

# Java EE

## Java Enterprise Edition

### Persistence et JPA



Mohamed SALLAMI



# Entity Bean, introduction

---

- Un Entity Bean représente
  - Des objets persistants stockés dans une base de donnée,
  - Des noms, des données
  - Gestion via JPA2 + sessions beans
- Dans ce chapitre on étudiera
  - Le concept de persistance,
  - Ce qu'est un entity bean, du point de vue du programmeur,
  - Les caractéristiques des entity beans,
  - Les concepts de programmation des entity beans.

# La persistance par sérialisation

- Sérialisation = sauvegarde de l'état d'un objet sous forme d'octets.
  - Rappel : l'état d'un objet peut être quelque chose de très compliqué.
  - Etat d'un objet = ses attributs, y compris les attributs hérités.
  - Si les attributs sont eux-même des instances d'une classe, il faut sauvegarder aussi les attributs de ces instances, etc...
- A partir d'un état sérialisé, on peut reconstruire l'objet
- En java, au travers de l'interface `java.io.Serializable`, des méthodes de `java.io.ObjectInputStream` et `java.io.ObjectOutputStream`

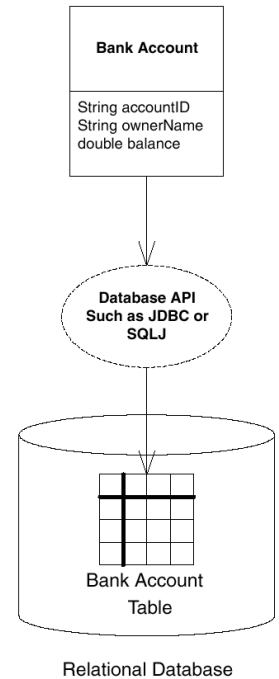
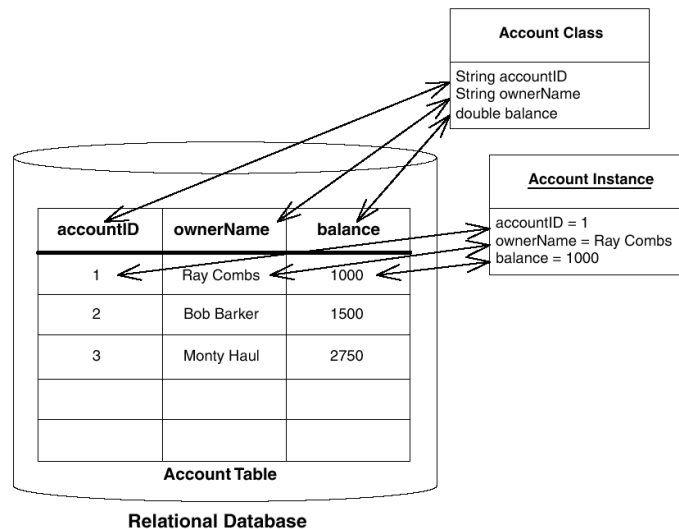
# La persistance par sérialisation

---

- Défauts nombreux...
- Gestion des versions, maintenance...
- Pas de requêtes complexes...
  - Ex : on sérialize mille comptes bancaires.  
Comment retrouver ceux qui ont un solde négatif ?
- Solution : stocker les objets dans une base de donnée!

# La persistance par mapping objet/BD relationnelle

- On stocke l'état d'un objet dans une base de donnée.
- Ex : la classe `Personne` possède deux attributs `nom` et `prenom`, on associe cette classe à *une table* qui possède deux colonnes : `nom` et `prenom`.
- On décompose chaque objet en une suite de variables dont on stockera la valeur dans une ou plusieurs tables.
- Permet des requêtes complexes.



# La persistance par mapping objet/BD relationnelle

---

- Pas si simple...
  - Détermination de l'état d'un objet parfois difficile, tout un art...
  - Il existe des produits pour nous y aider... EclipseLink, TopLink (WebGain), Hibernate (JBoss),
  - Aujourd'hui la plupart des gens font ça à la main avec JDBC ou SQL/J.
  - Mais SQL dur à tester/debugger... source de

# Le modèle de persistance JPA 2

---

- JPA 2 propose un modèle standard de persistance à l'aide des Entity beans
- Les outils qui assureront la persistance (Toplink, Hibernate, EclipseLink, etc.) sont intégrés au serveur d'application et devront être compatibles avec la norme JPA 2.
- Java EE 6 repose à tous les niveaux sur de « l'injection de code » via des annotations de code
  - Souvent, on ne fera pas de « new », les variables seront créées/initialisées par injection de code.

