

A dark blue vertical bar is positioned to the left of the title.

Java Persistence API



Plan

- ▶ Introduction JPA et Entity Bean (Rappel)
- ▶ Définition des Entity Beans : les annotations JPA
 - ▶ Définition du mapping Entity Bean – Table
 - ▶ Recherche de données avec JPQL
 - ▶ Modélisation des relations dans les Entity Beans
- ▶ JPA et OO



Plan

- ▶ Introduction JPA et Entity Bean (Rappel)
- ▶ Définition des Entity Beans : les annotations JPA
 - ▶ Définition du mapping Entity Bean – Table
 - ▶ Recherche de données avec JPQL
 - ▶ Modélisation des relations dans les Entity Beans
- ▶ JPA et OO



Le modèle de persistance JPA 2

- ▶ Java Persistence API (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.
- ▶ Annotation et « injection de code » depuis Java EE 6
 - ▶ Souvent, on ne fera pas de « new », les variables seront créées/initialisées par injection de code.



Qu'est-ce qu'un Entity Bean (Rappel)

- ▶ 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 (Rappel)

▶ Exemples

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

▶ Pourquoi 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 !



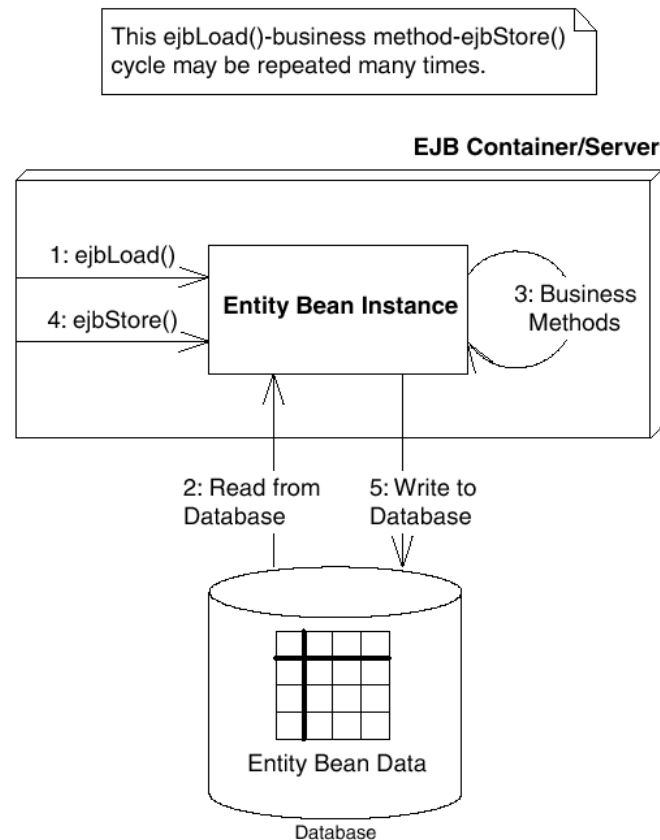
Exemple avec un compte bancaire

- ▶ 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



Caractéristiques des entity beans

- ▶ Survivent aux crashes du serveur, du SGBD
- ▶ Ce sont des vues sur des données dans un SGBD



Cycle de vie d'un entity bean

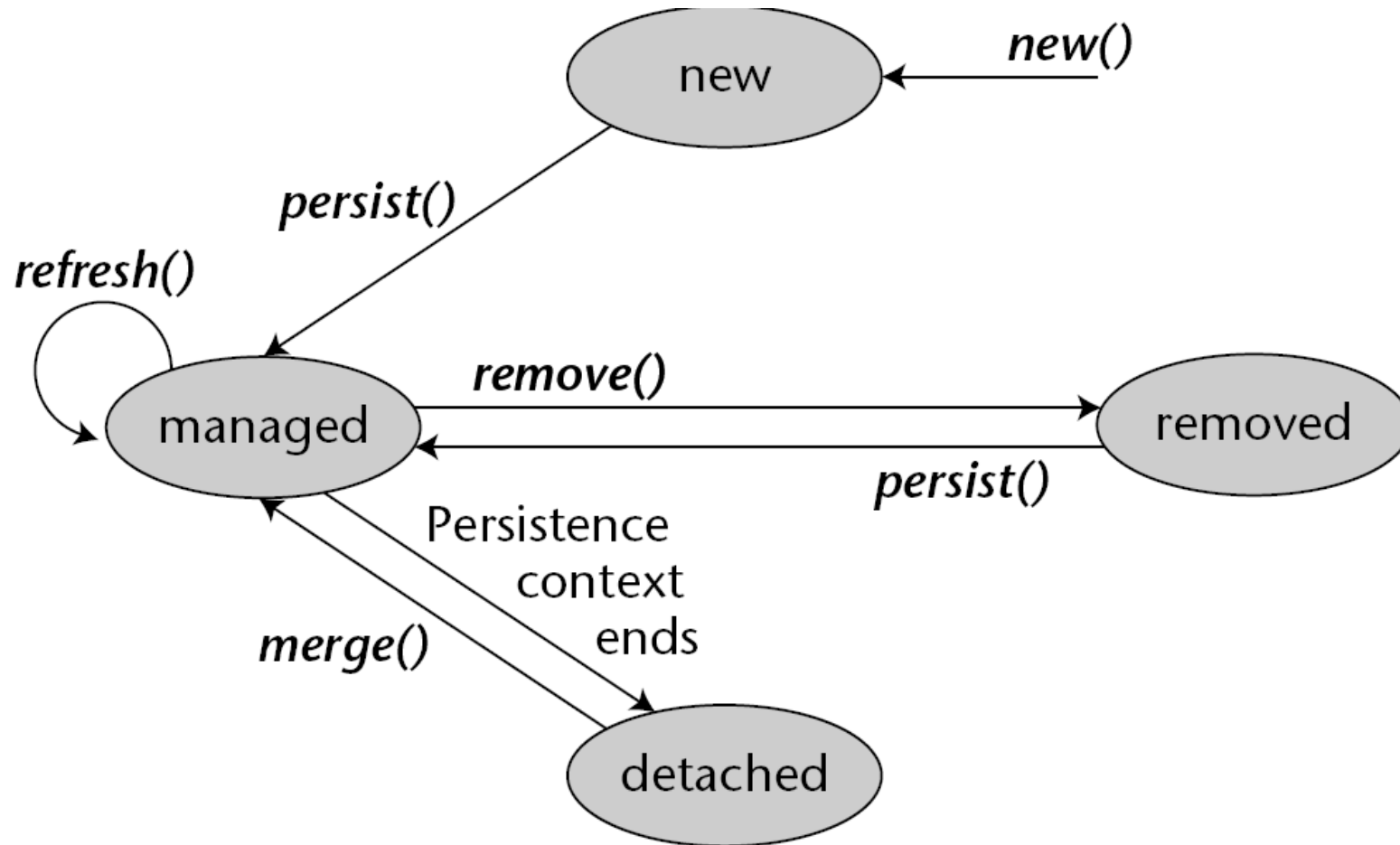


Figure 6.3 Entity life cycle.

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 à `flush()`)
3. **Detached** : le bean 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).



