Persistance avec JPA / Hibernate

Raquel Araujo de Oliveira

raquel.oliveira@irit.fr

Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

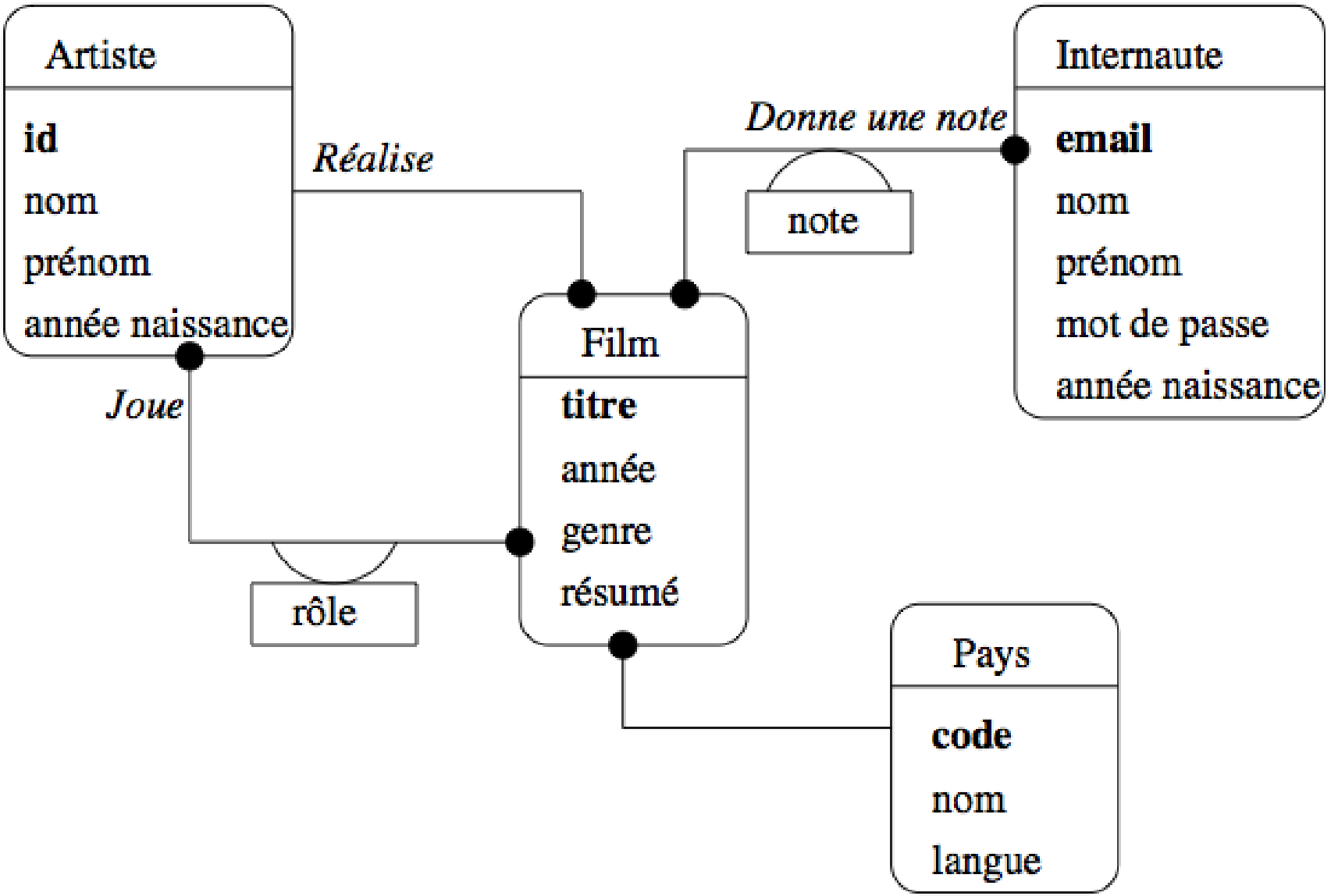
La plateforme Java EE

- MVC (Modèle-Vue-Contrôleur) : un motif d'architecture logicielle
- Technologies Java EE pour implémenter une application web selon MVC :
  - JPA (Java Persistence API) / Hibernate [Modèle]
  - JSP (Java Server Page) [Vues]
  - Servlets [Contrôleur]

Plan

- Exemple de base : *Films*
- La conception de cette base (notée en UML) est brièvement présentée
- Comment y accéder grâce à l'interface d'accès à une base relationnelle en Java: JDBC
- Optimiser l'accès aux données : JPA/Hibernate
- Lecture des données

Exemple : Une application de gestion de films



Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

La persistance avec JDBC

- JDBC : Java DataBase Connectivity
- Créée par Sun Microsystems – depuis racheté par Oracle Corporation
- API de base (intégrée à la Java Standard Edition) pour la communication avec les BD relationnelles
- JDBC normalise cette communication :
  - en principe une application s'appuyant sur JDBC peut de manière transparente passer d'une base (ex: MySQL) à l'autre (ex: PostgreSQL)
  - en pratique cela suppose une certaine rigueur pour s'en tenir à la partie normalisée de SQL et éviter les extensions particulières de chaque système
  - exemple...
- JDBC est implanté pour chaque système relationnel sous la forme d'un "pilote" (driver) à télécharger et à copiez dans le répertoire "WEB-INF/lib" de l'application (DEMO)

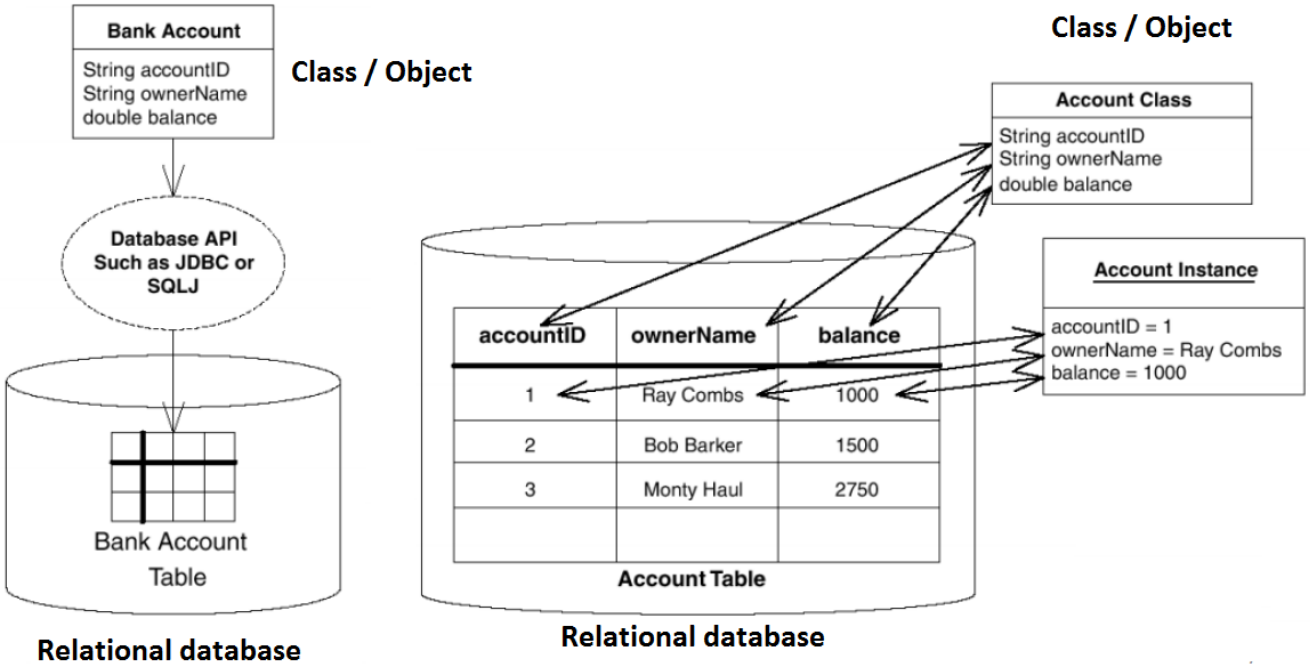
JDBC - principe

1. Chargement d'un driver (ici, MySQL)
2. Établissement d'une connexion vers la base (nom du serveur, nom de la base)
3. Création d'une requête
4. Exécution de la requête
5. Utilisation du résultat

```
public static void main(String[] args){
  (1) Class.forName(``com.mysql.jdbc.Driver'');

  (2) String url = ``jdbc:mysql://localhost/add'';
      Connection connection = DriverManager.getConnection(
                                                  url, login, passwd);
  (3) String query = ``SELECT * FROM Film'';

  (4) Statement s = connection.createStatement();
      ResultSet films = s.executeQuery(query);

  (5) while (films.next()){
        System.out.println(films.get(``titre''));
}
```

Incompatibilité de représentation

- *impedance mismatch* : l'incompatibilité de représentation des données entre une base relationnelle et une application objet
- dans une application objet, le domaine fonctionnel est modélisé par des classes, et les instances de ces classes (des objets donc) doivent être rendus persistants par stockage dans la base relationnelle
- cette incompatibilité nécessite des conversions répétitives : des objets en lignes, et réciproquement des lignes ramenées par des requêtes en objets



La persistance par mapping objet/BD relationnelle

```
public class Film{
        private String titre;

        public Film() {}

        public void setTitre(String leTitre) { titre = leTitre;}

        public String getTitre() {return titre;}
}
```

```
public List<Film> chercheFilmsC() throws SQLException{
  List<Film> resultat = new ArrayList<Film>();

  Statement statement = connexion.createStatement();
  ResultSet films = statement.executeQuery( "SELECT␣*␣FROM␣Film");
  while (films.next()) {
    Film film = new Film ();
    film.setTitre(films.getString("titre"));
    resultat.add(film);
  }
  return resultat;
}
```

La persistance par mapping objet/BD relationnelle

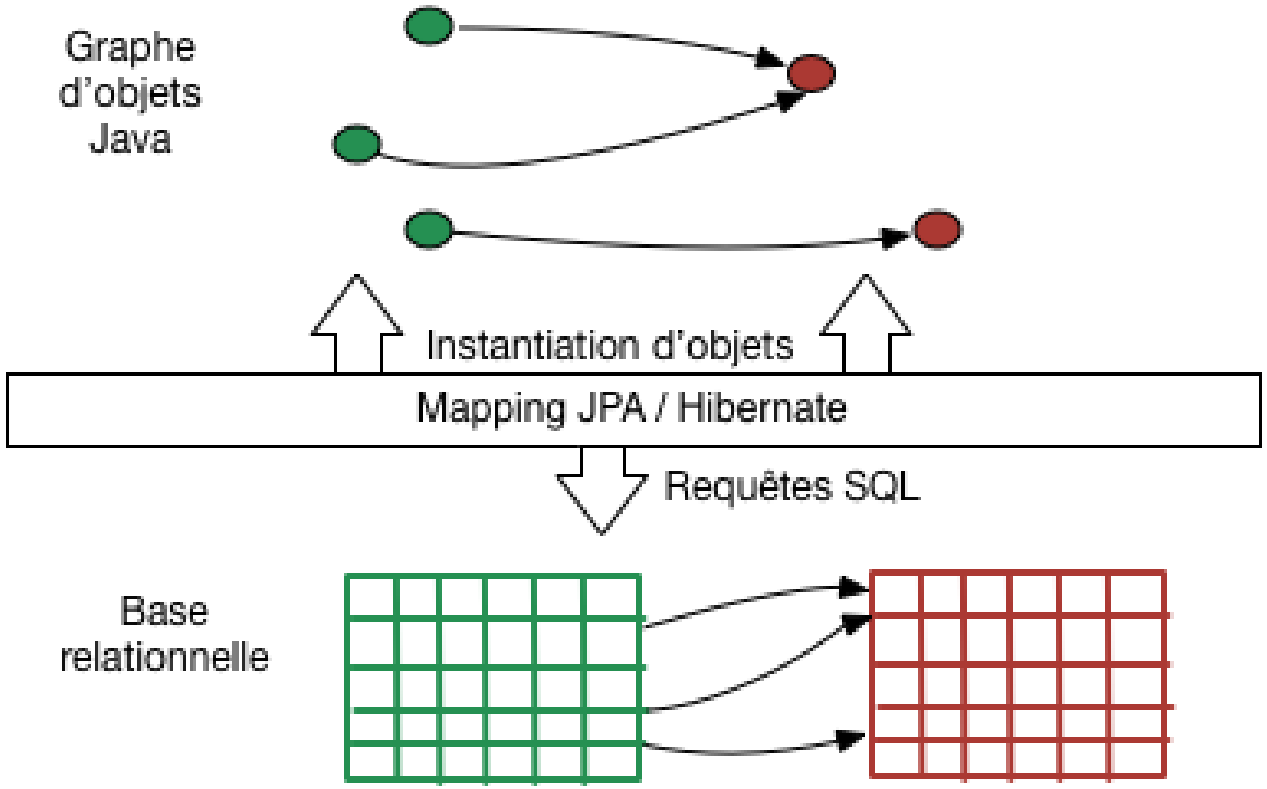
- à l'échelle d'une application non triviale (avec des dizaines de tables et des modèles complexes), la méthode consistant à coder manuellement les conversions est très fastidieuse et très répétitif
- dans une application objet, tout doit être objet; la couche de persistance, celle qui accède à la base de données, doit donc renvoyer des objets: elle est en charge de la conversion
- JDBC est une API d'assez bas niveau pour être utilisée dans le cadre d'une application basée sur le motif MVC
- Le rôle d'un système ORM (Object-Relational Mapping) est de convertir automatiquement, à la demande, la base de données sous forme d'un graphe d'objet
- MAIS : les ORM s'appuient toujours sur JDBC

Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

La persistence avec JPA

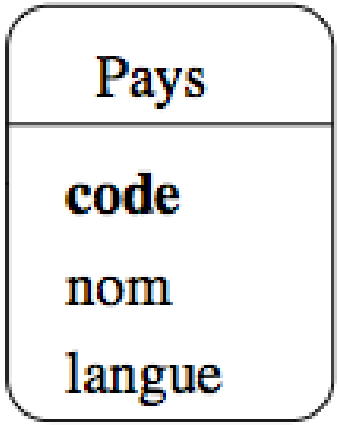
- Des outils ORM existent et sont normalisés dans la plateforme Java sous le nom générique de *Java Persistence API* ou `JPA`
- `JPA` est une spécification intégrée a Java EE qui vise à standardiser la couche d'association entre une base relationnelle et une application Java construite sur des objets



Hibernate

- Hibernate est un exemple de framework de persistance (un des plus connus) qui implémente la spécification JPA
- Plus la spécification JPA est respecté, plus il possible de passer d'un framework à l'autre sans problème

Une entité simple



Mapping d'une entité – Informations nécessaires :

- La table de la BD
- Les champs de la table
- La clé primaire

Mapping d'une entité via fichier XML

```
<hibernate-mapping>
    <class name="pojos.Person" table="t_person">
        <id name="Id" type="integer">
            <column name="id" sql-type="int(4)" />
            <generator class="increment" />
        </id>
        <property name="Name" type="string">
            <column name="nom" sql-type="char(50)" not-null="true" />
        </property>
        <property name="Age" type="integer">
            <column name="age" sql-type="int(4)" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Mapping d'une entité via fichier XML

```
<hibernate-mapping>
    <class name="pojos.Person" table="t_person">
        <id name="Id" type="integer">
            <column name="id" sql-type="int(4)" />
            <generator class="increment" />
        </id>
        <property name="Name" type="string">
            <column name="nom" sql-type="char(50)" not-null="true" />
        </property>
        <property name="Age" type="integer">
            <column name="age" sql-type="int(4)" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

- la configuration est séparée de la classe
- convention : un fichier de mapping par classe de persistance
- le nom de ces fichiers doit se terminer par *.hbm.xml*
- endroit du fichier ?

Mapping d'une entité via annotations

```
@Entity
public class Pays{
  public Pays() {}
  @Id
  private String code;
  public void setCode(String c) {code = c;}
  public String getCode() {return code ;}

  @Column
  private String nom;
  public void setNom(String n) {nom = n;}
  public String getNom() {return nom;}

  @Column(name = ‘‘language’’, unique = false)
  private String langue;
  public void setLangue(String l) {langue = l;}
  public String getLangue() {return langue;}
}
```

- la configuration est dans la classe
- `@Entity` : les instances de la classe sont stockées dans la BD
- `@Table(name="Country")` si nom de la table différent
- `@Transient` : la colonne n'est pas stockée dans la BD

- `@GeneratedValue(strategy = GenerationType.AUTO)` : pour générer des ids

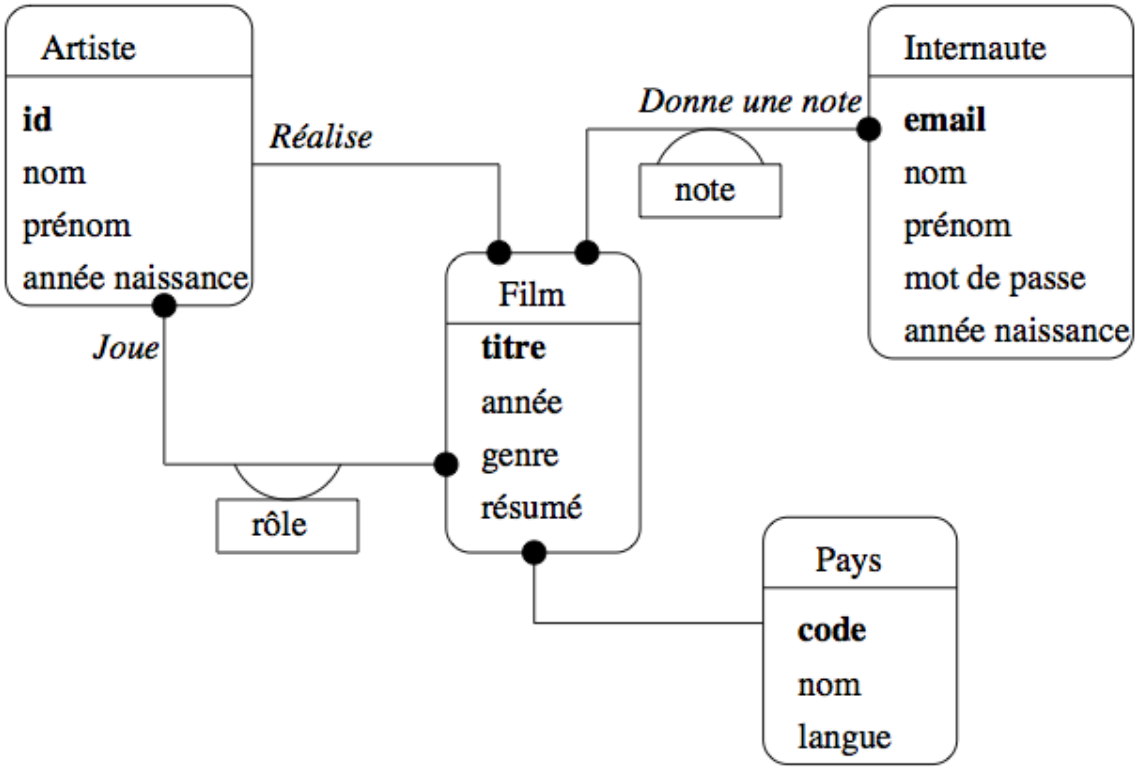
La représentation des associations

Types d'associations :

- un à un (peu fréquentes)
- plusieurs à un
- un à plusieurs
- plusieurs à plusieurs

Représentations :

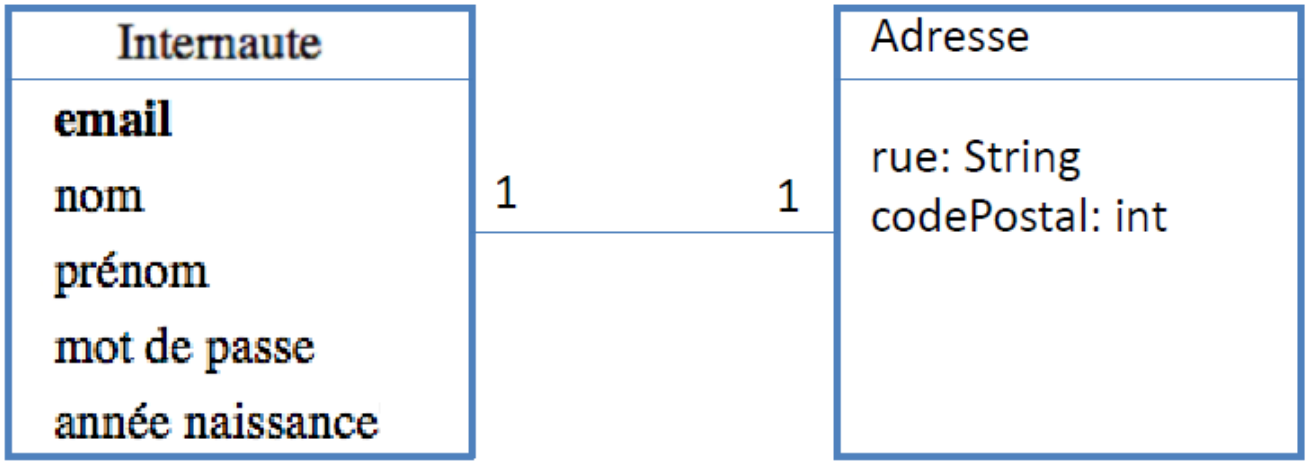
- Uni-directionnelles
- Bidirectionnelles



Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

Mapping un à un



- Différentes stratégies pour persister cette relation en base de données
- Supposons une seule table définie dans la base de donnée contenant l'ensemble des champs
  - Attributs de l'Internaute
  - Attributs de l'Adresse

Mapping un à un

Composants

- *Composant* : un objet sans identifiant, qui ne peut être persistant que par rattachement à une entité (classes identifiées par `@Entity`)
- On va considérer que l'adresse est un *composant* de la représentation d'une internaute; il dispose d'une unité propre et distinguable du reste de la table
  - nécessité de contrôler qu'un code postal est correct, une logique applicative qui n'est pas vraiment pertinente pour l'entité Internaute
  - on peut réutiliser la définition du composant pour d'autres entités, comme par exemple Société

Mapping un à un

Composants

```
@Embeddable
public class Adresse {
        @Column
        String rue;
        public void setRue(String v) {rue = v;}
        public String getRue() {return rue;}

        @Column
        String codePostal;
        public void setCodePostal(String v) {codePostal = v;}
        public String getCodePostal() {return codePostal;}
}
```

- `@Embeddable` : la classe est un composant (pas de `@Entity`)
- Un composant n'a pas d'identifiant (marqué `@Id`) et ne peut donc pas être sauvegardé dans la base indépendamment. Il faut au préalable le rattacher à une entité, ici `Internaute`

Mapping un à un

Composants

```
@Entity
public class Internaute {
        @Id
        private String email;
        public void setEmail(String e) {email = e;}

        @Column
        private String nom;
        public void setNom(String n) {nom = n;}
        public String getNom() {return nom;}
        ...

        @Embedded
        private Adresse adresse;
        public void setAdresse(Adresse a) {adresse = a;}
        public Adresse getAdresse() {return adresse;}
}
```

La propriété *adresse* devient, comme *nom*, une propriété persistante de l'entité

