

JAVA Persistence API JEE6

1

AGENDA

- Présentation
- Les *mappings* de base
- Relations
- EntityManager
- Queries
- NamedQuery
- Transactions
- Demo
- Support et outils
- Questions

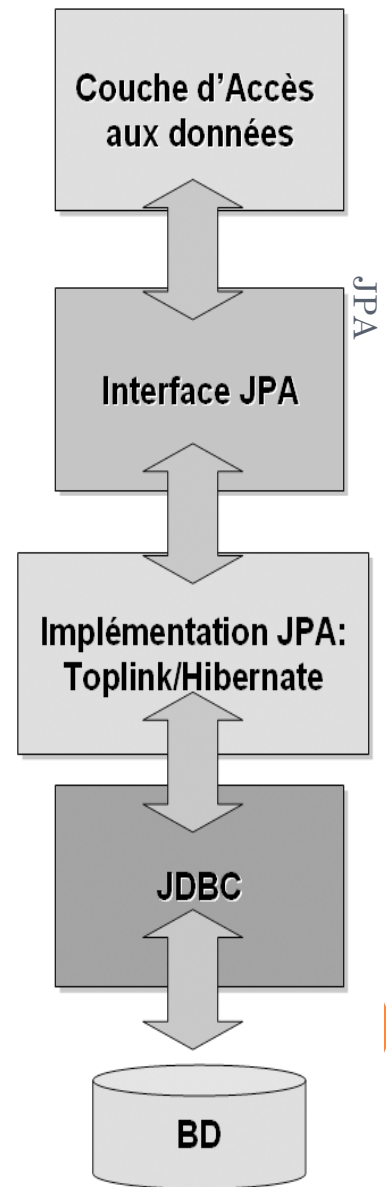
PRÉSENTATION

- L'API de persistance Java JPA est une spécification de Sun. Fondée sur le concept POJO
- Disponible depuis les premières versions du JDK 1.5, JPA est une **norme**, et **non une implémentation**
- Elle est apparue en réponse au manque de flexibilité et à la complexité de la spécification J2EE 1.4, en particulier en ce qui concerne la persistance des beans entités
- JPA est implémentée par deux produits de référence : TopLink, un produit commercial(Oracle) devenu libre, et Hibernate.

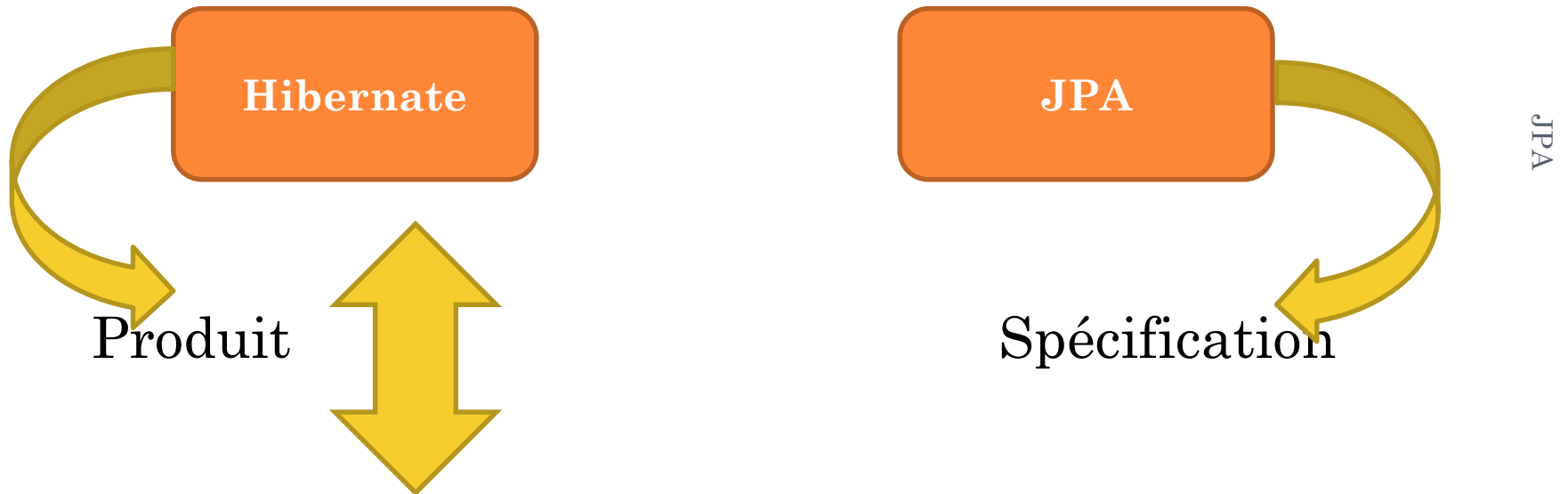
JPA ET SON INTÉGRATION DANS L'ARCHITECTURE N-TIERS?

