

# Conception d'applications internet

## JPA

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 24 mars 2016

# Annotations : présentation

- Une forme de métadonnée
- Détection d'erreurs supplémentaires
- Configuration du compilateur
- Génération de code (XML, etc.)
- Traitement runtime possible

# Types annotation

```
@interface NameAndNumber {  
    String name();  
    int number() default 1;  
}
```

- *Type* annotation : sorte d'interface ; type doté d'*éléments*
- Élément : type, identifiant, valeur par défaut
- Si pas d'élément, l'annotation est un *Marqueur*
- Si un seul élément, élément souvent nommé *value*
- Prédéfinies ou définies par l'utilisateur

# Annotations

```
@Deprecated; @SuppressWarnings(value = "unchecked");  
@SuppressWarnings("unchecked"); @NameAndNumber(name =  
"Benjamin Franklin", number = 3)
```

- Annotation : nom d'un type annotation et valeurs pour ses éléments
- Si seulement une valeur : utilisée pour value
- Appliquées sur des déclarations (classes, champs, méthodes, usages de types); multiples annotations pour une déclaration permises

Exemple d'usage : `@NonNull String str;`

# Java Persistence API : introduction

- Standard pour gérer la persistance
- Pour Java SE et Java EE
- Modèle Objet / Relationnel (ORM)
- JPA définit les concepts et interfaces, fournisseur JPA les implémente (exemple : Hibernate)
- Pour cette présentation : ORM ; Accès concurrents ; JPA
- Appuié sur JDBC

# Standards JCP

- Implication de « la communauté » pour standards Java
- JCP ?

# Standards JCP

- Implication de « la communauté » pour standards Java
- JCP ? Java Community Process

# Standards JCP

- Implication de « la communauté » pour standards Java
- JCP ? Java Community Process
- Définit les JSR : standards utilisés en Java SE ou Java EE
- Spécifications tiennent compte de nombreux avis d'horizons divers
- JSR 338 : JPA 2.1 ; JSR 345 : EJB 3.2 ; JSR 342 : Java EE 7 ; JSR 346 : CDI...
- Tensions entre standard ouvert et contrôle ! (2010, Apache [quitte](#) le comité JCP ; Doug Lea [également](#), en faveur de OpenJDK...)



# Faiblesses du modèle relationnel pur

Problème : « Mismatch » Objet / Relationnel

- Modèle de données (DM) : objets
- Références directionnelles
- Modèle Entité / Relationnel : sans comportement
- Pas de notion de navigation
- Chargement automatique ?

# Faiblesses du modèle relationnel pur

Problème : « Mismatch » Objet / Relationnel

- Modèle de données (DM) : objets
- Références directionnelles
- Modèle Entité / Relationnel : sans comportement
- Pas de notion de navigation
- Chargement automatique ? Éviter de charger tout le graphe !

# Faiblesses du modèle relationnel pur

Problème : « Mismatch » Objet / Relationnel

- Modèle de données (DM) : objets
- Références directionnelles
- Modèle Entité / Relationnel : sans comportement
- Pas de notion de navigation
- Chargement automatique ? Éviter de charger tout le graphe !
- Problème classique :  $n+1$  select
- Cohérence à maintenir entre BD et objets : types, colonnes, contraintes not null ou autres...
- Répétition lors écriture des requêtes de base
- Difficultés particulières : héritage et autres concepts objet

# Avantages d'une solution ORM

ORM ?

# Avantages d'une solution ORM

ORM ? Object / Relational Mapping

# Avantages d'une solution ORM

## ORM ? Object / Relational Mapping

- Détection des modifications avec accès BD optimisés
- Réduction code répétitif
- Meilleure portabilité
- Permet modèle objet fin
- Facilite refactoring et développement agile

De : [diapos](#) Maude Manouvrier, p. 22

# Modèle et Entité

- DM : Domain Model
- Lien entre DM et BD : *entités* et annotations sur entités
- DM contient des classes *entités*
- Entité : instance représente pas toujours une ligne d'une table

