerFactory().createEntityManager();

em.getTransaction().begin();

Event theEvent = (Event) em.getReference(Event.class, new Long (90));

em.remove(theEvent);

em.getTransaction().commit();
```

⇒ **select … from EVENT event0_ left outer join PERSON_EVENT participan1_ on event0_.EVENT_ID=participan1_.EVENT_ID left outer join PERSON person2_ on participan1_.PERSON_ID=person2_.PERSON_ID where event0_.EVENT_ID=?**

car chargement agressif par défaut en JPA!!
un `select` uniquement sur `EVENT` si `fetch=FetchType.LAZY` – on y renviendra!!

⇒ **delete from EVENT where EVENT_ID=?**

Même comportement avec **`session.delete(object)`**
avec chargement tardif par défaut

151

Opérations du gestionnaire de persistance (14/15)

Possibilité d'appeler ces méthodes au sein d'un DAO :

```
/*** Home object for domain model class Departement.*/
public class DepartementHome {
   private static final Log log = LogFactory.getLog(DepartementHome.class);
   private final SessionFactory sessionFactory = getSessionFactory();

  ...
  public void persist(Departement transientInstance) {
    log.debug("persisting Departement instance");
    try {
      sessionFactory.getCurrentSession().persist(transientInstance);
      log.debug("persist successful");
     } catch (RuntimeException re) {
        log.error("persist failed", re);
        throw re;
      }
  }
  public void delete(Departement persistentInstance) {
    log.debug("deleting Departement instance");
    try {
     sessionFactory.getCurrentSession().delete(persistentInstance);
     log.debug("delete successful");
    } catch (RuntimeException re) {
        log.error("delete failed", re);
        throw re;
      }
 }

 ...
}
```


152

Opérations du gestionnaire de persistance (15/15)

Collections persistantes :

- Implémentations des collections **`Set`**, **`List`** et **`Map`** propres à Hibernate
- Utilisation des collections persistantes de la même manière que pour les collections Java ordinaires.
- Initialisation par défaut des variables d'instance avec une instance de **`HashSet`**

```
private Set emailAddresses = new HashSet();
```

- Persistance d'une instance initialisée par **`HashSet`** (par **`persist()`** par exemple) ⇒ Remplacement par Hibernate de l'instance par une implémentation de **`Set`** propre à Hibernate

Faire exo 2 exam fév.. 2013

Méta-données et fichiers de *mapping* des classes métiers (1/2)

- Fichier de correspondance (*mapping*) :
  - Fichier XML permettant de faire la correspondance entre les classes persistantes et le modèle relationnel
  - Contenant un ensemble de paramètres ou méta-données
  - Un fichier par classe de préférence, d'extension `hbm.xml` et placé dans le même répertoire que le fichier `.java` de la classe
  - Langage de mapping *Java-centrique* : mappings construits à partir des déclarations des classes persistantes et non des déclarations des tables
- Référence à la DTD :

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
        (http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd
         depuis la version 3.6)
```

Méta-données et fichiers de *mapping* des classes métiers (2/2)

Éléments des fichiers de *mapping* XML :

- Déclaration de la DTD
- Élément racine : **<hibernate-mapping>**
  Possibilité d'y définir l'attribut `package` pour éviter de le spécifier à chaque déclaration de classe et d'association
- **<class>** : pour spécifier la correspondance entre une relation de base de données et une classe
- **<property>** : pour spécifier la correspondance entre une colonne de relation et une propriété de classe
- **<many-to-one>** et **<one-to-one>** : pour spécifier la correspondance d'une association vers une entité
- **<component>** : pour spécifier la correspondance d'une association vers un composant
- **<one-to-many>** et **<many-to-many>** : pour spécifier la correspondance d'une association vers une collection d'entités

Méta-données et annotations

Possibilité d'utiliser des annotations de JPA (*Java Persistence API*) :

- Inclusion du package `javax.persistence`
- Inclusion d'annotations dans les POJO
- Utilisation d'annotations JPA et d'annotations propres a Hibernate
  ⇒ Inclure pour les versions antérieures à 3.6 :
  `hibernate.annotation.jar + ejb3-persistence.jar`
  sinon `hibernate-jpa-2.0-api-1.0.0.Final.jar`
- Compilation des annotations en bytecode et lecture des annotations à l'exécution par Hibernate en utilisant la réflexion Java
- Possibilité de mélanger annotations et fichiers de correspondance XML

NB : Possibilité de décrire les annotations JPA en XML par le langage JPA-XML (pour faciliter le déploiement en cas de changement de configuration)

Dans la suite : présentation des annotations en parallèle du XML

Description de la balise `<class>`

- **`name :`** nom Java complet de la classe (ou interface) persistante
- **`table :`** nom de la relation en base de données – par défaut nom (non-qualifié) de la classe
- **`schema :`** Surcharge le nom de schéma spécifié par l'élément racine (optionnel)

Balise XML

```
<class name="package.Departement" table="departement"
schema="public">
```

Annotations

```
@Entity
@Table(name="departement",schema="public")
public class Departement { … }
```

- \+ d'autres pour gérer la génération des requêtes SQL à l'exécution, la stratégie de verrouillage, les cas où la classe appartient à une hiérarchie etc. *On y reviendra …*

  cf. documentation de référence Hibernate : http://www.hibernate.org/docs

Description de l'élément `<id>` (1/3)

Identité relationnelle de l'entité persistante

- Pour définir la correspondance entre la propriété identificateur de la classe et la clé primaire de la relation
- Obligatoire pour toute classe représentée par une relation de base de données
- **`name`** : nom de la propriété identifiant – si non référencé Hibernate considère que la classe n'a pas d'identifiant
- **`type`** : type Hibernate – déduction par Hibernate du type SQL et du type Java - déterminé par défaut par introspection de la classe
- **`column`** : nom de la colonne clé primaire – par défaut de même nom que la propriété (attribut **`name`**)
- **`unsaved-value`** : valeur de la propriété d'identifiant pour indiquer que l'instance est nouvellement instanciée (non sauvegardée), et la distinguer des instances transients sauvegardées ou chargées dans une session précédente – par défaut **`null`**
- **`access :`** stratégie utilisée par Hibernate pour accéder aux valeurs des propriétés – par défaut **`property`**

Description de la balise <id> (2/3)

<generator> : génération automatique d'id

- Pratique conseillée : clé artificielle
- Générateurs disponibles :
  - **increment** : génère des identifiants de type long, short ou int qui ne sont uniques que si aucun autre processus n'insère de données dans la même table
  - **native** : choisit le générateur en fonction de la base de données (**sequence** pour Oracle ou PostgreSQL, **identity** pour MYSQL par ex.)
  - **assigned** : laisse l'application affecter un identifiant à l'objet avant que la métode **save()** ou **persist()** ne soit appelée.- Stratégie par défaut si aucun **<generator>** n'est spécifié
  - + d'autres – cf. doc Hibernate

```
<id name="id" column="PERSON_ID">
     <generator class="native"/>
</id>
```