Plan

- ▶ Introduction JPA et Entity Bean (Rappel)
- ▶ Définition des Entity Beans : les annotations JPA
 - ▶ Définition du mapping Entity Bean – Table
 - ▶ Recherche de données avec JPQL
 - ▶ Modélisation des relations dans les Entity Beans
- ▶ JPA et OO



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 modificateurs, etc.
 - ▶ On utilisera les méta-données ou « attributs de code » ou **annotation** 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.



Annotations Java/JEE

- ▶ **Métadonnées ajoutées dans le code des classes java**
 - ▶ Faciliter les déploiement des applications
 - ▶ Ex. déjà vu @WebServlet fait éviter la déclaration dans le web.xml
 - ▶ Définir le type du composant
 - ▶ Ex. @Stateless, @Stateful
 - ▶ Définir la visibilité/portée d'une interface
 - ▶ Ex. @Remote, @Local
 - ▶ Injecter les dépendances entre composants
 - ▶ Ex. @EJB, @Resource



Annotations JPA 2

- ▶ Définition du mapping entre Entity Bean et Table SQL
 - ▶ @Table, @Column, @Id, ...
- ▶ Définition des relations entre les Entity Beans
 - ▶ @OneToOne, @OneToMany, ...
- ▶ Remarques
 - ▶ Valeurs par défaut
 - ▶ une classe entité Personne se mapperà dans la table PERSONNE par défaut
 - ▶ un attribut « nom » sur la colonne NOM, etc.
 - ▶ Attributs pour les annotations (voir les spécifications Java EE)
 - ▶ Conversion de types
 - ▶ String vers VARCHAR(255) par défaut mais peut dépendre des SGBD



Exemple d'Entity Bean avec annotation JPA : 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
```

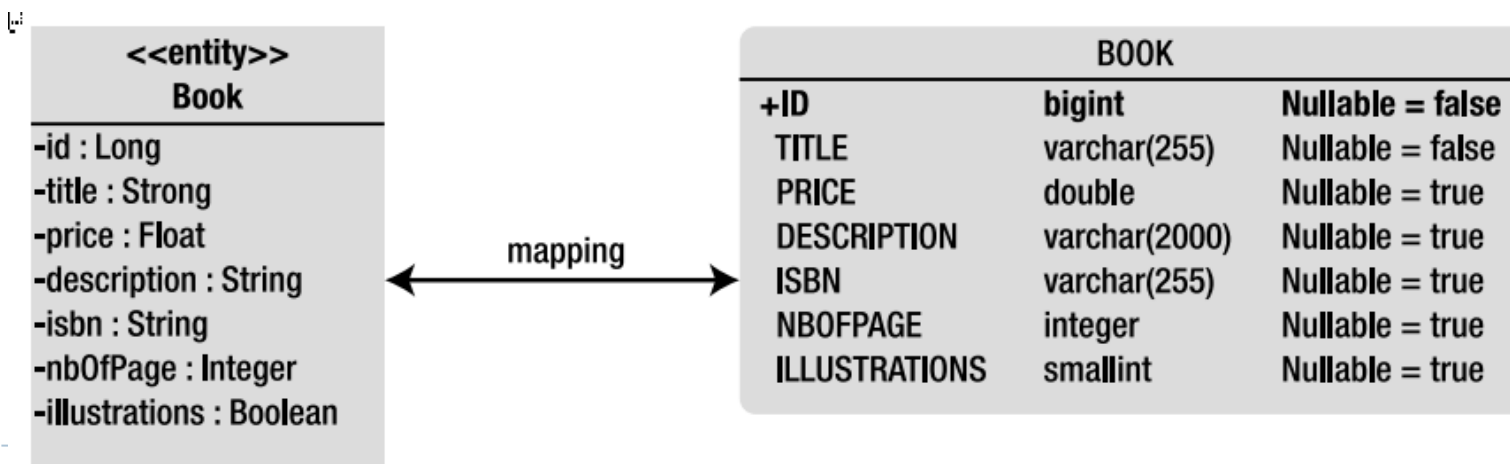
```
}
```



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;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations
}
```



Java SE 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();  
    }  
}
```



Java EE Session Bean

- ▶ Dans le cas où le client est un « session bean »
 - ▶ du code peut être « injecté »,
 - ▶ Les transactions sont déclenchées par défaut,

```
@Stateless
public class UnSessionBean {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public Employee createEmployee(String titre,...) {
        Book book = new Book();
        book.setTitle(...);
        ...

        em.persist(book);

        return book;
    }
}
```



Remarque : à quoi correspond le session bean ?

- ▶ Coder 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>



Utiliser des colonnes composites

@Embeddable

```
public class Address {  
    protected String street;  
    protected String city;  
    protected String state;  
    @Embedded  
    Zipcode zipcode;  
}
```

@Embeddable

```
public class Zipcode {  
    String zip;  
    protected String plusFour;  
}
```



Utiliser une clé primaire composite

- ▶ Similaire à l'exemple précédent sauf que au lieu d'utiliser `@Embedded` / `@Embeddable` on utilisera `@EmbeddedId` / `Embeddable`

```
@Embeddable  
public class CompositId {  
    String name;  
    String email  
}
```

```
@Entity  
public class Dependent {  
    @EmbeddedId // indique que la clé primaire est dans une autre classe  
    CompositId id;  
    @ManyToOne  
    Employee emp;  
}
```



Packager et déployer un Entity Bean

- ▶ Les EB sont déployés dans des « persistence Units »,
 - ▶ Spécifiées dans le fichier « persistence.xml »
 - ▶ Exemple (simplifié):

```
<?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.



Exemple de fichier persistence.xml

- Ce fichier « configure » le persistence manager

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/
  persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
  persistence http://java.sun.com/xml/ns/persistence/
  persistence_1_0.xsd">
  <persistence-unit name="IGift-ejbPU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/igift</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```



Utilisation de l'entity 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 ».



Cycle de vie d'un entity bean (rappel)