Les aspects importants de cette nouvelle architecture sont:

- ses relative stabilité
- standardisation.
- La couche d'accès aux données dialoguant avec les interfaces JPA.
- les développements gagnent en souplesse, puisqu'il n'est plus nécessaire de changer de modèle O/R ni de couche DAO (pour l'accès aux données) en fonction de l'outil de mapping Utilisé
- Quel que soit le produit qui implémente l'API, l'interface de la couche JPA reste inchangée.



CORRESPONDANCE HIBERNATE/JPA(1)



- hibernate génère le code SQL pour vous
- persistance transparente
- récupération de données optimisée.
- portabilité du code si changement de base de données.



COMPLEXITÉ DES ORM(1)

S'il est vrai que la valeur ajoutée d'hibernate a diminué avec l'avènement de la spécification JPA

NOTEZ hibernate reste un choix privilégié pour les développeurs qui :

- n'accordent pas une priorité à la portabilité de leur développement
- souhaitent profiter des dernières fonctionnalités, non présentes dans spécifications (qui prennent beaucoup de temps à être validées dans le cadre de la Java Community Process).

MAIS

- Hibernate est populaire mais complexe à maîtriser.
- La courbe d'apprentissage souvent présentée comme facile est en fait assez raide.

COMPLEXITÉ DES ORM(2)

- Utiliser un ORM est complexe pour un débutant.
- il y a des concepts à comprendre pour configurer le pont relationnel / objet. il y a la notion de contexte de persistance avec ses notions d'objets dans un état "persisté", "détaché", "neuf"
- il y a la mécanique autour de la persistance (transactions, pools de connexions), généralement des services délivrés par un conteneur
- il y a les réglages à faire pour les performances (cache de second niveau)
- ...