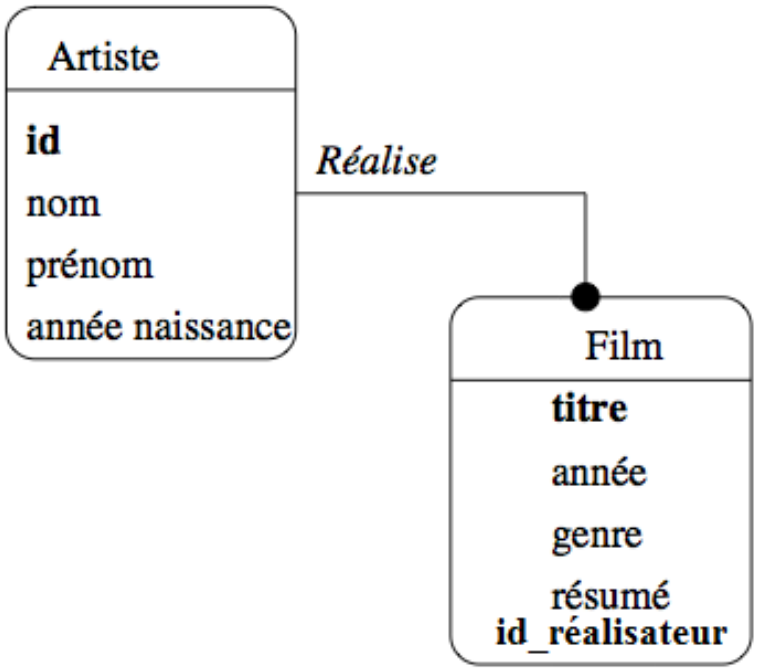
Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

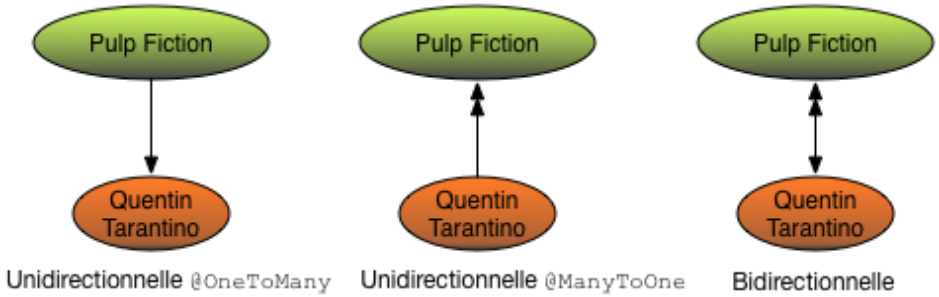
Mapping plusieurs à un

Représentations uni et bidirectionnelle

Plusieurs films réalisés par un (unique) réalisateur :



En relationnel, nous avons dans la table "Film" un attribut "id_réalisateur" qui sert de clé étrangère



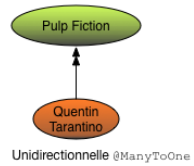
En Hibernate nous pouvons représenter l'association de trois manières:

- dans un film, on place un lien vers le réalisateur (unidirectionnel)
- dans un artiste, on place des liens vers les films qu'il a réalisés (unidirectionnel)
- on représente les liens des deux côtés (bidirectionnel)

Mapping plusieurs à un

représentation uni-directionnelle

"plusieurs films peuvent avoir été réalisés par un seul réalisateur"
En Hibernate :




```
@Entity
public class Film {
        ...
        @ManyToOne
        @JoinColumn (name="id_realisateur")
        private Artiste realisateur;
        public void setRealisateur(Artiste a) {realisateur = a;}
        public Artiste getRealisateur() {return realisateur;}
}
```

- `@ManyToOne` indique à Hibernate que l'objet "realisateur", instance de la classe "Artiste", encode un des côtés d'une association plusieurs-à-un entre les films et les réalisateurs
- `@JoinColum` indique la clé étrangère qui permet de trouver le réalisateur ("id_realisateur" dans la table "Film")

Différence entre la représentation relationnelle et le modèle de données en java

Concernant les associations

Il n'est pour l'instant pas possible de récupérer la liste des films réalisés par un réalisateur car nous n'avons pas représenté dans le modèle objet (Java) ce côté de l'association. En revanche, une simple jointure en SQL nous donnerait cette information

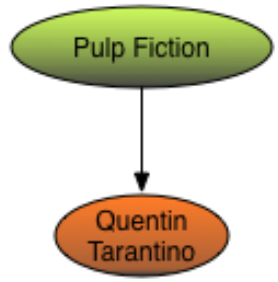
Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

Mapping un à plusieurs

représentation uni-directionnelle

"un réalisateur peut avoir réalisé plusieurs films"
En Hibernate :



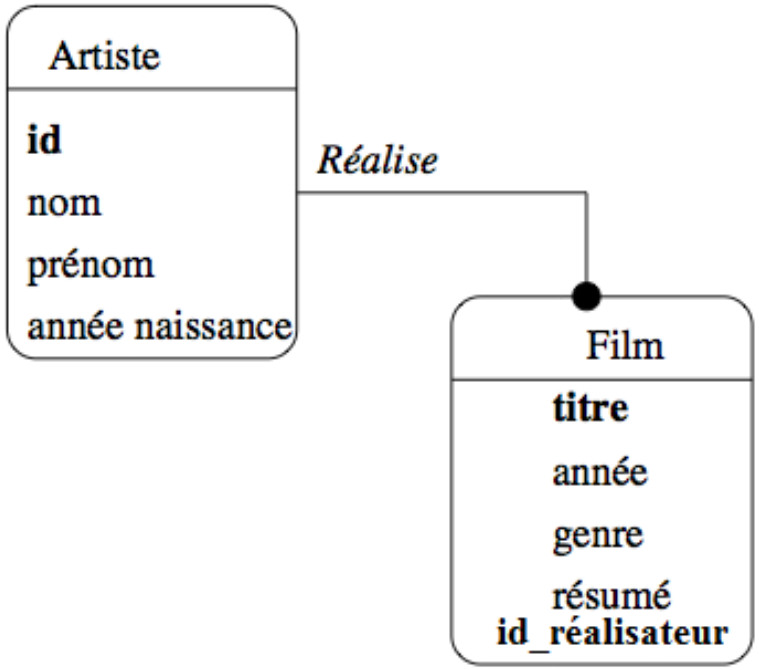
Unidirectionnelle `@OneToMany`

```
@Entity
public class Artiste {
        ...
        @OneToMany
        @JoinColumn(name="id_realisateur")
        private Set<Film> filmsRealises = new HashSet<Film>();
        public void addFilmsRealise(Film f) {filmsRealises.add(f) ;}
        public Set<Film> getFilmsRealises() {return filmsRealises;}
}
```

- De l'autre côté de l'association, on utilise l'annotation `@OneToMany`
- On indique, comme précédemment, que la clé étrangère est "id_realisateur" et Hibernate comprend qu'il s'agit d'un attribut de Film
- On fournit la méthode add (au lieu de set)

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film



Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
  - là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
  - là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel
  - quand on renseigne un côté de l'association, java ne sait pas déduire l'autre

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
  - là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel
  - quand on renseigne un côté de l'association, java ne sait pas déduire l'autre
  - code alourdi par la nécessité d'attacher les deux bouts de l'association aux objets correspondants

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
  - là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel
  - quand on renseigne un côté de l'association, java ne sait pas déduire l'autre
  - code alourdi par la nécessité d'attacher les deux bouts de l'association aux objets correspondants
  - Parce que les deux spécifications de chaque bout de l'association sont indépendantes l'une de l'autre, Hibernate effectue deux `update`

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
  - là où il y a deux emplacements distincts en java, il n'y en a qu'un en relationnel
  - quand on renseigne un côté de l'association, java ne sait pas déduire l'autre
  - code alourdi par la nécessité d'attacher les deux bouts de l'association aux objets correspondants
  - Parce que les deux spécifications de chaque bout de l'association sont indépendantes l'une de l'autre, Hibernate effectue deux `update`
  - l'inconvénient: un surplus de requêtes transmises à la base, ce qui peut pénaliser les performances

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
- Il faut déclarer qu'un côté de l'association est *responsable* de la mise à jour. Cela se fait avec l'attribut `mappedBy`, comme suit :

```
public class Artiste { ...
  @OneToMany(mappedBy="realisateur")
  private Set<Film> filmsRealises = new HashSet<Film>();   ... }
```

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
- Il faut déclarer qu'un côté de l'association est *responsable* de la mise à jour. Cela se fait avec l'attribut `mappedBy`, comme suit :

```
public class Artiste { ...
  @OneToMany(mappedBy="realisateur")
  private Set<Film> filmsRealises = new HashSet<Film>();   ... }
```

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
- Il faut déclarer qu'un côté de l'association est *responsable* de la mise à jour. Cela se fait avec l'attribut `mappedBy`, comme suit :

```
public class Artiste { ...
  @OneToMany(mappedBy="realisateur")
  private Set<Film> filmsRealises = new HashSet<Film>();    ... }
```

- Le mapping de l'association est pris en charge par la classe du côté "plusieurs" ("Film")

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
- Il faut déclarer qu'un côté de l'association est *responsable* de la mise à jour. Cela se fait avec l'attribut `mappedBy`, comme suit :

```
public class Artiste { ...
  @OneToMany(mappedBy="realisateur")
  private Set<Film> filmsRealises = new HashSet<Film>();    ... }
```

- Le mapping de l'association est pris en charge par la classe du côté "plusieurs" ("Film")
- Dans le côté "un" (classe "Artiste"), on supprime l'annotation `@JoinColumn` puisqu'elle devient inutile. `MappedBy` indique qu'un objet associé de la classe "Film" maintient un lien avec une instance de la classe courante ("Artiste") grâce à une propriété nommée "realisateur"

Représentation bidirectionnelle

Permet de naviguer dans l'association "réalise" à partir d'un réalisateur ou film

Une des principales difficultés conceptuelles du mapping objet-relationnel :

- Placer simultanément dans la classe "Film" la spécification avec `@ManyToOne` et dans la classe "Artiste" la spécification avec `@OneToMany` telles quelles décrites précédemment ne suffit pas
- Il faut déclarer qu'un côté de l'association est *responsable* de la mise à jour. Cela se fait avec l'attribut `mappedBy`, comme suit :

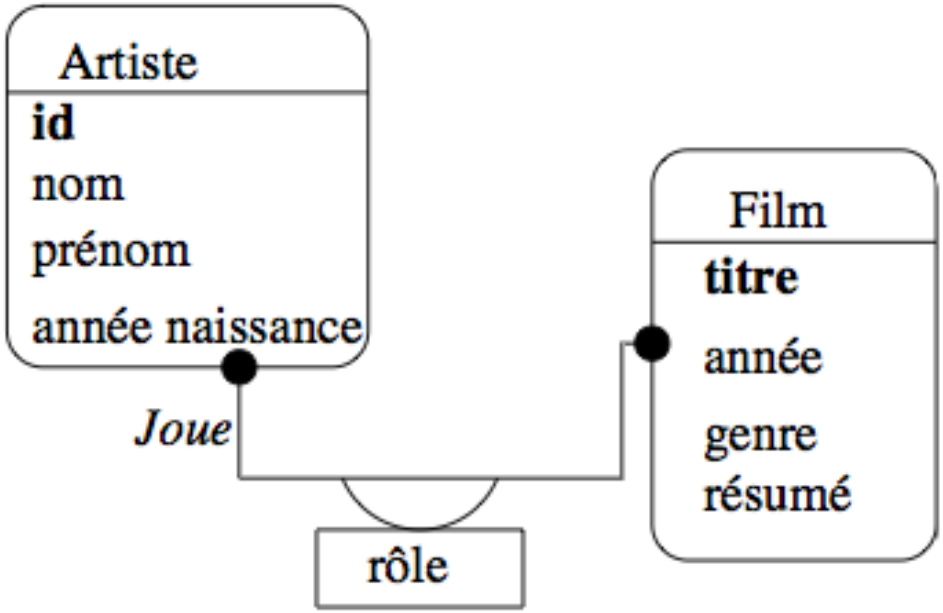
```
public class Artiste { ...
  @OneToMany(mappedBy="realisateur")
  private Set<Film> filmsRealises = new HashSet<Film>();    ... }
```

- Le mapping de l'association est pris en charge par la classe du côté "plusieurs" ("Film")
- Dans le côté "un" (classe "Artiste"), on supprime l'annotation `@JoinColumn` puisqu'elle devient inutile. `MappedBy` indique qu'un objet associé de la classe "Film" maintient un lien avec une instance de la classe courante ("Artiste") grâce à une propriété nommée "realisateur"
- Hibernate, en allant inspecter la classe "Film", trouvera que "realisateur" est mappé avec la base de données par le biais de la clé étrangère "id realisateur"

Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

Mapping plusieurs à plusieurs



- en relationnel, l'association devient une table dont la clé est la concaténation des clés des deux entités de l'association (i.e. une table "Role" avec une clé composée [id_film, id_acteur])
- Deux cas possibles :
  1. association sans attribut
  2. association avec attribut

Mapping plusieurs à plusieurs

association sans attribut (1/2)

```
@Entity
public class Film {
        ...
        @ManyToMany()
        @JoinTable(name = "Role",
          joinColumns = @JoinColumn(name = "id_film"),
          inverseJoinColumns = @JoinColumn(name = "id_acteur"))
        Set<Artiste> acteurs = new HashSet<Artiste>();
        public Set<Artiste> getActeurs() { return acteurs; }
        ...
}
```

- Ce côté est "responsable" du mapping avec la base de données
- Une association `@ManyToMany`, avec une seconde annotation `@JoinTable` qui décrit la table représentant l'association
- `joinColumns` indique la clé étrangère de la table "Role" vers la table de la classe courante ("Film")
- `inverseJoinColumns` indique la clé étrangère de la table "Role" vers la table représentant l'autre entité, "Artiste"

Mapping plusieurs à plusieurs

association sans attribut (2/2)

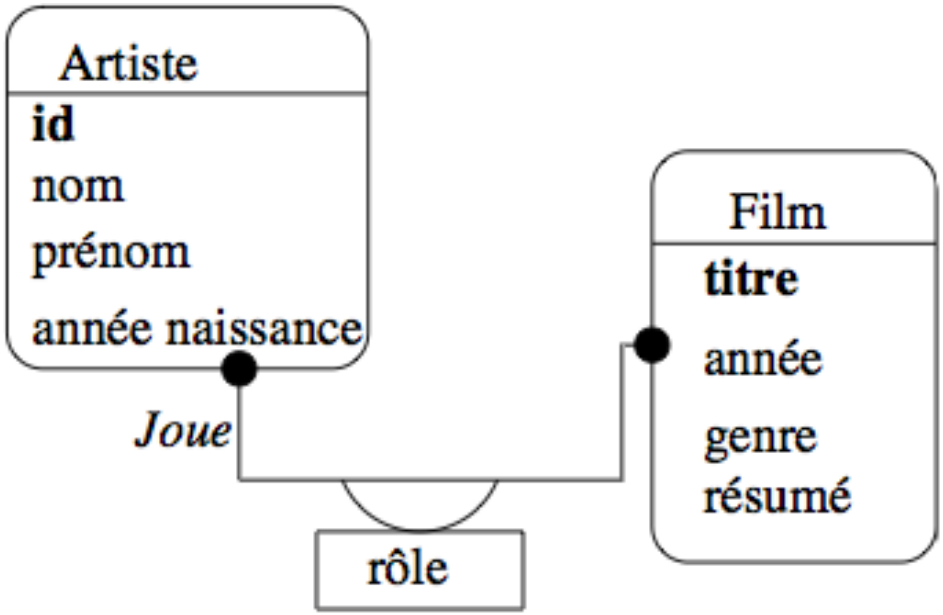
De l'autre côté :

```
@Entity
public class Artiste {
        ...
        @ManyToMany(mappedBy = "acteurs")
        Set<Film> filmologie;
        public Set<Film> getFilmologie() {
                return filmologie;
        }
        ...
}
```

- On ne répète pas la description du mapping. On indique avec `mappedBy` qu'elle peut être trouvée dans la classe "Film"
- Ça permet de naviguer d'un film vers ses acteurs et réciproquement

Mapping plusieurs à plusieurs

association avec attribut

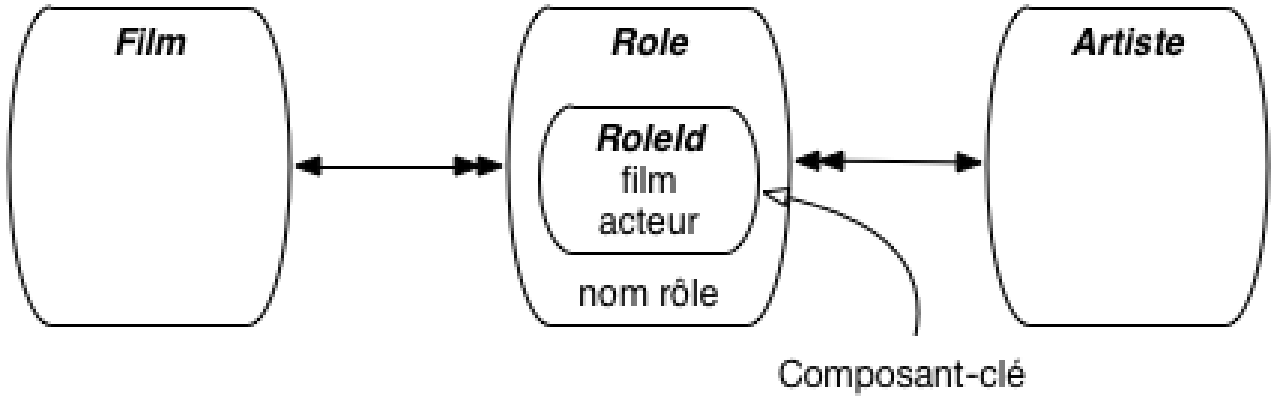


- Considérons que l'association "Role" contient le nom du rôle
- Pour y accéder, il faut représenter cette association dans le modèle JPA
  1. gérer la clé composée de l'association "Role"
  2. représenter cette association dans le modèle JPA comme une `@Entity`, avec des associations "un à plusieurs" vers, respectivement, "Film" et "Artiste"

Mapping plusieurs à plusieurs

association avec attribut

La classe “Role” est une entité composée d’une clé et d’un attribut, le nom du rôle:



- La clé est une instance d’une classe “RoleId” constituée de l’identifiant du film et de l’identifiant de l’acteur
- Il faut définir une classe (“RoleId”) avec les attributs constituant la clé seulement

Mapping plusieurs à plusieurs

association avec attribut

Utilisation de *composant* pour définir la clé (comme pour l'exemple de l'adresse dans le mapping un à un):

```
@Embeddable
public class RoleId implements java.io.Serializable {
        ...
        @ManyToOne
        @JoinColumn(name = "id_acteur")
        private Artiste acteur;
        public Artiste getActeur() { return acteur;     }
        public void setActeur(Artiste a) { this.acteur = a;     }

        @ManyToOne
        @JoinColumn(name = "id_film")
        private Film film;
        public Film getFilm() { return film; }
        public void setFilm(Film f) { this.film = f; }
}
```

- une classe dont les instances vont servir d'identifiants doit hériter de `serializable`

Mapping plusieurs à plusieurs

association avec attribut

Voici maintenant l'entité "Role"

```
@Entity
public class Role {
        @Id
        RoleId pk;
        public RoleId getPk() { return pk; }
        public void setPk(RoleId pk) { this.pk = pk; }

        @Column(name="nom_role")
        private String nom;
        public void setNom(String n) {nom= n;}
        public String getNom() {return nom;}
}
```

- La classe a comme identifiant une instance de "RoleId", et contient les attributs de l'association

Mapping plusieurs à plusieurs

association avec attribut

Comment coder les autres côtés de l'association (Pour le "Film"):

```
@Entity
public class Film {
        @OneToMany(mappedBy = "pk.film")
        private Set<Role> roles = new HashSet<Role>();
        public Set<Role> getRoles() { return this.roles; }
        public void setRoles(Set<Role> r) { this.roles = r; }
}
```

Même chose du côté des artistes :

```
@Entity
public class Artiste {
        @OneToMany(mappedBy = "pk.acteur")
        private Set<Role> roles = new HashSet<Role>();
        public Set<Role> getRoles() { return this.roles; }
        public void setRoles(Set<Role> r) { this.roles = r; }
}
```

La classe "Role" est la responsable de la mise à jour des 2 cotes de l'association

Plan du cours

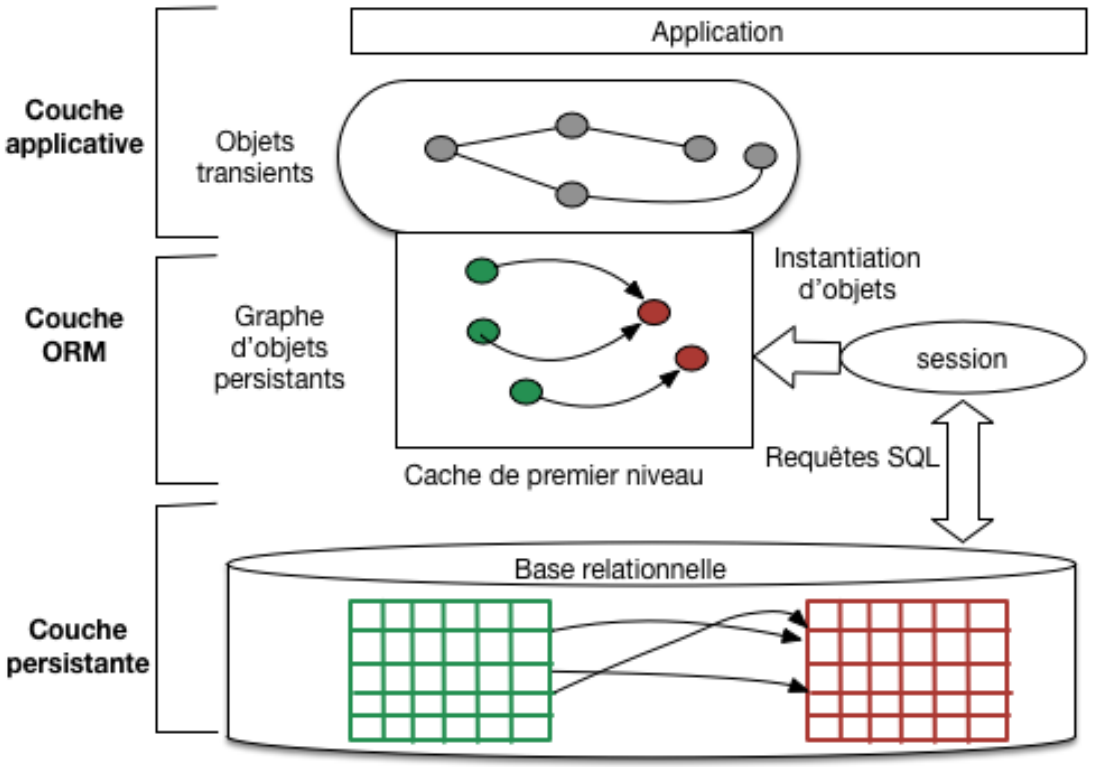
1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

Accès à une base de données via un ORM

- Hibernate s'appuie sur le mapping ORM défini précédemment pour accéder aux données
- Regardons maintenant comment sont gérés ces accès
- Pour l'instant nous n'allons considérer que les lectures de données
- Cela exclut donc la question des transactions

Architecture

La structuration en couches d'une application s'appuyant sur un framework ORM



- *Objets transients* : des objets standard, instanciés par l'opérateur *new*, dont le cycle de vie est géré par le garbage collector;
- *Persistants* : des instances d'une classe persistante (a l'annotation @Entity), représentant des lignes d'une table de la base
- les objets persistants sont placés dans le cache de premier niveau d'Hibernate
- cet espace est contrôlé et surveillé par un objet `session` d'Hibernate
- à chaque accès à la base ramenant un objet, Hibernate vérifie, dans le cache de premier niveau, si la ligne correspondante n'a pas déjà été instanciée, et si oui renvoie l'objet déjà présent dans le cache

Les opérations de lecture

- par navigation dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;

```
return film.getRealisateur().getNom();
```

Les opérations de lecture

- par navigation dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;

```
return film.getRealisateur().getNom();
```

- par identifiant: méthode basique, utilisée implicitement par d'autres méthodes comme la navigation;

```
return (Film) session.get(Film.class, id);
```