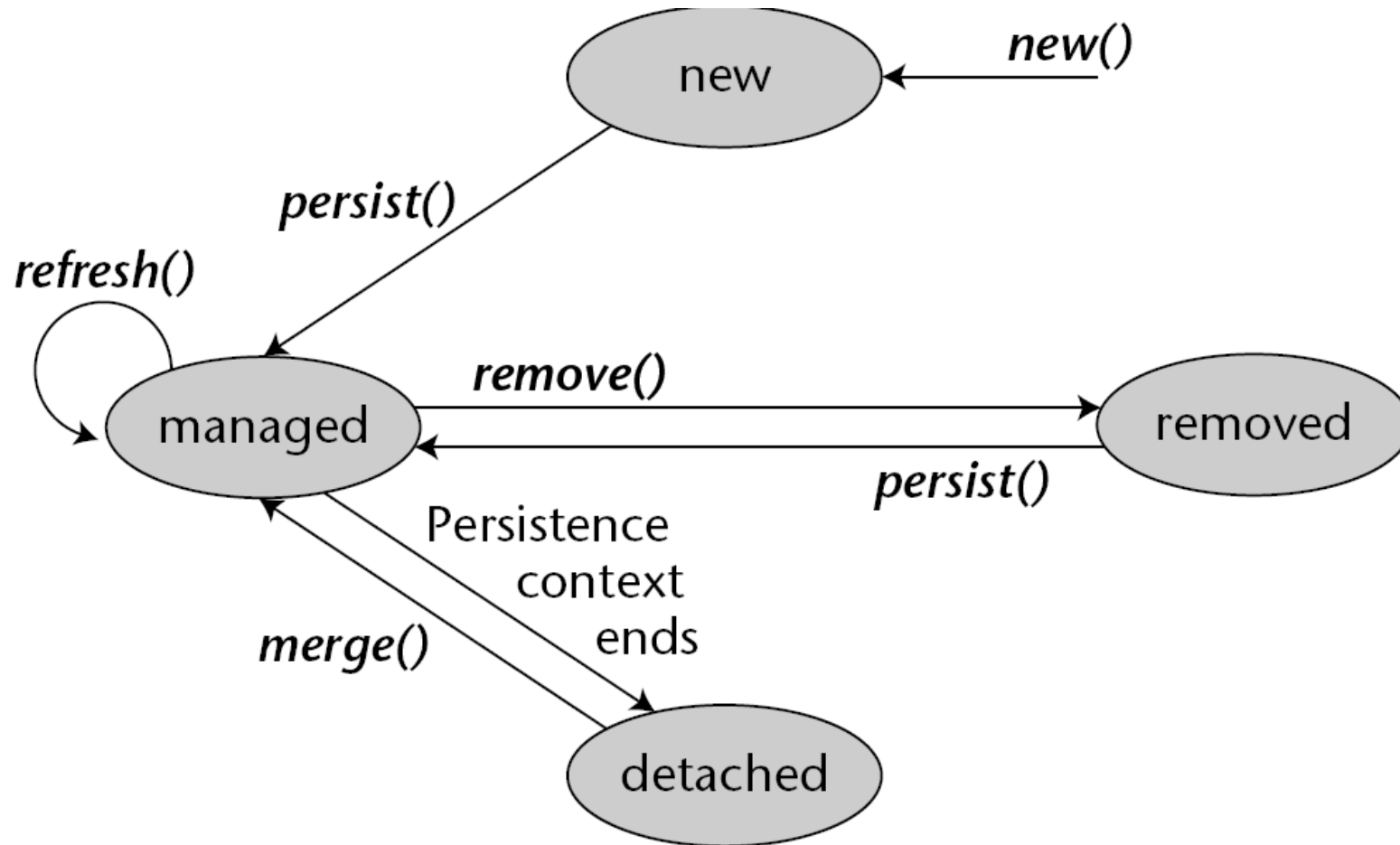


Figure 6.3 Entity life cycle.



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);  
}
```



Plan

- ▶ Introduction JPA et Entity Bean (Rappel)
- ▶ Définition des Entity Beans : les annotations JPA
 - ▶ Définition du mapping Entity Bean – Table
 - ▶ Recherche de données avec JPQL
 - ▶ Modélisation des relations dans les Entity Beans
- ▶ JPA et OO



Recherche d'entity beans

- ▶ Les instances d'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);
```

```
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 {...}
```



JPQL : Quelques exemples

```
// customers 20-30 named 'Joe', ordered by last name  
Query q = em.createQuery("select c from Customer c where  
    c.firstName = :fname order by c.lastName");  
q.setParameter("fname", "Joe");  
q.setFirstResult(20);  
q.setMaxResults(10);  
List<Customer> customers = (List<Customer>) q.getResultList();
```



JPQL : Quelques exemples (suite)

// all orders, as a named query

`@Entity`

`@NamedQuery(name="Order:findAllOrders", query="select o from Order o");`

`public class Order { ... }`

`Query q = em.createNamedQuery("Order:findAllOrders");`



JPQL : Quelques exemples (suite)

// all people, via a custom SQL statement

```
Query q = em.createNativeQuery("SELECT ID, VERSION, SUBCLASS,  
    FIRSTNAME, LASTNAME FROM PERSON", Person.class);  
List<Person> people = (List<Person>) q.getResultList();
```

// single-result aggregate: average order total price

```
Query q = em.createQuery("select avg(i.price) from Item i");  
Number avgPrice = (Number) q.getSingleResult();
```



JPQL : Quelques exemples (suite)

- ▶ Liste toutes les commandes qui ne comprennent pas (LEFT) de produit dont le prix est supérieur à une certaine quantité (et celles qui ne comprennent pas de produits)

```
// traverse to-many relations
```

```
Query q = em.createQuery("select o from Order o  
    left join o.items li where li.price > :price");  
q.setParameter("price", 1000);  
List<Order> orders = (List<Order>) q.getResultList();
```



JPQL : Quelques exemples (suite)

- Requête **sur plusieurs attributs** renvoie soit un tableau d'Object, soit une collection de tableaux d'Object

```
String req = "select e.nom, e.salaire from Employe as e";
query = em.createQuery(req);
List<Object[]> liste = (List<Object[]>)query.getResultList();
for (Object[] info : liste) {
    System.out.println(info[0] + "gagne » + info[1]);
}
```



► Table des compagnies

ID	NAME
1	M*Power Internet Service, Inc.
2	Sun Microsystems
3	Bob's Bait and Tackle

Table D.1 Company

► Table des employés

ID	NAME	COMPANY_ID
1	Micah Silverman	1
2	Tes Silverman	1
3	Rima Patel	2



JPQL : Quelques exemples (suite)

- ▶ Cette requête récupère trois compagnies :

```
SELECT DISTINCT c FROM CompanyOMBid c
```

- ▶ Mais celle-ci uniquement deux :

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN c.employees
```

- ▶ Celle-là : les trois (même si join condition absente)

```
SELECT DISTINCT c FROM CompanyOMBid c LEFT JOIN c.employees
```



JPQL : Quelques exemples (suite)

- ▶ Provoque le chargement des entités reliées

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN FETCH c.employees
```

- ▶ Prend le devant sur @FetchType.LAZY
- ▶ Autre exemple :

```
SELECT DISTINCT c  
FROM CompanyOMBid c, IN(c.employees) e  
WHERE e.name='Micah Silverman'
```



JPQL : Quelques exemples (suite)

► WHERE et requêtes paramétrées

```
Query q =  
    em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = ?1").  
        setParameter(1, "M*Power Internet Services, Inc.");
```

► Autre exemple avec paramètres nommés

```
Query q =  
    em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = :cname").  
        setParameter("cname", "M*Power Internet Services, Inc.");
```



JPQL : Quelques exemples (suite)

► Expressions

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers between 4 AND 5
```

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers IN(2,5)
```

```
SELECT r FROM RoadVehicleSingle r WHERE r.make LIKE 'M%'
```

► Le % dans le LIKE = suite de caractères, le _ = un caractère

```
SELECT r FROM RoadVehicleSingle r WHERE r.model IS NOT NULL
```

```
SELECT c FROM CompanyOMBid c WHERE c.employees IS NOT EMPTY
```



JPQL : Quelques exemples (suite)

► MEMBER OF

```
"SELECT e FROM EmployeeOMBid e, CompanyOMBid c  
WHERE e MEMBER OF c.employees"
```

► Sous-Requêtes

```
SELECT c FROM CompanyOMBid c WHERE  
(SELECT COUNT(e) FROM c.employees e) = 0
```



