

# Java Persistence API

Master TIIL & ILIADE 1<sup>ère</sup> année

Eric Cariou

Université de Bretagne Occidentale  
UFR Sciences et Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

Octobre 2016

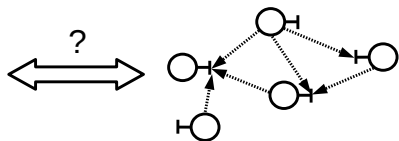
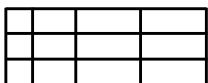
1

## Limites de JDBC

3

## Problématique

- ◆ Java avec stockage BDD et accès via JDBC
  - ◆ Au delà de l'accès à faire à distance
    - ◆ Qui complexifie forcément les choses mais ne peut pas y couper
  - ◆ Représentation des données très différentes
    - ◆ Coté BDD
      - ◆ Structure en table avec langage de requête dédié
      - ◆ Récupère toujours une « sous-table » via le résultat de l'exécution d'une requête de type SELECT
    - ◆ Coté Java
      - ◆ Structure des données définies par les classes que l'on instancie
      - ◆ Les objets sont associés entre eux et forment un graphe d'objet dans l'application



5

## Java Persistence API

- ◆ JPA : Java Persistence API
- ◆ Framework de persistance en Java
  - ◆ ORM : Object-Relationnal Mapping
  - ◆ API définie sur la base d'Hibernate, ORM de référence dans le monde Java
    - ◆ Peut fonctionner avec d'autres moteurs de persistance
- ◆ Principes
  - ◆ On définit la correspondance entre la structure des classes objet et le schéma relationnel de la BDD
  - ◆ On manipule directement des objets dans le code
  - ◆ Le framework fait la lecture/enregistrement du contenu des objets sur le support physique
    - ◆ Plus besoin de coder des requêtes SQL
    - ◆ Même si un langage de requête orienté objet reste utile

2

## Problématique

- ◆ Java avec stockage des données dans une BDD relationnelle
  - ◆ Nécessite des requêtes SQL
    - ◆ Utilisation d'un framework dédié : JDBC
  - ◆ Sort du format « standard » de représentation Java des données (classes et objets)
    - ◆ Une requête de type SELECT retourne un ResultSet
      - ◆ C'est à dire un ensemble de ligne de plusieurs colonnes
    - ◆ On accède aux éléments du ResultSet en naviguant selon les lignes et les colonnes, case par case
    - ◆ Peu pratique mais difficile de faire autrement vu ce que retourne de manière native les requêtes SQL
      - ◆ Sauf à passer par des SGBD objet-relationnel

4

## Retour sur l'exemple du cours JDBC

- ◆ Gestion de sports et de disciplines
  - ◆ Un sport se compose de plusieurs disciplines
  - ◆ Deux tables définies dans un schéma nommé « sports »
    - ◆ sport (*code\_sport*, intitule)
    - ◆ discipline (*code\_discipline*, intitule, code\_sport)
- ◆ Première version coté Java
  - ◆ Deux classes et correspondance totale avec la structure des tables

```
public class Sport {
    private int codeSport;
    private String intitule;
    // getter, setters
    // constructeurs
}

public class Discipline {
    private int codeDiscipline;
    private String intitule;
    private int codeSport;
    // getter, setters
    // constructeurs
}
```

6

## Retour sur l'exemple du cours JDBC

- ◆ Première version des classes pas satisfaisante
- ◆ Attributs codeSport dans la classe Sport et codeDiscipline dans la classe Discipline
  - ◆ Clés primaires coté BDD : données techniques ici, pas métier
  - ◆ Mais on va les garder quand même
    - ◆ Utile pour identifier les objets et faire le lien avec la BDD
- ◆ Classe Discipline, attribut codeSport
  - ◆ Permet de connaître le sport auquel se rattache la discipline
  - ◆ Pourquoi un entier comme en BDD ?
    - ◆ Si on veut récupérer l'instance de la classe Sport associée, on devra parcourir tous les objets de type Sport pour trouver celui qui a le bon codeSport ou faire une requête SQL sur la base
    - ◆ Autant mettre une référence vers un objet de type Sport directement
- ◆ Classe Sport
  - ◆ Conceptuellement, un sport se compose de plusieurs disciplines même si cela n'est pas directement représenté techniquement 7
  - ◆ Peut rajouter dans la classe un attribut disciplines

## Retour sur l'exemple du cours JDBC

- ◆ Nouvelle version des classes
- ◆ En « pur objet/métier », sans se préoccuper du stockage en BDD
  - ◆ Sauf pour clés primaires qui serviront d'identifiant coté Java
- ◆ Les méthodes d'accès à la BDD prendront ces classes en paramètres ou type de retour

```

public class Sport {
    private int codeSport;
    private String intitule;
    private Set<Discipline> disciplines;

    // getter, setters
    // constructeurs
}

public class Discipline {
    private int codeDiscipline;
    private String intitule;
    private Sport sport;

    // getter, setters
    // constructeurs
}
    
```

9

## Nouvelle version ajout discipline

- ◆ Pour ajouter une discipline, il faut une instance de Sport
- ◆ Peut la récupérer dans la base à partir de son intitulé

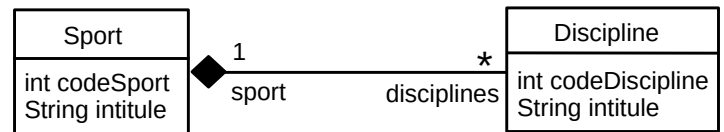
```

public Sport getSport(String intitule) throws SQLException {
    Statement req = conn.createStatement();
    ResultSet res = req.executeQuery("SELECT * FROM SPORT
                                     WHERE INTITULE = '"+intitule+"'");
    if (res.next()) {
        Sport sport = new Sport();
        sport.setCodeSport(res.getInt(1));
        sport.setIntitule(res.getString(2));
        // chargement des disciplines du sport
        sport.setDisciplines(this.getDisciplinesSport(sport));
        return sport;
    }
    // sport non trouvé
    else throw new SQLException("Sport "+intitule+"non trouve");
}
    
```