# Qu'est-ce qu'un Entity Bean

---

- Ce sont des objets qui savent se *mapper* dans une base de donnée.
- Ils utilisent un mécanisme de persistance (parmi ceux présentés)
- Ils servent à représenter sous forme d'objets des données situées dans une base de donnée
  - Le plus souvent un objet = une ou plusieurs ligne(s) dans une ou plusieurs table(s)



# Qu'est-ce qu'un Entity Bean

## ■ Exemples

- ☐ Compte bancaire (No, solde),
- ☐ Employé, service, entreprises, livre, produit,
- ☐ Cours, élève, examen, note,

## ■ Mais au fait, pourquoi nous embêter à passer par des objets ?

- ☐ Plus facile à manipuler par programme,
- ☐ Vue plus compacte, on regroupe les données dans un objet.
- ☐ On peut associer des méthodes simples pour manipuler ces données...
- ☐ On va gagner la couche middleware !

# Exemple avec un compte bancaire

- On lit les informations d'un compte bancaire en mémoire, dans une instance d'un entity bean,
- On manipule ces données, on les modifie en changeant les valeurs des attributs d'instance,
- Les données seront mises à jour dans la base de données automatiquement !
- Instance d'un entity bean = *une vue* en mémoire des données physiques

# Fichiers composant un entity bean

- Schéma classique :
  - La classe du bean se mappe dans une base de données.
  - C'est une classe java « normale » (POJO) avec des attributs, des accesseurs, des modifieurs, etc.
  - On utilisera **les méta-données ou « annotations de code »** pour indiquer le mapping, la clé primaire, etc.
    - Clé primaire = un objet sérializable, unique pour chaque instance. C'est la clé primaire au sens SQL.
    - Note : on peut aussi utiliser un descripteur XML à la place des annotations de code
  - On manipulera les données de la BD à l'aide des EntityBeans + à l'aide d'un PERSISTENT MANAGER.
  - Le PM s'occupera de tous les accès disque, du cache, etc.
    - Lui seul contrôle quand et comment on va accéder à la BD, c'est lui qui génère le SQL, etc.

# Exemple d'entity bean : un livre

**@Entity**

```
public class Book {
```

```
    @Id @GeneratedValue
```

```
    private Long id;
```

```
    @Column(nullable = false)
```

```
    private String title;
```

```
    private Float price;
```

```
    @Column(length = 2000)
```

```
    private String description;
```

```
    private String isbn;
```

```
    private Integer nbOfPage;
```

```
    private Boolean illustrations;
```

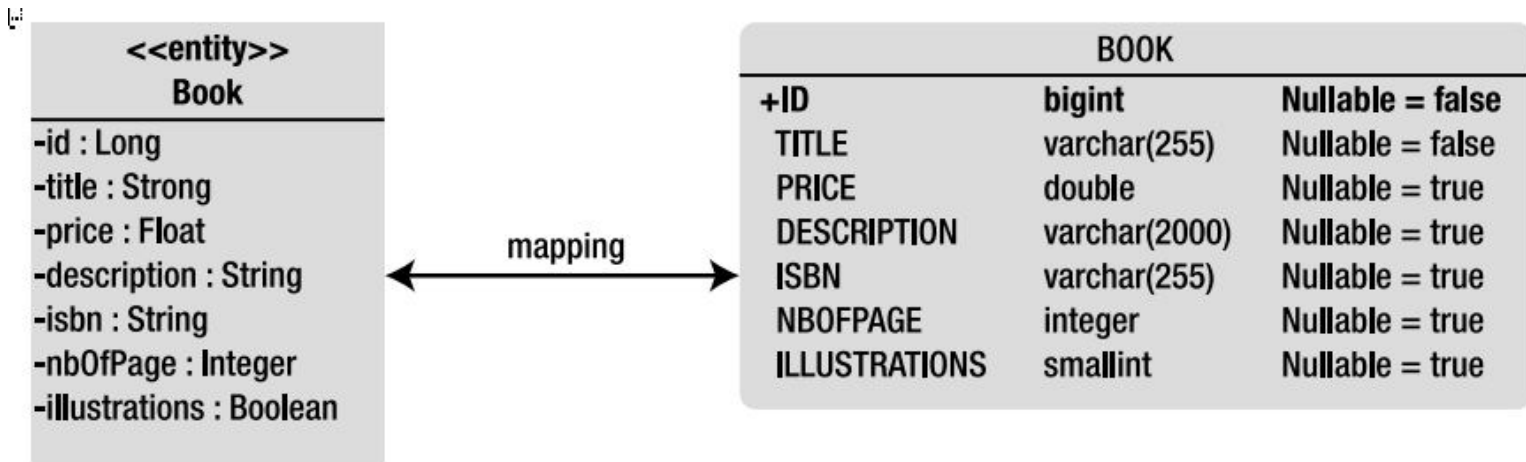
```
    // Constructors, getters, setters
```

```
}
```

# Exemple d'entity bean : un livre

```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
```



# Les annotations de code JPA 2

---

- Nombreuses valeurs par défaut, par exemple une classe entité `Personne` se mapperà dans la table `PERSONNE` par défaut, un attribut « nom » sur la colonne `NOM`, etc.
- Il existe de très nombreux attributs pour les annotations, ce cours présente les principaux, pour une étude détaillée, voir la spécification.