Fonctions sur chaînes, arithmétique

```
functions returning strings ::=
    CONCAT(string primary, string primary) |
    SUBSTRING(string_primary,
               simple arithmetic expression,
               simple arithmetic expression) |
    TRIM([[trim_specification] [trim_character] FROM]
          string primary) |
    LOWER(string primary) |
    UPPER(string_primary)
trim specification ::= LEADING | TRAILING | BOTH

functions_returning_numerics ::=
    LENGTH(string primary) |
    LOCATE(string_primary, string_primary[,
           simple_arithmetic_expression])
```

Fonctions sur chaînes, arithmétique (suite)

```
functions returning numerics ::=  
  ABS(simple_arithmetic_expression) |  
  SQRT(simple_arithmetic_expression) |  
  MOD(simple_arithmetic_expression, simple_arithmetic_expression) |  
  SIZE(collection_valued_path_expression)
```



JPQL : Quelques exemples (suite)

// bulk update: give everyone a 10% raise

```
Query q = em.createQuery("update Employee emp  
    set emp.salary = emp.salary * 1.10");  
int updateCount = q.executeUpdate();
```

// bulk delete: get rid of fulfilled orders

```
Query q = em.createQuery("delete from Order o  
    where o.fulfilledDate is not null");  
int deleteCount = q.executeUpdate();
```



JPQL : Quelques exemples (suite)

// subselects: all orders with an expensive line item

```
Query q = em.createQuery("select o from Order o where exists  
    (select li from o.items li where li.price > 10000)");
```



Plan

- ▶ Introduction JPA et Entity Bean (Rappel)
- ▶ Définition des Entity Beans : les annotations JPA
 - ▶ Définition du mapping Entity Bean – Table
 - ▶ Recherche de données avec JPQL
 - ▶ **Modélisation des relations dans les Entity Beans**
- ▶ JPA et OO



Besoin des relations

- ▶ 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 (*directionality*)

- ▶ **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



Cardinalité

- ▶ 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

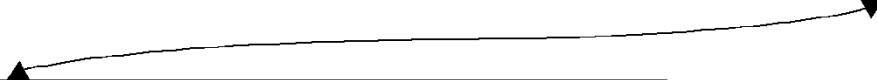


Relations 1:1

- ▶ 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 1:1, le bean Order

```
@Entity(name="OrderUni")
public class Order implements Serializable {
    private int id;
    private String orderName;
private Shipment shipment;

    public Order() {
        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```



Relations 1:1, le bean Order

```
...
// other setters and getters go here
...

@OneToOne(cascade={CascadeType.PERSIST})
public Shipment getShipment() {
    return shipment;
}

public void setShipment(Shipment shipment) {
    this.shipment = shipment;
}
}
```



Relations 1:1, le bean Shipment

```
...
@Entity(name="ShipmentUni")
public class Shipment implements Serializable {
    private int id;
    private String city;
    private String zipcode;

    public Shipment() {

        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    ...
    // other setters and getters go here
    ...
}
```

Exemple de code pour insérer une commande avec une livraison reliée

```
...
@Stateless
public class OrderShipmentUniBean implements OrderShipment {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {

        Shipment s = new Shipment();
        s.setCity("Austin");
        s.setZipcode("78727");

        Order o = new Order();
        o.setOrderName("Software Order");
        o.setShipment(s);

        em.persist(o);
    }

    public List getOrders() {
        Query q = em.createQuery("SELECT o FROM OrderUni o");
        return q.getResultList();
    }
}
```

Relations 1:1, exemple de client (ici un main...)

```
...
InitialContext ic = new InitialContext();
OrderShipment os =
    (OrderShipment)ic.lookup(OrderShipment.class.getName());

os.doSomeStuff();

System.out.println("Unidirectional One-To-One client\n");

for (Object o : os.getOrders()) {
    Order order = (Order)o;
    System.out.println("Order "+order.getId()+" : "+
        order.getOrderName());
    System.out.println("\tShipment details: "+
        order.getShipment().getCity()+" "+
        order.getShipment().getZipcode());
}
...
```



Version bidirectionnelle (on modifie Shipment)

```
...
@Entity(name="ShipmentBid")
public class Shipment implements Serializable {
    private int id;
    private String city;
    private String zipcode;
    private Order order;

    public Shipment() {
        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```



Version bidirectionnelle (suite)

```
...
// other setters and getters go here
...

@OneToOne(mappedBy="shipment")
public Order getOrder() {
    return order;
}

public void setOrder(Order order) {
    this.order = order;
}
}
```



Version bi-directionnelle (suite, code qui fait le persist)

- ▶ On peut maintenant ajouter au code de tout à l'heure (celui qui écrit une commande) :

```
...  
public List getShipments() {  
    Query q = em.createQuery("SELECT s FROM ShipmentBid s");  
    return q.getResultList();  
}  
...
```



Version bi-directionnelle (suite, code du client)

```
...
for (Object o : os.getShipments()) {
    Shipment shipment = (Shipment)o;
    System.out.println("Shipment: "+
        shipment.getCity()+" "+
        shipment.getZipcode());
    System.out.println("\tOrder details: "+
        shipment.getOrder().getOrderName());
}
...
```



Relations 1:N

- ▶ Exemple : une entreprise a plusieurs employés

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

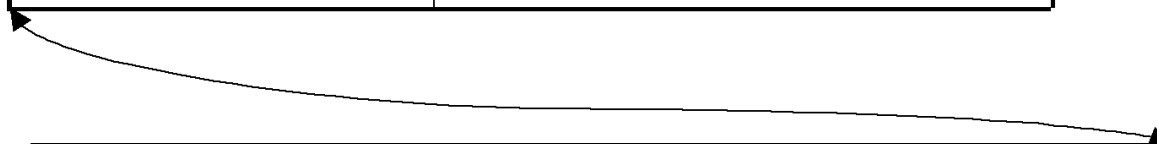
EmployeePK	Name	Sex
20202	Ed	M
20203	Floyd	M

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 1:N exemple

```
...
@Entity(name="CompanyOMUni")
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;

    ...
    // other getters and setters go here
    // including the Id
    ...

    @OneToMany(cascade={CascadeType.ALL}, fetch=FetchType.EAGER)
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
}
```



Relations 1:N exemple

```
...
@Entity(name="EmployeeOMUni")
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;

    ...
    // other getters and setters go here
    // including the Id
    ...
}
```



Exemple de code qui insère des compagnies

```
...
@Stateless
public class CompanyEmployeeOMUniBean implements CompanyEmployeeOM {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {
        Company c = new Company();
        c.setName("M*Power Internet Services, Inc.");

        Collection<Employee> employees = new ArrayList<Employee>();
        Employee e = new Employee();
        e.setName("Micah Silverman");
        e.setSex('M');
        employees.add(e);

        e = new Employee();
        e.setName("Tes Silverman");
        e.setSex('F');
        employees.add(e);

        c.setEmployees(employees);
        em.persist(c);
    }
}
```

Exemple de code qui liste des compagnies

```
public List getCompanies() {  
    Query q = em.createQuery("SELECT c FROM CompanyOMUni c");  
    return q.getResultList();  
}
```



Exemple de client

```
...
    InitialContext ic = new InitialContext();
    CompanyEmployeeOM ceom = (CompanyEmployeeOM)ic.lookup(
        CompanyEmployeeOM.class.getName());

    ceom.doSomeStuff();

    for (Object o : ceom.getCompanies()) {
        Company c = (Company)o;
        System.out.println("Here are the employees for company: "+
            c.getName());
        for (Employee e : c.getEmployees()) {
            System.out.println("\tName: "+
                e.getName()+"", Sex: "+e.getSex());
        }
        System.out.println();
    }
...
```