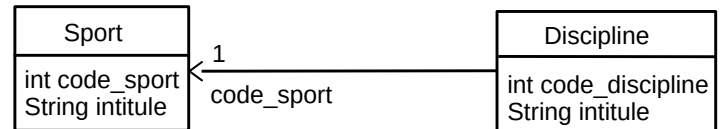
11

## Retour sur l'exemple du cours JDBC

- ◆ Coté objet/Java on veut implémenter cette modélisation métier



- ◆ Coté BDD, la base est implémentée selon cette logique technique



8

## Nouvelle version ajout discipline

```

public Discipline ajouterDiscipline(String intitule, Sport sport)
    throws SQLException {
    // requête pour définir la clé primaire de la discipline : max + 1
    Statement req = conn.createStatement();
    ResultSet res = req.executeQuery("
        SELECT MAX(CODE_DISCIPLINE) FROM DISCIPLINE");
    res.next();
    int codeDisc = res.getInt(1);
    codeDisc++;

    // insertion des données dans la table discipline
    req = conn.createStatement();
    int nb = req.executeUpdate("INSERT INTO DISCIPLINE VALUES ('
        + codeDisc + ', '" + intitule + "', " + sport.getCodeSport() + "')");
    System.out.println(" Discipline insérée : " + nb);

    // création de l'objet discipline et ajout dans les disciplines du sport
    Discipline disc = new Discipline(codeDisc, intitule, sport);
    sport.addDiscipline(disc);
    return disc;
}
    
```

10

## Nouvelle version ajout discipline

- ◆ Méthode qui retourne toutes les disciplines d'un sport

```

public Set<Discipline> getDisciplinesSport(Sport sport) throws SQLException {
    Statement req = conn.createStatement();
    ResultSet res = req.executeQuery("SELECT * FROM DISCIPLINE
                                     WHERE CODE_SPORT="+sport.getCodeSport());
    HashSet<Discipline> disciplines = new HashSet<Discipline>();
    Discipline disc;
    while (res.next()) {
        disc = new Discipline(res.getInt(1), res.getString(2), sport);
        disciplines.add(disc);
    }
    return disciplines;
}
    
```

- ◆ Exemple d'ajout d'une discipline

```

AccesSportsJDBC acces = new AccesSportsJDBC();
acces.connexionSGBD();
    
```

```

Sport sport = acces.getSport("ski");
Discipline disc = acces.ajouterDiscipline("biathlon", sport);
    
```

12

## Analyse du code

- ◆ Analyse du code présenté
- ◆ Marche très bien
  - ◆ Dans le sens où il fait ce qui doit être fait : ajouter une discipline à un sport dans la BDD via l'utilisation d'objets
  - ◆ Mais potentiels problèmes ou points faibles
- ◆ Ex : quand on récupère un sport dans la BDD, on charge et instancie toutes ses disciplines
  - ◆ Peut être très lourd (mémoire JVM, accès réseau, surcharge SGBD ...) pour des données pas forcément (encore) utiles
  - ◆ Avec des classes/tables avec beaucoup de références entre elles
    - ◆ Charger pour une requête une grosse partie du contenu de la base
  - ◆ Solutions
    - ◆ Ne pas mettre l'attribut Set<Discipline> disciplines dans la classe Sport mais c'est dommage ...
    - ◆ Le garder mais modifier getDisciplines() pour charger les disciplines à la demande et non pas automatiquement

13

## De JDBC à JPA

- ◆ Avec un framework ORM comme JPA
  - ◆ Définition des correspondances classes / tables
    - ◆ Avec des classes Java classiques (POJO : Plain Old Java Object)
  - ◆ Gère ensuite tout seul la cohérence objet / contenu base
- ◆ JDBC
  - ◆ Très simple d'exécuter des requêtes SQL
  - ◆ Mais ensuite le programmeur doit tout gérer à la main
- ◆ JPA
  - ◆ Framework assez complexe mais très puissant
  - ◆ Qui supprime une très grande partie du code nécessaire avec JDBC

15

## Mappings entités Java – tables SQL

17

## Analyse du code

- ◆ Problèmes de cohérence entre objets et contenu de la BDD
  - ◆ Il se passe quoi si on exécute ce code :

```
Sport peche = new Sport(4, "pêche");
acces.ajouterDiscipline("mouche", peche);
```

alors qu'il n'y a pas de sport « pêche » avec une clé primaire de 4 dans la base ?
    - ◆ Ça plante parce que la clé étrangère vers la table sport n'existe pas
  - ◆ Finalement je préfère écrire « biathlon » avec une majuscule :

```
disc.setIntitule("Biathlon");
```

    - ◆ La mise à jour sur la base n'est pas faite avec un setter basique
- ◆ Le développeur doit gérer lui-même la cohérence entre le contenu des objets et la BDD
  - ◆ Code relativement complexe
  - ◆ Peut s'appuyer sur un patron DAO (Data Access Object)
    - ◆ Cf TP

14

## Fonctionnement général de JPA

- ◆ Définition de correspondances entre classes et tables
  - ◆ On ne manipulera que des objets métier coté Java
- ◆ API de JPA offre des fonctionnalités pour
  - ◆ Récupérer des objets à partir de leur contenu en BDD
    - ◆ Usage au besoin d'un langage de requête ressemblant à SQL mais se basant sur la structure des classes
  - ◆ Rendre des objets créés persistants
    - ◆ Leur contenu est inséré dans la BDD
  - ◆ Faire toute opération de création / modification / suppression sur des objets avec persistance en BDD
    - ◆ Se fait en mode transactionnel
- ◆ JPA est une API qui nécessite un moteur l'implémentant
  - ◆ Hibernate, EclipseLink ...
  - ◆ On évitera d'utiliser des fonctionnalités propriétaires des moteurs si on veut du code portable

16

## Principes de définition des mappings

- ◆ Indépendance de définition des structures de données
  - ◆ On modélise d'un coté le métier et de l'autre la BDD en s'attachant à faire les meilleures implémentations / conceptions selon le domaine
    - ◆ Coté métier : structure métier et facilité de manipulation des données
    - ◆ Coté BDD : optimiser la gestion des données et performance d'accès
  - ◆ On définira ensuite les mappings requis entre les classes et tables
- ◆ Une classe Java est mappée vers une table dite primaire
  - ◆ La plupart des attributs de la classe ont une correspondance avec les colonnes de la table
  - ◆ Pourra utiliser d'autres tables dites secondaires pour la correspondance d'autres attributs
    - ◆ Via des jointures réalisées automatiquement par JPA
  - ◆ Une classe pourra fusionner le contenu de deux (ou plus) tables
  - ◆ Pourra gérer tout type de jointures entre tables / associations entre classes