Description de la balise <id> (3/3)

Possibilité de spécifier le générateur à un seul endroit pour faciliter le changement de stratégie

Exemple du projet caveatemptor d'Hibernate :

- **Dans le fichier globals.dtd :**

```
<!ENTITY idgenerator    "identity" >
```

- **Dans l'entête de chaque fichier de mapping XML :**

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY % globals SYSTEM
  "classpath://auction/persistence/globals.dtd">
%globals;
]>
```

- **Dans chaque fichier de mapping XML :**

```
<id name= "id" type="…" column="…"
   <generator class= "&idgenerator;" />
</id>
```

Annotation pour l'identificateur

@Id : définition de la propriété identifiant

- Placement de l'annotation avant le getter pour spécifier que l'accès est indirect (~ **`property`** en XML) ou avant la propriété si l'accès est direct – comportement JPA par défaut
- Possibilité d'utiliser l'annotation Hibernate **`@AccessType`**(**`"<strategy>"`**)
- Générateurs disponibles : **`AUTO, TABLE, IDENTITY, SEQUENCE`**

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
public Long getId() { ... }
```

- cf. doc Hibernate

Description de l'élément `<composite-id>` (1/2)

Clé composite :

- Gestion interne :
  - Possibilité de faire correspondre plusieurs attributs de la classe comme propriétés identifiantes
  - **`<key-property>`** : pour faire correspondre une colonne sans contrainte de clé étrangère vers une autre relation
  - **`<key-many-to-one>`** : pour faire correspondre une colonne soumise à une contrainte de clé étrangère vers une autre relation
- Gestion externe (+élégante) : implémenter l'identifiant par une classe composée
  - **`name`** : propriété de type composant qui contient l'identifiant composé
  - **`class`** : classe composant utilisée comme identifiant composé

La classe identifiante doit redéfinir **`equals()`** et **`hashCode()`** et implémenter **`Serializable`**

Description de l'élément <composite-id> (2/2)

```
<class name="test1_package.Enseignement" table="enseignement" schema="public">
        <composite-id name="id" class="test1_package.EnseignementId">
            <key-property name="enseignementId" type="int">
                <column name="enseignement_id" />
            </key-property>
            <key-property name="departementId" type="int">
                <column name="departement_id" />
            </key-property>
        </composite-id>
...
</class>
public class EnseignementId implements java.io.Serializable {
      // Fields
      private int enseignementId;
      private int departementId;
      public EnseignementId() {} /** default constructor */
      public boolean equals(Object other) { …}
      public int hashCode() {…}
     // Property accessors
  …
}
public class Enseignement implements java.io.Serializable {
     // Fields
     private EnseignementId id;
…}
```


Annotations pour les clés composites (1/2)

- **`@Embeddable` :** pour indiquer qu'une classe « embarquée » dans une autre

```
@Embeddable
public class EnseignementId {
     // Fields
      private int enseignementId;
      private int departementId;
    // Property accessors
    @Column(name="enseignement_id", unique=false,
      nullable=false, insertable=true, updatable=true)
    public int getEnseignementId() {
            return this.enseignementId;
        }
    @Column(name="departement_id", unique=false,
      nullable=false, insertable=true, updatable=true)
       public int getDepartementId() {
            return this.departementId;
        }
…}
```

Annotations pour les clés composites (2/2)

- **`@EmbeddedId` :** Pour indiquer que l'identificateur est composite et est géré par une autre classe

```
@Entity
@Table(name="enseignement",schema="public")
public class Enseignement {
    // Fields
    private EnseignementId id;
  …
  // Property accessors
  @EmbeddedId
  @AttributeOverrides( {
   @AttributeOverride(name="enseignementId",
    column=@Column(name="enseignement_id") ),
   @AttributeOverride(name="departementId",
    column=@Column(name="departement_id)
   } )
   public EnseignementId getId() {
       return this.id;
   }
```


Description de la balise `<natural-id>` (1/2)

- Recommandation de l'utilisation de clé primaire générée
- Mais nécessité d'identifier des clés métier (naturelles) pour toutes les entités
- Utilisation de la balise **`<natural-id>`**
- Recommandation d'implémenter **`equals()`** et **`hashCode()`** pour comparer les clés naturelles de l'entité

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

Méta-données et fichiers de *mapping* des classes métiers

Description de la balise `<natural-id>` (2/2)

Fichier de correspondance

```
<class name="User" table="AuctionUser" lazy="true">
    <comment>Users may bid for or sell auction items.</comment>

    <id name="id">
    <generator class="native"/>
    </id>

    <natural-id mutable="true">
          <property name="userName" length="10"/>
    </natural-id>
…
</class>
```

Instruction SQL

```
CREATE TABLE Auctionuser
(
  id SERIAL NOT NULL,
  username varchar(10) NOT NULL,
  …
  CONSTRAINT auctionuser_pkey PRIMARY KEY (id),
  CONSTRAINT auctionuser_username_key UNIQUE (username)
)
```


Méta-données des classes métiers

Annotation pour les clés métiers : `@NaturalId`

```
@Entity
 public class Citizen {
    @Id
    @GeneratedValue
    private Integer id;
    private String firstname;
    private String lastname;
    @NaturalId
    @ManyToOne
    private State state;
    @NaturalId
    private String ssn;
    ...
}

// Exemple de requête
List results = s.createCriteria( Citizen.class )
                .add( Restrictions.naturalId()
                .set( "ssn", "1234" )
                .set( "state", ste ) ).list();
```

Annotation pour les contraintes d'unicité

`@UniqueConstraint` en conjonction avec `@Table`

```
@Entity
@Table(name="departement",schema="public",
uniqueConstraints = { @UniqueConstraint(
    columnNames = { "nom_departement" } ) }
)
public class Departement { …
@Column(name="nom_departement", unique=true,
nullable=false, insertable=true, updatable=true,
length=25)
public String getNomDepartement() {
        return this.nomDepartement;
    }
…
}
```


Description de l'élément <property> (1/2)

Déclaration de propriété persistante :

- **name** : nom de la propriété, avec une lettre initiale en minuscule (cf. convention *JavaBean*)
- **column** : nom de la colonne de base de données correspondante – par défaut de même nom que la propriété
- **type** : nom indiquant le type Hibernate - déterminé par défaut par introspection de la classe
- **update**, **insert** : indique que les colonnes *mappées* doivent être incluses dans les **UPDATE** et/ou les **INSERT -** par défaut à **true** Mettre les deux à **false** empêche la propagation en base de données (utile si vous savez qu'un trigger affectera la valeur à la colonne)
- **access :** stratégie utilisée par Hibernate pour accéder aux valeurs des propriétés – par défaut **property**
- **unique** : Génère le DDL d'une contrainte d'unicité pour les colonnes - optionnel
- **not-null** : Génère le DDL d'une contrainte de non nullité pour les colonnes – optionnel
- + d'autres – cf. doc Hibernate