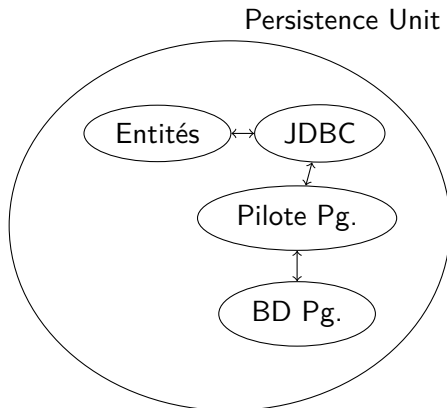
# Définition d'une entité

- Marquer classe `@Entity` (voir aussi `@Table`)
- Marquer champs transients (ou méthodes `get*`) `@Transient` (et persistants `@Column` si désiré)
- Marquer un champ persistant id `@Id` (et `@GeneratedValue`)
- Id représente la clé primaire
- Id initialisé par le fournisseur de persistance (pas de `setId public`) (sauf si clé naturelle, généralement déconseillé)
- Pour permettre concurrence optimiste : marquer un champ persistant version `@Version` (écriture : slmt fournisseur)



# Unité de persistance

- Ensemble d'entités contenu dans une « persistence unit »
- Unité liée à un pilote JDBC et des propriétés de connexion
- Liaison effectuée par configuration xml



## Configuration unité de persistance

- Unité de pers. définie dans META-INF/persistence.xml ([xsd](#))
- Java EE : dans un jar comme bibliothèque du .ear (aussi possible dans EJB ou .war)
- Par défaut, unité contient toutes les entités du projet (non-standard JPA en Java SE)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="...">
  <persistence-unit name="MyPersistenceUnit">
    <properties><property ... /></properties>
  </persistence-unit>
</persistence>
```

# Propriétés unité de persistance

- Génération du schéma par JPA : propriété `javax.persistence.schema-generation.database.action`, valeur `drop-and-create` dans `persistence.xml`
- Propriétés propriétaires, exemple Hibernate
- `"hibernate.show_sql"` (valeur : `true`)
- `"hibernate.format_sql"`
- Java SE : configurer JDBC
- Propriétés `javax.persistence.jdbc.url`,  
`javax.persistence.jdbc.user`,  
`javax.persistence.jdbc.password` (`javax.persistence.jdbc.driver` :  
semble non nécessaire (spec ambiguë))

# Fournisseur JPA

- Fournisseur trouvé dans classpath
- Java EE : fournisseur JPA donné par serveur d'application

Java SE + Maven :

- Dépendance API JPA (fournie par Hibernate) :  
[hibernate-jpa-2.1-api](#)
- [Dépendances](#) implémentation Hibernate (`groupId = org.hibernate`) :
- `hibernate-core` : Hibernate natif
- [hibernate-entitymanager](#) : implémentation JPA
- `hibernate-java8` : types spécifiques java 8, principalement [Date-Time](#)
- Quels scopes ?
- Quelles versions ?

# EntityManager

- Unité associée à `EntityManagerFactory`
- `EntityManager` gère entités dans unité persistance
- Java SE :  
`Persistence.createEntityManagerFactory("MyUnit");`
- Fournisseur JPA lit la configuration et instancie une Factory
- Obtenir la Factory une seule fois (coûteux)
- Java EE : le conteneur s'occupe de l'`EntityManagerFactory`

# Persistance initiale avec EntityManager

Création d'une nouvelle entité :

- `e = new MyEntity(); e.setTruc(...);` // e est « new »
- `entityManager.persist(e);` // e est « managed »
- fermeture du `entityManager` : e est « detached »

## Contexte de persistance

- EntityManager (EM) lié à un *contexte de persistance*
- Contexte : cache d'entités
- Chaque entité peut être associée à son identité persistante

### États d'une instance d'entité

**new** sans identité persistante, pas dans le contexte

**managed** avec identité persistante, dans le contexte

**detached** avec identité persistante, pas dans le contexte

**removed** avec identité persistante, dans le contexte de persistance, marquée pour effacement

# EM et requêtes

- Modifications des entités : modification en mémoire
- Quand nécessaire ou au commit : flush du entity manager
- Flush : synchronisation état contexte avec BD

## Requête retardée

- `e = new MyEntity(); e.setTruc(...);`
  - `entityManager.persist(e);` // *pas de requête à ce stade*
  - `entityManager.flush();` // (ou commit) : INSERT...
- 
- JPA fera par défaut un flush quand nécessaire



# EM et transactions

- Les modifications doivent avoir lieu dans une transaction
- JPA désactive mode auto-commit de JDBC
- Transaction normalement terminée par un commit
- Changements à la BD annulés par un rollback
- Rollback n'annule *pas* les changements en mémoire

## Transaction simple