18

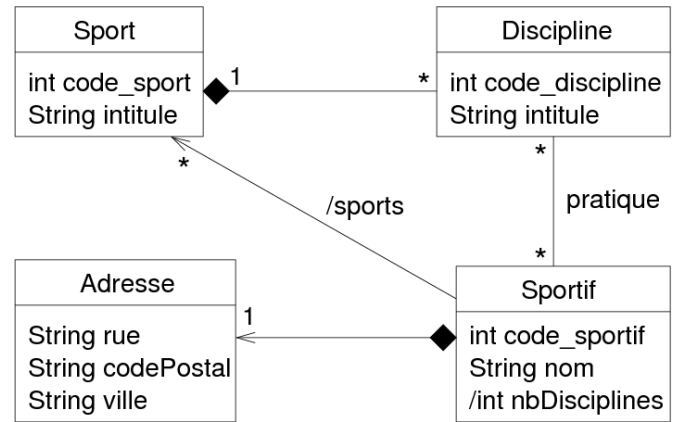
## Entité

- ◆ Classe Java qui a vocation à avoir le contenu de ses objets rendu persistant en BDD
- ◆ Classe standard : un POJO (Plain Old Java Object)
  - ◆ Définit des attributs de tout type
    - ◆ Et éventuellement des méthodes métier
  - ◆ Peu de contraintes à respecter (programmation classique)
    - ◆ Constructeurs dont un sans paramètre
    - ◆ Getters et setters pour les attributs
    - ◆ Un attribut jouera un rôle d'identifiant
    - ◆ Implémente `java.io.Serializable`
    - ◆ Rédéfinir `equals()` et `hashCode()` pour gestion des collections
      - ◆ En évitant d'utiliser l'identifiant dans le calcul du `hashCode`

19

## Exemple métier

- ◆ On reprend la modélisation métier de l'application de sports avec ajout d'une classe / entité sportif



20

## Exemple métier

- ◆ Classes Sport et Disciplines
  - ◆ Reprend les mêmes contenus que dans le cours JDBC
    - ◆ Un sport se compose de plusieurs disciplines
- ◆ Ajoute une entité Sportif
  - ◆ Un sportif pratique plusieurs disciplines et une discipline est pratiquée par plusieurs sportifs
    - ◆ Association bidirectionnelle \* <-> \* pratique entre les 2 classes
  - ◆ Un sportif a une adresse gérée comme un concept métier dédié
- ◆ Deux éléments dérivés
  - ◆ Attribut `nbDisciplines` dans Sportif
    - ◆ Le nombre de disciplines pratiquées par un sportif se détermine à partir de l'association pratique
  - ◆ Association unidirectionnelle dérivée `/sports` entre Sportif et Sport
    - ◆ Se détermine de l'association pratique en récupérant les sports des disciplines du sportif

21

## POJO de Sport

```
public class Sport implements java.io.Serializable {
    private short codeSport;
    private String intitule;
    private Set<Discipline> disciplines;

    public Sport() {
        this.disciplines = new HashSet<>();
    }

    public Sport(short codeSport) {
        this.codeSport = codeSport;
        this.disciplines = new HashSet<>();
    }

    public Sport(short codeSport, String intitule,
        Set disciplines) {
        this.codeSport = codeSport;
        this.intitule = intitule;
        this.disciplines = disciplines;
    }

    public short getCodeSport() {
        return this.codeSport;
    }

    public void setCodeSport(short codeSport) {
        this.codeSport = codeSport;
    }

    public String getIntitule() {
        return this.intitule;
    }

    public void setIntitule(String intitule) {
        this.intitule = intitule;
    }

    public Set getDisciplines() {
        return this.disciplines;
    }

    public void setDisciplines(Set disciplines) {
        this.disciplines = disciplines;
    }

    @Override
    public boolean equals(Object o) {
        ...
    }

    @Override
    public int hashCode() {
        ...
    }

    @Override
    public String toString() {
        ...
    }
}
```

22

## POJO de Sport

- ◆ Contenu de la classe
  - ◆ `codeSport` jouera le rôle d'identifiant
  - ◆ Un getter / setter par attribut + plusieurs constructeurs
  - ◆ Implémentation de `Serializable` et redéfinition de `equals()`, `hashCode()` et `toString()`
- ◆ Pour faciliter l'ajout et suppression de disciplines, peut rajouter 2 méthodes

```
public void addDiscipline(Discipline disc) {
    if (this.disciplines == null) disciplines = new HashSet<>();
    disciplines.add(disc);
}
```

```
public void removeDiscipline(Discipline disc) {
    if (this.disciplines != null) disciplines.remove(disc);
}
```

23

## Attributs des autres POJO

- ◆ Classe Sportif
  - `int codeSportif; // identifiant`
  - `String nom;`
  - `Adresse adresse;`
  - `Set<Disciplines> disciplines;`
  - ◆ Rajoutera deux pseudos getter (pas d'attribut dans la classe)
    - `int getNbDisciplines()`
    - `Set<Sport> getSports()`
    - ◆ On verra plus loin comment les implémenter
- ◆ Classe Adresse
  - `String rue, ville, codePostal;`
  - ◆ Pas d'identifiant dans cette classe, on verra pourquoi plus tard
- ◆ Classe Discipline
  - `int codeDiscipline; // identifiant`
  - `String intitule;`
  - `Sport sport;`
  - `Set<Sportif> sportifs;`
  - ◆ Dans Sportif et Discipline
    - ◆ Pourra aussi rajouter des méthodes d'ajout / suppression d'éléments dans les collections

24

## Tables de l'exemple

- ◆ Tables associées dans le SGBD
  - ◆ Conserve les tables sport et discipline déjà vues
    - ◆ sport (*code\_sport*, intitulé)
    - ◆ discipline (*code\_discipline*, intitulé, *code\_sport*)
      - ◆ *code\_sport* : clé étrangère venant de sport
  - ◆ Données sur un sportif
    - ◆ Une adresse est spécifique à un sportif : peut inclure les données de l'adresse avec les données du sportif
      - ◆ sportif(*code\_sportif*, nom, rue, *code\_postal*, ville)
  - ◆ Association entre disciplines et sportifs : nécessite une table d'association
    - ◆ pratique(*code\_sportif*, *code\_discipline*)
      - ◆ Clé primaire : combinaison de *code\_sportif* et *code\_discipline*
      - ◆ Clés étrangères : *code\_sportif* de sportif et *code\_discipline* de discipline

## Annotations

- ◆ Les mappings classes / tables sont définis par des annotations Java dans les classes des entités
  - ◆ Ou dans des fichiers XML mais moins pratique
- ◆ Annotations Java : méta-données
  - ◆ Données sur des données
  - ◆ S'appliquent sur des éléments du code
  - ◆ Précise qu'un élément joue un certain rôle, a certaines caractéristiques, informer de quelque chose ...
    - ◆ Une annotation peut contenir des paramètres (nommés s'il y en a plusieurs)
  - ◆ Une annotation commence par @
    - ◆ Exemple : @Override pour préciser que c'est la redéfinition d'une méthode de la classe mère
  - ◆ Une annotation est définie par une classe Java dédiée