Version bidirectionnelle

```
...
@Entity(name="CompanyOMBid")
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;

    ...

    @OneToMany(cascade={CascadeType.ALL},
        fetch=FetchType.EAGER,
        mappedBy="company")
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
    ...
}
```



Version bidirectionnelle

```
...
@Entity(name="EmployeeOMBid")
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;
    private Company company;

    ...

    @ManyToOne
    public Company getCompany() {
        return company;
    }

    public void setCompany(Company company) {
        this.company = company;
    }
}
...
```



```

...
@Stateless
public class CompanyEmployeeOMBidBean implements CompanyEmployeeOM {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {
        Company c = new Company();
        c.setName("M*Power Internet Services, Inc.");

        Collection<Employee> employees = new ArrayList<Employee>();
        Employee e = new Employee();
        e.setName("Micah Silverman");
        e.setSex('M');
e.setCompany(c);

        employees.add(e);

        e = new Employee();
        e.setName("Tes Silverman");
        e.setSex('F');
e.setCompany(c);
        employees.add(e);

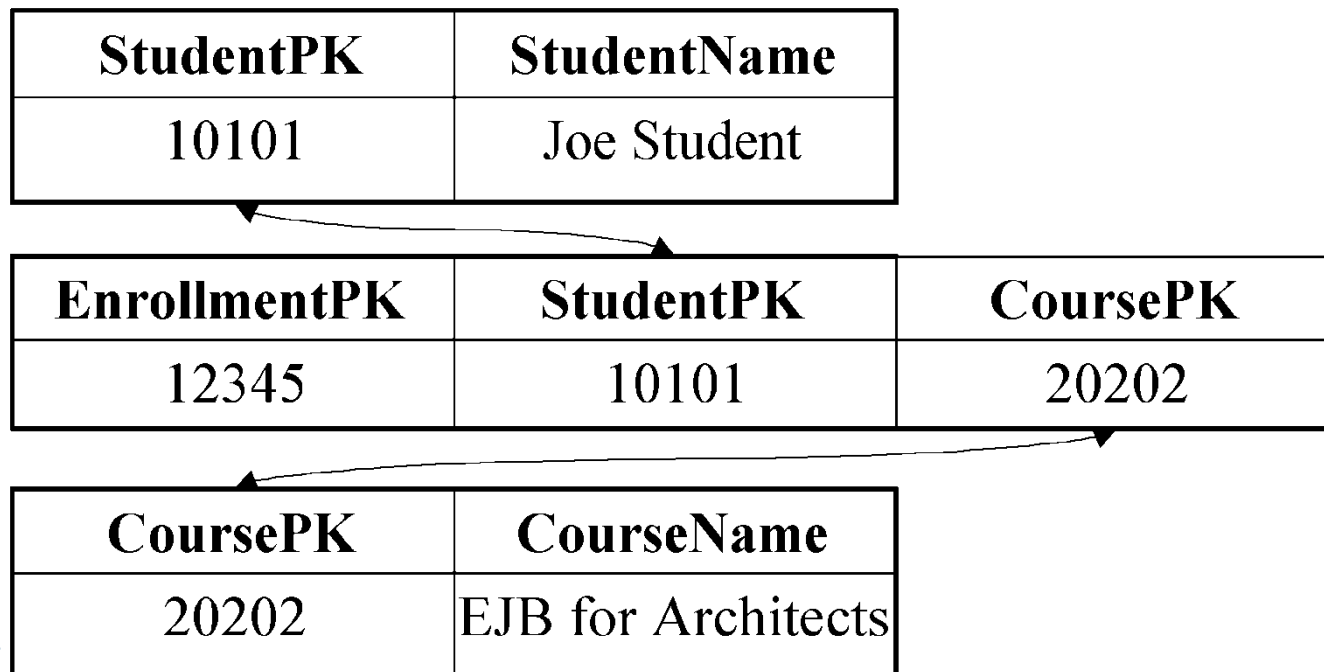
        c.setEmployees(employees);
        em.persist(c);

        ...
        // the other Company and Employee code
        // comes after this
        ...
    }
    ...
}

```

Relations M:N

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



Relations M:N, choix de conception

- ▶ Deux possibilités lorsqu'on modélise cette relation avec des EJBs
 1. Utiliser un troisième EJB pour modéliser la table de jointure. On veut peut-être mémoriser la date où un étudiant s'est inscrit, etc... Cet EJB possèdera deux relations 1:N vers le bean Student et le vers le bean Course
 2. Si on n'a rien besoin de plus à part la relation, on peut utiliser simplement deux EJB, chacun ayant un attribut correspondant à une Collection de l'autre EJB...



Relations M:N, exemple

```
...
@Entity(name="StudentUni")
public class Student implements Serializable {
    private int id;
    private String name;
    private Collection<Course> courses = new ArrayList<Course>();

    public Student() {
        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

    ...
    //other setters and getters go here
    ...

    @ManyToMany(cascade={CascadeType.ALL}, fetch=FetchType.EAGER)
    @JoinTable(name="STUDENTUNI_COURSEUNI")
    public Collection<Course> getCourses() {
        return courses;
    }

    public void setCourses(Collection<Course> courses) {
        this.courses = courses;
    }
}
```

Relations M:N, exemple

```
...
@Entity(name="CourseUni")
public class Course implements Serializable {
    private int id;
    private String courseName;
    private Collection<Student> students = new ArrayList<Student>();

    ...
    //setters and getters go here
    ...
}
```



```

...
@Stateless
public class StudentCourseUniBean implements StudentCourse {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {
        Course c1 = new Course();
        c1.setCourseName("EJB 3.0 101");

        Course c2 = new Course();
        c2.setCourseName("EJB 3.0 202");

        Student s1 = new Student();
        s1.setName("Micah");

        s1.getCourses().add(c1);

        c1.getStudents().add(s1);

        Student s2 = new Student();
        s2.setName("Tes");

        s2.getCourses().add(c1);
        s2.getCourses().add(c2);

        c1.getStudents().add(s2);
        c2.getStudents().add(s2);

        em.persist(s1);
        em.persist(s2);
    }

    public List<Student> getAllStudents() {
        Query q = em.createQuery("SELECT s FROM StudentUni s");
        return q.getResultList();
    }
}

```

```
...
    InitialContext ic = new InitialContext();
    StudentCourse sc = (StudentCourse)ic.lookup(
        StudentCourse.class.getName());

    sc.doSomeStuff();

    for (Student s : sc.getAllStudents()) {
        System.out.println("Student: "+s.getName());
        for (Course c : s.getCourses()) {
            System.out.println("\tCourse: "+c.getCourseName());
        }
    }
...
```



La directionnalité et le modèle de données dans la DB

- ▶ Qu'on soit en présence d'un modèle normalisé ou pas, les outils d'ORM s'adaptent.

Schéma normalisé

PersonPK	PersonName	Address ForeignPK
12345	Ed Roman	10101

AddressPK	City	ZipCode
10101	Austin	78727

Schéma dénormalisé

PersonPK	PersonName	Address ForeignPK
12345	Ed Roman	10101

AddressPK	City	ZipCode	Person ForeignPK
10101	Austin	78727	12345

Choisir la directionnalité ?

