JPA

EclipseLink

1 Présentation

- JPA 1.0 a été introduite dans la spécification JAVA EE 5, la spécification JEE 6 utilise JPA 2.0
- JPA est une abstraction de la couche JDBC, les classes et les annotations JPA sont dans le package javax.persistance. Les annotations de mapping peuvent être regroupées en deux catégories:
  - Logique: description du modèle d'entités.
  - Physique: description des tables et des relations de la base de données
- Composants:
  - ORM: mécanisme de mapping relationnel objet
  - L'API Entity Manager qui gère les opérations CRUD
  - JPQL : langage de requête orienté objet.
  - JTA (Java Transaction API) : Mécanisme de gestion des verrouillages et des transactions dans un environnement concurrent
  - CallBack et listeners
- Frameworks de persistance JPA
  - TopLink.
  - EclipseLink (framework de référence JPA, basé sur TopLink).
  - Hibernate.

2- Exemple

Etape 1 Mise en place de l'unité de persistance dans le fichier META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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_2_0.xsd">
  <persistence-unit name="jpa1" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>jpa.Client</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/bdJpa"/>
      <property name="javax.persistence.jdbc.password" value="jpa"/>
      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.user" value="jpa"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Etape2 création de l'entité de persistance

```
package jpa;
import javax.persistence.*;
@Entity
public class Client implements Serializable {
   @Id
private Long id;
private String prenom;
private String nom;
private String dateNaissance;
private String dateCreation;
private int age;
private String email;
private Adresse adresse;

//méthodes get et set }
```

Etape 3 Utilisation de l'EntityManager

```
 EntityManagerFactory
emf=Persistence.createEntityManagerFactory("jpa1");
      EntityManager em=emf.createEntityManager();
      Client cl=new Client();
      cl.setAge(3);
      cl.setEmail("cl@un.ma");
      cl.setNom("unclient");
      cl.setDateNaissace("01/01/1967");
      cl.setPrenom("ali");
      EntityTransaction tx=em.getTransaction();
      tx.begin();
      em.persist(cl);
String jql = "Select c from Client  c";
         Query requete = em.createQuery(jql);
         List<Client> liste = requete.getResultList();
            for (Client cli : liste) {
               System.out.println(cli.getNom());
            }
 tx.commit();
      em.close();
      emf.close();
// Utiliser une requête JPQL
```

L'utilisation du mécanisme des transactions est obligatoire, dans le cas des opérations (persist, merge et remove), seule la méthode find() ne nécessite pas de transaction pour l'accès aux données.

Dans le cadre d'une application Java SE l'utilisation des méthodes Begin et commit est obligatoire, dans le cadre d'une application Java EE l'API JTA se charge du mécanisme des transactions.

3 Entity

- Entity
  - Une classe Entité est une classe java qui doit justifier les conditions suivantes:
    - Etre annotée par javax.persistence.Entity ou bien elle doit être décrite dans le descripteur xml comme une entité.
    - La classe doit posséder une clé primaire annotée par Id
    - La classe doit posséder un constructeur par défaut publique ou protégé.
    - La classe ne peut être de type enum ou interface
    - La classe ne doit pas être final et ne doit pas posséder des méthodes déclarées final, ni des attributs persistants déclarés final.
    - Si des instances de la classe doivent être passées par valeur, la classe entité doit implémenter l'interface Serializable.
    - Les attributs de la classe déclarés public sont ignorés par l'unité de persistance.
    - L'accès aux propriétés est conforme aux JavaBeans.
  - Annotations
    - L'annotation @Table permet de modifier le nom par défaut de la table associée à l'entité @Table(name=" Tlivre").
    - @NamedQuery(name="Romans", query="Select l from Livre l where l.categorie=5"): permet de définir une requête JPQL.
    - @Column: pour personnaliser les attributs