=> l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes

Les opérations de lecture

- par navigation dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;

```
return film.getRealisateur().getNom();
```

- par identifiant: méthode basique, utilisée implicitement par d'autres méthodes comme la navigation;

```
return (Film) session.get(Film.class, id);
```

=> l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes

Les opérations de lecture

- par navigation dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;

```
return film.getRealisateur().getNom();
```

- par identifiant: méthode basique, utilisée implicitement par d'autres méthodes comme la navigation;

```
return (Film) session.get(Film.class, id);
```

=> l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes

- directement par SQL: pas une méthode portable et devrait être évité
  =>un des objectifs d'Hibernate est de s'éloigner de la base de données. Or, l'utilisation du SQL natif va à l'encontre de cet objectif

Les opérations de lecture

- par navigation dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;

```
return film.getRealisateur().getNom();
```

- par identifiant: méthode basique, utilisée implicitement par d'autres méthodes comme la navigation;

```
return (Film) session.get(Film.class, id);
```

=> l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes

- directement par SQL: pas une méthode portable et devrait être évité =>un des objectifs d'Hibernate est de s'éloigner de la base de données. Or, l'utilisation du SQL natif va à l'encontre de cet objectif
- par le langage de requêtes HQL;

Les opérations de lecture

- par navigation dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;

```
return film.getRealisateur().getNom();
```

- par identifiant: méthode basique, utilisée implicitement par d'autres méthodes comme la navigation;

```
return (Film) session.get(Film.class, id);
```

=> l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes

- directement par SQL: pas une méthode portable et devrait être évité =>un des objectifs d'Hibernate est de s'éloigner de la base de données. Or, l'utilisation du SQL natif va à l'encontre de cet objectif
- par le langage de requêtes HQL;
- par l'API Criteria, qui permet de construire par programmation objet une requête à exécuter;

Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

Le language HQL

- *Hibernate Query Language*: un langage de requêtes objet qui sert à interroger le **graphe des objets Java** constituant la vue orientée-objet de la base
  Exemple : recherche de films par titre en HQL:

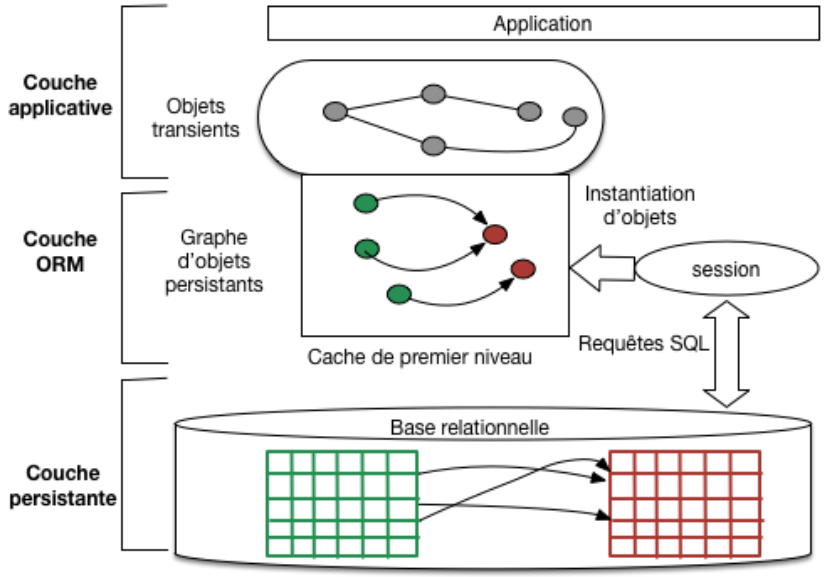
```
public List<Film> parTitre(String titre){
        Query q = session.createQuery("from Film f where f.titre=:titre");
        q.setString ("titre", titre);
        return q.list();        }
```

Le language HQL

- *Hibernate Query Language*: un langage de requêtes objet qui sert à interroger le **graphe des objets Java** constituant la vue orientée-objet de la base
  Exemple : recherche de films par titre en HQL:

```
public List<Film> parTitre(String titre){
        Query q = session.createQuery("from Film f where f.titre=:titre");
        q.setString ("titre", titre);
        return q.list();          }
```

- Hibernate se charge d'effectuer les requêtes SQL pour matérialiser la partie du graphe qui satisfait la requête



Le language HQL

- *Hibernate Query Language*: un langage de requêtes objet qui sert à interroger le **graphe des objets Java** constituant la vue orientée-objet de la base
  Exemple : recherche de films par titre en HQL:

```
public List<Film> parTitre(String titre){
        Query q = session.createQuery("from Film f where f.titre=:titre");
        q.setString ("titre", titre);
        return q.list();          }
```

- Hibernate se charge d'effectuer les requêtes SQL pour matérialiser la partie du graphe qui satisfait la requête
- L'intégration de HQL à Java se fait par des méthodes de la `Session`

Le language HQL

- *Hibernate Query Language*: un langage de requêtes objet qui sert à interroger le **graphe des objets Java** constituant la vue orientée-objet de la base
  Exemple : recherche de films par titre en HQL:

```
public List<Film> parTitre(String titre){
        Query q = session.createQuery("from Film f where f.titre=:titre");
        q.setString ("titre", titre);
        return q.list();          }
```

- Hibernate se charge d'effectuer les requêtes SQL pour matérialiser la partie du graphe qui satisfait la requête
- L'intégration de HQL à Java se fait par des méthodes de la `Session`
- La clause `select` est optionnelle en HQL: on interroge des objets (instances des classes *mappées*), et la projection sur certains attributs offre peu d'intérêt

Le language HQL

- HQL est un langage *objet*, même s'il ressemble beaucoup à SQL
- Il permet de naviguer dans le graphe par exemple avec la clause where. Voici un exemple :

```
Query q = session.createQuery("from Film f
                                where f.realisateur.nom='Eastwood'"
```

Le language HQL

Chaînage des méthodes :

```
session.createQuery("from Film as film
                where film.titre like :titre and film.annee < :annee")
        .setString ("titre", "er")
        .setInteger("annee", 2000)
        .list();
```

Le language HQL

Chaînage des méthodes :

```
session.createQuery("from Film as film
                where film.titre like :titre and film.annee < :annee")
        .setString ("titre", "er")
        .setInteger("annee", 2000)
        .list();
```

Opérateurs et clauses similaires à SQL : =, <, >, between, in, not between, like, not like, and, or, not, order by, is null

Le language HQL

On peut également utiliser comme paramètre un objet persistant :

```
// bergman est une instance de Artiste
Artiste bergman = ...;

session.createQuery("from Film as film where film.realisateur=:mes")
  .setEntity ("mes",bergman)
␣␣.list();
```

Le language HQL

Limitation

L'utilisation d'une approche basée sur des chaînes de caractères n'entraîne pas la validation de la syntaxe des requêtes à la compilation

Exemple:

```
Query q = session.createQuery("fro␣Film␣f␣whee␣f.titre=:titre");
```

Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

L'API Criteria

- Criteria est une API de recherche orientée objet
- Elle propose un ensemble de classes et de méthodes pour construire des requêtes avec une syntaxe spécifique très proche de java

```
public List<Film> parTitreCriteria(String titre){
        Criteria criteria = session.createCriteria(Film.class);
        criteria.add (Restrictions.like("titre", titre));
        return criteria.list();
}
```

- On ajoute des *expressions* pour indiquer les restrictions de la recherche
- Le respect d'une approche tout-objet c'est l'argument principal pour cette API au lieu de HQL

L'API Criteria

Regrouper différents critères de manière logique :

```
Criteria criteria = session.createCriteria(Product.class)
        .add(Restrictions.between("weigth", 100,200) )
        .add(Restrictions.or(
                Restrictions.isNotNull("etiquette"),
                Restrictions.isNull("description")
                ) )
List results = criteria.list();
```

L'API Criteria

Trier les résultats d'une requête :

```
Criteria criteria = session.createCriteria(Od.class)
        .add(Restrictions.between("weigth", 200, 500))
        .addOrder(Order.asc("weigth"))
        .addOrder(Order.desc("nom"));
List results = criteria.list();
```

L'API Criteria

Méthodes

Méthode	Description
list	Retourne tous les résultats correspondant à la requête
get	Retourne exactement un résultat ; à utiliser quand l unique résultat est attendu.
listDistinct	Retourne une liste de résultats distincts (suppression des doublons)
count	Retourne le nombre de résultats.
scroll	Retourne une liste de résultats "scrollable"

HQL x Criteria

- Une requête construite avec Criteria peut être vérifiée à la compilation
- Mais, on peut aussi estimer qu'une requête HQL est plus concise et plus lisible

Criteria :

```
Criteria criteria = session.createCriteria(Film.class);
criteria.add (Restrictions.like("titre", titre));
return criteria.list();
```

HQL :

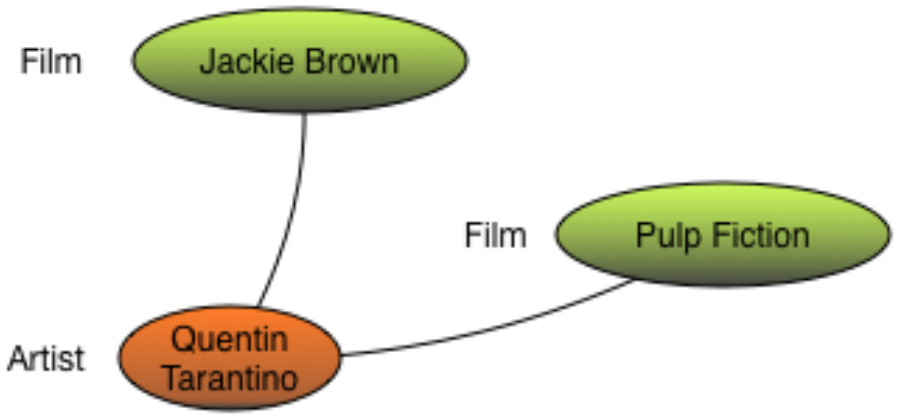
```
session.createQuery("from Film where titre like :titre")
        .setString ("titre", titre)
        .list();
```

Plan du cours

1. Introduction
2. La persistance avec JDBC
3. La persistence avec JPA
   - Mapping un à un
   - Mapping plusieurs à un
   - Mapping un à plusieurs
   - Mapping plusieurs à plusieurs
4. Accès à une base de données via un ORM
   - Le language HQL
   - L'API Criteria
   - Optimisation des lectures

Optimisation des lectures

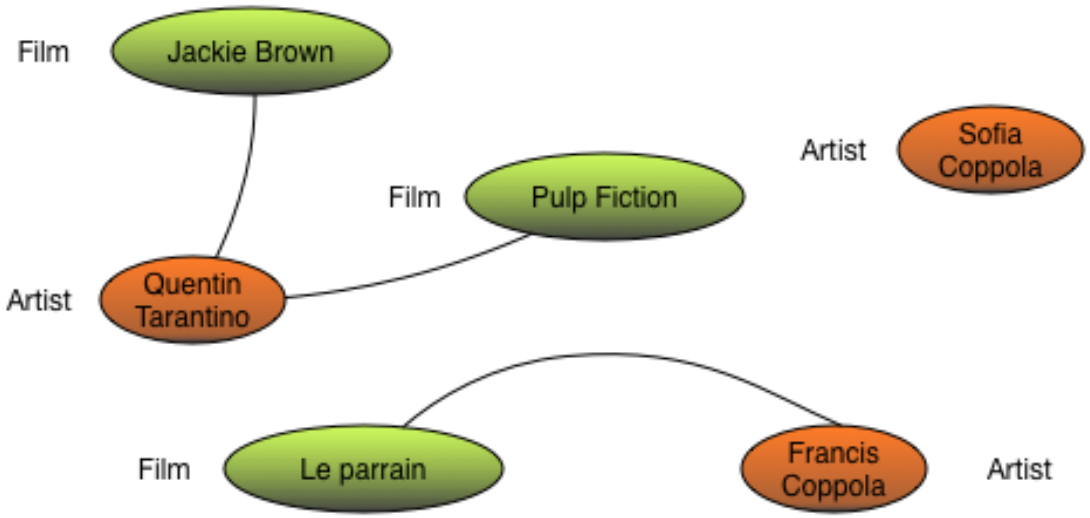
- On peut optimiser les requêtes SQL engendrées par Hibernate pour éviter de se retrouver dans la situation où de nombreuses requêtes sont exécutées pour charger, petit à petit, des parties minimes du graphe de données