```
EntityManager em = ...;  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
e = new MyEntity(); e.setTruc...();  
em.persist(e); // pas de requête à ce stade  
tx.commit(); // INSERT ...  
em.close();
```

# Éviter les courants d'air

- Ne pas oublier de fermer le contexte : `em.close()`
- Fermer le contexte *après* le commit
- Possible : plusieurs transactions dans un contexte de persistance

# Transactions et Java EE

- Pilotes JDBC supportent généralement JTA (Java Transaction API)
- Permet entre autres la gestion des transactions par le conteneur
- EJBs : transactions gérées par le conteneur (ou utiliser `@TransactionManagement`)
- Chaque méthode participe alors par défaut à une transaction
- Si pas de transaction en cours lors de l'appel : le conteneur démarre puis termine automatiquement une transaction (commit si ok, rollback si exception sauf exception application)
- Si transaction en cours, la méthode y participe
- Ou annoter la méthode `@TransactionAttribute`

## Unité de persistance en Java EE

- Conteneur Java EE définit `java:comp/DefaultDataSource` : nom utilisé par défaut
- Sinon, indiquer, dans `persistence.xml`, `jta-data-source` : un nom JNDI
- Injection dans EJB : `@PersistenceContext` EntityManager
- Persistence context (par défaut) a le scope de la transaction en cours

# Mise à jour état entités

- État *managed* : entités gérées par persistence unit
- Retient les changements de données
- Synchronisés au flush

## Changement de données

Supposons entité *u managed*

- `u.setTruc(...);` // em enregistre que *u* modifiée en mémoire
- `em.flush();` // UPDATE...

# Récupérer une entité

## Récupération

```
User u = em.find(User.class, 1234); // u est managed (si  
non null)
```

Exemple où le find ne génère pas de SELECT ?

# Récupérer une entité

## Récupération

```
User u = em.find(User.class, 1234); // u est managed (si  
non null)
```

Exemple où le find ne génère pas de SELECT ? Si u est déjà dans le contexte de persistance

# Effacer une entité

## Effacement

Supposons entité *u* *managed*

- `u.remove();` // *u* est *removed*
- `em.flush();` // DELETE...

- Effacer une entité : entité *doit* être « managed »
- Exemple où entité n'est pas « managed » ?



# Effacer une entité

## Effacement

Supposons entité *u* *managed*

- `u.remove();` // *u* est *removed*
- `em.flush();` // DELETE...

- Effacer une entité : entité *doit* être « managed »
- Exemple où entité n'est pas « managed » ? *u* est *new*, ou... ?

```
User u = em.find(User.class, 1234);  
em.getTransaction().commit();  
em.close();  
EntityManager em2 = emfactory.createEntityManager();  
⇒ Dans em2, u est... ?
```

# Effacer une entité

## Effacement

Supposons entité *u* *managed*

- `u.remove();` // *u* est *removed*
- `em.flush();` // DELETE...

- Effacer une entité : entité *doit* être « managed »
- Exemple où entité n'est pas « managed » ? *u* est *new*, ou... ?