Description de l'élément <property> (2/2)

```
<hibernate-mapping>
        <class name="events.Event" table="EVENTS">
            <id name="id" column="EVENT_ID">
                <generator class="native"/>
            </id>
            <property name="date" type="timestamp"
                 column="EVENT_DATE"/>
            <property name="title"/>
    …
    </class>
    </hibernate-mapping>

    <property name="nomDepartement" type="string">
          <column name="nom_departement" length="25"
                  not-null="true" unique="true" />
    </property>
```


Annotation @Column

```
@Column(name="reservation_id", unique=true, nullable=false,
     insertable=true, updatable=true)
 public int getReservationId() {
        return this.reservationId;
 }

@Temporal(TemporalType.DATE)
@Column(name="date_resa", unique=false, nullable=false,
     insertable=true, updatable=true, length=4)
public Date getDateResa() {
        return this.dateResa;
}

@Temporal(TemporalType.TIME)
@Column(name="heure_debut", unique=false, nullable=false,
     insertable=true, updatable=true, length=8)
public Date getHeureDebut() {
        return this.heureDebut;
 }

@Enumerated(EnumType.STRING)
public Note getNote() //persistance en String dans la BD
   { ... }
```

Correspondance des associations (1/12)

- Partie la plus complexe dont dépend les performances de l'application
- Balises de correspondance des collections : **`<set>`, `<list>`; `<map>`, `<bag>`, `<array>`** …
- Tags de correspondance des cardinalités/multiplicités : **`<one-to-one>`, `<many-to-one>`, `<many-to-many>`**
- Si correspondance d'une collection : utilisation d'une balise de collection contenant un tag de cardinalité
- Si correspondance d'une association uni ou bi-directionnelle vers une entité : utilisation des balises de cardinalité

Correspondance des associations (2/12)

`<many-to-one>` et `<one-to-one>` : associations uni ou bi-directionnelle vers une entité

- **`name`** : nom de la propriété
- **`column`** : nom de la clé étrangère – possibilité de balise **`<column>`**
- **`class`** : nom de la classe associée – optionnel par défaut le type de la propriété déterminé par réflexion
- **`cascade`** : Propagation des opérations de l'objet père vers les objets associés – optionnel
- **`lazy`** : pour le chargement de l'association - valeur : **`proxy|no-proxy|false`** – optionnel par défaut : **`proxy`**
- **`fetch`** : Pour choisir entre le chargement de type **`outer-join`** ou le chargement par **`select`** successifs - optionnel par défaut à **`select`**
- **`update, insert`** : Indication des colonnes mappées devant être incluses dans des **`UPDATE`** et/ou des **`INSERT`** - optionnel par défaut à **`true`**
- + d'autres – cf. doc Hibernate

Correspondance des associations (3/12)

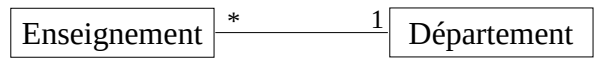
Exemple d'association uni-directionnelle :

```
<class name="test1_package.Enseignement" table="enseignement"
       schema="public">
        ...
        <many-to-one name="departement"
                 class="test1_package.Departement">
            <column name="departement_id" not-null="true" />
        </many-to-one>
        ...
</class>
```


```
/*** Enseignement generated by hbm2java*/
public class Enseignement implements java.io.Serializable {
     // Fields
     private EnseignementId id;
     private Departement departement;
     private String intitule;
     private String description;
     ...
}
```


Correspondance des associations (3bis/12)

Exemple d'annotation pour une association uni-directionnelle :




```
@Entity
@Table(name="enseignement",schema="public"
  , uniqueConstraints = {  } )
public class Enseignement  implements java.io.Serializable {
    // Fields
     private EnseignementId id;
     private Departement departement;
...
     @ManyToOne(fetch=FetchType.LAZY)

     @JoinColumn(name="departement_id", unique=false, nullable=false,
        insertable=false, updatable=false)
     public Departement getDepartement() {
             return this.departement;
     }
...
}
```

Correspondance des associations (4/12)

<one-to-one> :

constrained : dans le cas de clés primaires « partagées » entre deux relations

Dans **Employee.hbm.xml** :

```
<one-to-one name="person" class="Person"/>
```

Dans **Person.hbm.xml** :

```
<one-to-one name="employee" class="Employee"
            constrained="true"/>
```

La clé primaire de `person` fait référence à la clé primaire de `employee`

Correspondance des associations (4/12)

<one-to-one> :

Stratégie Hibernate spéciale de génération d'identifiants par **foreign**

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

⇒ Attribution de la même valeur de clé primaire à une instance fraîchement enregistrée de **Person** et l'instance de **Employee** référencée par la propriété **employee** de **Person**

Correspondance des associations (4bis/12)

Annotation `@OneToOne` :

```
@Entity
public class Employee { ...
     @Id
     public Long getId() { return id; }
     @OneToOne(cascade = CascadeType.ALL)
     @PrimaryKeyJoinColumn
     public Person getPerson() {
         return person;
     }
     ...
}

@Entity
public class Person { ...
     @GeneratedValue(generator = "system-foreign")
       @GenericGenerator(name = "system-foreign", strategy =
       "foreign", parameters = { @Parameter(name = "property",
                                            value = "employee") })
   public String getId() {return _id;}
   @OneToOne(mappedBy = "employee")
   public Employee getEmployee() {return _employee;}
   ...
}
```


Correspondance des associations (5/12)

`<component>` : associations vers une valeur

- Pour faire correspondre deux classes liées par une association **`one-to`** à une seule relation de base de données
- Classe référencée dans la balise **`<component>`** = valeur (par opposition à entité - i.e. pas de cycle de vie propre et pas de possibilité d'être référencée par plusieurs entités)
- **`name`** : nom de la propriété
- **`class`** : nom de la classe du composant – optionnel par défaut le type de la propriété déterminé par réflexion
- + d'autres – cf. doc Hibernate
- **`lazy`** : pour charger le composant dés le premier accès à la variable d'instance – par défaut **`false`**

  On reviendra sur les stratégies de chargement …

Correspondance des associations (6/12)

`<component>` : associations vers une valeur

```
<class name="Person" >
  …
  <component name="Address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
  </component>
</class>
```

Une classe **`Person`** et une classe **`Address`**
Mais une seule relation de base de données : **`Person`**

Possibilité d'inclure une référence au conteneur : balise **`parent`**

```
<parent name="AdressedPerson"/>
```


Correspondance des associations (6bis/12)

```
@Embeddable
    public class Adresse {
       private int numero;
       private String rue;
       private String ville;
    . . .
    }

 @Entity
 public class Person {
       @Embedded private Adresse adresse;
    ou @Embedded Adresse getAdresse() { …};
    . . .
    }
```