## Annotation basique d'attributs

- ◆ Cas où le contenu d'une colonne mappe directement le contenu d'un attribut de type simple (pas de collection)
- ◆ @Basic : mapping colonne/attribut
  - ◆ Optionnel, s'applique par défaut : pour préciser des options
    - ◆ fetch : FetchType (voir plus loin)
    - ◆ optional : boolean, si remplir l'attribut est obligatoire ou pas
- ◆ @Id : si l'attribut est un identifiant d'objet (aucun paramètre)
- ◆ @Column : optionnel, précision de paramètres
  - ◆ Application implicite par défaut sur la base de même nom entre un attribut et une colonne
  - ◆ Plusieurs paramètres optionnels dont
    - ◆ name : String, par défaut égal au nom de la propriété
    - ◆ table : String, si colonne vient d'une autre table que la primaire
    - ◆ unique : boolean, si unicité des valeurs de la colonne

## Mappings généraux de l'exemple

- ◆ Les tables sport, sportif et discipline ont respectivement une classe associée
  - ◆ Une colonne d'une table = un attribut d'une classe
  - ◆ La clé primaire = l'identifiant de la classe
  - ◆ Les jointures sont implémentées par des Set<XXX>
- ◆ La table sportif a une correspondance vers deux classes
  - ◆ Sportif et Adresse
  - ◆ Avoir deux tables = jointure alors que très peu d'adresses seront partagées par plusieurs sportifs
- ◆ La table d'association pratique n'est pas représentée par une classe
  - ◆ On aura directement des méthodes dans les classes Sportif et Discipline pour récupérer les éléments correspondants
  - ◆ Les mappings permettront de directement gérer cette association

## Annotations des entités

- ◆ Annotations JPA définies dans le package javax.persistence
- ◆ @Entity
  - ◆ Définit une classe comme étant une entité EJB dont les instances pourront être rendues persistantes
  - ◆ Paramètre name="nomEntite"
    - ◆ Nom optionnel de l'entité : par défaut le nom de la classe
    - ◆ A utiliser si deux classes de même nom dans 2 packages différents
- ◆ @Table(name = "sport")
  - ◆ Définit la table primaire mappée sur la classe
  - ◆ Paramètres
    - ◆ name : nom de la table, optionnel si la classe a le même nom
    - ◆ schema, catalog : optionnel, si utile à préciser, sinon valeurs par défaut
    - ◆ uniqueConstraints : optionnel, contraintes d'unicité sur des colonnes

## Exemple : entité Sport

- ◆ Avec ajout des annotations, la classe Sport devient

```
// définit l'entité mappant la table « sport » dans le schéma par défaut
@Entity
@Table(name = "sport")
public class Sport implements Serializable {

    // définit l'attribut identifiant « codeSport » qui mappe la clé primaire
    // « code_sport » et qui est obligatoire
    @Id
    @Basic(optional = false)
    @Column(name = "code_sport")
    private Integer codeSport;

    // l'attribut « intitulé » mappe la colonne « intitulé » : optionnel car ont
    // le même nom, le mapping se ferait implicitement et par défaut
    @Column(name = "intitulé")
    private String intitulé;

    ...
}
```



## Annotation basique d'attributs

- ◆ Annotations pour types de données particulières
  - ◆ `@Temporal` : types temporels (Date ...), `@Enumerated` : pour une énumération, `@Lob` : donnée de grande taille, ...
- ◆ Pour un identifiant
  - ◆ `@GeneratedValue` : si la clé primaire est générée automatiquement par ex. par le SGBD (IDENTITY) ou le moteur JPA (AUTO)
    - ◆ Paramètres optionnels
      - ◆ `strategy` : enum GenerationType (TABLE, SEQUENCE, IDENTITY, AUTO)
      - ◆ `generator` : nom du générateur de clé autre que celui par défaut
  - ◆ Dans ces cas là, peut se passer d'un setter public sur l'attribut ou d'un paramètre pour le constructeur
    - ◆ Ne sera pas pris en compte à la création de l'objet
  - ◆ De manière plus générale : ne peut pas changer une clé primaire, pas besoin de setter si id précisé ou généré à la création de l'objet
- ◆ `@Transient` si on ne veut pas qu'un attribut ait une correspondance dans la table 31

## Exemple : Sportif et Adresse

- ◆ Classe Sportif

```
// mapping de l'entité vers la table « sportif »
@Entity
@Table(name = "sportif")
public class Sportif implements Serializable {

    // codeSportif : identifiant obligatoire auto-généré par le SGBD
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "code_sportif")
    private Integer codeSportif;

    // nom : mapping implicite, valeur non nulle
    @Basic(optional = false)
    private String nom;

    // adresse qui est définie par une classe incluse
    @Embedded
    private Adresse adresse;
    ....
}
```

33

## Surcharge de mappings

- ◆ Modification de la table « sportif » pour rajouter une adresse professionnelle avec mêmes valeurs que l'adresse perso
  - ◆ `sportif(code_sportif, rue, ville, code_postal, rue_pro, ville_pro, code_postal_pro)`
- ◆ Dans la classe Sportif, on va rajouter un deuxième objet inclus de type Adresse
  - ◆ Pb : le mapping précisé dans le composant Adresse se fait sur les colonnes de l'adresse perso (rue, ville, code\_postal)
  - ◆ Solution : surcharger les mappings des attributs lors de l'inclusion dans la classe Sportif

```
@Embedded
private Adresse adresse;

@Embedded
@AttributeOverrides({
    @AttributeOverride(name="rue", column = @Column(name="rue_pro")),
    @AttributeOverride(name="codePostal", column = @Column(name="code_postal_pro")),
    @AttributeOverride(name="ville", column = @Column(name="ville_pro"))
})
private Adresse adressePro;
```

35

## Composant inclus

- ◆ Dans notre exemple
  - ◆ Une adresse n'a pas d'existence indépendamment d'un sportif : relation « physique » de composition 1 – 1
  - ◆ La classe Adresse est « incluse » dans la classe Sportif
- ◆ `@Embedded` : annotation sur l'attribut de la classe incluse
- ◆ `@Embeddable` : annotation dans la classe incluse
  - ◆ On y définira là les mappings par rapport à la table primaire de l'entité principale
  - ◆ Ne définit pas d'identifiant qui sera celui de l'entité
- ◆ Variante si on veut que l'attribut inclus soit la clé primaire de l'entité principale (pratique pour définir des clés composées)
  - ◆ `@EmbeddedId` à la place de `@Embedded` sur l'attribut
  - ◆ Pas d'attribut marqué `@Id` dans l'entité principale (sinon on aurait deux identifiants) 32