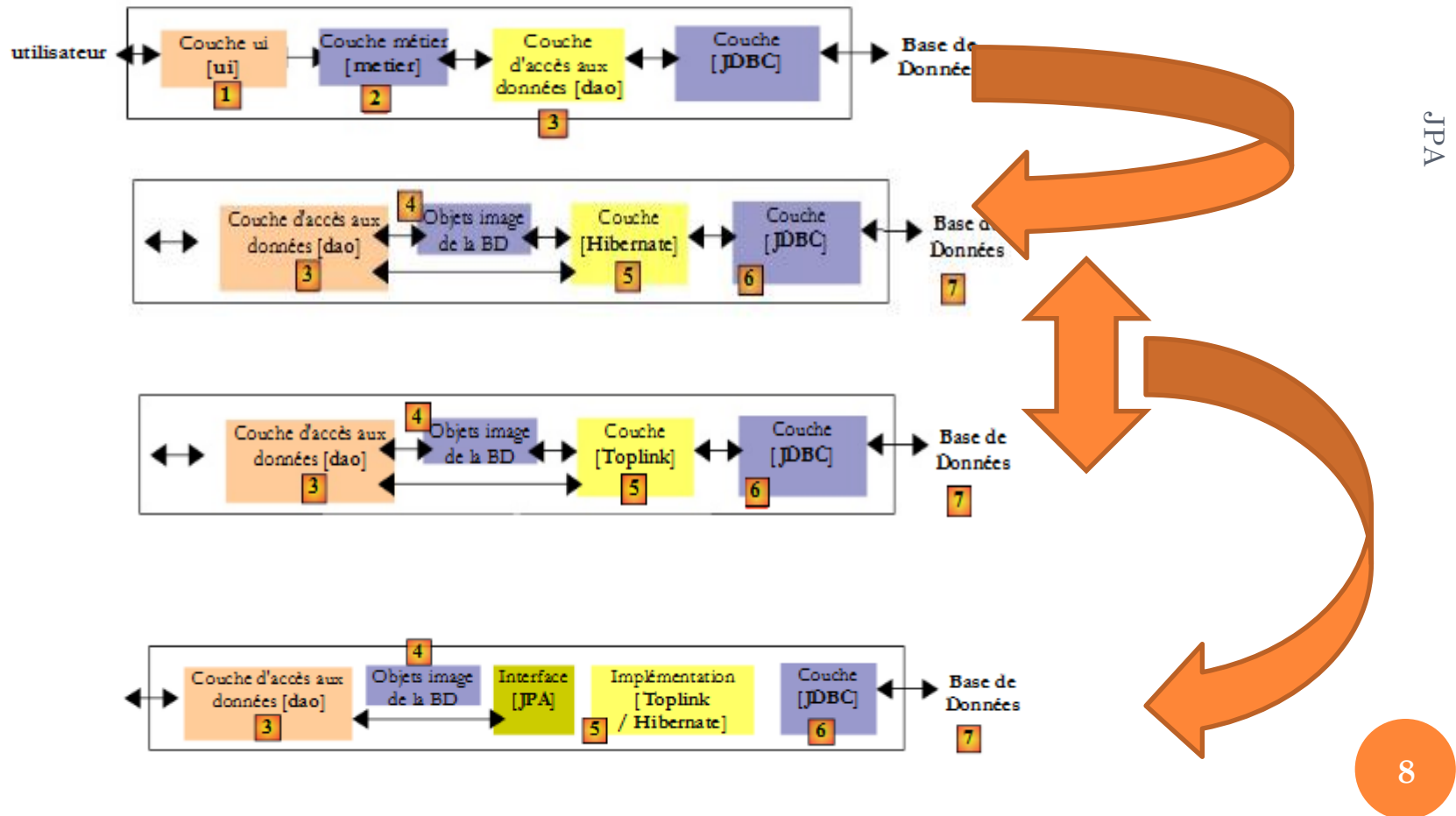


Fichier mapping

Fichier de
configuration

Relations
<one-to-many>
<many-to-one>

EVOLUTION VERS L'API JPA (1)...



EVOLUTION VERS L'API JPA (2)...

La couche [dao] dialogue maintenant avec la spécification JPA, un ensemble d'interfaces.

Le développeur y a gagné en standardisation.

Avant, s'il changeait sa couche ORM, il devait également changer sa couche [dao] qui avait été écrite pour dialoguer avec un ORM spécifique.

Maintenant, il va écrire une couche [dao] qui va dialoguer avec une couche JPA.

Quelque soit le produit qui implémente celle-ci, l'interface de la couche JPA présentée à la couche [dao] reste la même.

JAVA PERSISTENCE API: QUOI DE NEUF?

On a littéralement évacué la complexité

- Plus besoin de ces innombrables interfaces (Home, Remote, Local ...)
- On peut l'utiliser tant à l'extérieur, qu'à l'intérieur d'un *container* JEE
- Chaque entité est maintenant un simple POJO *
- Les mappings sont facilement mis en place, à l'aide d'annotations (Java SE 5.0)

* **POJO** – n.m. [pôdjô]

POJO is an acronym for **Plain Old Java Object**, and is favoured by advocates of the idea that the simpler the design, the better. - Wikipedia.

"We wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely."

- Martin Fowler

AGENDA

- Présentation
- Configuration
- **Les *mappings* de base**
- Relations
- EntityManager
- Queries
- NamedQuery
- Transactions
- Demo
- Support et outils
- Questions

CONFIGURATION VIA HIBERNATE

Avec l'ORM Hibernate

Fichier Cfg.xml

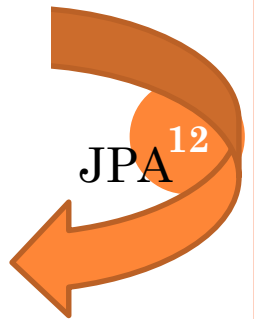
```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/hibernate1</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>

    <!-- Pool de connection (interne) -->
    <property name="connection.pool_size">1</property>

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

    <!-- Supprimer et re-crée le schéma de base de données au démarrage -->
    <property name="hbm2ddl.auto">create</property>
```

JPA



JPA 12

CONFIGURATION VIA JPA(1)

○ Persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="TestJPA/Hibernate" transaction-
type="RESOURCE_LOCAL">