- L'application sait souvent à l'avance à quelles données elle va accéder, et peut anticiper le chargement grâce à des requêtes SQL qui vont matérialiser la partie du graphe contenant ces données visitées
- Cela est défini selon la *stratégie de chargement*

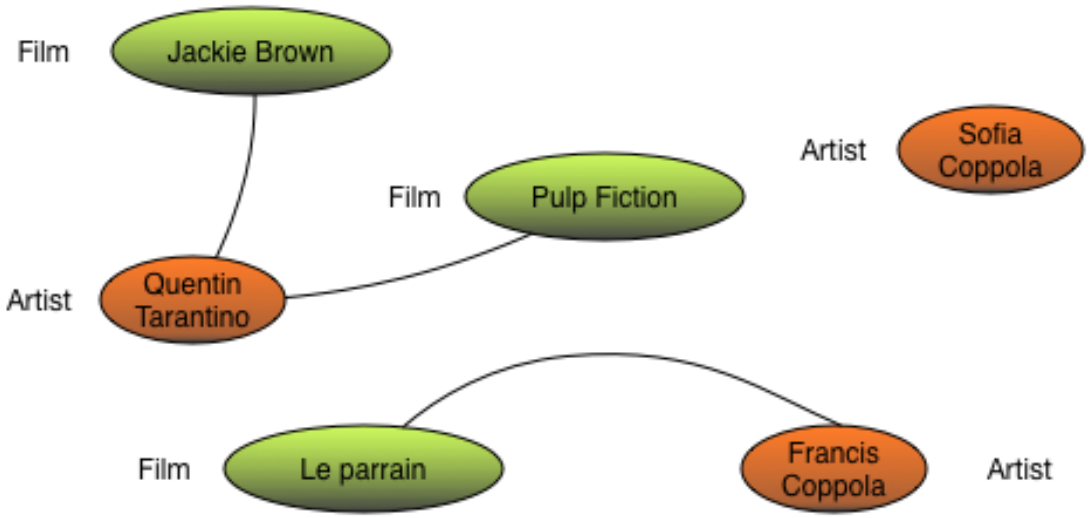
Stratégies de chargement

- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé



Stratégies de chargement

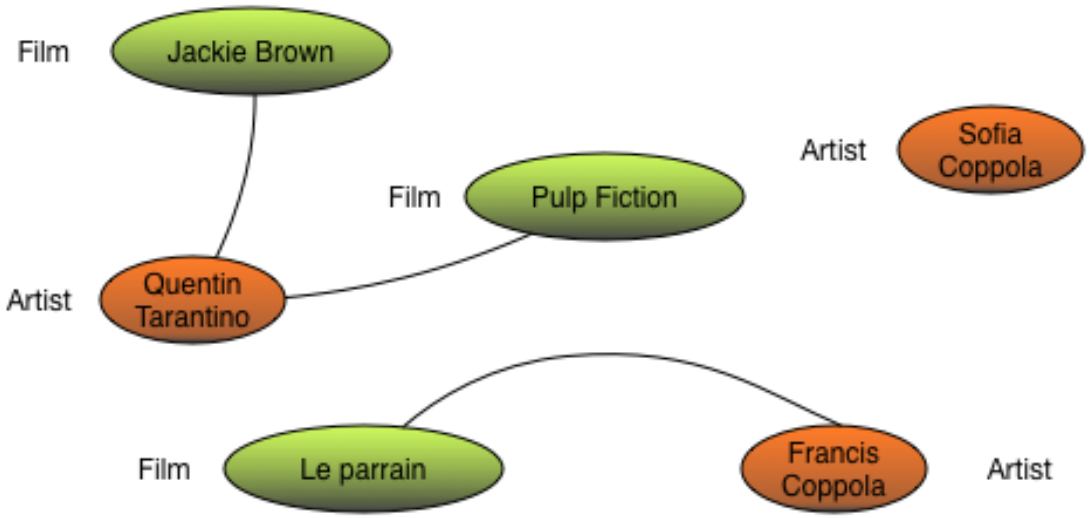
- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé



- 2 types de stratégies:

Stratégies de chargement

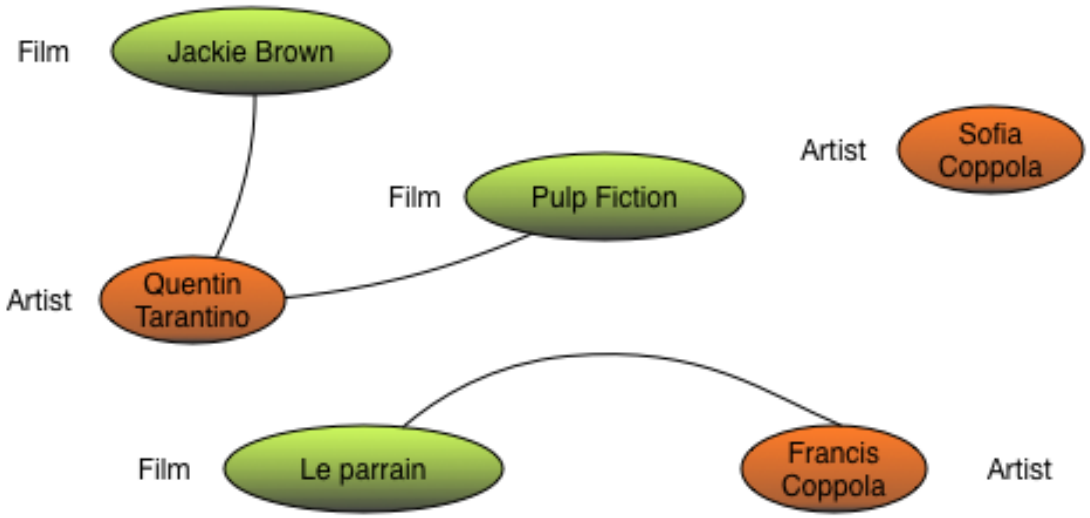
- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé



- 2 types de stratégies:
  - *eager* (ou glouton, le chargement immédiat)

Stratégies de chargement

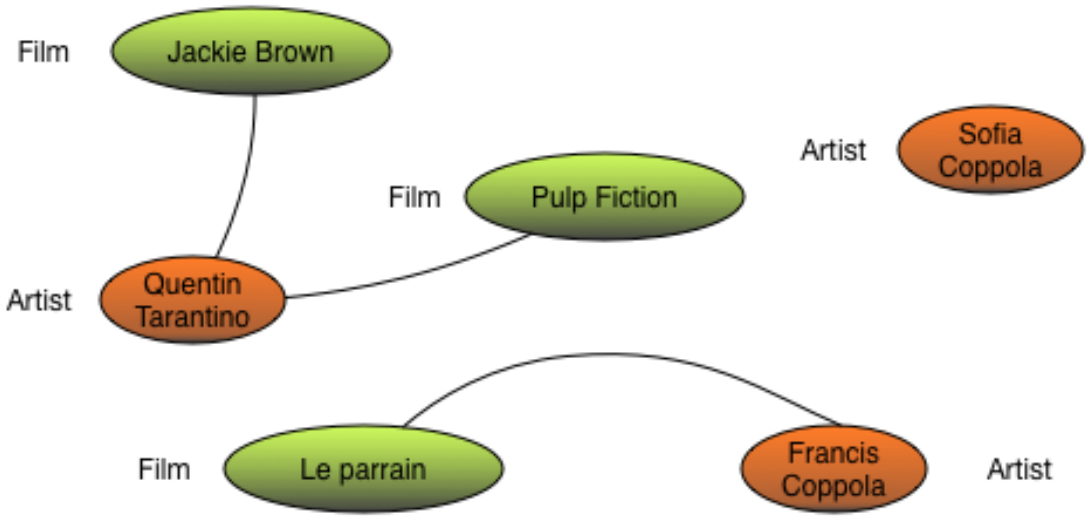
- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé



- 2 types de stratégies:
  - *eager* (ou glouton, le chargement immédiat)
  - *lazy* (ou paresseux)

Stratégies de chargement

- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé

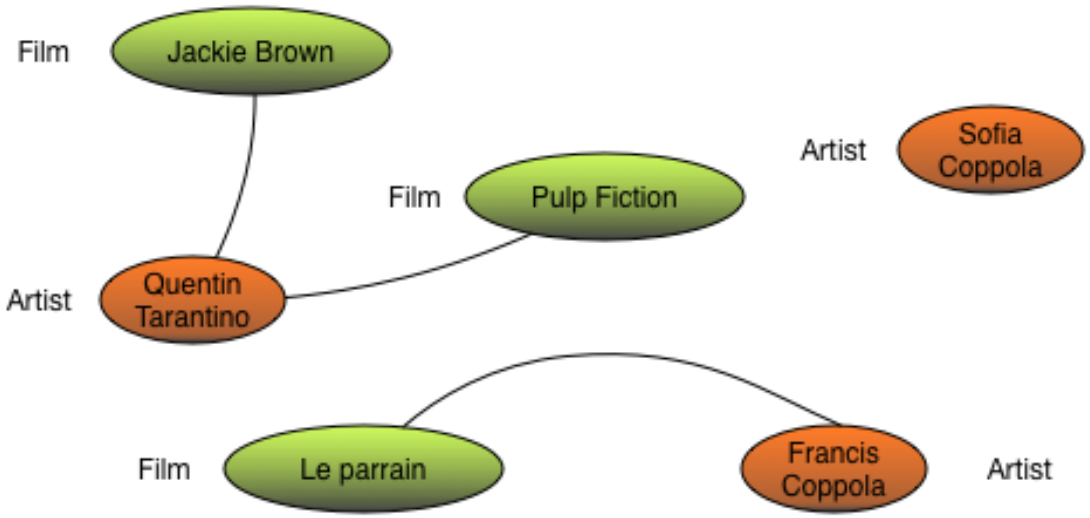


- 2 types de stratégies:
  - *eager* (ou glouton, le chargement immédiat)
  - *lazy* (ou paresseux)

  -> La stratégie défaut est différente en Hibernate et en JPA, et change aussi d'une version d'Hibernate à une autre. Mieux vaut donner la stratégie explicitement.