# Exemple d'insertion d'un livre

```
public class Main {  
  
    public static void main(String[] args) {  
  
        // On crée une instance de livre  
        Book book = new Book();  
        book.setTitle("The Hitchhiker's Guide to the Galaxy");  
        book.setPrice(12.5F);  
        book.setDescription("Science fiction comedy book");  
        ...  
        // On récupère un pointeur sur l'entity manager  
        // Remarque : dans une appli web, pas besoin de faire tout cela !  
        EntityManagerFactory emf =  
            Persistence.createEntityManagerFactory("chapter02PU");  
        EntityManager em = emf.createEntityManager();  
        // On rend l'objet « persistant » dans la base (on l'insère)  
        EntityTransaction tx = em.getTransaction();  
        tx.begin();  
        em.persist(book);  
        tx.commit();  
        em.close();  
        emf.close();  
    }  
}
```

# Client sous forme de session bean

```
@Stateless
```

```
public class BookBean {
```

```
    @PersistenceContext(unitName = "chapter04PU")
```

```
    private EntityManager em;
```

```
    public void createBook() {
```

```
        Book book = new Book();
```

```
        book.setId(1234L);
```

```
        book.setTitle("The Hitchhiker's Guide to the Galaxy");
```

```
        book.setPrice(12.5F);
```

```
        book.setDescription("Science fiction created by Douglas Adams.");
```

```
        book.setIsbn("1-84023-742-2");
```

```
        book.setNbOfPage(354);
```

```
        book.setIllustrations(false);
```

```
        //Les transactions sont déclenchées par défaut,
```

```
        em.persist(book);
```

```
        // Récupère le livre dans la BD par sa clé primaire
```

```
        book = em.find(Book.class, 1234L);
```

```
        System.out.println(book);
```

```
    }
```

```
}
```



# Remarques : à quoi correspond le session bean ?

---

- On codera la partie « métier »
  - Souvent on utilise un session bean pour la couche « DAO » (avec des fonctions de création, recherche, modification et suppression d'entity beans)
    - Exemple : GestionnaireUtilisateurs
  - On utilisera aussi des session beans pour implémenter des services composites
    - Exemple : GestionnaireDeCommandes, qui utilisera d'autres gestionnaires

# Autres annotations

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "book_title", nullable = false, updatable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    @Column(name = "nb_of_page", nullable = false)
    private Integer nbOfPage;
    private Boolean illustrations;
    @Basic(fetch = FetchType.LAZY)
    @Lob
    private byte[] audioText;

    // Constructors, getters, setters
}
```

# Autres annotations (suite)

- @Column permet d'indiquer des préférences pour les colonnes
  - Attributs possibles : name, unique, nullable, insertable, updatable, table, length, precision, scale...
- @GeneratedValue
  - Indique la stratégie de génération automatique des clés primaires,
  - La valeur : GenerationType.auto est recommandée,
  - Va ajouter une table de séquence
- @Lob indique « large object » (pour un BLOB)
  - Souvent utilisé avec @Basic(fetch = FetchType.LAZY) pour indiquer qu'on ne chargera l'attribut que lorsqu'on fera un get dessus

# Autres annotations (suite)

---

- Il existe de nombreuses autres annotations,
  - Voir par exemple : JPA Reference  
<http://www.objectdb.com/api/java/jpa>
- Les curieux peuvent consulter la spécification java EE 6 ou le tutorial (ou un bon livre)

# Packager et déployer un Entity Bean

- Les EB sont déployés dans des « persistence Units »,
  - Spécifié dans le fichier « persistence.xml » qui est dans le jar contenant les EJBs.
  - Exemple le plus simple :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="intro"/>
</persistence>
```

- Mais on peut ajouter de nombreux paramètres :
  - <description>, <provider>, <transaction type>, <mapping file> etc.

# Entity Manager

- L'objet EntityManager est responsable de la gestion des entités et de leurs états. Il va ainsi permettre les opérations de base offertes par le langage relationnel que sont :
  - l'ajout ; la lecture ; la mise à jour ; la suppression.
- L'EntityManager est donc l'objet qui va permettre au développeur de manipuler ses objets Java devenus des entités et ainsi lui permettre de les persister. Il est donc nécessaire d'obtenir une référence vers un objet EntityManager; cela s'effectue par l'appel à la méthode factory de la classe EntityManagerFactory, comme montré ci-dessous :