Correspondance des associations (6ter/12)

```
@Embeddable
public class Address implements Serializable {
    @Column(length = 255, nullable = false)
     private String street;
    @Column(length = 16, nullable = false)
     private String zipcode;
    @Column(length = 255, nullable = false)
     private String city;
…}

@Entity
@Table(name = "USERS")
public class User implements Serializable {
@Embedded
    @AttributeOverrides( {
        @AttributeOverride(name   = "street",
           column = @Column(name="HOME_STREET", length = 255) ),
        @AttributeOverride (name   = "zipcode",
           column = @Column(name="HOME_ZIPCODE", length = 16) ),
        @AttributeOverride (name   = "city",
            column = @Column(name="HOME_CITY", length = 255) )
        })
 private Address homeAddress;
…}
```


Correspondance des associations (7/12)

Correspondance des collections :

- Implémentations des collections **`Set`**, **`List`** et **`Map`** propres à Hibernate
- Balise propre à chaque type d'interface : **`<set>`**, **`<list>`**, **`<map>`** etc.
  - **`name`** : nom de la propriété contenant la collection
  - **`table`** : nom de la relation contenant la collection - optionnel par défaut nom de la propriété - non utilisé pour les associations **`one-to-many`**
  - **`lazy`** : pour désactiver l'initialisation tardive - par défaut = **`true`**
  - **`inverse`** : Pour définir la collection comme extrêmité "inverse" de l'association bidirectionnelle -
  - + d'autres – cf. doc Hibernate
  - + précision des cardinalité et type de collections

Correspondance des associations (8/12)

Balise `<key column= ... >` :

- Pour spécifier comment effectuer la jointure entre les deux relations entrant en jeu dans l'association **`to-many`**
- **`column`** : nom de la clé étrangère

```
<set name="events" table="PERSON_EVENT">
    <key column="PERSON_ID"/>
    <many-to-many column="EVENT_ID"
    class="events.Event"/>
</set>
```

```
<set name="enseignements" inverse="true">
  <key>
    <column name="departement_id" not-null="true" />
  </key>
  <one-to-many class="package.Enseignement" />
</set>
```

- Possibilité de définir des propriétés pour **`key`** (**`on delete`** etc.) – cf. doc Hibernate

Correspondance des associations (8bis/12)

```
@Entity
@Table(name="departement",schema="public", uniqueConstraints = {
@UniqueConstraint( columnNames = { "nom_departement" } ) })
public class Departement  implements java.io.Serializable {
 ...
   @OneToMany(cascade={CascadeType.ALL}, fetch=FetchType.LAZY,
      mappedBy="departement")
   public Set<Enseignement> getEnseignements() {
        return this.enseignements;
    }
...}
```

```
@Entity
@Table(name="enseignement",schema="public", uniqueConstraints = {})
public class Enseignement  implements java.io.Serializable {
...
   @ManyToOne(cascade={},
        fetch=FetchType.LAZY)

    @JoinColumn(name="departement_id", unique=false, nullable=false,
            insertable=false, updatable=false)
    public Departement getDepartement() {
        return this.departement;
    }
...}
```

Correspondance des associations (9/12)

Balise `<element>` : collection de valeurs

- **`column`** : nom de la colonne contenant les valeurs de l'élément de la collection
- **`formula`** : formule SQL utilisée pour évaluer l'élément – optionnel
- **`type`** : type de l'élément de la collection

```
<set name="emailAddresses"
     table="PERSON_EMAIL_ADDR">
  <key column="PERSON_ID"/>
  <element type="string"
  column="EMAIL_ADDR"/>
</set>
```

Balise équivalente à l'annotation **@ElementCollection**

Correspondance des associations (9bis/12)

Annotation `@ElementCollection` : collection de valeurs

```
@Entity
public class User {

   [...]
   public String getLastname() { ...}


   @ElementCollection
   @CollectionTable(name="Nicknames",
     joinColumns=@JoinColumn (name="user_id"))
     @Column(name="nickname")
   public Set<String> getNicknames() { ... }

}
```

Correspondance des associations (10/12)

Balise `<one-to-many>` : Liaison de deux relations de BD de deux classes par une clé étrangère

- **`class`** : nom de la classe associée
- **`not-found`** : spécification de comment gérer les identifiants cachés qui référencent des lignes manquantes – optionnel par défaut **`exception`**

```
<! Dans le fichier Departement.hnm.xml >
<set name="enseignements" inverse="true">
  <key>
    <column name="departement_id" not-null="true" />
  </key>
  <one-to-many class="test1_package.Enseignement" />
</set>
```

Balise équivalente à l'annotation **@OneToMany**

Correspondance des associations (10bis/12)

`<one-to-many>` et **@OneToMany : exemple**

```
<set name="categorizedItems"
            cascade="all, delete-orphan"
            inverse="true"
            fetch="subselect">
        <key column="CATEGORY_ID" not-null= "true"/>
        <one-to-many class="CategorizedItem"/>
</set>
```

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "category")
@org.hibernate.annotations.Cascade(value =
   org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
@org.hibernate.annotations.Fetch
   (org.hibernate.annotations.FetchMode.SUBSELECT)
private Set<CategorizedItem> categorizedItems =
    new HashSet<CategorizedItem>();
```

Correspondance des associations (11/12)

Balise `<many-to-many>` : en cas de création d'une relation dite d'association

- **`column`** : nom de la colonne clé étrangère
- **`class`** : nom de la classe associée
- **`fetch`** : Activation des récupérations par jointures externes ou par **`selects`** séquentiels pour cette association - par défaut **`join`**

Dans le fichier Person.hbm.xml

```
<set name="events" table="PERSON_EVENT">
   <key column="PERSON_ID"/>
   <many-to-many column="EVENT_ID" class="events.Event"/>
</set>
```

Dans le fichier Event.hbm.xml

```
<set name="participants" table="PERSON_EVENT"
   inverse="true">
   <key column="EVENT_ID"/>
   <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

Balise équivalente à l'annotation @**ManyToMany**

Correspondance des associations (12/12)

Correspondance des associations implementées de manière bidriectionnelle :

- Ne pas oublier de gérer les deux extrémités des associations
- En relationnel, contrairement au Java, réalisation d'une seule opération : mise à jour ou initialisation de la clé étrangère
- En Java : deux opérations – une à chaque extrémité de l'association
- Ne pas oublier de spécifier **`inverse= "true"`** à l'une des extrémité dans le fichier de correspondance (ou l'annotation **`mappedBy`** dans l'une des classes Java) pour éviter la création de deux ordres SQL
- Penser à regrouper les méthodes d'instanciation au sein d'une méthode métier « de cohérence »

Correspondance des clés naturelles

- Balise **<many-to-one>** dans la balise **<natural-id>**
- Création par Hibernate des contraintes `UNIQUE` et des contraintes de clés étrangères correspondantes

Fichier de correspondance

```
<class name="Bid" ...>
    <id name="id">
          <generator class="native"/>
    </id>
    <natural-id>
          <many-to-one name="item"/>
          <property name="amount"/>
    </natural-id>
...
 </class>
```