1-2-5 Types d'accès

- Une annotation peut être appliquée aux attributs (accès de type champ) ou bien au méthodes get (accès de type propriété)
- Pour utiliser les deux types d'accès dans une même classe entité, il faut alors le définir d'une manière explicite à l'aide de l'annotation Access

```
@Entity
@Access(AccessType.PROPRETY)
public class Client {
private String id;
private String prenom;
private String nom;
private String dateNaissace;
private String dateCreation;
@Access(AccessType.FIELD)
@Transient // L'attribut ne sera pas associé à aucun champ de la base de données
private int age;
private String email;
   public void setAge(int age) {
      this.age = age;
   }
   public void setDateCreation(String dateCreation) {
      this.dateCreation = dateCreation;
   }
   public void setDateNaissace(String dateNaissace) {
      this.dateNaissace = dateNaissace;
   }
   public void setId(String id) { this.id = id;   }
   public void setNom(String nom) {       this.nom = nom;    }
   public void setPrenom(String prenom) {       this.prenom = prenom;    }
   public String getEmail() {       return email;    }
   public void setEmail(String email) {       this.email = email;    }
  public int getAge() {       return age;    }
@Temporal(TemporalType.TIMESTAMP)
   public String getDateCreation() {       return dateCreation;    }
@Temporal(TemporalType.DATE)
   public String getDateNaissace() {       return dateNaissace;    }
@Id
@GeneratedValue
   public String getId() {       return id;    }
   public String getNom() {       return nom;    }
   public String getPrenom() {       return prenom;    }}
```

l’annotation @Column

- L’annotation @Column permet de déterminer les propriétés d’une colonne
  - String name() default :
  - boolean unique() default false;
  - boolean nullable() default true;
  - boolean insertable() default true;
  - boolean updatable() default true;
  - String columnDefinition() default "";
  - String table() default "";
  - int length() default 255;
  - int precision() default 0; // decimal precision
  - int scale() default 0; // decimal scale

L’annotation @Temporal et @Transient

- 3 attributs possibles: DATE, TIME, TIMESTAMP
- Un attribut annoté par @Transient ne sera pas mappé

```
@Entity
public class Client {
@Id
@GeneratedValue
private String id;
private String prenom;
private String nom;
@Temporal(TemporalType.DATE)
private String dateNaissace;
@Temporal(TemporalType.TIMESTAMP)
private String dateCreation;
}
```

Clés primaires composées

- Une clé primaire composée peut être implémentée de deux manières: @EmbeddedId ou @IdClass

```
@Embeddable
 class NewsId {
   private String titre;
   private String langue;
   // constructeurs , getters, setters}

@Entity
 class News {
   @EmbeddedId
private NewsId id;
private String contenu;
// Constructeurs, getters, setters}
```

```
/* pour rechercher une news par id :
 * NewsId pk = new NewsId("Java EE 6",
"EN")
News news = em.find(News.class, pk);
*/
```

```
//Méthode 2: IdClass
/* la classe NewsId est une simple classe
POJO */
public class NewsId {
   private String titre;
   private String langue;
   // constructeurs , getters, setters
}

@Entity
@IdClass(NewsId.class)
public class News {
@Id private String titre;
@Id private String langue;
private String contenu;
// Constructeurs, getters, setters
}
```

@Enumerated

- Par défaut JPA stocke la valeur ordinale d'une énumération (EnumType.ORDINAL)

```
public enum CarteCreditType {
VISA,
MASTERCARD,
MAESTRO,
}
@Entity
public class CarteCredit {
@Id
private String numero;
@Temporal(TemporalType.DATE)
private Date dateExpiration;
private Integer cvc;//Card verification code
@Enumerated(EnumType.STRING)
private CarteCreditType ccType;
// Constructors, getters, setters
}
```

Collections de types (JPA 2.0)

- La collection peut être de type List, Set ou Collection

```
@Entity
public class Livre2 {
@Id
@GeneratedValue
(strategy=GenerationType.AUTO)
private Long id;
@Column(nullable=false,updatable=false)
private String titre;
private Float prix;
@Column(length=2000)
private String description;
private String isbn;
private Integer nbPages;
private Boolean illustrations;

@CollectionTable(name="tag" ) //Par défaut le
nom de la table est le nom de la classe _ nom de la
collection
@Column(name="valeur" )

private  ArrayList<String> tags;
…
```