## Exemple : Sportif et Adresse

- ◆ Classe Adresse

```
// définition de la classe embarquée
// mappings définis par rapport à la table primaire de la classe Sportif,
// c'est-à-dire la table « sportif »
@Embeddable
public class Adresse implements Serializable {

    // mapping implicite de la colonne « rue »
    private String rue;

    // mapping implicite de la colonne « ville »
    private String ville;

    // mapping de la colonne « code_postal »
    @Column(name = "code_postal")
    private String codePostal;

    // getters, setters, constructeurs ...
}
```

34

## Deux tables vers une entité

- ◆ On veut préciser qu'un sport est géré par une fédération
- ◆ Ajout d'une nouvelle table dans la BDD
  - ◆ `federation(code_federation, nom, acronyme)`
- ◆ Coté objet, on veut directement inclure le nom et l'acronyme d'une fédération dans la classe Sport
  - ◆ Il n'y aura pas de classe Federation
- ◆ La structure de la classe sera mappée vers deux tables, la principale « sport » et la secondaire « federation »
  - ◆ `@SecondaryTable` avec paramètres
    - ◆ `name` : le nom de la table secondaire (obligatoire)
    - ◆ `pkJoinColumns` : la/les clé(s) privée(s) correspondant à la clé primaire de la table secondaire (optionnel si mêmes noms de clés)
  - ◆ Si plus de deux tables requises : `@SecondaryTables`

36

## Modification de la classe Sport

```
@Entity
// « sport » est la table primaire pour le mapping de la classe
@Table(name = "sport")
// « federation » est la table secondaire pour le mapping de la classe qui se fait via sa
// clé primaire « code_federation »
@SecondaryTable(
    name="federation",
    pkJoinColumn = @PrimaryKeyJoinColumn(name = "code_federation")
)
public class Sport implements Serializable {

    // définit l'attribut identifiant « codeSport » qui mappe la clé de la table primaire
    @Id
    @Basic(optional = false)
    @Column(name = "code_sport")
    private Integer codeSport;

    // l'attribut « intitule » mappe la colonne « intitule » de la table primaire
    @Column(name = "intitule")
    private String intitule;

    // attribut mappé sur la colonne « nom » de la table secondaire « federation »
    @Column(name = "nom", table="federation")
    private String nomFederation;

    // attribut mappé sur la colonne du même nom de la table secondaire « federation »
    @Column(table="federation")
    private String acronyme;
}
```

37

## Jointures / associations

- ◆ Quatre types de jointures/associations entre tables/classes
  - ◆ 1 vers 1 : `@OneToOne 1 --- 1`
  - ◆ 1 vers plusieurs : `@OneToMany 1 --- *`
  - ◆ Plusieurs vers un : `@ManyToOne * --- 1`
  - ◆ Plusieurs vers plusieurs `@ManyToMany * --- *`
    - ◆ Nécessite une table d'association entre les 2 tables
    - ◆ Utilisera `@JoinTable` pour préciser cette table
  - ◆ `@JoinColumn`
    - ◆ Sur un attribut, précisera avec quelle clé étrangère est fait la jointure
- ◆ Les associations pourront être uni ou bi-directionnelles

39

## OneToOne

- ◆ Classe Sport
  - ◆ Définit un attribut de type Federation par une jointure `@OneToOne` sur la clé étrangère `code_federation`
    - ◆ `@OneToOne`  
`@JoinColumn(name="code_federation")`  
`private Federation federation;`
    - ◆ `@JoinColumn` est optionnel si la clé étrangère et l'attribut ont le même nom (pas le cas ici). Champs optionnels de `@JoinColumn`:
      - ◆ `name` : nom de la colonne de la clé privée
      - ◆ `referencedColumnName` : nom de la clé primaire dans table primaire
      - ◆ `table` : nom de la table si autre que celle référencée par défaut
- ◆ Classe Federation
  - ◆ Utilise le paramètre `mappedBy` de `@OneToOne`
  - ◆ Précise simplement que c'est l'autre bout de l'association définie dans la classe Sport par l'attribut `federation`
    - ◆ `@OneToOne(mappedBy = "federation")`  
`private Sport sport;`

41

## Deux tables vers une entité

- ◆ Mapping des attributs
  - ◆ Par défaut sur la table primaire, sur la secondaire si précisé explicitement
- ◆ Précision de la clé primaire de la table secondaire
  - ◆ Ne fait pas une vraie jointure
  - ◆ Récupère la ligne avec la clé primaire de la table secondaire de même valeur que la clé primaire de la ligne de la table primaire
    - ◆ Conceptuellement, une même donnée répartie entre deux tables donc avec la même clé
- ◆ Exemple
  - ◆ Si on a dans la table sport(`code_sport`, intitule)  
3 | natation
  - ◆ On a dans la table federation(`code_federation`, nom, acronyme)  
3 | fédération française de natation | FFN

38

## OneToOne

- ◆ Modification de la table sport pour considérer la fédération comme une jointure
  - ◆ Un sport a une fédération : association 1-1
  - ◆ Une fédération gère un sport : association 1-1
  - ◆ Peut définir une association bidirectionnelle
- ◆ Tables
  - ◆ sport(`code_sport`, intitule, `code_federation`)
    - ◆ Avec `code_federation` clé étrangère vers la table federation
  - ◆ federation(`code_federation`, nom, acronyme)
    - ◆ Retrouvera le sport géré par la fédération via la clé étrangère dans la table sport

40

## ManyToMany et OneToMany

- ◆ Association bidirectionnelle entre sports et disciplines
  - ◆ Un sport possède plusieurs disciplines : `OneToMany`
  - ◆ Une discipline appartient à un sport : `ManyToOne`
    - ◆ Clé étrangère `code_sport` vers la table sport
- ◆ Classe Discipline
  - ◆ Jointure via la clé étrangère `code_sport` qui est obligatoire  
`@JoinColumn(name = "code_sport")`  
`@ManyToOne(optional = false)`  
`private Sport sport;`
- ◆ Classe Sport
  - ◆ Jointure définie comme l'opposée de celle de l'attribut sport dans la classe Discipline  
`@OneToMany(cascade = CascadeType.ALL,`  
`fetch = FetchType.LAZY,`  
`mappedBy = "sport")`  
`private Set<Discipline> disciplines;`

42

## ManyToMany

- ◆ Association bidirectionnelle entre sportifs et disciplines
  - ◆ Un sportif pratique plusieurs disciplines
  - ◆ Une discipline est pratiquée par plusieurs sportifs
- ◆ Utilisera une table d'association pour la jointure
  - ◆ Une clé étrangère pour chacune des deux tables
  - ◆ La clé primaire est la composition des deux clés étrangères
- ◆ Noms par défaut pour JPA
  - ◆ Nom de la table est la concaténation des deux noms de tables : « DISCIPLINE\_SPORTIF »
  - ◆ Nom clé étrangère : nom de la table puis « \_id », ex : « sportif\_id »
  - ◆ Si on utilise d'autres noms : @JoinTable
    - ◆ Cas de notre exemple, la table d'association est pratique(code\_sportif, code\_discipline)