Stratégies de chargement

- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé

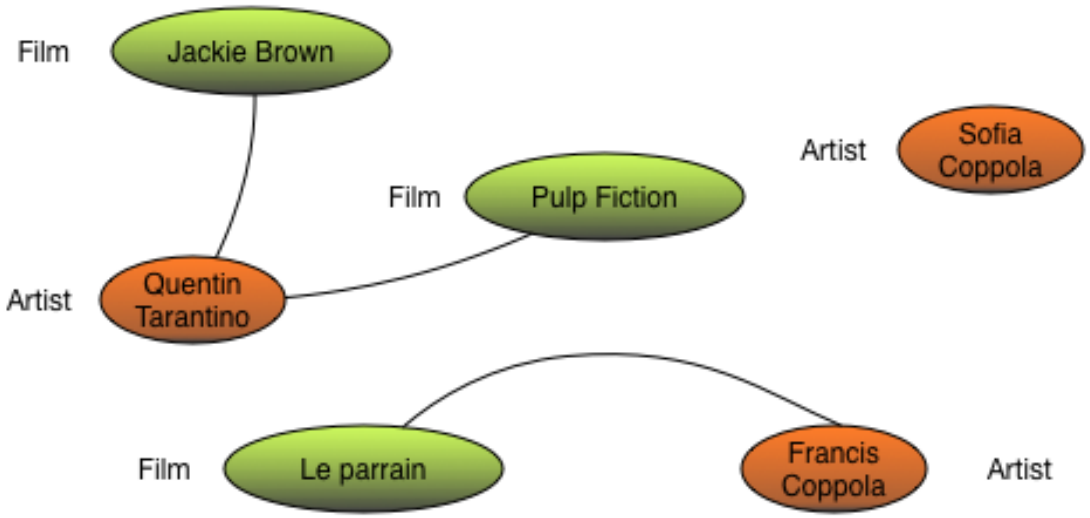


- 2 types de stratégies:
  - *eager* (ou glouton, le chargement immédiat)
  - *lazy* (ou paresseux)

  -> La stratégie défaut est différente en Hibernate et en JPA, et change aussi d'une version d'Hibernate à une autre. Mieux vaut donner la stratégie explicitement.
- La stratégie de chargement peut être paramétrée à deux niveaux :

Stratégies de chargement

- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé

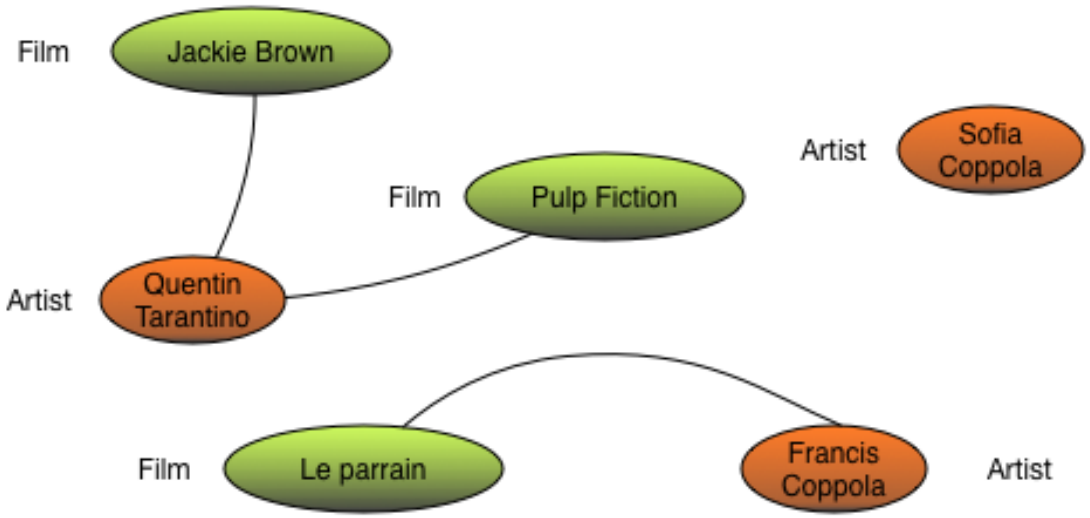


- 2 types de stratégies:
  - *eager* (ou glouton, le chargement immédiat)
  - *lazy* (ou paresseux)

  -> La stratégie défaut est différente en Hibernate et en JPA, et change aussi d'une version d'Hibernate à une autre. Mieux vaut donner la stratégie explicitement.
- La stratégie de chargement peut être paramétrée à deux niveaux :
  - dans la configuration du mapping, sous forme d'une annotation JPA;

Stratégies de chargement

- indique quel est le voisinage d'un objet persistant qui doit être matérialisé quand cet objet est lui même chargé



- 2 types de stratégies:
  - *eager* (ou glouton, le chargement immédiat)
  - *lazy* (ou paresseux)

  -> La stratégie défaut est différente en Hibernate et en JPA, et change aussi d'une version d'Hibernate à une autre. Mieux vaut donner la stratégie explicitement.
- La stratégie de chargement peut être paramétrée à deux niveaux :
  - dans la configuration du mapping, sous forme d'une annotation JPA;
  - à l'exécution de requêtes **HQL**, grâce au mot-clé *fetch*

Stratégies de chargement

Paramétrage de la stratégie de chargement :

- dans la configuration du mapping, sous forme d'une annotation JPA;
- à l'exécution de requêtes **HQL**, grâce au mot-clé *fetch*

Stratégies de chargement

Paramétrage de la stratégie de chargement :

- dans la configuration du mapping, sous forme d'une annotation JPA;
- à l'exécution de requêtes **HQL**, grâce au mot-clé *fetch*

Exemple :

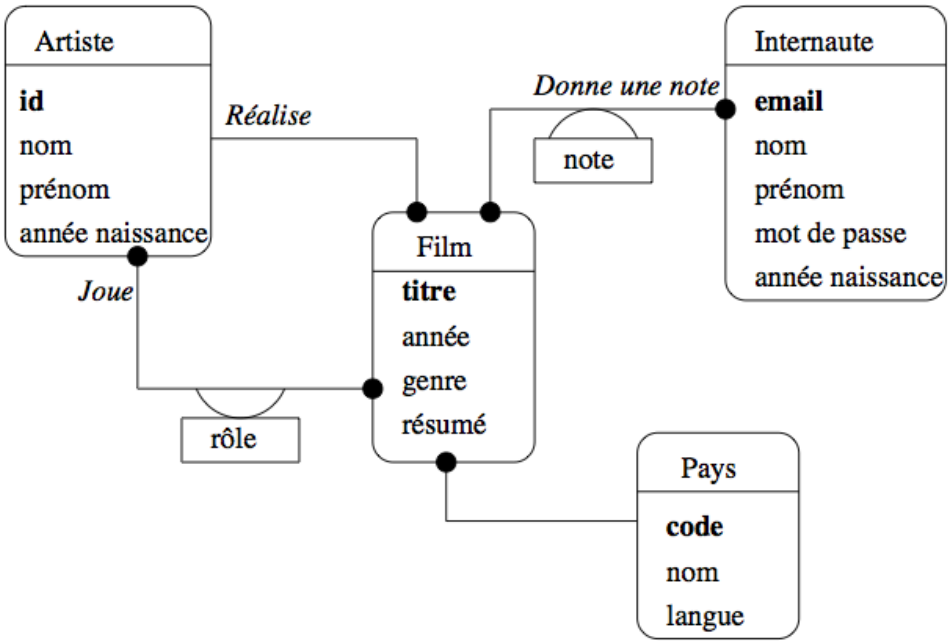
```
session.createQuery("from Film f where f.titre= :titre")
        .setString ("titre", titre)
        .list();
```

Action qui affiche pour chaque film son titre et le nom du réalisateur :

```
<c:forEach items="${films}" var="film">
        <li>Nous avons bien ramene le film ${film.titre} dont le
                realisateur est ${film.realisateur.nom}</li>
</c:forEach>
```

Stratégies de chargement

Type Eager



Paramétrée dans la configuration du mapping :

```
@ManyToOne(fetch=FetchType.EAGER)
```

Hibernate engendre au moins 4 requêtes SQL dès qu'un film est placé dans le cache :

```
select * from Film film0_ where film0_.id=
select * from Artiste artiste0_ where artiste0_.id=?
select * from Note genre0_ where note0_.id=? (inutile)
select * from Pays pays0_ where pays0_.id=?  (inutile)
```

Stratégies de chargement

Type Lazy

Paramétrée dans la configuration du mapping :

```
@ManyToOne(fetch=FetchType.LAZY)
```

Hibernate engendre 2 requêtes SQL :

```
select * from Film film0_ where film0_.id=?
select * from Artiste artiste0_ where artiste0_.id=?
```

- La seconde requête est déclenchée par *navigation*: quand on veut afficher le nom du réalisateur, Hibernate déclenche l'accès à l'artiste et son chargement. Cet accès est nécessaire car l'objet n'est pas présent dans le cache
- Mais: Une jointure avec la table "Artiste" quand on recherche le film serait plus appropriée

Stratégies de chargement

Paramétrage de la stratégie de chargement :

- dans la configuration du mapping, sous forme d'une annotation JPA;
- à l'exécution de requêtes **HQL**, grâce au mot-clé *fetch*

Stratégies de chargement

L'option fetch

- Si on sait à l'avance que l'on va traiter un film, son metteur en scène et ses acteurs, on peut décider de charger par une seule requête tout ce sous-graphe

Stratégies de chargement

L'option fetch

- Si on sait à l'avance que l'on va traiter un film, son metteur en scène et ses acteurs, on peut décider de charger par une seule requête tout ce sous-graphe
- Paramétré à l'exécution de requêtes HQL : Permet d'adapter la stratégie de chargement au contexte particulier d'utilisation

Stratégies de chargement

L'option fetch

- Si on sait à l'avance que l'on va traiter un film, son metteur en scène et ses acteurs, on peut décider de charger par une seule requête tout ce sous-graphe
- Paramétré à l'exécution de requêtes HQL : Permet d'adapter la stratégie de chargement au contexte particulier d'utilisation
- Le mot-clé *fetch* dans la requête HQL entraine une *surcharge* de la stratégie de chargement par défaut

Stratégies de chargement

L'option fetch

- Si on sait à l'avance que l'on va traiter un film, son metteur en scène et ses acteurs, on peut décider de charger par une seule requête tout ce sous-graphe
- Paramétré à l'exécution de requêtes HQL : Permet d'adapter la stratégie de chargement au contexte particulier d'utilisation
- Le mot-clé *fetch* dans la requête HQL entraine une *surcharge* de la stratégie de chargement par défaut

Stratégies de chargement

L'option fetch

- Si on sait à l'avance que l'on va traiter un film, son metteur en scène et ses acteurs, on peut décider de charger par une seule requête tout ce sous-graphe
- Paramétré à l'exécution de requêtes HQL : Permet d'adapter la stratégie de chargement au contexte particulier d'utilisation
- Le mot-clé *fetch* dans la requête HQL entraine une *surcharge* de la stratégie de chargement par défaut

Exemple :

```
session.createQuery("from Film f join fetch f.realisateur")
```

La requête SQL devrait être devenue:

```
select [...]
from Film film0_
        left outer join Artiste artiste3_
        on film0_.id_realisateur=artiste3_.id
where film0_.id=?
```

Quand on navigue du film vers son réalisateur, l'objet Artiste est trouvé dans le cache et il n'est plus nécessaire d'effectuer une requête SQL supplémentaire.

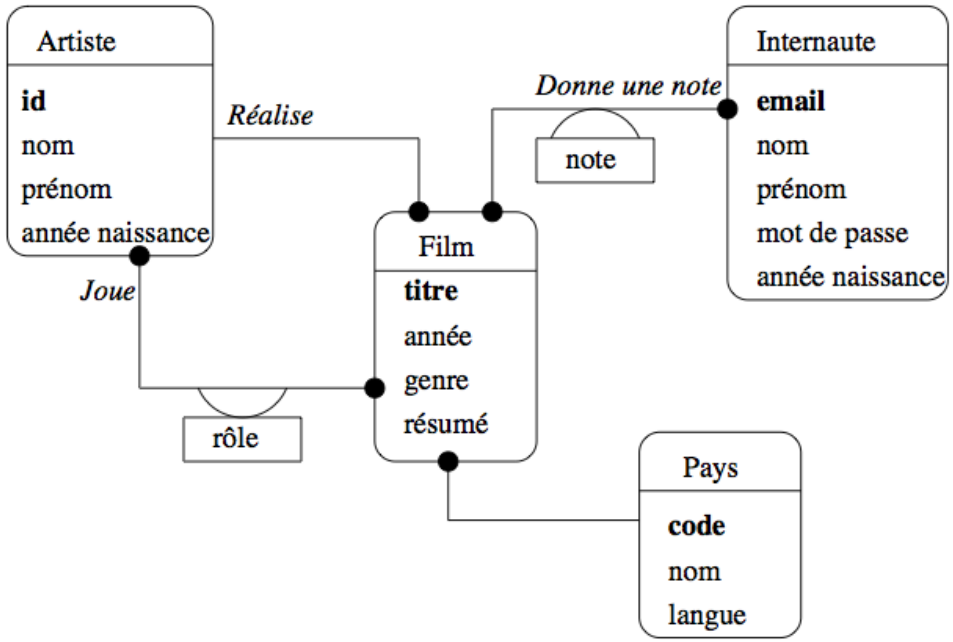
Stratégies de chargement

Paramétrage de la stratégie de chargement :

- dans la configuration du mapping, sous forme d'une annotation JPA;
- à l'exécution de requêtes **HQL**, grâce au mot-clé *fetch*