        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
<property name="hibernate.jdbc.Schema" value="test"
<property name="hibernate.connection.driver_class"
value="com.mysql.jdbc.Driver" />
<property name="hibernate.connection.url"
value="jdbc:mysql://localhost:3306/GestionFormation;" />
<property name="hibernate.connection.username" value="root" />
<property name="hibernate.connection.password" value="root" />
<property name="hibernate.Log" value="none" />
<property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect" />
        </properties>

        <class>bean.MyTest</class>

    </persistence-unit>
</persistence>
```

CONFIGURATION VIA JPA(2)

- Il contient un ou plusieurs tags <persistence-unit> qui va contenir les paramètres d'un persistence unit.
- Ce tag possède deux attributs : name (obligatoire) qui précise le nom de l'unité et qui servira à y faire référence et transaction-type (optionnel) qui précise le type de transaction utilisée (ceci dépend de l'environnement d'exécution :Java SE ou Java EE).



Transaction type
=Resource_local



Transaction type=JTA

CONFIGURATION VIA JPA(3)

- Le tag <persistence-unit> peut avoir les tags fils suivants :

Tag	Rôle
<description>	Description purement informative de l'unité de persistance(optionnel)
<provider>	Nom pleinement qualifié d'une classe de type javax.persistence.PersistenceProvider (optionnel). Généralement fournie par le fournisseur de l'implémentation de l'API : une utilisation de ce tag n'est requise que pour des besoins spécifiques
<jta-data-source>	Nom JNDI de la DataSource utilisée dans un environnement avec support de JTA (optionnel)
<non-jta-data-source>	Nom JNDI de la DataSource utilisée dans un environnement sans support de JTA (optionnel)
<mapping-file>	Précise un fichier de mapping supplémentaire (optionnel)
<jar-file>	Précise un fichier jar qui contient des entités à inclure dans l'unité de persistance : le chemin précisé est relatif par rapport au fichier persistence.xml (optionnel)
<class>	Précise une classe d'une entité qui sera incluse dans l'unité de persistance (optionnel)
<properties>	Fournir des paramètres spécifiques au fournisseur. Comme Java SE ne propose pas de serveur JNDI, c'est fréquemment via ce tag que les informations concernant la source de données sont définis (optionnel)
<exclude-unlisted-classes>	Inhibition de la recherche automatique des classes des entités (optionnel)

AGENDA

- Présentation
- Configuration
- **Les *mappings* de base**
- Relations
- EntityManager
- Queries
- NamedQuery
- Transactions
- Demo
- Support et outils
- Questions

MAPPING AVEC ORM:HIBERNATE

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.esprit.hibernate.demohibernate.simple.Formation" table="FORMATIONS">
    <id name="id" column="FORMATION_ID">
      <generator class="increment" />
    </id>
    <property name="theme" column="FORMATION_THEME" />
  </class>
</hibernate-mapping>
```

Diagram illustrating the mapping configuration with numbered annotations:

- 1: `<hibernate-mapping>` tag
- 2: `<class name="com.esprit.hibernate.demohibernate.simple.Formation" table="FORMATIONS">` tag
- 3: `<id name="id" column="FORMATION_ID">` tag
- 4: `<generator class="increment" />` tag
- 5: `<property name="theme" column="FORMATION_THEME" />` tag