POJO

```
public class Bid {
 private AuctionItem item;
 private float amount;
 ...
}
```

Instruction SQL

```
CONSTRAINT bid_pkey PRIMARY KEY (id),
CONSTRAINT fk104dd26c25fbe FOREIGN KEY (item)
    REFERENCES auctionitem (id)
CONSTRAINT bid_item_key UNIQUE (item, amount)
```


Correspondance d'une hiérarchie de classes (1/10)

- Trois stratégies d'héritage de base possible sous Hibernate :
  - Une relation de BD par hiérarchie de classe
  - Une relation de BD par classe
    - Avec ou sans discriminateur
  - Une relation de BD par classe concrète
    - Avec ou sans polymorphisme implicite
- Possibilité d'utiliser différentes stratégies de *mapping* pour différentes branches d'une même hiérarchie

Correspondance d'une hiérarchie de classes (2/10)

Une relation par hiérarchie de classes :

- Correspondance de la totalité de la hiérarchie au sein d'une seule relation
- Balise **<subclass>** : pour introduire une sous-classe de la racine de la hiérarchie (pouvant contenir une sous-balise **<subclass>**)
- Balise **<discriminateur>** : pour indiquer la colonne qui contient la valeur permettant de différencier les lignes correspondant aux sous-classes
- Rappels :
  - ⇒ Pas de jointure nécessaire
  - ⇒ Gestion du polymorphisme
  - ⇒ Mais pas de contrainte **NOT NULL** d'où possibilité d'incohérence

Correspondance d'une hiérarchie de classes (3/10)

Une relation par hiérarchie de classes :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment"
              discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Correspondance d'une hiérarchie de classes (3bis/10)

Une relation par hiérarchie de classes :

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="PAYMENT-TYPE",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("PAYMENT")
public abstract class Payment { ... }

@Entity
@DiscriminatorValue("CREDIT")
public class CreditCardPayment extends Payment { ...
}
```


Correspondance d'une hiérarchie de classes (4/10)

Une relation par classe :

- Correspondance de chaque classe (abstraite ou concrète) avec une relation de base de données
- Balise **<joined-subclass>** :
  - Pour introduire une sous-classe de la racine de la hiérarchie (pouvant contenir une sous-balise **<joined-subclass>**)
  - Contenant une balise **<key>** pour spécifier la clé étrangère vers la relation correspondant à la classe mère (clé primaire de la relation correspondant à la sous-classe)
- Rappels :
  - ⇒ Nécessité de jointures
  - ⇒ Garantie de la cohérence
  - ⇒ Accès au polymorphisme

Correspondance d'une hiérarchie de classes (5/10)

Une relation par classe :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment"
                     table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment"
                     table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```


Correspondance d'une hiérarchie de classes (5bis/10)

Une relation par classe :

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Payment implements Serializable
{
   ... }

// Pas besoin de spécifier quoique cela soit pour les
// sous-classes
@Entity
public class CreditCardPayment extends Payment { ...
}

// Possibilité de spécifier explicitement le nom de
// colonne de jointure
@Entity
@PrimaryKeyJoinColumn(name="CHEQUE_ID")
public class ChequePayment  extends Payment { ... }
```

Correspondance d'une hiérarchie de classes (6/10)

Une relation par classe avec discriminateur :

- Mélange des deux stratégies précédentes
- Mélange des balises **<subclass>**, **<discriminateur>** et **<join>**

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
    ...
        </join>
    </subclass>
...
</class>
```

∀ stratégie : Possibilité de créer une association polymorphique vers la classe racine en utilisant **<many-to-one>** pour chaque sous-classe :

```
<many-to-one name="payment" column="PAYMENT_ID"  class="Payment"/>
```


Correspondance d'une hiérarchie de classes (7/10)

Une relation par classe concrète :

- Balise **<union-subclass>**
- Contraintes à respecter **:**
  - Même nom de colonne pour toutes les classes filles pour toute propriété de la classe mère mise en correspondance avec la base de données
  - Stratégie du générateur d'identifiant non permise dans l'héritage de classes filles par union
  - **abstract="true"** : pour une classe mère abstraite (sinon nécessité d'avoir une relation de BD correspondante)
- Rappels **:**
  - Insertion dans une seule relation par instance (si classe mère abstraite)
  - Sélection par UNION (⇒ + lourd et- performant)
  - Garantie de l'intégrité des données
  - Accès au polymorphisme

Correspondance d'une hiérarchie de classes (8/10)

Une relation par classe concrète :

```
<class name="Payment">
  <id name="id" type="long" column="PAYMENT_ID">
      <generator class="sequence"/>
  </id>
  <property name="amount" column="AMOUNT"/>
  ...
  <union-subclass name="CreditCardPayment"
   table="CREDIT_PAYMENT">
      <property name="creditCardType" column="CCTYPE"/>
      ...
  </union-subclass>
  <union-subclass name="CashPayment" table="CASH_PAYMENT">
      ...
  </union-subclass>
  <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
      ...
  </union-subclass>
</class>
```


Correspondance d'une hiérarchie de classes (8bis/10)

Une relation par classe concrète :

```
@Entity
@Inheritance(strategy =
  InheritanceType.TABLE_PER_CLASS)
public abstract class Payment implements Serializable
  {... }

@Entity
@Table(name="CREDIT_PAYMENT")
public class CreditCardPayment extends Payment { ... }
```

Correspondance d'une hiérarchie de classes (9/10)

Une relation par classe concrète avec polymorphisme implicite:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```


Correspondance d'une hiérarchie de classes (9bis/10)

Une relation par classe concrète avec polymorphisme implicite:

```
@MappedSuperclass
public abstract class Payment implements
  Serializable {... }