```
User u = em.find(User.class, 1234);  
em.getTransaction().commit();  
em.close();  
EntityManager em2 = emfactory.createEntityManager();  
⇒ Dans em2, u est... ? detached
```

# Exercices

- Créer une entité pour un type de votre application, sans liens avec d'autres entités pour le moment
- Faire en sorte que la table correspondante soit créée automatiquement lors du déploiement de l'entité
- Vérifier si ça fonctionne. Se féliciter.
- Permettre CRUD
- Lui ajouter des liens à d'autres objets
- Étendre à toutes les entités de votre application et ajouter des fonctionnalités

# Définition des value types

- Entity type : une classe représentant une table
- Value type : une classe représentant une *partie* d'une table
- Entité a un cycle de vie propre
- Value type attachée à une entité
- Cycle de vie dépend de l'entité parente

## Sortes de value types

- Value type de base : champ `int` par exemple
  - Représente une colonne avec un type simple Java
  - Utilise les conversions JDBC
  - Exemple : Date, int...
- Value type collection : pour liens multiples entre instances
  - Nous n'en parlerons pas ici
- Value type embarqué
  - Représente *plusieurs* colonnes avec un type Java non élémentaire

# Utilité du value type embarqué

## Value type embarqué

- Dans BD, une seule table User
  - User a une adresse
  - Dans modèle objet, classe User et classe Adresse
- 
- Modèle objet : *granularité* plus fine peut être désirable
  - Pourquoi cette différence ?

# Utilité du value type embarqué

## Value type embarqué

- Dans BD, une seule table User
  - User a une adresse
  - Dans modèle objet, classe User et classe Adresse
- 
- Modèle objet : *granularité* plus fine peut être désirable
  - Pourquoi cette différence ?
  - Classes ont des responsabilités
  - Réutiliser une classe, ne pas la dupliquer
  - Schéma BD stable dans le temps

# Utilisation

- Annoter le value type `@Embeddable`
- Annoter son utilisation `@Embedded`
- On peut aussi utiliser un value type dans un value type

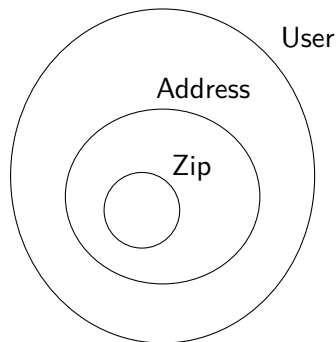


## Exemple d'utilisation

```
@Embeddable
public class Zip {
    private String postalCode;
    private String city;
}

@Embeddable
public class Address {
    private String street;
    @Embedded
    private ZipCode zipCode;
}

@Entity
public class User {
    private String name;
    @Embedded
    private Address address;
}
```



# Utilisation via EntityManager

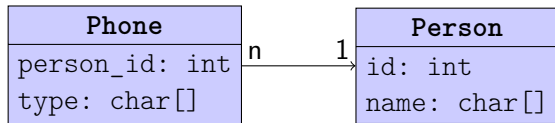
- Créer une instance user
- Créer une instance address
- `user.setAddress(address)`
- `em.persist(user)`
- L'EM persiste le user *et* son adresse
- Cycle de vie address lié à cycle user
- De même, effacement d'un user efface son adresse

## Association one-to-many

- Une personne a plusieurs numéros de téléphone
- BD ?

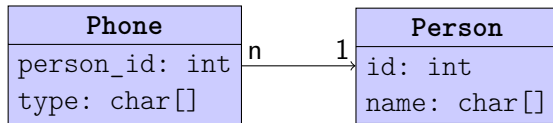
## Association one-to-many

- Une personne a plusieurs numéros de téléphone
- BD ? clé étrangère Phone référence Person



## Association one-to-many

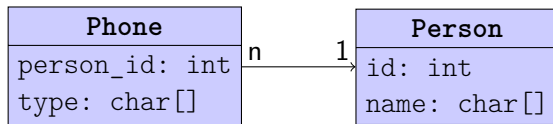
- Une personne a plusieurs numéros de téléphone
- BD ? clé étrangère Phone référence Person



Représentation avec ORM ?

## Association one-to-many

- Une personne a plusieurs numéros de téléphone
- BD ? clé étrangère Phone référence Person



Représentation avec ORM ?

- Classe Phone référence Person
- Classe Person a une collection de Phones
- Les deux

Côté Person appelé parent ; côté Phone appelé enfant

# Mapping proche BD

## Classe Phone référence Person

- Entités normales Phone et Person
- Dans Phone, inclure champ Person
- L'annoter `@ManyToOne` voir aussi `@JoinColumn`
- Retenir : "...ToOne" plutôt que "Many" !
- Crée schema présenté précédemment

`@Entity`

```
public class Phone {  
    @Id ...  
  