- ▶ Premier critère : la logique de votre application,
- ▶ Second critère : si le schéma relationnel existe, s'adapter au mieux pour éviter de mauvaises performances.



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 en CMP, 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
```

Renvoie tous
les clients qui
ont placé une
commande

- ▶ Principale différence avec SQL, l'opérateur "."
 - ▶ Pas besoin de connaître le nom des tables, ni le nom des colonnes...



Relations et EJB-QL

- ▶ On peut aller plus loin...

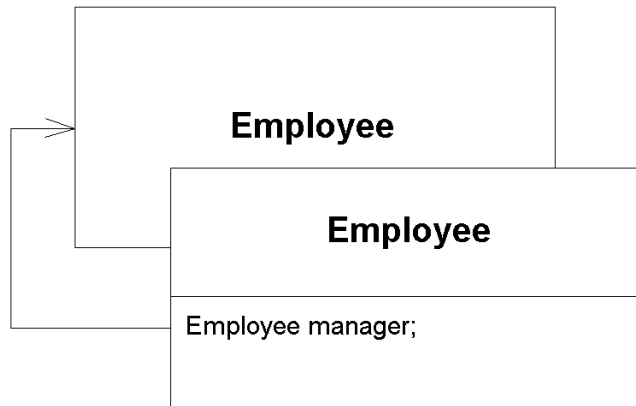
```
SELECT o.customer.address.homePhoneNumber  
FROM Order o
```

- ▶ On se promène le long des relations...



Relations récursives

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

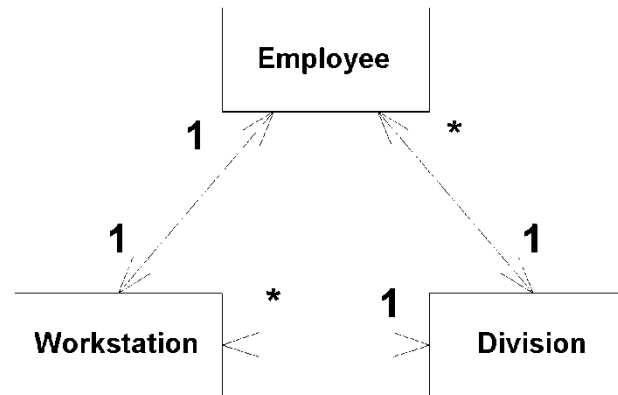


- ▶ Rien de particulier, ces relations sont implémentées exactement comme les relations non récursives...



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.



Intégrité référentielle

- ▶ Un bean Compagnie, Division, etc... a des relations avec un bean Employé
 - ▶ Si on supprime un employé, il faut vérifier qu'il est bien supprimé partout où on a une relation avec lui.
- ▶ Problème classique dans les SGBDs
 - ▶ Résolus à l'aide de *triggers*. Ils se déclenchent sitôt qu'une perte d'intégrité risque d'arriver et effectuent les opérations nécessaires.
 - ▶ On peut aussi utiliser des procédures stockées via JDBC. Ces procédures effectuent la vérification d'intégrité.



Intégrité référentielle

- ▶ Gérer l'intégrité dans le SGBD est intéressant si la BD est attaquée par d'autres applications que les EJBs...
- ▶ **Autre approche : gérer l'intégrité dans les EJBs**
 - ▶ Solution plus propre,
 - ▶ Le SGBD n'est plus aussi sollicité,
 - ▶ Avec les EJB: le travail est fait tout seul !



Intégrité référentielle

- ▶ Et dans un contexte distribué ?
- ▶ Plusieurs serveurs d'application avec le même composant peuvent accéder à des données sur le même SGBD,
- ▶ Comment mettre à jour les relations ?
- ▶ **Problème résolu par les transactions !!!**



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



Trier des relations : annotation `@OrderBy`

- ▶ On peut utiliser l'opérateur « . » si on trie sur une colonne qui est définie dans une autre classe par `@Embedded`

```
@Entity public class Person {  
    ...  
    @ElementCollection  
    @OrderBy("zipcode.zip,  
                zipcode.plusFour")  
    Set<Address> residences;  
    ...  
}
```



Trier des relations : annotation `@OrderBy`

`@Embeddable`

```
public class Address {  
    protected String street;  
    protected String city;  
    protected String state;  
    @Embedded  
    Zipcode zipcode;  
}
```

`@Embeddable`

```
public class Zipcode {  
    String zip;  
    protected String plusFour;  
}
```



Plan

- ▶ Introduction JPA et Entity Bean (Rappel)
- ▶ Définition des Entity Beans : les annotations JPA
 - ▶ Définition du mapping Entity Bean – Table
 - ▶ Recherche de données avec JPQL
 - ▶ Modélisation des relations dans les Entity Beans
- ▶ JPA et OO
 - ▶ Héritage et polymorphisme

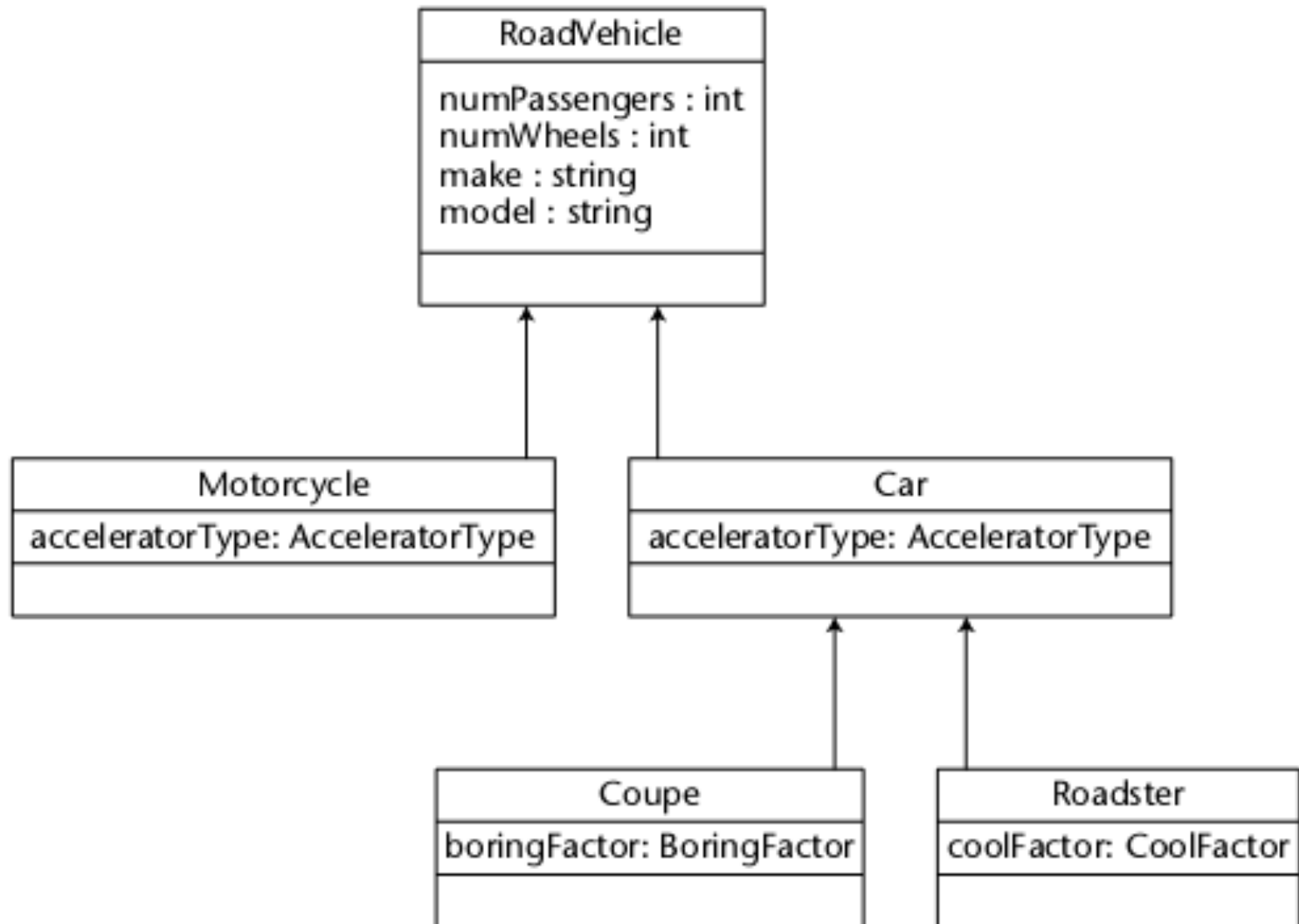


Héritage

- ▶ Stratégies de mapping entre classes et tables quand on a de l'héritage ?
 - ▶ Une table pour toute la hiérarchie de classes ?
 - ▶ Une table par classe/sous-classe ?
 - ▶ Autres solutions ?



Un exemple !



Code de RoadVehicle.java (classe racine)