Collection de type Map

```
@Entity
public class Cd1 {
@Id
@GeneratedValue
private Long id;
private String titre;
private Float prix;
private String description;
@Lob
private byte[] couverture;
@ElementCollection
@CollectionTable(name="piste")
@MapKeyColumn (name = "position")
@Column(name = "titre")
private Map<Integer, String> pistes;

}
```

Composition

```
@Embeddable
public class Adresse {
private String adresse2;
    private String ville;
    private String cp;
    private String pays;
}

@Entity
public class Client {
private String id;
private String prenom;
private String nom;
private String dateNaissance;

private String dateCreation;
private int age;
private String email;
@Embedded
 private Adresse adresse;
```

Relations

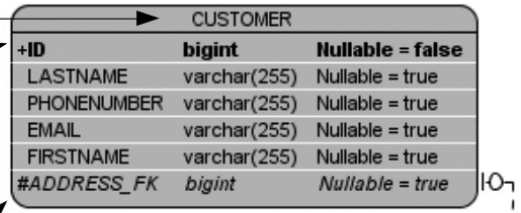
Table 3-1. *All Possible Cardinality-Direction Combinations*

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

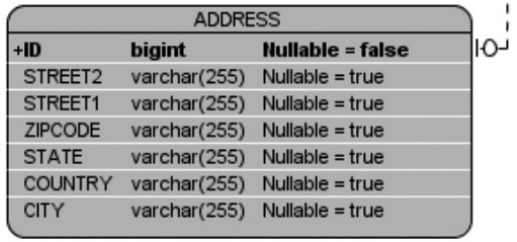
- Une association possède les attributs suivants:
  - cascade : cette propriété permet de gérer les contraintes de cascade entre les entités.
  - fetch : cette propriété permet de gérer le chargement de la classe liée.
  - mappedBy : cette propriété permet de gérer la relation bidirectionnelle.
  - optional : cette propriété permet de gérer les options de l'association.
  - targetEntity : cette propriété permet de gérer la classe entité qui est la cible de l'association.

1-5-1 OneToOne bidirectionnel

```java
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```



CUSTOMER		
+ID	bigint	Nullable = false
LASTNAME	varchar(255)	Nullable = true
PHONENUMBER	varchar(255)	Nullable = true
EMAIL	varchar(255)	Nullable = true
FIRSTNAME	varchar(255)	Nullable = true
#ADDRESS_FK	*bigint*	*Nullable = true*



ADDRESS		
+ID	bigint	Nullable = false
STREET2	varchar(255)	Nullable = true
STREET1	varchar(255)	Nullable = true
ZIPCODE	varchar(255)	Nullable = true
STATE	varchar(255)	Nullable = true
COUNTRY	varchar(255)	Nullable = true
CITY	varchar(255)	Nullable = true

OneToOne unidirectionnel

- La relation entre les deux entités est tout simplement matérialisée par la propriété adresse de l'entité Client, le nom par défaut de la clé étrangère sera Adresse_id et sera nullable par défaut sinon on peut configurer les annotations OneToOne et JoinColumn (cette annotation permet de configurer une clé étrangère et possède les mêmes attributs que l'annotation Column :

```
@Entity
public class Client implements Serializable {
   @Id  @GeneratedValue
private Long id;
private String prenom;
private String nom;
private String dateNaissance;
private String dateCreation;
private int age;
private String email;
private Adresse adresse;
}
```

```
@Entity
public class Adresse implements Serializable {
   @Id  @GeneratedValue
   private Long id;
   private String adresse1;
   private String adresse2;
   private String ville;
   private String cp;
   private String pays;}
```

1-5-3 OneToMany Unidirectionnel

- La relation OneToMany unidirectionnelle ne nécessite pas d'annotations
- La relation peut être personnalisée à l'aide des annotations JoinTable et JoinColumn

```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}
```

```
@Entity
@Table(name = "order_line")
public class OrderLine {

    @Id @GeneratedValue
    private Long id;
    private String item;
    private Double unitPrice;
    private Integer quantity;

    // Constructors, getters, setters
}
```

ManyToMany

```
@Entity
public class CD {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @ManyToMany(mappedBy = "appearsOnCDs")
    private List<Artist> createdByArtists;