    @ManyToOne  
    private Person person;  
}
```

## Usage mapping ManyToOne

- Deux entités aux cycles de vie indépendants
- Par défaut : nécessaire persister *Phone* et *Person*
- `phone.setPerson(person)`
- `em.persist(phone)`
- `em.persist(person)`
- De même, nécessaire effacer *Phone* en plus de *Person*
- Pour éviter ceci : `cascade=ALL` sur annotation `ManyToOne`
- Dès lors persister, effacer, etc. *person* s'applique également à *phones* liés
- NB `cascade` va toujours du parent (*person*) vers ses enfants (*phones*)



## Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ?

## Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ? Peu naturelle

## Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ? Peu naturelle
- Comment trouver les n° de téléphone d'une personne ?

## Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ? Peu naturelle
- Comment trouver les n° de téléphone d'une personne ?
- On peut toujours le faire par requête SQL !
- Mais on peut souhaiter pouvoir le faire via navigation de pointeurs

## Mapping OneToMany unidirectionnel

- Mapping inverse : ref `@OneToMany` unidirectionnelle
- Hibernate utilise une table intermédiaire `Person_Phone`

```
@Entity
```

```
public class Phone {
```

```
    @Id ...
```

```
@Entity
```

```
public class Person {
```

```
    @Id ...
```

```
    @OneToMany
```

```
    private List<Phone> phones;
```

```
}
```

## Usage mapping OneToMany

- Effacement peu efficace : Hibernate enlève tous les phones et réinsère les non-effacés
- Il faut maintenir la référence vers une même collection
- Retrait d'un phone depuis la liste depuis person : penser à effacer également l'entité phone correspondante
- `person.getPhones().remove(phone)`
- `em.remove(phone)`
- Ou effacement automatique avec `orphanRemoval` sur annotation `OneToMany`
- Différence avec cascade ?

## Usage mapping OneToMany

- Effacement peu efficace : Hibernate enlève tous les phones et réinsère les non-effacés
- Il faut maintenir la référence vers une même collection
- Retrait d'un phone depuis la liste depuis person : penser à effacer également l'entité phone correspondante
- `person.getPhones().remove(phone)`
- `em.remove(phone)`
- Ou effacement automatique avec `orphanRemoval` sur annotation `OneToMany`
- Différence avec cascade ? Avec cascade :  
`em.remove(person)`  $\Rightarrow$  efface également ses téléphones

## Mapping OneToMany bidirectionnel

- Et si on veut pouvoir naviguer depuis les deux côtés ?
- OneToMany côté parent, ManyToOne côté enfant
- Il faut un seul propriétaire de la relation
- Propriétaire toujours côté enfant (ManyToOne)
- Côté inverse : préciser mappedBy = "person" sur OneToMany

@Entity

```
public class Phone {  
    @ManyToOne  
    private Person person;  
}
```

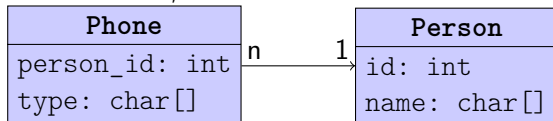
@Entity