```
package examples.entity.single_table;

public class RoadVehicle {
    public enum AcceleratorType {PEDAL, THROTTLE};

    protected int numPassengers;
    protected int numWheels;
    protected String make;
    protected String model;

    // setters and getters go here
    ...

    public String toString() {
        return "Make: "+make+
            ", Model: "+model+
            ", Number of passengers: "+numPassengers;
    }
}
```



Code de Motorcycle.java

```
package examples.entity.single_table;

public class Motorcycle extends RoadVehicle {
    public final AcceleratorType acceleratorType =
        AcceleratorType.THROTTLE;

    public Motorcycle() {
        numWheels = 2;
        numPassengers = 2;
    }

    public String toString() {
        return "Motorcycle: "+super.toString();
    }
}
```



Code de Car.java

```
package examples.entity.single_table;

public class Car extends RoadVehicle {
    public final AcceleratorType acceleratorType =
        AcceleratorType.PEDAL;

    public Car() {
        numWheels = 4;
    }

    public String toString() {
        return "Car: "+super.toString();
    }
}
```



Code de Roadster.java

```
package examples.entity.single_table;

public class Roadster extends Car {
    public enum CoolFactor {COOL,COOLER,COOLEST};

    private CoolFactor coolFactor;

    public Roadster() {
        numPassengers = 2;

    }

    // setters and getters go here
    ...

    public String toString() {
        return "Roadster: "+super.toString();
    }
}
```


Code de Coupe.java

```
package examples.entity.single_table;

public class Coupe extends Car {
    public enum BoringFactor {BORING,BORINGER,BORINGEST};

    private BoringFactor boringFactor;

    public Coupe() {
        numPassengers = 5;
    }

    // setters and getters go here
    ...

    public String toString() {
        return "Coupe: "+super.toString();
    }
}
```



Premier cas : une seule table !

- ▶ Une seule table représente toute la hiérarchie.
- ▶ Une colonne de « discrimination » est utilisée pour distinguer les sous-classes.
- ▶ **Avantage :**
 - ▶ Cette solution supporte le polymorphisme.
- ▶ **Désavantages :**
 - ▶ Une colonne pour chaque champ de chaque classe,
 - ▶ Comme une ligne peut être une instance de chaque classe, des champs risquent de ne servir à rien (nullable)



Code avec les annotations !

```
package examples.entity.single_table;

// imports go here

@Entity(name="RoadVehicleSingle")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISC",
    discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("ROADVEHICLE")
public class RoadVehicle implements Serializable {
    public enum AcceleratorType {PEDAL, THROTTLE};
}
```



(suite)

```
@Id  
protected int id;  
protected int numPassengers;  
protected int numWheels;  
protected String make;  
protected String model;  
  
public RoadVehicle() {  
    id = (int) System.nanoTime();  
}  
  
// setters and getters go here  
...  
}
```



Motorcycle.java annotée !

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("MOTORCYCLE")
public class Motorcycle extends RoadVehicle implements Serializable {

    AcceleratorType.THROTTLE;

    public Motorcycle() {
        super();
        numWheels = 2;
        numPassengers = 2;
    }
}
```



Car.java annotée

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("CAR")
public class Car extends RoadVehicle implements Serializable {
    public final AcceleratorType acceleratorType =
        AcceleratorType.PEDAL;

    public Car() {
        super();
        numWheels = 4;
    }
}
```



Roadster.java annotée

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("ROADSTER")
public class Roadster extends Car implements Serializable {
    public enum CoolFactor {COOL, COOLER, COOLEST};

    private CoolFactor coolFactor;

    public Roadster() {
        super();
        numPassengers = 2;

    }

    // setters and getters go here
    ...
}
```

Coupe.java annotée

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("COUPE")
public class Coupe extends Car implements Serializable {
    public enum BoringFactor {BORING,BORINGER,BORINGEST};

    private BoringFactor boringFactor;

    public Coupe() {
        super();
        numPassengers = 5;
    }

    // setters and getters go here
    ...
}
```



Table corrispondante

```
CREATE TABLE ROADVEHICLE (  
    ID INTEGER NOT NULL,  
    DISC VARCHAR(31),  
    NUMWHEELS INTEGER,  
    MAKE VARCHAR(255),  
    NUMPASSENGERS INTEGER,  
    MODEL VARCHAR(255),  
    ACCELERATORTYPE INTEGER,  
    COOLFACTOR INTEGER,  
    BORINGFACTOR INTEGER  
);
```



Quelques objets persistants !

```
...
    @PersistenceContext
    EntityManager em;
...