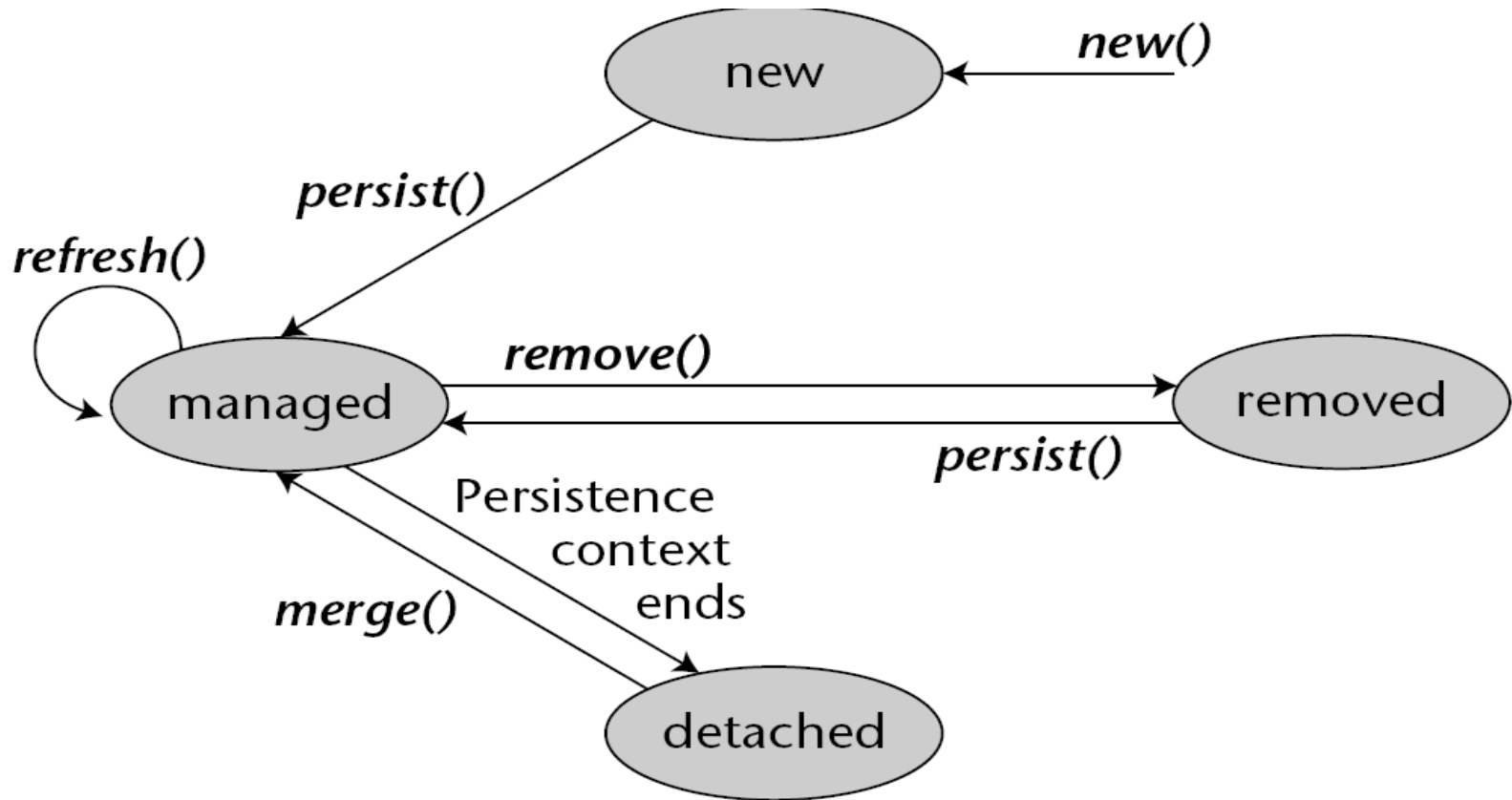
```
EntityManagerFactory factory = Persistence.createEntityManagerFactory();  
EntityManager em = factory.createEntityManager();
```

# Entity Manager

- On peut ensuite utiliser cette instance pour réaliser les opérations expliquées ci-dessous :

```
Person p = new Person(); // create a new Person
p.setFirstName("Foo");
p.setLastName("Barr");
em.persist(p); // save p in database
p.setLastName("BAR");
em.merge(p); // update p
// retrieve p, it's a Person with the primary key id
p = em.find(Person.class, p.getId());
// remove p
em.remove(p);
```

# Que faire avec un entity manager ?



**Figure 6.3** Entity life cycle.



# Etats d'un Entity Bean

- Un EB peut avoir 4 états
  1. **New**: le bean existe en mémoire mais n'est pas encore associé à une BD, il n'est pas encore associé à un contexte de persistance (via l'entity manager)
  2. **Managed** : après le persist() par exemple. Le bean est associé avec les données dans la BD. Les changements seront répercutés (transaction terminées ou appel a flush())
  3. **Detached** : le bean est n'est plus associé au contexte de persistance
  4. **Removed** : le bean est associé à la BD, au contexte, et est programmé pour être supprimé (les données seront supprimées aussi).

# Utilisation du persistent manager

---

- Remove() pour supprimer des données,
- Set(), Get(), appel de méthodes de l'entity bean pour modifier les données, mais le bean doit être dans un état « managed »,
- Persist() pour créer des données, le bean devient managé,
- Merge pour faire passer un bean « detached » dans l'état « managed ».