```
public class Person {  
    @OneToMany(mappedBy = "person")  
    private List<Phone> phones;  
}
```



## Usage mapping OneToMany bidirectionnel

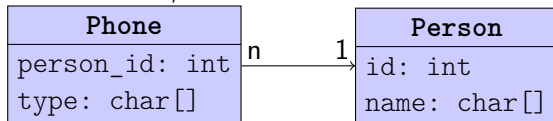
Deux côtés dans modèle, mais un côté dans BD :



- Maintenir les pointeurs corrects dans le modèle des *deux côtés*
- Méthode ajout téléphone :  
`person.getPhones().add(phone) ;`  
`phone.setPerson(person) ;`
- Pourquoi cette nécessité ?

## Usage mapping OneToMany bidirectionnel

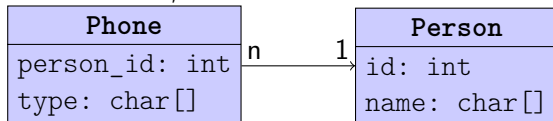
Deux côtés dans modèle, mais un côté dans BD :



- Maintenir les pointeurs corrects dans le modèle des *deux côtés*
- Méthode ajout téléphone :  
`person.getPhones().add(phone) ;`  
`phone.setPerson(person) ;`
- Pourquoi cette nécessité ? Pour code correct indépendamment du code de persistance

## Usage mapping OneToMany bidirectionnel

Deux côtés dans modèle, mais un côté dans BD :



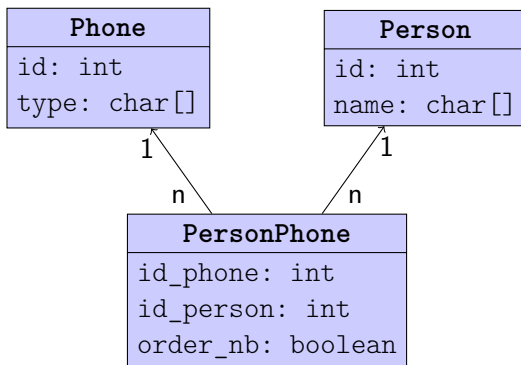
- Maintenir les pointeurs corrects dans le modèle des *deux côtés*
- Méthode ajout téléphone :  
`person.getPhones().add(phone);`  
`phone.setPerson(person);`
- Pourquoi cette nécessité ? Pour code correct indépendamment du code de persistance
- Si cascades, partent toujours du côté parent : effacer person peut être répercuté sur phones
- Nettement plus efficace effacer un phone : seulement une opération

## Association many-to-many

- Deux personnes peuvent partager un téléphone par ex. co-habitants partageant un fixe
- Représentation en BD ?

# Association many-to-many

- Deux personnes peuvent partager un téléphone par ex. co-habitants partageant un fixe
- Représentation en BD ?



# Implémentation

- Déclarer entité PersonPhone autres possibilités existent
- Associations JPA ManyToOne, OneToMany selon contraintes de navigabilité
- Exemple : deux associations bidirectionnelles Phone – PersonPhone et Person – PersonPhone

## Liens entre entités

Exemple : un item a plusieurs bid associés tiré de Java Persistence with Hibernate

- Chaque bout d'une association : to one ou to many
- To one (Item forItem dans Bid) : annoter `@ManyToOne`, préciser éventuellement `optional=false` et `fetch`
- Si bidirectionnelle : votre code doit maintenir la synchronisation, malgré annotations (Pourquoi ?)

## Liens entre entités

Exemple : un item a plusieurs bid associés tiré de Java Persistence with Hibernate

- Chaque bout d'une association : to one ou to many
- To one (Item forItem dans Bid) : annoter `@ManyToOne`, préciser éventuellement `optional=false` et `fetch`
- Si bidirectionnelle : votre code doit maintenir la synchronisation, malgré annotations (Pourquoi ? pour indépendance à ORM)



## Liens entre entités

Exemple : un item a plusieurs bid associés tiré de Java Persistence with Hibernate

- Chaque bout d'une association : to one ou to many
- To one (Item forItem dans Bid) : annoter `@ManyToOne`, préciser éventuellement `optional=false` et `fetch`
- Si bidirectionnelle : votre code doit maintenir la synchronisation, malgré annotations (Pourquoi ? pour indépendance à ORM)
- Pour autre bout d'un lien bidirectionnel (ne compte pas dans DDL) : `@OneToMany` et préciser `mappedBy`
- Liens one to one similaire, utiliser `@OneToOne`

## Chargement d'entités via référence

- Entité u d'état *managed*
- `u.getChildren()` ;
- L'entity manager charge les enfants

## Références vers entités

- Obtenir une représentation d'une entité sans la chercher dans BD
- `em.getReference(User.class, 1234);`
- Seule l'id est initialisée
- Chargement *tardif* (*lazy*) : SELECT seulement si une autre propriété accédée dans le même contexte
- Exemple d'usage ?

## Références vers entités

- Obtenir une représentation d'une entité sans la chercher dans BD
- `em.getReference(User.class, 1234);`
- Seule l'id est initialisée
- Chargement *tardif* (*lazy*) : SELECT seulement si une autre propriété accédée dans le même contexte
- Exemple d'usage ?
- Effacement de l'entité
- Mise au point d'un pointeur

## Références vers entités

- Obtenir une représentation d'une entité sans la chercher dans BD
- `em.getReference(User.class, 1234);`
- Seule l'id est initialisée
- Chargement *tardif* (*lazy*) : SELECT seulement si une autre propriété accédée dans le même contexte
- Exemple d'usage ?
- Effacement de l'entité
- Mise au point d'un pointeur
- Danger ?

## Références vers entités

- Obtenir une représentation d'une entité sans la chercher dans BD
- `em.getReference(User.class, 1234);`
- Seule l'id est initialisée
- Chargement *tardif* (*lazy*) : `SELECT` seulement si une autre propriété accédée dans le même contexte
- Exemple d'usage ?
- Effacement de l'entité
- Mise au point d'un pointeur
- Danger ? Quand contexte fermé, trop tard pour accéder aux autres propriétés de l'entité !
- Conseil : ne pas balader des entités non chargées

# Chargement tardif collections

- Éviter problème  $n+1$  selects
- Attention au problème du produit cardinal
- Défaut JPA : `FetchType` = EAGER sur champs simples, LAZY sur collections

## Définition requêtes

- Définir requête sur entité via `NamedQuery` ou `NamedQueries`
- Lui donner un nom