@Entity
@Table(name="CREDIT_PAYMENT")
@AttributeOverride( name="overrided_att", column =
  @Column(name="column") )
@AssociationOverride( name="association",
  joinColumns = @JoinColumn(name="association_fk") )
public class CreditCardPayment extends Payment { ...
  }
```

Correspondance d'une hiérarchie de classes (10/10)

Une relation par classe concrète avec polymorphisme implicite:

Association polymorphique définie par `<any>`

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT"
    class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```


Préférer la stratégie par **`union`**

Possibilité pour des raisons de lisibilité de faire un fichier par classe (∀ stratégie) : utilisation de **`extends="SuperclassName"`**

Récupération des instances persistantes (1/3)

- Chargement à la demande ou tardif (*lazy loading*) :
  - Pour anticiper les problèmes de performance dus au chargement de gros graphes d'objets
  - Possibilité de charger les entités et collections d'entités persistantes en une fois ou de manière progressive
  - Configuration des fichiers de *mapping* / annotations et possibilité de redéfinir les paramètres à l'exécution
- Requêtes :
  - HQL (*Hibernate Query Language*)
  - EJB-QL (EJB *Query Language*)
  - API Criteria
  - Requêtes natives en SQL (*Native Query*)

Récupération des instances persistantes (2/3)

Stratégies de chargement d'Hibernate3 :

- **Chargement par jointure** : Récupération de l'instance ou la collection dans un même `SELECT`, en utilisant un `OUTER JOIN`
- **Chargement par select** : Utilisation d'un second `SELECT` pour récupérer l'instance ou la collection
  Chargement tardif ⇒ second `SELECT` exécuté uniquement lors de l'accès réel à l'association
- **Chargement par sous-select** : Utilisation d'un second `SELECT` pour récupérer les associations pour toutes les entités récupérées dans une requête ou un chargement préalable
  Chargement tardif ⇒ second `SELECT` exécuté uniquement lors de l'accès réel à l'association
- **Chargement par lot** : stratégie d'optimisation pour le chargement par `SELECT =` récupèration d'un lot d'instances ou de collections en un seul `SELECT` en spécifiant une liste de clé primaire ou de clé étrangère

Récupération des instances persistantes (3/3)

Types de chargement d'Hibernate3 :

- **Chargement immédiat d'une association, une collection ou un attribut** : dès le chargement de l'objet auquel appartient l'élément
- **Chargement tardif d'une collection** : lors de l'invocation d'une méthode sur cette collection par l'application (mode de chargement par défaut pour les collections)
- **Chargement "super tardif" d'une collection** : récupération individuelle des éléments de la collection depuis la base de données lorsque nécessaire ⇒ pas de chargement de toute la collection en mémoire sauf si absolue nécessité (bien adapté aux très grandes collections)
- **Chargement par *proxy* d'une association vers un seul objet** : lors de l'appel sur l'objet associé d'une méthode autre que le getter sur l'identifiant est appelée
- **Chargement "sans *proxy*" d'une association vers un seul objet** : lors de l'accès à cet objet
  Approche moins tardive que par proxy mais plus transparente
- **Chargement tardif des attributs** : un seul attribut ou objet associé chargé lors de son accès

Paramètres de chargement (1/3)

- Par défaut dans le noyau Hibernate :
  - Chargement tardif par `SELECT` pour les collections

Stratégie différente en JPA!!

  - Chargement tardif par *proxy* pour les associations vers un seul objet

Pas de chargement tardif possible pour les objets détachés

- Activation/Désactivation du chargement tardif par l'attribut **`fetch`**

Paramètres de chargement (2/3)

- Valeurs de **`fetch`** :
  - **`select`** : valeur par défaut - ⇒ exécution d'un ordre `SELECT` supplémentaire à la demande pour récupérer les informations
  - **`join`** : utilisation d'une jointure externe (`left outer join`) pour récupérer en une requête la partie du graphe concerné
- **`fetch`** disponible pour les balises `<many-to-one>`, `<one-to-one>`, `<many-to-many>` et pour les collections et balise `<join>`

Paramètres de chargement (3/3)

- Affectation par la stratégie de chargement définie par `fetch` dans les fichiers de mapping :
  - Récupération via `get()` ou `load()`
  - Récupération implicite lors de la navigation à travers une association
  - Requêtes de type `Criteria`
  - Requêtes HQL si utilisation du chargement par `subselect`
- En général : conservation du comportement par défaut et surcharge pour une transaction particulière via HQL ou API Criteria

Exemples de stratégies de chargement (1/4)

- **Chargement tardif (`lazy`) par défaut pour les deux collections `events` et `participants` :**

```
Person aPerson1 = (Person) session.get(Person.class, new
Long(4));
```

⇒ **`select … from PERSON person0_ where person0_.PERSON_ID=?`**

- **Chargement agressif (`eager`) pour la collection `events` avec stratégie par défaut (`select`):**

```
Person aPerson1 = (Person) session.get(Person.class, new
Long(4));
```

⇒ **`select … from PERSON person0_ where person0_.PERSON_ID=?`**
⇒ **`select … from PERSON_EVENT events0_ inner join EVENTS event1_ on events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?`**

- **Chargement agressif pour la collection `events` avec stratégie charg. par jointure (`join`) :**

```
Person aPerson1 = (Person) session.get(Person.class, new
Long(4));
```

⇒ **`select … from PERSON person0_ left outer join PERSON_EVENT events1_ on person0_.PERSON_ID=events1_.PERSON_ID left outer join EVENTS event2_ on events1_.EVENT_ID=event2_.EVENT_ID where person0_.PERSON_ID=?`**

214

Exemples de stratégies de chargement (2/4)

- **Chargement agressif pour les deux collections et avec stratégie charg. par sous-select pour `participants` :**

```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
// La personne 2 est associée à 2 événements dont un associé à 2
// autres personnes
① select … from PERSON person0_ where person0_.PERSON_ID=?
② select … from PERSON_EVENT events0_ inner join EVENTS
    event1_ on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
③ select … from PERSON_EVENT participan0_ inner join PERSON
    person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where
    participan0_.EVENT_ID in (select event1_.EVENT_ID from
    PERSON_EVENT events0_ inner join EVENTS event1_ on
    events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?)
④ select … from PERSON_EVENT events0_ inner join EVENTS event1_
    on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
⑤ select  … from PERSON_EVENT events0_ inner join EVENTS event1_
    on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
⑥ select … from PERSON_EVENT participan0_ inner join PERSON
    person1_ on participan0_.PERSON_ID=person1_.PERSON_ID where
    participan0_.EVENT_ID in (select event1_.EVENT_ID from
    PERSON_EVENT events0_ inner join EVENTS event1_ on
    events0_.EVENT_ID=event1_.EVENT_ID where events0_.PERSON_ID=?)
```

215

Exemples de stratégies de chargement (3/4)

Table PERSON_EVENT

	event_id bigint	person_id [PK] bigint
1	3	2
2	6	2
3	3	4
4	3	5
5	7	5
6	8	5
*		

① Récupération du nuplet correspondant à la personne 2
② Récupération des événements associés à la personne 2 : 3 et 6
③ Récupération des personnes associées aux événements associés à la personne 2 : (3,2) ; (3,4); (3,5) et (6,2)
④ Récupération des événements associés à la personne 4 : il n'y a que 3
⑤ Récupération des événements associés à la personne 5 : 3, 7 et 8
⑥ Récupération des personnes associées aux événements associés à la personne 5 : aucun

216

Exemples de stratégies de chargement (4/4)

- **Chargement agressif pour les deux collections et avec stratégie charg. par select pour `participants` :**