# Exemple de merge() avec le bean stateless

```
public Account openAccount(String ownerName) {  
    Account account = new Account();  
    account.ownerName = ownerName;  
    manager.persist(account);  
    return account;  
}
```

```
public void update(Account detachedAccount) {  
    Account managedAccount = manager.merge(detachedAccount);  
}
```

# Recherche d'entity beans

---

- Les entity beans correspondant à des lignes dans une BD, on peut avoir besoin de faire des recherches.
- Similaire à un SELECT
- Plusieurs fonctions sont proposées par l'entity manager

# Recherche d'entity beans

## ■ Recherche par clé primaire :

```
/** Find by primary key. */  
public <T> T find(Class<T> entityClass, Object primaryKey);
```

## ■ Exécution de requêtes JPQL

```
public List<Account> listAccounts() {  
    Query query = manager.createQuery("SELECT a FROM Account a");  
    return query.getResultList();  
}
```

# Recherche d'entity beans

## ■ Requêtes SQL:

```
public Query createNativeQuery(String sqlString, Class resultClass);

public Query createNativeQuery(String sqlString,
    String resultSetMapping);
```

## ■ Requêtes nommées:

```
public List<Account> listAccounts() {
    Query query = manager.createNamedQuery("findThem");
    return query.getResultList();
}
```

```
@Entity
@NamedQuery(name="findThem", queryString="SELECT a FROM Account a")
public class Account implements Serializable {...}
```

# Le langage JPQL

- Le langage JPQL est un langage de requête dont la grammaire est définie par la spécification J.P.A. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. On trouve également le nom d'EJBQL dans la littérature Java, il s'agit du nom donné à ce langage dans la norme EJB2.
- Voici un exemple de requête SQL repérant l'ensemble des étudiants puis sa transcription en langage JPQL :  

```
// SQL Query  
SELECT * FROM STUDENT  
  
// JPQL Query  
SELECT s FROM Student s
```
- On remarque une similitude évidente, ce qui garanti une facilité d'utilisation pour toutes personnes ayant déjà pratiqué le langage SQL.

# Le langage JPQL

- De la même façon on peut modifier ou supprimer un étudiant par les requêtes suivantes :

// Update Query

```
UPDATE Student s SET s.firstName = "foo", s.lastName =  
    "bar" WHERE s.firstName = "toto"
```

// Delete Query

```
DELETE Student s WHERE s.firstName = "foo" AND  
    s.lastName = "bar"
```

- JPA permet l'utilisation des langages SQL et JPQL. Cependant, pour des soucis d'abstraction et de portabilité, il est fortement recommandé d'utiliser uniquement le langage de la norme JPA.



# Les requêtes et l'objet Query

- L'objet Query va permettre de créer une requête (JPQL ou SQL). Pour pouvoir construire un objet Query, il est nécessaire de posséder une instance d'EntityManager comme démontré précédemment, puis d'appeler l'une des méthodes factory suivantes en fonction du langage utilisé pour formuler la requête :

```
// em is an instance of EntityManager
```

```
// create a JPQL Query
```

```
Query jQuery = em.createQuery("Select s From Student s");
```

```
// create a SQL Query
```

```
Query sQuery = em.createNativeQuery("SELECT * FROM  
STUDENT");
```

- On peut ensuite procéder à la récupération des données de la façon suivante :

```
List<Student> rList = jQuery.getResultList();
```

# Requête nommée

- J.P.A permet d'ajouter encore plus de clarté et de cohérence au code grâce à l'utilisation des requêtes nommées.
- Les requêtes nommées sont définies par des méta-données (via annotation ou fichier XML) et sont associées à une entité. Elles vont permettre d'attribuer un nom à une requête JPQL.
- Voici un exemple de requête nommée associée à la classe Person grâce aux annotations :

```
@NamedQuery("Person.findAll", "Select s from Person s")
```

```
@Entity
```

```
public class Person { ... }
```

- Notre requête ainsi créée est utilisée pour construire l'objet Query associé, via la méthode adéquate de l'EntityManager :

```
Query nQuery = em.createNamedQuery("Person.findAll");
```

# Requête nommée

---

- L'usage des requêtes nommées est une bonne pratique qu'il est bon d'empreinter dès la création des entités et lors de la mise en place de nouvelles requêtes.
- Leur usage permet de mutualiser le code et rapproche l'objet de la requête qui l'interroge.
- Plus de cohérence.
- De plus ces requêtes sont compilées lors du déploiement des entités via les modules EJB3, cela permet d'en vérifier la validité syntaxique et sémantique.