```
@NamedQuery(  
    name = "getUserByName",  
    query = "SELECT u FROM User u WHERE name = :name"  
)
```

- Dans code, invoquer requête par son nom
- Indiquer les paramètres sur l'objet `Query`

```
TypedQuery<User> typedQuery = entityManager.  
    createNamedQuery("getUserByName", User.class);
```



# JPA QL

```
SELECT DISTINCT pr FROM Person pr
LEFT OUTER JOIN pr.phones ph
WHERE ph is null or ph.type = :phoneType
```

- Dans clause FROM : entités (Person) et variables d'identification (ou aliases) (pr)
- Une seule entité dans FROM : renvoie List<Person>

```
SELECT pr, ph FROM Person pr, Phone ph
WHERE phone.owner = pr
```

- Renvoie List<Object[]>
- Pour chaque entrée entry : entry[0] de type Person, entry[1] de type Phone

## Dynamic fetch en JPA QL

- Ajouter fetch pour obliger fetch de type EAGER

```
SELECT DISTINCT pr FROM Person pr  
    LEFT OUTER JOIN FETCH pr.phones
```

# Merge

- Entité détachée
- Merge : entité redevient managed
- Renvoie une référence
- Ne plus utiliser l'ancienne
- Modifications éventuelles seront reflétées dans DB au flush