    // Constructors, getters, setters
}
```

```
@Entity
public class Artist {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", ➥
        joinColumns = @JoinColumn(name = "artist_fk"), ➥
        inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> appearsOnCDs;

    // Constructors, getters, setters
}
```

- Valeur par défaut de l'attribut fetch selon le type d'association:
  - OneToOne: EAGER
  - ManyToOne: EAGER
  - OneToMany:LAZY
  - ManyToMany: LAZY

@OrderBy: Tri dynamique

```java
@Entity
public class News {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String content;
    @OneToMany(fetch = FetchType.EAGER)
    @OrderBy("postedDate DESC")
    private List<Comment> comments;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Comment {

    @Id @GeneratedValue
    private Long id;
    private String nickname;
    private String content;
    private Integer note;
    @Column(name = "posted_date")
    @Temporal(TemporalType.TIMESTAMP)
    private Date postedDate;

    // Constructors, getters, setters
}
```

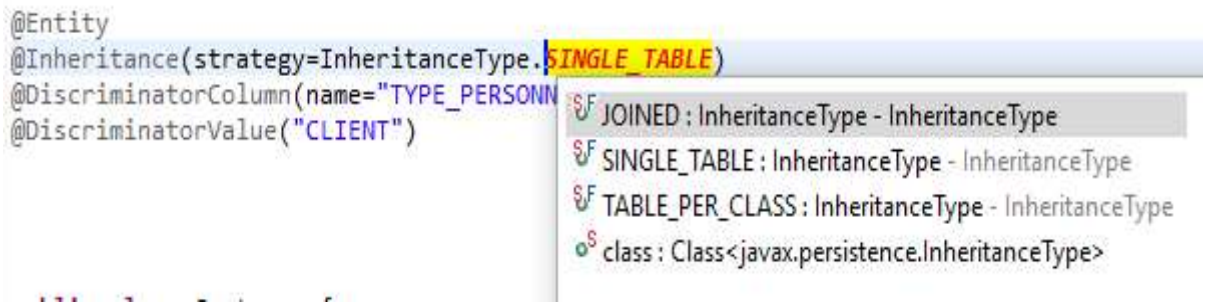
Héritage

- Pour réaliser le mapping d'une relation d'héritage JPA propose trois stratégies:
  - Une seule pour stocker les attributs de toutes les classes d'une arborescence d'héritage (stratégie par défaut). Un champ (le champ discriminant) permet de déterminer le type de la classe (le nom par défaut de ce champ est DTYPE de type varchar et peut être défini par @DiscriminatorColumn (l'attribut DiscriminatorType définit le type de ce champ), la valeur par défaut est le nom de l'entité et cette valeur peut être personnalisée à l'aide de l'annotation @DiscriminatorValue
    - Remarque:
      - Les attributs des entités filles doivent être nullable
  - Jointure des sous classes: Une table par classe de la hiérarchie d'héritage (que la classe soit abstraite ou concrète)
    - Inconvénients: performance
  - Une table pour chaque classe concrète.(le support de cette stratégie est optionnel dans la spécification JPA 2.0), les propriétés de la classe de base sont dupliquées dans les classes filles.
    - Inconvénients: les appels polymorphiques sont plus coûteux que dans les deux autres stratégies.
    - Les annotations @AttributeOverride et @AttributeOverrides permettent de renommer les attributs dupliqués dans les classes filles.