# Relations avec les entity beans

# On complique un peu l'étude des entity beans !

---

- Les entity beans représentant des données dans une BD, il est logique d'avoir envie de s'occuper de gérer des relations
- Exemples
  - ☐ Une commande et des lignes de commande
  - ☐ Une personne et une adresse
  - ☐ Un cours et les élèves qui suivent ce cours
  - ☐ Un livre et ses auteurs
- Nous allons voir comment spécifier ces relations dans notre modèle EJB

# Concepts abordés

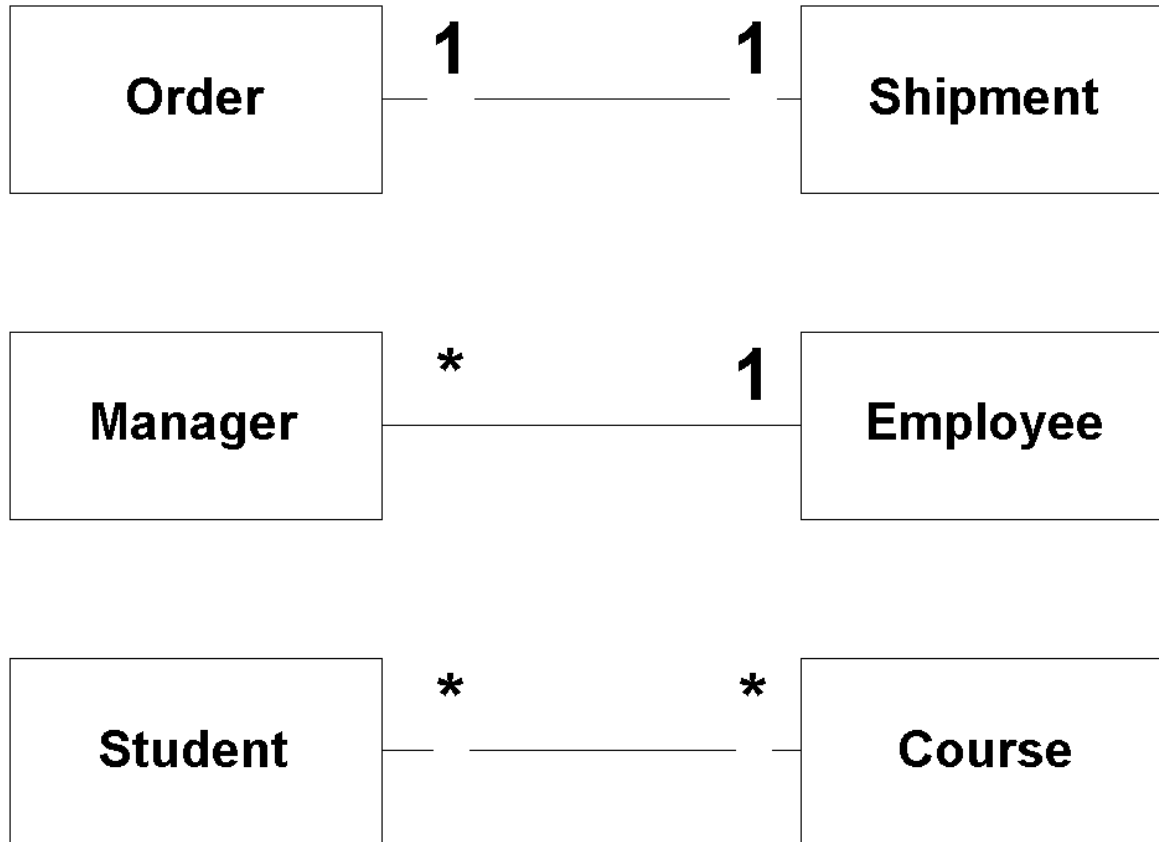
---

- Cardinalité (1-1, 1-n, n-n...),
- Direction des relations (bi-directionnelles, uni-directionnelles),
- Agrégation vs composition et destructions en cascade,
- Relations récursives, circulaires, aggressive-load, lazy-load,
- Intégrité référentielle,
- Accéder aux relations depuis un code client, via des Collections,
- Comment gérer tout ça !

# Direction des relations et cardinalité

- Direction des relations
  - Unidirectionnelle: On ne peut aller que du bean A vers le bean B
  - Bidirectionnelle: On peut aller du bean A vers le bean B et inversement
- La cardinalité indique combien d'instances vont intervenir de chaque côté d'une relation
- One-to-One (1:1)
  - Un employé a une adresse...
- One-to-Many (1:N)
  - Un PDG et ses employés...
- Many-to-Many (M:N)
  - Des étudiants suivent des cours...

# Cardinalité






# Relations 1:1

- Représentée typiquement par une clé étrangère dans une BD
- Ex : une commande et un colis