```
User u = em.find(User.class, 1234);  
em.getTransaction().commit();  
em.close(); // u détaché du contexte  
u.setName("HerName");  
EntityManager em2 = emFactory.createEntityManager();  
em.getTransaction().begin();  
User uNew = em2.merge(u); // uNew managed  
em2.getTransaction().commit(); // mise à jour BD
```

# Concurrence

- Entité avec `@Version` : protection optimiste par défaut
- Après update, effacement, merge : vérification automatique du n° de version en cache
- Vérification effectuée au moment même ou au flush
- Si versions ne correspondent pas :  
`OptimisticLockException`
- Protection pessimiste : utiliser  
`EntityManager.lock(entity, lockModeType)`

# Différentes égalités

## Trois types d'égalités

- Égalité en mémoire : `a == b`
- Égalité objet : `a.equals(b)`
- Égalité DB : `a.getId().equals(b.getId())`

Quelles propriétés prendre en compte dans `hashCode` et `equals` ?

- Table de hachage : objet ne peut changer de hash / d'égalité
- Dans une session il faut éviter deux objets not `equals()` et concernant la même ligne de la table
- Id pour égalité : pourquoi pas ?

# Différentes égalités

## Trois types d'égalités

- Égalité en mémoire : `a == b`
- Égalité objet : `a.equals(b)`
- Égalité DB : `a.getId().equals(b.getId())`

Quelles propriétés prendre en compte dans `hashCode` et `equals` ?

- Table de hachage : objet ne peut changer de hash / d'égalité
- Dans une session il faut éviter deux objets not `equals()` et concernant la même ligne de la table
- Id pour égalité : pourquoi pas ? Tant que non persistantes, ne fonctionne pas ; change lors sauvegarde

# Différentes égalités

## Trois types d'égalités

- Égalité en mémoire : `a == b`
- Égalité objet : `a.equals(b)`
- Égalité DB : `a.getId().equals(b.getId())`

Quelles propriétés prendre en compte dans `hashCode` et `equals` ?

- Table de hachage : objet ne peut changer de hash / d'égalité
- Dans une session il faut éviter deux objets not `equals()` et concernant la même ligne de la table
- Id pour égalité : pourquoi pas ? Tant que non persistantes, ne fonctionne pas ; change lors sauvegarde

⇒ Ensemble d'attributs déterminants pt de vue utilisateur !  
(username...)

# Patterns

- DM *sans* dépendance persistance (tests ; simplification dépendances)
- Entités transversales (couche web et business)
- Qqs classes service business
- Qqs classes persistance visibles du business
- Ou pattern DAO...



## Place mémoire du contexte

- Le contexte de persistance connaît toutes les entités qu'il gère
- Peut prendre trop de place en mémoire
- Utiliser `em.detach(...)` pour détacher des entités
- Voir aussi `em.clear()`

# Références

- Hibernate 5.1 [User Guide](#)
- Java Persistence with Hibernate : [1<sup>re</sup>](#) édition, [2<sup>e</sup>](#) édition
- [JSR 338](#) (JPA 2.1) ([direct](#))
- [JSR 907](#) (JTA) (moins utile pour développeur final)
- ENORM: An Essential Notation for Object-Relational Mapping, ACM SIGMOD Record 43(2), June 2014, pp 23–28 ([doi](#), [pdf](#) article)

## Votre mission : tout au long du cours

- Indiquer, sur chaque classe, l'auteur principal de la classe (javadoc @author) et éventuellement le pair programmeur (indiquer : Reviewer: ...)
- Relire diapo Auteurs précédente, toujours valable
- Je vous demanderai des remises intermédiaires à intervalles réguliers (cf. diapo suivante)
- Note CC 4 sera sur l'ensemble de ces remises intermédiaires
- Conseil : mettez en place vos entités et gestion de base rapidement. La partie difficile vient après ! (Synchronisation des contextes de persistance et gestion de la concurrence, liens GUI et BD...)

## Votre mission : à court terme

Avant la fin du prochain cours (31 mars, 17h)

- Créer une branche `review`
- Y placer une partie présentable de votre travail
- Requiert de vous accorder avec votre groupe
- Je commenterai cette partie pour vous aiguiller
- L'existence d'un travail de chacun d'entre vous dans cette branche est obligatoire
- Le respect de la date butoir est également obligatoire
- Je vous interrogerai en cas de doute concernant votre participation au code

# Pattern DAO

Extraction des aspects propres à la persistance

- DAO ?

# Pattern DAO

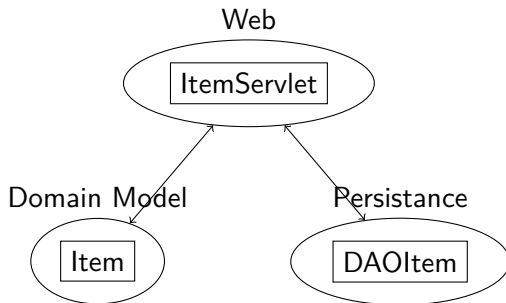
Extraction des aspects propres à la persistance

- DAO ? Data Access Object

# Pattern DAO

Extraction des aspects propres à la persistance

- DAO ? Data Access Object
- « Modèle » découpé en aspects Persistance et Domain Model
- Domain Model : opérations logiques, connaissance métier
- Persistance : seule autorisée à communiquer avec la BD



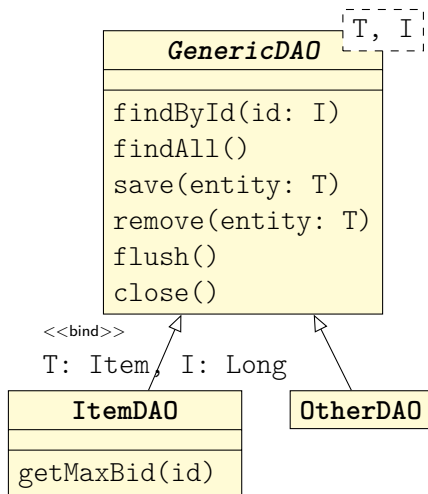
# DAO : mise en œuvre

## Classe parent

- abstraite
- générique :
  - T = type entité
  - I = type ID
- contient les méthodes *CRUD* : Create / Read / Update / Delete

## Sous-classes

- contiennent les méthodes spécialisées



Inspiré par : Java persistence with hibernate



# Licence

Cette présentation, et le code LaTeX associé, sont sous [licence MIT](#). Vous êtes libres de réutiliser des éléments de cette présentation, sous réserve de citer l'auteur.

Le travail réutilisé est à attribuer à [Olivier Cailloux](#), Université Paris-Dauphine.

(Ceci ne couvre pas les images incluses dans ce document, puisque je n'en suis généralement pas l'auteur.)