```
@Entity
@AttributeOverrides({
@AttributeOverride(name = "id",➥
column = @Column(name = « livre_id")),
@AttributeOverride(name = "titre,➥
column = @Column(name = « livre_titre")),
@AttributeOverride(name = "description",➥
column = @Column(name =
"livre_description"))
})
```

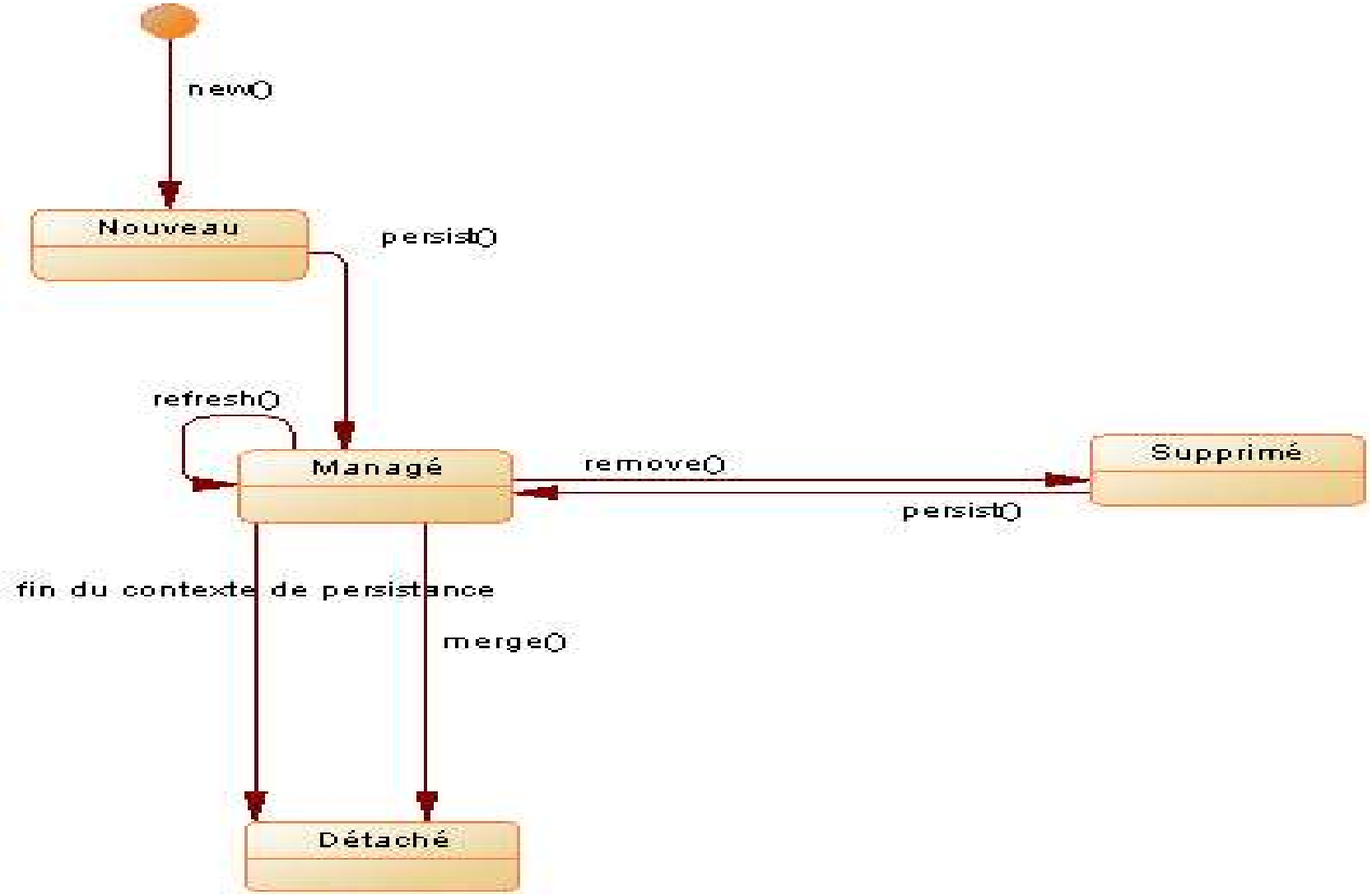
Annotations possibles (optionnelles)

- Classe de base



- Sous classes
  - DiscriminatorValue("Important")

Le cycle de vie d'une entité



On distingue quatre états Managés (managed) :

- Nouveau (new) : c'est un bean qui vient d'être instancié. Il n'est associé à aucun contexte de persistance.
- Détaché (detached) : le bean est détaché du contexte de persistance. Il n'est plus synchronisé avec la base.
- Supprimé (removed) : le bean est attaché au contexte de persistance mais est prévu pour être supprimé de la base.

Les méthodes de l'interface javax.persistence.EntityManager (API JPA) conduisant à ces états sont les suivantes :

- refresh() : dans la mesure où les entités peuvent utiliser un cache pour stocker les données, cette méthode permet de mettre à jour les entités par rapport à la base. L'entité devient managée.
- merge() : cette méthode synchronise une entité détachée avec le contexte de persistance. L'entité détachée devient alors managée.
- persist() : l'entité est insérée en base. Elle devient du coup managée.
- remove() : l'entité est supprimée de la base après commit. Ce qui la conduit à l'état supprimé.
- flush() : cette méthode synchronise de façon explicite le contexte de persistance avec la base.

JPQL

- Exemple de requêtes:
  - SELECT li From Livre li
  - SELECT li From Livre li Where li.titre='UML'
  - SELECT li.titre, li.Editeur from livre li
  - SELECT c.Adresse.pays from client c
- Syntaxe d'une requête:

  SELECT
  FROM
  [WHERE]
  [ORDER BY]
  [GROUP BY]
  [HAVING]
- Sélection selon une condition (JPA 2.0)

  SELECT CASE l.editeur WHEN 'Eyrolles"
  THEN l.prix * 0.5
  ELSE l.prix * 0.8
  END
  FROM Livre l

- L'exécution d'une requête peut retourner une valeur, une collection de 0 ou plusieurs entités (ou attributs) , les données retournées peuvent contenir des doublons.
- Pour ignorer les doublons:
  - SELECT DISTINCT c
    FROM Client c
  - SELECT DISTINCT c.Nom
    FROM Client c
- Les fonctions d'agrégation suivantes peuvent être utilisées : AVG, COUNT, MAX, MIN, SUM.
- exemples de requêtes (clause WHERE):