OrderPK	OrderName	Shipment ForeignPK
12345	Software Order	10101

ShipmentPK	City	ZipCode
10101	Austin	78727

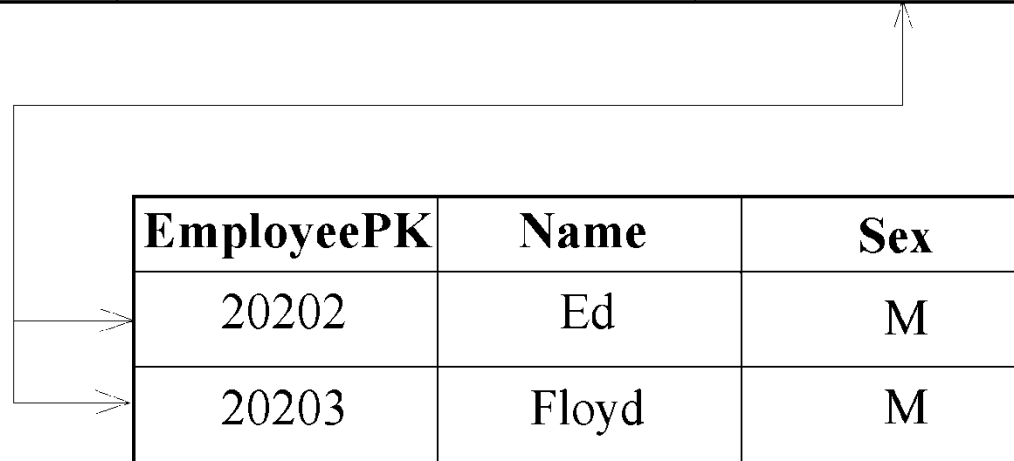


# Relations 1:N

- Exemple : une entreprise a plusieurs

CompanyPK	Name	Employee FKs
12345	The Middleware Company	<Vector BLOB>

EmployeePK	Name	Sex
20202	Ed	M
20203	Floyd	M




# Relations 1:N

- Exemple : une entreprise a plusieurs employés

□ Solution plus propre (éviter les BLOBs!)

CompanyPK	Name
12345	The Middleware Company

EmployeePK	Name	Sex	Company
20202	Ed	M	12345
20203	Floyd	M	12345



# Relations M:N

- Un étudiant suit plusieurs cours, un cours a plusieurs étudiants inscrits
  - Table de jointure nécessaire.

<b>StudentPK</b>	<b>StudentName</b>
10101	Joe Student

<b>EnrollmentPK</b>	<b>StudentPK</b>	<b>CoursePK</b>
12345	10101	20202

<b>CoursePK</b>	<b>CourseName</b>
20202	EJB for Architects

# Relations M:N, conception

---

- on peut utiliser simplement deux EJB,  
chacun ayant un attribut correspondant à  
une Collection de l'autre EJB...

# Lazy-loading des relations

## ■ Agressive-loading

- Lorsqu'on charge un bean, on charge aussi tous les beans avec lesquels il a une relation.
- Cas de la Commande et des Colis plus tôt dans ce chapitre.
- Peut provoquer un énorme processus de chargement si le graphe de relations est grand.

## ■ Lazy-loading

- On ne charge les beans en relation que lorsqu'on essaie d'accéder à l'attribut qui illustre la relation.
- Tant qu'on ne demande pas quels cours il suit, le bean Etudiant n'appelle pas de méthode *finder* sur le bean Cours.

# Agrégation vs Composition et destructions en cascade

## ■ Relation par Agrégation

- ☐ Le bean *utilise* un autre bean
- ☐ Conséquence : si le bean A *utilise* le bean B, lorsqu'on détruit A on ne détruit pas B.
- ☐ Par exemple, lorsqu'on supprime un étudiant on ne supprime pas les cours qu'il suit. Et vice-versa.

## ■ Relation par Composition

- ☐ Le bean *se compose d'un* autre bean.
- ☐ Par exemple, une commande se compose de lignes de commande...
- ☐ Si on détruit la commande on détruit aussi les lignes correspondantes.
- ☐ Ce type de relation implique des destructions en cascade..

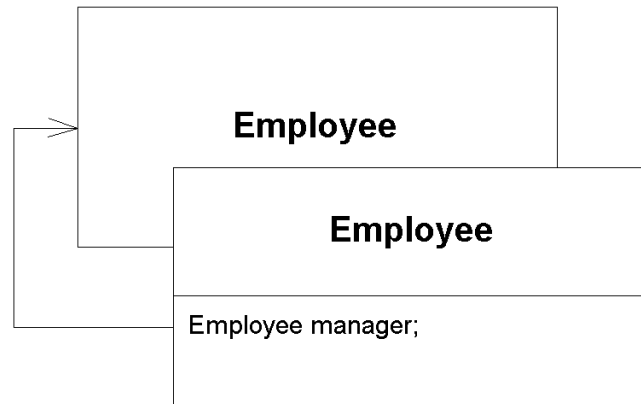
# Relations et JPQL

- Lorsqu'on définit une relation, on peut aussi indiquer la requête qui permet de remplir le champs associé à la relation.
- On fait ceci à l'aide de JPQL  
**SELECT o.customer FROM Order o**
- Principale différence avec SQL, l'opérateur "."
  - Pas besoin de connaître le nom des tables, ni le nom des colonnes...



# Relations récursives

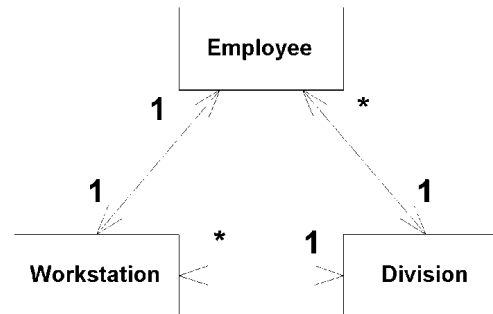
- Relation vers un bean de la même classe
  - Exemple : Employé/Manager



- Rien de particulier, ces relations sont implémentées exactement comme les relations non récursives...