    Coupe c = new Coupe();
    c.setMake("Bob");
    c.setModel("E400");
    c.setBoringFactor(BoringFactor.BORING);
    em.persist(c);

    Roadster r = new Roadster();
    r.setMake("Mini");
    r.setModel("Cooper S");
    r.setCoolFactor(CoolFactor.COOLEST);
    em.persist(r);

    Motorcycle m = new Motorcycle();
    em.persist(m);
...
```



Et les données correspondantes

ID	DISC	MAKE	MODEL	COOL FACTOR	BORINGFACTOR
1818876882	COUPE	Bob	E400	NULL	0
1673414469	MOTORCYCLE	NULL	NULL	2	NULL
1673657791	ROADSTER	Mini	Cooper S	NULL	NULL



Deuxième stratégie : une table par classe

- ▶ Il suffit de modifier quelques annotations !

- ▶ Dans RoadVehicle.java

```
@Entity(name="RoadVehicleJoined")
@Table(name="ROADVEHICLEJOINED")
@Inheritance(strategy=InheritanceType.JOINED)
public class RoadVehicle {
    ...
}
```

- ▶ On peut retirer les @Discriminator des sous-classes (on aura des valeurs par défaut)
- ▶ Le champ Id de la classe RoadVehicle sera une clé étrangère dans les tables des sous-classes,
- ▶ Remarque : on utilise ici @TABLE pour ne pas que la table porte le même nom que dans l'exemple précédent (facultatif)



Les tables !

Table 9.2 ROADVEHICLEJOINED Table

ID	DTYPE	NUMWHEELS	MAKE	MODEL
1423656697	Coupe	4	Bob	E400
1425368051	Motorcycle	2	NULL	NULL
1424968207	Roadster	4	Mini	Cooper S

Table 9.3 MOTORCYCLE Table

ID	ACCELERATORTYPE
1425368051	1



Les tables (suite)

Table 9.4 CAR Table

ID	ACCELERATORTYPE
1423656697	0
1424968207	0

Table 9.5 COUPE Table

ID	BORINGFACTOR
1423656697	0

Table 9.6 ROADSTER Table

ID	COOLFACTOR
1423656697	2



Requête SQL pour avoir tous les Roadsters

- ▶ Il faut faire des joins !
- ▶ Plus la hierarchie est profonde, plus il y aura de jointures : problèmes de performance !

```
SELECT
    rvj.NumWheels, rvj.Make, rvj.Model,
    c.AcceleratorType, r.CoolFactor
FROM
    ROADVEHICLEJOINED rvj, CAR c, ROADSTER r
WHERE
    rvj.Id = c.Id and c.Id = r.Id;
```



Conclusion sur cette approche

- ▶ Supporte le polymorphisme,
- ▶ On alloue juste ce qu'il faut sur disque,
- ▶ Excellente approche si on a pas une hiérarchie trop profonde,
- ▶ A éviter sinon...



Autres approches

- ▶ Des classes qui sont des entity bean peuvent hériter de classes qui n'en sont pas,
- ▶ Des classes qui ne sont pas des entity beans peuvent hériter de classes qui en sont,
- ▶ Des classes abstraites peuvent être des entity beans,
- ▶ (déjà vu : une classe qui est un entity bean hérite d'une autre classe qui est un entity bean)



Cas 1 : Entity Bean étends classe java

- ▶ On utilise l'attribut `@MappedSuperclass` dans la classe mère
 - ▶ Indique qu'aucune table ne lui sera associée

```
...
@MappedSuperclass
public class RoadVehicle {
    public enum AcceleratorType {PEDAL, THROTTLE};

    @Id
    protected int id;
    protected int numPassengers;
    protected int numWheels;

    protected String make;
    protected String model;
    ...
}
...
```

Cas 1 (les sous-classes entities)

```
@Entity
public class Motorcycle extends RoadVehicle {
    public final AcceleratorType ac = AcceleratorType.THROTTLE ;
    ...
}
...

@Entity
public class Car extends RoadVehicle {
    public final AcceleratorType ac = AcceleratorType.PEDAL;
    ...
}
...
```



Cas 1 : les tables

Table 9.7 Database Table Layout Mapped for Roadster.java

ID	NUMPASSENGERS	NUMWHEELS	MAKE	MODEL	ACCELERATORTYPE	COOLFACTOR
----	---------------	-----------	------	-------	-----------------	------------

Table 9.8 Database Table Layout Mapped for Coupe.java

ID	NUMPASSENGERS	NUMWHEELS	MAKE	MODEL	ACCELERATORTYPE	BORINGFACTOR
----	---------------	-----------	------	-------	-----------------	--------------

Table 9.9 Database Table Layout Mapped for Motorcycle.java

ID	NUMPASSENGERS	NUMWHEELS	MAKE	MODEL	ACCELERATORTYPE
----	---------------	-----------	------	-------	-----------------



Remarques sur le cas 1

- ▶ RoadVehicle n'aura jamais sa propre table,
- ▶ Les sous-classes auront leur propre table, avec comme colonnes les attributs de RoadVehicle en plus des leurs,
- ▶ Si on n'avait pas mis `@MappedSuperclass` dans `RoadVehicle.java`, les attributs hérités n'auraient pas été des colonnes dans les tables des sous-classes.



Classe abstraite et entity bean

- ▶ Une classe abstraite peut être un entity bean (avec @Entity)
- ▶ Elle ne peut pas être instanciée, ses sous-classes concrètes oui,
- ▶ Elle aura une table dédiée,
- ▶ Elle pourra faire l'objet de requêtes (polymorphisme) : très intéressant !



Polymorphisme ! Exemple avec un SessionBean

```
...
@Stateless
public class RoadVehicleStatelessBean implements RoadVehicleStateless {
    @PersistenceContext(unitName="pu1")
    EntityManager em;

    public void doSomeStuff() {
        Coupe c = new Coupe();
        c.setMake("Bob");
        c.setModel("E400");
        c.setBoringFactor(BoringFactor.BORING);
        em.persist(c);

        Roadster r = new Roadster();
        r.setMake("Mini");
        r.setModel("Cooper S");
        r.setCoolFactor(CoolFactor.COOLEST);
        em.persist(r);

        Motorcycle m = new Motorcycle();
        em.persist(m);
    }
}
```



Polymorphisme (suite)

- Des requêtes polymorphes ! Si ! Si !

```
public List getAllRoadVehicles() {  
    Query q = em.createQuery(  
        "SELECT r FROM RoadVehicleSingle r");  
    return q.getResultList();  
}  
...  
}  
...
```



Polymorphisme : code client

```
...
public class RoadVehicleClient {
    public static void main(String[] args) {
        InitialContext ic;
        try {
            ic = new InitialContext();
            String name =

                RoadVehicleStateless.class.getName();
            RoadVehicleStateless rvs =
                (RoadVehicleStateless)ic.lookup(name);

            rvs.doSomeStuff();

            for (Object o : rvs.getAllRoadVehicles()) {
                System.out.println("RoadVehicle: "+o);
            }
        }
        catch (NamingException e) {
            e.printStackTrace();
        }
    }
}
```

Polymorphisme : oui, ça marche !

- ▶ C'est bien la méthode toString() de chaque sous-classe qui est appelée !
- ▶ La requête à récupéré tous les RoadVehicle (s)

```
RoadVehicle: Coupe: Car:  
    Make: Bob, Model: E400, Number of passengers: 5  
RoadVehicle: Motorcycle:  
    Make: null, Model: null, Number of passengers: 2  
RoadVehicle: Roadster: Car:  
    Make: Mini, Model: Cooper S, Number of passengers: 2
```



Sources / Références

- ▶ Ce cours couvre les besoins les plus importants, vous pourrez retrouver un cours très complet sur JPA2 et sur la persistance en java en général sur la page de Richard Grin
 - ▶ <http://deptinfo.unice.fr/~grin/mescours/minfo/modpersobj/supports/index.html>
- ▶ Aussi, une bonne référence en ligne sur JPA2
 - ▶ <http://www.objectdb.com/api/java/jpa>, en particulier la section JPA2 annotations
- ▶ La javadoc Java EE 6 pour le package et sous-packages **javax.persistence**
 - ▶ <http://docs.oracle.com/javaee/6/api/index.html?javax/persistence/package-summary.html>