Stratégies de chargement

- Ces deux niveaux sont complémentaires
- La spécification au niveau de la configuration a l'inconvénient d'être insensible aux différents contextes dans lesquels la base de données est interrogée (ex: charger un film avec les acteurs, avec les notes, tout seul)
- La bonne méthode consiste à indiquer une stratégie de chargement par défaut au niveau de la configuration, et à la *surcharger* grâce à des requêtes HQL spécifiques selon le contexte



Le problème des 1+n requêtes

En mode Lazy

Supposions que l'on veut parcourir tous les pays de la base, et pour chacun analyser les films tournés. La structure de l'action serait la suivante:

Le problème des 1+n requêtes

En mode Lazy

Supposions que l'on veut parcourir tous les pays de la base, et pour chacun analyser les films tournés. La structure de l'action serait la suivante:

```
Set<Pays> pays = session.createQuery ("from␣Pays");
for (Pays pays1: pays) {
        for (Film film: pays1.getFilm()) {
                // Faire qq chose avec le film
        }
}
```

Le problème des 1+n requêtes

En mode Lazy

Supposions que l'on veut parcourir tous les pays de la base, et pour chacun analyser les films tournés. La structure de l'action serait la suivante:

```
Set<Pays> pays = session.createQuery ("from␣Pays");
for (Pays pays1: pays) {
        for (Film film: pays1.getFilm()) {
                // Faire qq chose avec le film
        }
}
```

Hibernate engendre 2 requêtes SQL :

```
(1) select * from Pays
(2) select * from Film where id_pays=:pays.id
```

Le problème des 1+n requêtes

En mode Lazy

Supposions que l'on veut parcourir tous les pays de la base, et pour chacun analyser les films tournés. La structure de l'action serait la suivante:

```
Set<Pays> pays = session.createQuery ("from␣Pays");
for (Pays pays1: pays) {
        for (Film film: pays1.getFilm()) {
                // Faire qq chose avec le film
        }
}
```

Hibernate engendre 2 requêtes SQL :

```
(1) select * from Pays
(2) select * from Film where id_pays=:pays.id
```

- La première est exécutée une fois, la seconde autant de fois qu'il y a de pays, d'où la caractérisation par l'expression *'1+n' requêtes*

Le problème des 1+n requêtes

En mode Lazy

Supposions que l'on veut parcourir tous les pays de la base, et pour chacun analyser les films tournés. La structure de l'action serait la suivante:

```
Set<Pays> pays = session.createQuery ("from␣Pays");
for (Pays pays1: pays) {
        for (Film film: pays1.getFilm()) {
                // Faire qq chose avec le film
        }
}
```

Hibernate engendre 2 requêtes SQL :

```
(1) select * from Pays
(2) select * from Film where id_pays=:pays.id
```

- La première est exécutée une fois, la seconde autant de fois qu'il y a de pays, d'où la caractérisation par l'expression *'1+n' requêtes*
- Inefficace car il y aura potentiellement beaucoup de requêtes à la base de données (ex: s'il y a 10 000 pays, la 2ème requête sera exécutée 10 000 fois, alors que les SGBD sont conçus, grâce à l'opération de jointure, pour ramener tous les objets en une seule requête)

Le chargement par lot, une variante de "Lazy"

- *Batch fetching*: permet de factoriser les requêtes effectuées pour obtenir les collections associées aux objets de la boucle extérieure

Le chargement par lot, une variante de "Lazy"

- *Batch fetching*: permet de factoriser les requêtes effectuées pour obtenir les collections associées aux objets de la boucle extérieure
- Il est caractérisé par la taille d'un lot, k, exprimant le nombre de collections recherchées en une seule fois

Le chargement par lot, une variante de "Lazy"

- *Batch fetching*: permet de factoriser les requêtes effectuées pour obtenir les collections associées aux objets de la boucle extérieure
- Il est caractérisé par la taille d'un lot, k, exprimant le nombre de collections recherchées en une seule fois

Le chargement par lot, une variante de "Lazy"

- *Batch fetching*: permet de factoriser les requêtes effectuées pour obtenir les collections associées aux objets de la boucle extérieure
- Il est caractérisé par la taille d'un lot, k, exprimant le nombre de collections recherchées en une seule fois

Configuration :

```
@Entity
public class Pays {
        ...
        @OneToMany
        @BatchSize(size=10)
        @JoinColumn(name="id\_pays")
        private Set<Film> films = new HashSet<Film>();
        ...
}
```

on veut parcourir tous les pays de la base, et pour chacun analyser les films tournés

Le chargement par lot, une variante de "Lazy"

=> on veut parcourir tous les pays de la base, et pour chacun, les films tournés

- Avec un chargement par lot de taille 10, on va grouper la recherche des collections Pays (et ses films) 10 par 10.

Le chargement par lot, une variante de "Lazy"

=> on veut parcourir tous les pays de la base, et pour chacun, les films tournés

- Avec un chargement par lot de taille 10, on va grouper la recherche des collections Pays (et ses films) 10 par 10.
- Supposons pour simplifier que les identifiants des pays sont séquentiels: 1, 2, 3, ..., etc. Quand l'application cherche à accéder à la collection "films" du premier pays, la requête suivante sera effectuée:

Le chargement par lot, une variante de "Lazy"

=> on veut parcourir tous les pays de la base, et pour chacun, les films tournés

- Avec un chargement par lot de taille 10, on va grouper la recherche des collections Pays (et ses films) 10 par 10.
- Supposons pour simplifier que les identifiants des pays sont séquentiels: 1, 2, 3, ..., etc. Quand l'application cherche à accéder à la collection "films" du premier pays, la requête suivante sera effectuée:

Le chargement par lot, une variante de "Lazy"

=> on veut parcourir tous les pays de la base, et pour chacun, les films tournés

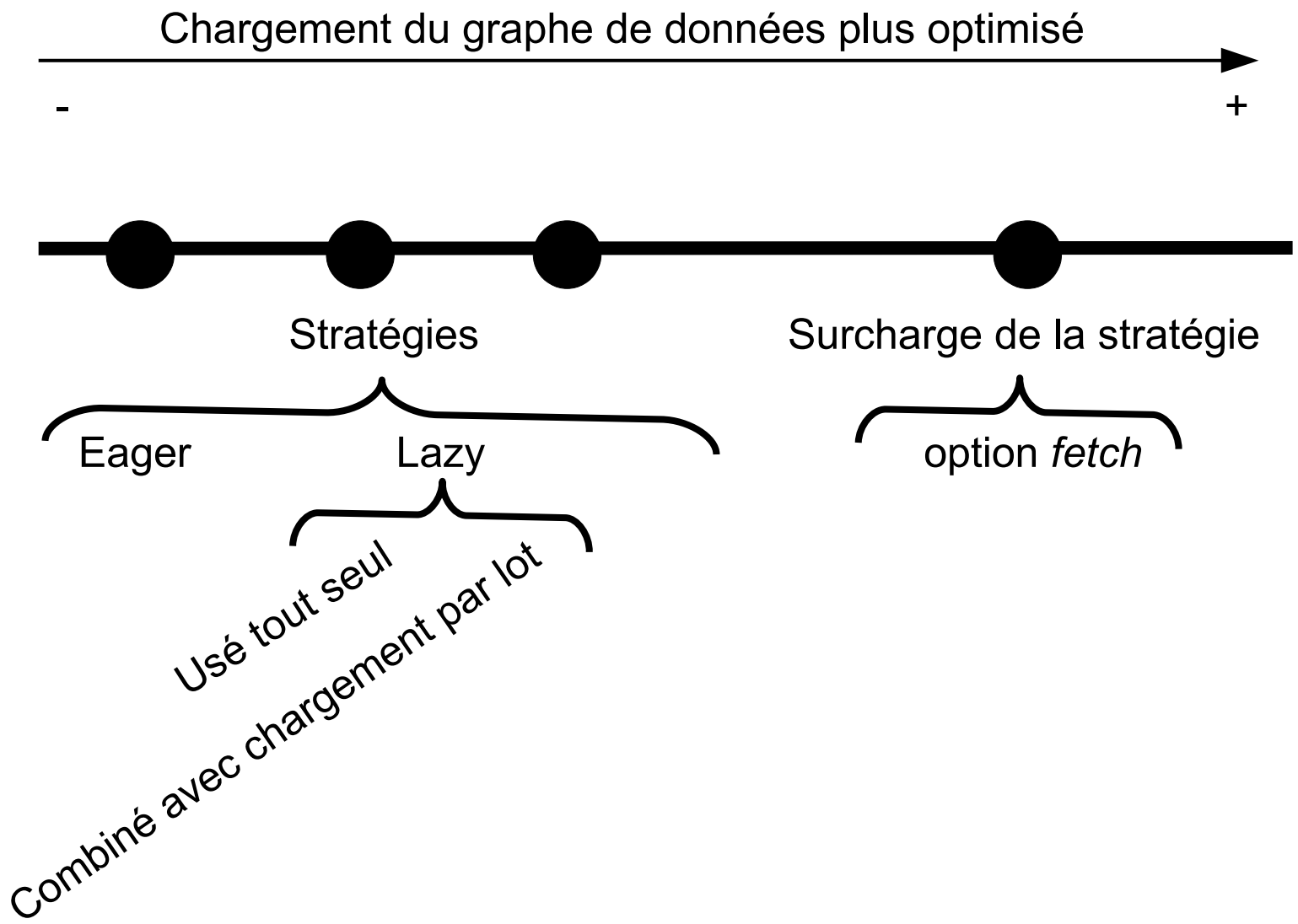
- Avec un chargement par lot de taille 10, on va grouper la recherche des collections Pays (et ses films) 10 par 10.
- Supposons pour simplifier que les identifiants des pays sont séquentiels: 1, 2, 3, ..., etc. Quand l'application cherche à accéder à la collection "films" du premier pays, la requête suivante sera effectuée:

```
select * from Film where id_pays in (1, 2, 3, 4, 5, 6, 7, 8 , 9, 10)
```

Avec ce paramétrage, on est passé du problème des 1+n requêtes au problème des 1+n/10 requêtes !

Stratégies de chargement

Résumé des stratégies



Stratégies de chargement

Recap

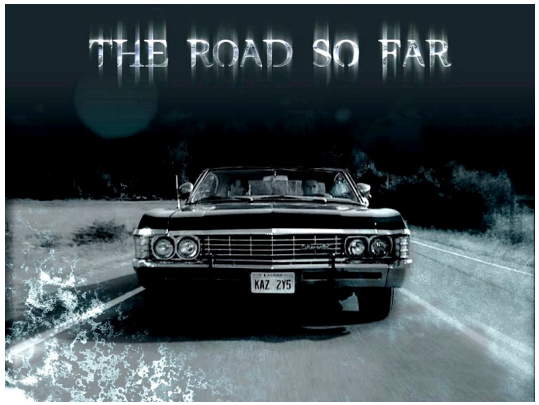
- La question du chargement des objets par des requêtes SQL est importante (performance, mémoire, ...)
- Hibernate (ou JPA) applique une stratégie par défaut qui tend à multiplier les requêtes SQL ou les jointures externes inutilement
- Cette stratégie par défaut peut être remplacée par une configuration explicite dans les annotations de *mapping*
- Une stratégie au niveau de la configuration ne peut être optimale dans tous les cas: on peut alors, dans un contexte donnée, la remplacer par un chargement adapté au contexte, exprimé en HQL avec l'option *fetch*

Sources

- `http://orm.bdpedia.fr/intro.html`
- `http://www.google.com`

Recap Général du Module

- MVC (Modèle-Vue-Contrôleur) : un motif d'architecture logicielle
- Java EE (Java Enterprise Edition) : Une plateforme pour le développement d'applications d'entreprise; un ensemble de spécifications



- Technologies Java EE pour implémenter une application web selon MVC:
  - Servlets [Contrôleur]
  - JSP (Java Server Page) [Vues]
  - JPA (Java Persistence API) / Hibernate [Modèle]
- Un framework de développement est un ensemble de ressources permettant de développer tout ou partie d'une application déployable dans un conteneur Java EE (Ex: Grails, SEAM, Spring)