# Relations circulaires

- Similaire aux relations récursives sauf qu'elles impliquent plusieurs types de beans
  - Ex : un employé travaille dans une division, une division possède plusieurs ordinateurs (workstation), une workstation est allouée à un employé...



- Ce type de relation, en cas de aggressive-loading peut mener à une boucle sans fin...
  - Même problème avec les destructions en cascade...

# Relations circulaires

- Plusieurs stratégies sont possibles
  1. Certains containers proposent d'optimiser le chargement d'un bean en chargeant toutes ses relations en cascade. Attention si relations circulaires !
  2. Supprimer une des relations (!!!) si le modèle de conception le permet.
  3. Supprimer la bidirectionnalité d'une des relations pour briser le cercle, si le modèle de conception le permet.
  4. Utiliser le lazy-loading et ne pas faire de destruction en cascade.
  5. Les meilleurs moteurs CMP détectent les relations circulaires et vous permettent de traiter le problème avant le runtime.

# Trier les relations

- Lorsqu'on accède aux relations par un getter, on ne contrôle pas par défaut l'ordre des éléments.
- Plusieurs solutions sont possibles pour récupérer des relations sous forme de collections triées
  - Utiliser l'annotation `@OrderBy` juste avant la déclaration de la relation ou juste avant le getter
  - Utiliser une requête avec un Order By
  - Annoter l'attribut correspondant à la colonne qui sera ordonnée, dans l'entity de la relation

# Trier des relations : annotation @OrderBy

```
@Entity public class Course {  
    ...  
    @ManyToMany  
    @OrderBy("lastname ASC")  
    List<Student> students;  
    ...  
}
```

## ■ Remarques

- ASC ou DESC pour l'ordre de tri, ASC par défaut,
- lastname est une propriété de l'entité Student.java,
- Si la propriété n'est pas spécifiée -> tri par l'id

# Trier des relations : annotation @OrderBy

```
@Entity public class Student {  
    ...  
    @ManyToMany(mappedBy="students")  
    @OrderBy // tri par clé primaire (défaut)  
    List<Course> courses;  
    ...  
}
```

## ■ Remarques

- ❑ ASC ou DESC pour l'ordre de tri, ASC par défaut,
- ❑ lastname est une propriété de l'entité Student.java,
- ❑ Si la propriété n'est pas spécifiée -> tri par l'id

# Hibernate

# Définition

---

- Hibernate est un framework de mapping Objet/Relationnel pour applications JAVA (et .NET avec Nhibernate). Hibernate permet de créer une couche d'accès aux données (DAO) plus **modulaire**, plus **maintenable**, plus **performante** qu'une couche d'accès aux données 'classique' reposant sur l'API JDBC
- **La promesse d'hibernate : libérer le développeur de 95% des taches de programmation classique d'une couche d'accès aux données via JDBC.**



# Historique

---

- Né suite à la complexité EJB entity (EJB1.X et EJB2.X)
- A été écrit en juillet 2000 sous la responsabilité de Gavin King qui fait partie de l'équipe Jboss
- Inclut dans JBOSS
- Standardisé par spécifications JPA/EJB3 Entity
- De plus en plus populaire, Hibernate est devenu le standard de facto de mapping objet-relationnel du monde open source, il possède une API personnalisée qui permet une meilleure flexibilité.

# Avantages

## ■ **Hibernate génère le code SQL pour vous**

- Pas de requête SQL à écrire
- Pas d'Objet ResultSet à gérer : cycle récupération manuelle  
ResultSet + Casting de chaque ligne du resultset (type Object) vers un type d'objet métier (Ex : Employe)
- Application plus portable. S'adapte à la base de données cible

## ■ **Persistance transparente**

Possibilité de faire des classes métiers des classes persistantes sans ajout de code tiers. C'est là une différence forte et fondamentale vis à vis de EJB2.x qui étaient utilisées dans les années de création d'Hibernate.

- Cette différence n'est plus d'actualité avec les EJB3, dont les spécifications ont été écrites avec la collaboration de Gavin King et l'équipe d'Hibernate.

# Avantages

---

- **Récupération de données optimisée**

- ☐ Hibernate fournit plusieurs stratégies pour interroger la base de données. Requête SQL, langage HQL ou Api Criteria, avec des options de fetching et de mise en cache sophistiquées.

- **Portabilité du code si changement de base de données.**

- ☐ Hibernate vous permet de changer de base de données cible (ex : remplacer SQL Server par PostGresql) en modifiant un minimum de paramètres de configuration (comme le Dialect).

# Hibernate ou JPA/EJB3 Entity ?

---

- Hibernate est un produit.
- JPA est une spécification.
- EJB3 Entity est une implémentation de la spécification JPA..