```
SELECT c
FROM Client c
WHERE c.age NOT BETWEEN 40 AND 60
SELECT c
FROM Client c
WHERE c.addresse.pays IN ('USA', 'France')
SELECT c
FROM Client c
WHERE c.email LIKE '%mail.com'
```

3-3 L’interface Query

```
public interface Query {
// Exécute une requête et retourne un résultat
public List getResultList();
public Object getSingleResult();
public int executeUpdate();
// Définition des paramètres d’une requête
public Query setParameter(String name, Object value);
public Query setParameter(String name, Date value,➥
TemporalType temporalType);
public Query setParameter(String name, Calendar value,➥
TemporalType temporalType);
public Query setParameter(int position, Object value);
public Query setParameter(int position, Date value,➥
TemporalType temporalType);
public Query setParameter(int position, Calendar value,➥
TemporalType temporalType);
public Map<String, Object> getNamedParameters();
public List getPositionalParameters();
public Query setMaxResults(int maxResult);
public int getMaxResults();
public Query setFirstResult(int startPosition);
public int getFirstResult();

public Query setHint(String hintName, Object value);
public Map<String, Object> getHints();
public Set<String> getSupportedHints();
//
public Query setFlushMode(FlushModeType flushMode);
public FlushModeType getFlushMode();
//
public Query setLockMode(LockModeType lockMode);
public LockModeType getLockMode();
// Autorise l’ccès à l’API propriétaire du fournisseur
public <T> T unwrap(Class<T> cls);
}
```

The methods that are mostly used in this

Exécution de requêtes

- JPA 2.0 supporte 4 types d'exécution de requêtes
  - Requêtes dynamiques:
    - Query createQuery(String jpqlString)
  - Requêtes nommées
    - @NamedQuery
    - Query createNamedQuery(String name)
  - Requêtes natives
    - @NamedNativeQuery
    - Query createNamedQuery(String name)
    - Query createNativeQuery(String sqlString)
  - API criteria
    - Query createQuery(QueryDefinition qdef)

JPQL

- Query createQuery(String jpqlString)
- Query createNamedQuery(String name)
- Query createNativeQuery(String sqlString)
- Query createNativeQuery(String sqlString, Class resultClass)
- <T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass)
- <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery)
- <T> TypedQuery<T> createQuery(String jpqlString, Class<T> resultClass)