43

## Gestion des associations entre entités

- ◆ Une discipline appartient à un sport et un sport contient des disciplines
  - ◆ Bien positionner les bonnes références dans les deux objets même si on a défini l'association des entités comme bidirectionnelle
  - ◆ Privilégier une méthode dédiée pour cela, de préférence du côté de la classe définissant la jointure
- ◆ Exemple

```
public class Discipline {
    ...
    @JoinColumn(name = "code_sport")
    @ManyToOne(
        optional = false,
        cascade=CascadeType.ALL)
    private Sport sport;
    ...
    public void addSport(Sport sport) {
        this.sport = sport;
        sport.getDisciplines().add(this);
    }
    ...
}
// ajout d'une discipline
...
trans.begin();
Sport s = em.find(Sport.class, 3);
Discipline d = new Discipline(10, "3000m");
d.addSport(s);
em.persist(d);
trans.commit();
...
```

45

## Paramètres d'annotations

- ◆ cascade de type CascadeType (suite)
  - ◆ Autre exemple
- ◆ orphanRemoval=true
  - ◆ Si positionné à vrai, supprime les entités dites orphelines : entités qui doivent être associées à d'autres mais ne le sont plus
  - ◆ S'applique sur @OneToOne et @OneToMany
- ◆ Exemple

```
// supprime la relation d'un sport avec sa fédération
sport.setFederation(null);
```

Si orphanRemoval est à vrai sur l'attribut federation de Sport, alors son ancienne fédération sera automatiquement supprimée

47

## ManyToMany

- ◆ Classe Discipline
  - ◆ Utilise un @JoinTable qui définit chacune des deux clés utilisées sur la table d'association pratique
    - ◆ Via un @JoinColumn ou plusieurs si clés composées
  - ◆ L'une est l'inverse de l'autre (au choix)
- ◆ Classe Sportif
  - ◆ Définit simplement les disciplines comme l'opposé de l'association dans la classe Discipline

```
@JoinTable(
    name = "pratique",
    joinColumns = { @JoinColumn(name = "code_discipline") },
    inverseJoinColumns = { @JoinColumn(name = "code_sportif") }
)
@ManyToMany
private Set<Sportif> sportifs;
```

```
@ManyToMany(mappedBy = "sportifs")
private Set<Discipline> disciplines;
```

44

## Paramètres d'annotations

- ◆ cascade de type CascadeType
  - ◆ Utilisable avec @OneToOne, @OneToMany et @ManyToMany
  - ◆ Précise les relations de cycle de vie entre une entité et celles avec qui elle est liée par des jointures
    - ◆ Si une action est effectuée sur une entité, est-elle appliquée en cascade sur les entités avec qui on a une jointure ?
  - ◆ Valeurs : PERSIST, MERGE, REMOVE, REFRESH, DETACH
    - ◆ ALL pour tout en même temps
- ◆ Exemple

```
Sport peche = new Sport(4, "pêche");
Discipline disc = new Discipline(12, "mouche", peche);
// on rend persistante la discipline mais pas son sport
em.persist(disc);
```

Si l'attribut sport de Discipline est marqué comme CascadeType.PERSIST, le sport « pêche » sera enregistré dans la BDD en même temps que la discipline, sinon une erreur est générée

46

## Paramètres d'annotations

- ◆ fetch de type FetchType
  - ◆ Deux valeurs : LAZY, EAGER
  - ◆ Utilisable sur toutes les annotations de jointures et @Basic
  - ◆ Précise quand on charge une entité à partir de la base, si on charge aussi les entités liées
    - ◆ EAGER : chargées directement
    - ◆ LAZY : chargées à la demande (accès à l'attribut du POJO, appel du getter ...)
  - ◆ Par défaut
    - ◆ @Basic, @OneToOne, @ManyToOne : EAGER
    - ◆ @OneToMany, @ManyToMany : LAZY

48



## Gestion de l'héritage entre entités

- ◆ Héritage entre classes : fonctionnalité native et importante des langages objets
- ◆ N'est pas géré nativement coté BDD en relationnel
- ◆ Trois solutions générales, si B et C héritent de A
  - ◆ Une table par classe avec duplication
    - ◆ Les champs communs avec A d'une ligne de B ou C sont dupliqués dans A
  - ◆ Une table par classe sans duplication
    - ◆ Les données de A sont « importées » pour une ligne de B ou C via une jointure sur A
  - ◆ Une seule table qui contient tous les attributs de A, B et C
    - ◆ Des champs laissés vides selon les besoins dans chaque ligne
- ◆ `@Inheritance(strategy=InheritanceType.[TABLE_PER_CLASS|JOINED|SINGLE_TABLE])`  
public class A {...
- ◆ Choisira une des trois valeurs pour indiquer à JPA quelle stratégie de gestion d'héritage est utilisée coté SGBD

49

## Factorisation d'attributs/associations

```
@MappedSuperclass
public abstract class Personne
    implements Serializable {

    @Id
    private int id ;

    private String nom ;

    @Embedded
    private Adresse adresse ;

    ...
}

@Entity
@Table(name="sportif")
@AttributeOverrides {
    name="id",
    column=@Column(name="code_sportif")
}
public class Sportif extends Personne {

    // Attributs : id, nom et adresse sont hérités
    // Mappings : même nom de colonne pour
    // nom, dans classe embarquée pour
    // adresse et redéfini par surcharge pour id

    @ManyToMany(mappedBy = "sportifs")
    private Set<Discipline> disciplines;

    ...
}
```

51

## Manipulation d'entités persistantes

53

## Factorisation d'attributs/associations

- ◆ `@MappedSuperclass`
  - ◆ Définit une classe dont les attributs et associations avec leur mappings seront hérités dans des sous-classes
  - ◆ Cette classe n'est pas une entité
    - ◆ Pas de table associée en BDD
    - ◆ Pas de référence de la part d'entités sur des instances de cette classe
  - ◆ Si on veut surcharger des mappings dans les classes filles
    - ◆ `@AttributeOverride(s)`, `@AssociationOverride(s)`
  - ◆ Exemple
    - ◆ Une classe Personne spécialisée par Sportif
    - ◆ On factorisera l'identifiant de la personne, son nom et son adresse
    - ◆ Dans la classe Sportif, on surchargera le mapping de l'identifiant pour préciser la colonne de clé primaire et on rajoutera l'ensemble des disciplines pratiquées par le sportif

50

## Autres annotations

- ◆ `@IdClass` ou `@EmbeddedId`
  - ◆ Gestion d'une clé primaire composée
- ◆ `@MapKey`
  - ◆ Jointure renvoyant une Map au lieu d'un ensemble ou une liste
  - ◆ Utile pour gérer les associations ternaires
- ◆ `@NamedQuery`
  - ◆ Prédéfinition de requête (paramétrée) pour une entité
- ◆ `@Version`
  - ◆ Verrou sur une donnée en cas d'accès concourant pour assurer la cohérence
- ◆ `@Sort` et `@OrderBy`
  - ◆ Tri des données pour une liste lors de leur récupération en BDD en fonction de critères
- ◆ ...