```
Person aPerson1 = (Person) session.get(Person.class, new Long(2));
// La personne 2 est associée à 2 événements dont un associé à 2
// autres personnes
① select … from PERSON person0_ where person0_.PERSON_ID=?
② select … from PERSON_EVENT events0_ inner join EVENTS
    event1_ on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
③ select … from PERSON_EVENT participan0_ inner join PERSON person1_
    on participan0_.PERSON_ID=person1_.PERSON_ID
    where participan0_.EVENT_ID=?
  2 fois car 2 événements au lieu du sous select!!
④ select … from PERSON_EVENT events0_ inner join EVENTS event1_
    on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
⑤ select  … from PERSON_EVENT events0_ inner join EVENTS event1_
    on events0_.EVENT_ID=event1_.EVENT_ID
    where events0_.PERSON_ID=?
⑥ select … from PERSON_EVENT participan0_ inner join PERSON person1_
    on participan0_.PERSON_ID=person1_.PERSON_ID
    where participan0_.EVENT_ID=?
  2 fois car 2 événements au lieu du sous select!!
```

Faire exo 2 exam déc. 2013

Requête HQL (1/4)

- "Langage de requêtes orientées objet" ou encapsulation du SQL selon une logique orientée objet
- Requêtes HQL (et SQL natives) représentées avec une instance de **`org.hibernate.Query`**
- Obtention d'une **`Query`** en utilisant la **`Session`** courante : **`session.createQuery (string)`**
- Clauses : **`from, select, where`**
- Invocation de la méthode **`list()`** ⇒ retour du résultat sous forme de liste

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

Requête HQL (2/4)

- **`from`** :
  - Clause suivie d'un nom de classe et non de relation de BD : **`from Event`**
  - Possibilité d'appliquer la clause sur tout type de classe (abstraite ou concrète, interface)
- **`select`** :
  - Non obligatoire
  - Raisonnement objet ⇒ possibilité de naviguer à travers le graphe d'objets

    ```
    select elements(emailAddresses) from Person
    ```
  - **`element :`** pour récupérer les éléments d'une collection

Requête HQL (3/4)

- **`join`** :
  - Pour exécuter des jointures (**`inner join`**)
  - Exemple :

    ```
    select p from Person p
      join p.events where p.id = 4
    ```
- **`left join`** :
  - Jointure externe (**`left outer join`**)

    ```
    select p from Person p
      left join p.events where p.id = 4
    ```

Requête HQL (4/4)

- **Lier les paramètres** :

```
Long personId ;
Person aPerson = (Person) session
     .createQuery("select p from Person p
          left join fetch p.events where p.id = :pid")
      .setParameter("pid", personId)
      .uniqueResult();
```

- **Générer dynamiquement les requêtes** : API `Criteria`

```
Person aPerson = (Person) session
    .createCriteria(Person.class)
        .setFetchMode("events", FetchMode.JOIN)
     .add( Expression.eq("id", personId) )
     .uniqueResult();
```


API Criteria (1/2)

API d'interrogation par critères dite intuitive et extensible – appartenant au noyau

- **Obtention des instances persistances d'une classe**

```
Criteria crit = session.createCriteria(Person.class);
List resultat = crit.list();
```

← *Exécution de la requête*

- **Définition de critères de recherche**

```
List resultat = session.createCriteria(Person.class)
   .add( Restrictions.like("name", "Manou%") )
   .add( Restrictions.or(
       Restrictions.eq( "age", new Integer(0) ),
       Restrictions.isNull("age")
   ) )
   .list();
```

API Criteria (2/2)

- **Peuplement d'associations de manière dynamique**

```
List resultat = session.createCriteria(Person.class)
    .add( Restrictions.like("name", "Manou%") )
    .setFetchMode("events", FetchMode.JOIN)
    .list();
```

pour choisir le mode de chagement
Valeurs: `FetchMode.JOIN` (pour imposer une jointure externe), `FetchMode.SELECT` (pour charger l'association par un SELECT supplémentaire)
cf. mode de chargement ◁

- **Requêtes par l'exemple**

```
Person p = new Person();
p.setName("Manouvrier");
List results = session.createCriteria(Person.class)
                .add( Example.create(p) ) .list();
```


Requêtes SQL natives

Pour utiliser des requêtes optimisées et tirer partie des spécificités du SGBD utilisé

- **Requêtes natives du noyau**

```
session.createSQLQuery("SELECT * FROM PERSON").list();
```

Retourne une liste d'`Object[]` avec des valeurs scalaires pour chaque colonne de la table PERSON (i.e. retourne une table comme pour les requêtes classiques JDBC)

```
session.createSQLQuery("SELECT * FROM PERSON")
       .addEntity(Person.class);
```

Retourne une liste d'objets de la classe `Person`

- **Requêtes natives en JPA**

Besoin de méta-données pour définir le résultat de la requête sauf si le résultat correspond à des données dont le *mapping* est déjà décrit

```
entitymanager.createNativeQuery("SELECT * FROM PERSON",
                                 Person.class);
```

cf. doc *Hibernate Entity Manager*

Transactions (1/10)

Pour le noyau ou *Hibernate Entity Manager* :

- Utilisation directe des connexions JDBC ainsi que des ressources JTA sans y ajouter davantage de mécanisme de blocage
- Pas de verrouillage des objets en mémoire ni de modification du niveau d'isolation défini au niveau du SGBD
- En cas d'utilisation d'Hibernate dans un environnement léger (application autonome, application web simple ou applications Swing) => délimitation manuelle des transactions
- En cas d'utilisation d'Hibernate dans un environnement J2EE => utilisation des services de gestion automatique des transactions JTA (cf. cours Serveurs d'Appli. Java)
- Règles de base conseillées : **pas d'auto-commit** et **communication avec le SGBD par transaction uniquement** (même pour les lectures)

Transactions (2/10)

```
Session sess = factory.openSession(); // Même chose en utilisant
Transaction tx = null;                // EntityManager
try {
    tx = sess.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Déclenchement automatique de la synchronisation avec la BD

Toutes exceptions soulevées ne sont pas récupérables et doivent être considérées comme fatales pour le gestionnaire de persistance en cours (il faut annuler la transaction et fermer l'unité de travail)

Transactions (3/10)

3 types d'anomalies d'isolation défini dans ANSI SQL92 :

- **Lectures sales ou impropres**
  Une transaction T1 lit des modifications non validées d'items effectuées par T2.
  En cas de annulation de T2, T1 a lu des valeurs invalides
- **Lecture non reproductibles**
  T1 lit un item, T2 modifie ce même item, T1 relit ce item et obtient une valeur différente
- **Lectures fantômes**
  T1 lit un ensemble de nuplets, T2 ajoute/supprime des nuplets, T1 relit l'ensemble de nuplets et obtient un ensemble différent comme résultat

Transactions (4/10)

Degrés d'isolation :

Degré	Lecture impropre	Lecture non reproductible	Références fantômes
READ UNCOMMITTED	OUI	OUI	OUI
READ COMMITTED	NON	OUI	OUI
REPEATABLE READ	NON	NON	OUI
SERIALIZABLE	NON	NON	NON

Définition du niveau d'isolation :

- En SQL :
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
- ***Dans le fichier de config (`hibernate.hbm.xml` ou `persistence.xml`):***
  `<property name="hibernate.connection.isolation" value="x"/>`

  avec `x = 1` pour `read-uncommitted`, `2` pour `read committed`, `4` pour `repeatable read` et `8` pour `serializabe`