52

## Principes de fonctionnement

- ◆ Configuration d'une unité de persistance
  - ◆ Fichier XML définissant la liste des classes correspondant à des entités, la connexion au SGBD, divers paramètres ...
- ◆ Dans l'application Java
  - ◆ Récupère un « entity manager »
  - ◆ A partir de ce gestionnaire, on manipulera les entités
    - ◆ Récupération d'instances d'entités en base, rendre persistant de nouvelles instances, modifier des entités ...
    - ◆ Les modifications se font via une transaction
- ◆ Note
  - ◆ Le code présenté dans ces transparents se base sur une implémentation en Java SE
  - ◆ Avec Java EE, peut utiliser d'autres fonctionnalités comme JTA pour les transactions ou JNDI pour les annuaires

54

## Unité de persistance

Fichier « persistence.xml » se trouvant dans le répertoire « META\_INF »

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" ... >
<persistence-unit name="SportsPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>data.Discipline</class>
  <class>data.Sport</class>
  <class>data.Sportif</class>
  <class>data.Adresse</class>
  <class>data.Federation</class>
  <properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/sports"/>
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    <property name="javax.persistence.jdbc.user" value="eric"/>
    <property name="javax.persistence.jdbc.password" value="eric"/>
  </properties>
</persistence-unit>
</persistence>
```

55

## Entity manager

On le récupère à partir de la fabrique de gestionnaire d'entité et via le nom donné à l'unité de persistance

Code qui positionne un attribut « em » correspondant au gestionnaire d'entité pour notre unité de persistance « SportsPU »

```
public class AccesSportsJPA {
    // le gestionnaire d'entité qu'on utilisera pour accéder à la base « sports »
    private EntityManager em = null;

    // retourne le gestionnaire d'entité en l'instanciant au besoin
    public EntityManager getEntityManager() {
        if (em == null) {
            EntityManagerFactory emf = Persistence.createEntityManagerFactory("SportsPU");
            em = emf.createEntityManager();
        }
        return em;
    }

    // ferme le gestionnaire d'entité
    public void closeEntityManager() {
        if (em != null) { em.close(); em = null; }
    }
    ...
}
```

57

## Opérations sur les instances d'entités

Récupérer une instance d'une certaine entité en précisant sa classe et son identifiant

<T> T find(Class<T> entityClass, Object id)  
<T> T getReference(Class<T> entityClass, Object id)

Différences

- find fait le chargement de l'objet directement, getReference seulement quand on accédera à son contenu
- find renvoie null si pas d'instance trouvée, getReference lève l'exception EntityNotFoundException

Exemple : récupérer l'instance du sport de clé 3

```
EntityManager em = this.getEntityManager();
Sport sp = em.find(Sport.class, 3);
System.out.println("Le sport de clé 3 est : "+sp.getTitre());
```

Peut aussi récupérer une/des instance(s) d'entités via des requêtes en JPQL

59

## Unité de persistance

<persistence-unit ...>

Définit le nom de l'unité de persistance (« SportsPU ») et le type de transaction utilisée (celle de JPA en local)

<provider>

Le moteur de persistance utilisé par JPA

Ici EclipseLink

<class>

Définit qu'une classe Java sera une entité dont les instances seront persistantes en base de données

<properties>

Ensemble de propriétés de configuration

Ici on retrouve les paramètres de connexion à la base via JDBC

URL, driver, utilisateur et mot de passe

....

56

## États d'un objet persistant

Plusieurs états pour l'instance d'une classe entité

Persistante : elle a une correspondance de contenu en BDD

Gérée : son état est synchronisé par le gestionnaire d'entité avec le contenu en BDD

Détachée : son état n'est plus géré, les modifications ne seront plus synchronisées avec la BDD

Transient : objet java classique avec existence uniquement en mémoire de la JVM

Cas de l'instanciation de l'objet (avec un new)

Supprimé : instance persistante dont on a supprimé le contenu associé en BDD

L'objet existe toujours en mémoire de la JVM mais n'a plus de correspondance en base

58

## Opérations sur les instances d'entités

Les opérations modifiant le contenu de la BDD se font en mode transactionnel

Fonctionnement similaire à ce que l'on a vu avec JDBC

Schéma général

```
EntityManager trans = null;
try {
    // débute la transaction
    trans = em.getTransaction();
    trans.begin();
```

// actions sur des objets impliquant des modifications en BDD

```
// pas d'erreur, on valide les modifications
trans.commit();
}
```

```
// exception levée, on annule les modifications
catch (Exception e) {
    if (trans != null) trans.rollback();
}
```

60

## Opérations sur les instances d'entités

- ◆ void persist(Object entity)
  - ◆ Rend persistant en base de données un objet qui devient géré par le gestionnaire d'entités
    - ◆ Utilisable pour un objet transient ou supprimé
    - ◆ Retourne une erreur avec un objet détaché
  - ◆ Si l'objet était déjà géré, ne fait rien sauf si
    - ◆ Des objets en associations (jointures) ont été modifiés et que CascadeType.PERSIST a été positionné
    - ◆ Dans ce cas le persist applique les persist en cascade sur les nouveaux objets
  - ◆ Exemple : ajout d'un nouveau sport

```
trans.begin();
Sport sp = new Sport("pêche", 4);
em.persist(sp);
trans.commit();
```

61

## Opérations sur les instances d'entités

- ◆ void detach(Object entity)
  - ◆ Détache l'objet du gestionnaire d'entités qui ne le gère plus
    - ◆ Exception levée si l'objet n'était pas géré
  - ◆ Les modifications sur l'objet ne seront plus synchronisées sur la base
- ◆ void clear()
  - ◆ Détache tous les objets gérés par le gestionnaire d'entités
- ◆ void remove(Object entity)
  - ◆ Supprime un objet : efface ses données en base
  - ◆ S'applique sur un objet géré
- ◆ boolean contains(Object entity)
  - ◆ Vérifie si l'objet passé en paramètre est géré ou pas par le gestionnaire d'entités
- ◆ void refresh(Object entity)
  - ◆ Remet à jour le contenu de l'objet par rapport au contenu en base
  - ◆ A utiliser si on sait qu'un trigger a pu modifier les données en base

63

## Langage de requête JPQL

- ◆ JPA définit un langage de requête similaire à SQL
  - ◆ Mais travaillant sur la structure des classes directement
    - ◆ On peut également faire des requêtes SQL classiques
- ◆ Succincte présentation de JPQL via quelques exemples
  - ◆ Récupérer un sport via son nom plutôt que son identifiant