Transactions (5/10)

Pour Hibernate et JPA :

- Choix par le développeur et/ou l'administrateur de BD du niveau d'isolation utilisé au niveau de la base de données
- Contrôle de la concurrence de manière optimiste par versionnement – en général associé à un niveau d'isolation `read-committed`
- Garantie des lectures répétables par le contexte de persistance (cache)
- Possibilité de faire du verrouillage pessimiste pour augmenter le niveau d'isolation pour une unité de travail particulière

Transactions (6/10)

Gestion optimiste de la concurrence par versionnement :

- **Optimiste** = supposition d'une exécution correcte et de la rareté des conflits de mise à jour
- **Par versionnement** = utilisation de numéros de version ou d'horodatage pour détecter les mises à jour causant des conflits avec d'autres actualisations antérieures
  - Association d'un numéro de version (nombre ou estampille temporelle) à chaque entité
  - Mise à jour du numéro de version à chaque mise à jour
  - Comparaison automatique des numéros de version
  - Levée d'une exception en cas de conflit
- Méta-données pour la mise en place du versionnement :
  - **`<version>`** ou **`<timestamp>`** en XML (à placer après la prop. `id`)
  - **`@Version`** en annotation (de type `int, Integer, Short, Long` ou `Timestamp`)

Transactions (7/10)

Plusieurs stratégies de gestion optimiste de la concurrence par versionnement :

- En XML :
  `optimistic-lock="none|version|dirty|all"`
  dans la balise **`<class>`**
- Par annotation :
  `@org.hibernate.annotations.Entity(optimisticLock = OptimisticLockType.VERSION | OptimisticLockType.NONE | OptimisticLockType.DIRTY | OptimisticLockType.ALL)`
- **`NONE`** : désactivation de la gestion optimiste
- **`VERSION`** : exploitation de la propriété `<version>` ou `@Version`
- **`DIRTY`**: exploitation des valeurs de propriétés sales (conflit si mises à jour de la même propriétés par deux transactions concurrentes sinon pas de conflit)
- **`ALL`**: exploitation des valeurs de toutes les propriétés (comparaison sur les valeurs et non sur la propriété de version) – si pas de possibilité d'ajouter de numéro de version dans la base – Attention il faut que **`dynamicUpdate=true`** et problème si gestion d'objets détachés

Transactions (8/10)

Gestion optimiste de la concurrence par versionnement :

- Possibilité d'effectuer le contrôle au niveau applicatif : vérification du numéro de version par l'application afin de maintenir l'isolation transactionnelle (approche semblable à celle des EJB)
- Sinon gestion automatique par Hibernate
  - Vérification de la version des objets persistants avant d'effectuer le `flush()`
  - Lancement d'une exception si une modification concurrente est détectée
  - Gestion de l'exception par le développeur

Vérification du numéro de version et de l'entité dans un seul UPDATE

⇒ possibilité d'exécuter des UPDATE inutiles en cas de ré-attachement d'entités détachées

Pour imposer un SELECT avant l'UPDATE : **`select-before-update="true"`** en XML ou **`selectBeforeUpdate=true`** en annotation Hibernate (**`@org.hibernate.annotations.Entity`**)

Transactions (9/10)

Gestion pessimiste de la concurrence :

- Utilisation du mécanisme de verrouillage de la base de données et pas de verrouillage en mémoire
- Niveaux de verrouillage pouvant être obtenus par le noyau Hibernate (*Hibernate Native*) via **`LockMode`** :
  - **`LockMode.NONE`** : absence de verrouillage (pas d'accès à la BD sauf si l'objet n'est pas en mémoire)
  - **`LockMode.WRITE`** obtenu automatiquement quand Hibernate actualise ou insère un enregistrement
  - **`LockMode.READ`** : vérification que le numéro de version d'un objet en mémoire est le même que le numéro de version en BD - obtenu automatiquement quand Hibernate lit des données dans un contexte d'isolation `Repeatable Read` ou `Serializable`
  - **`LockMode.UPGRADE`** (équivalent à `LockModeType.READ` en JPA) : vérification du numéro de version (si applicable) et transformation d'un verrou en lecture par un verrou en écriture si autorisé par le SGBD – retour au mode **`LockMode.READ`** si `SELECT … FOR UPDATE` non permis par le SGBD
  - **`LockMode.UPGRADE_NOWAIT`** : idem que `UPGADE` mais utilisation de `NO WAITS` (lancement d'une exception et pas d'attente de verrou)
  - **`LockMode.FORCE`** (équivalent à `LockModeType.WRITE` en JPA) : incrémentation forcée du numéro de version en BD pour indiquer la MàJ par la transaction courante

Transactions (9bis/10)

Gestion pessimiste de la concurrence :

- Niveaux de verrouillage pouvant être obtenus par l'*EntityManager* d'Hibernate via **`LockMode`** :
  - **`OPTIMISTIC`** (précédemment **`READ`**) : gestion optimiste de la concurrence avec vérification du numéro de version avant toute validation.
  - **`OPTIMISTIC_FORCE_INCREMENT`** (précédemment **`WRITE`**) : gestion optimiste de la concurrence avec vérification du numéro de version avant toute validation et incrémentation forcée du numéro de version.
  - **`PESSIMISTIC_READ:`** application d'un verrouillage en lecture – lectures concurrentes autorisées mais pas d'autorisation d'écriture.
  - **`PESSIMISTIC_WRITE:`** application d'un verrouillage en écriture –pas d'autorisation d'écriture ou de lecture concurrente.

⇒ **Quel que soit le mode de gestion de la concurrence : pas de lecture sale ou non reproductible.**

⇒ **Gestion optimiste de la concurrence : pas de mise en attente des transactions et pas de verrouillage, détection des conflits par versionnement et hypothèse de la rareté des conflits**

⇒ **Gestion pessimiste : utilisation de mécanisme de verrouillage et mise en attente des transactions.**

Transactions (10/10)

- **Méthodes permettant d'obtenir explicitement un niveau de verrouillage :**
  - `Session.load()` en spécifiant un niveau verrouillage `LockMode`.
  - `Session.lock()` (réattache l'objet) ou `EntityManager.lock()` (valable uniquement pour les entités persistantes – ne réattache pas l'objet)
  - `Query.setLockMode()`
- **Accès non transactionnel aux données**
  - Recommandé par EJB3 pour les conversation (longues transactions applicatives) longue
  - Possibilité d'implémenter les conversations en Hibernate, JPA/EJB3
    - Par un gestionnaire d'entités multiples et des objets détachés (manipulation d'entités détachées – problème de ré-attachement)
    - Par la mise en place d'un contexte de persistance étendu (pas de manipulation d'entités détachées – problème de stockage du cycle de vie du gestionnaire d'entités et pour empêcher la synchronisation avec la BD avant la fin de la conversation)