```
public Sport getSport(String intitule) {
```

```
// définit la requête via le gestionnaire d'entités
Query query = this.getEntityManager().createQuery("
    select s from Sport s where s.intitule='"+intitule+"'");

// je sais que j'aurai au plus un seul sport, j'utilise getSingleResult
// pour récupérer un objet en retour de l'exécution de la requête
Sport s = (Sport)query.getSingleResult();
return s;
}
```

65

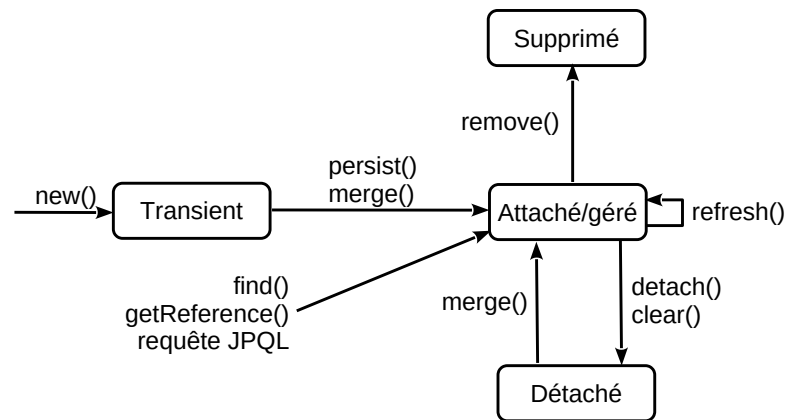
## Opérations sur les instances d'entités

- ◆ <T> T merge(T entity)
  - ◆ Retourne une copie gérée par le gestionnaire d'entité de l'objet passé en paramètre
    - ◆ Utilisé typiquement pour rattacher une instance détachée
    - ◆ Si l'objet n'avait pas de correspondance en BDD, est rendu persistant et géré par le gestionnaire d'entité
    - ◆ Applique en cascade le merge sur les objets associés si CascadeType.MERGE est positionné
  - ◆ N'est pas utilisable avec un objet supprimé
    - ◆ Utiliser persist() pour le rendre à nouveau persistant
  - ◆ Exemple : on suppose que la discipline « biathlon » était détachée, on la rattache et met une majuscule à son nom

```
Discipline biathlonBis = em.merge(biathlon);
// on fait les modifications sur l'objet retourné par le merge, pas l'initial
biathlonBis.setName("Biathlon");
trans.commit();
```

62

## Cycle de vie d'un objet persistant



64

## Langage de requête JQPL

- ◆ Variante de l'exemple précédent avec une requête paramétrée
  - ◆ 

```
public Sport getSport(String intitule) {
    Query query = this.getEntityManager().createQuery("
        select s from Sport s where s.intitule=:leNom");
    query.setParameter("leNom", intitule);
    Sport s = (Sport)query.getSingleResult();
    return s;
}
```

66

## Langage de requêtes JPQL

- ◆ Récupérer l'ensemble des sports pratiqués par un sportif
- ◆ Idée de l'association dérivée « /sport » partant de la classe Sportif

```
public List<Sport> getSportsSportif(Sportif sportif) {  
  
    // la requête recherche tous les sports des disciplines qui sont  
    // pratiqués par le sportif passé en paramètre  
    Query query = this.getEntityManager().createQuery("  
        select distinct(sport) from Sport sport, Discipline disc  
        where disc.sport = sport and  
        :leSportif in elements(disc.sportifs)");  
    query.setParameter("leSportif", sportif);  
  
    // on récupère directement la liste des résultats  
    List<Sport> sports = (List<Sport>)query.getResultList();  
    return sports;  
}
```

- ◆ Ne marche pas pour une erreur de syntaxe, mais marche très bien en HQL d'Hibernate :-)

67

## Requêtes nommées

- ◆ Une « named query » est une requête JPQL identifiée par un nom
- ◆ Quand on génère automatiquement les POJO, l'IDE propose souvent automatiquement des named query
- ◆ Exemple pour la classe Sport

```
@Entity  
@Table(name = "sport")  
@NamedQueries({  
    @NamedQuery(name = "Sport.findAll", query = "SELECT s FROM Sport s"),  
    @NamedQuery(name = "Sport.findByCodeSport", query = "SELECT s FROM  
        Sport s WHERE s.codeSport = :codeSport"),  
    @NamedQuery(name = "Sport.findByIntitule", query = "SELECT s FROM  
        Sport s WHERE s.intitule = :intitule")})  
public class Sport implements Serializable {  
    ...  
}
```

69

## Langage de requêtes JPQL

- ◆ Récupérer tous les sports d'un sportif en SQL natif
- ◆ Jointure sur les 4 tables pratique, discipline, sportif et sport, en fonction de la valeur `code_monSportif`
  - ◆ **select** distinct(sport.code\_sport), sport.intitule  
**from** discipline, pratique, sportif, sport  
**where** sportif.code\_sportif = `code_monSportif` **and**  
discipline.code\_discipline = pratique.code\_discipline **and**  
pratique.code\_sportif = sportif.code\_sportif **and**  
sport.code\_sport = discipline.code\_sport
- ◆ On exécuterait cette requête via JDBC et on devrait construire à la main en parcourant case par case le ResultSet pour instancier les sports et les ajouter un par un dans la liste retournée

68

## Requêtes nommées

- ◆ Pour appeler la requête nommée qui retourne un sport à partir de son intitulé :

```
try {  
    TypedQuery<Sport> query =  
        em.createNamedQuery("Sport.findByIntitule", Sport.class);  
    query.setParameter("intitule", "natation");  
    Sport sport = query.getSingleResult();  
    System.out.println("Les disciplines de natation : ");  
    for (Discipline disc : sport.getDisciplines())  
        System.out.println("-> " + disc.getIntitule());  
} catch (NoResultException ex) {  
    System.out.println("Pas trouvé le sport !");  
}
```

- ◆ On utilise ici une TypedQuery pour préciser le type attendu de la requête (une instance de Sport)
- ◆ Cela évite de faire le cast en retour de l'appel de « getSingleResult() »

70