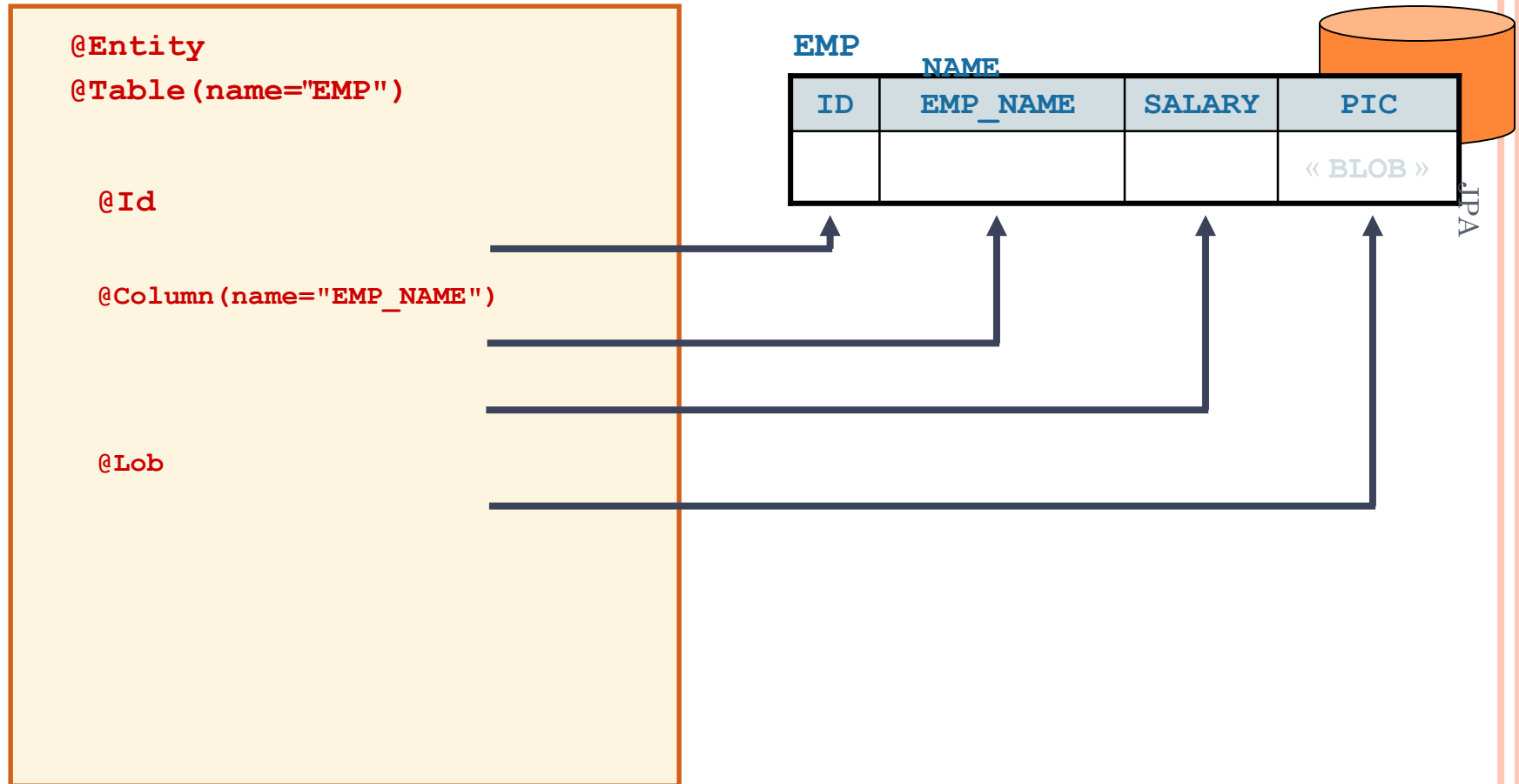
MAPPING VIA JPA

- Au niveau de entity bean sans creation de fichier..

```
@Entity 1
@Table(name="formation") 2
public class Formation {

    private Long id; // Identifiant formation (Clé primaire)
    private String theme;
    @Id 3
    @GeneratedValue 4
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    @Column 5
    public String getTheme() {
        return theme;
    }
}
```

JAVA PERSISTENCE API: MAPPINGS DE BASE



AGENDA

- Présentation
- Les *mappings* de base
- **Relations**
- EntityManager
- Queries
- NamedQuery
- Transactions
- Demo
- Support et outils
- Questions

LES REALTIONS(1)

Tout comme en SQL, on peut définir quatre types de relations entre entités JPA :

- relation 1:1 : annotée par @OneToOne
- relation n:1 : annotée par @ManyToOne
- relation 1:p : annotée par @OneToMany
- relation n:p : annotée par @ManyToMany

LES RELATIONS(2)

Les relations entre entités, telles que définies en JPA peuvent être:

- **Unidirectionnelles**



- **Bidirectionnelles:**



l'une des deux entités doit être **maître** et l'autre **esclave**. le cas des relations 1:1 et n:p, on peut choisir le côté maître comme on le souhaite. Dans le cas des relations 1:p et n:1, l'entité du côté 1 est l'entité esclave.

RELATION 1:1 CAS UNIDIRECTIONNELLE(1)



Schématisation:

-Coté Java:

Entité maitre :Employe

Entité esclave : Contrat

Donc la relation est schématisé au niveau Employe.java

o @OneToOne

@JoinColumn(name="id_contrat")

-Coté SQL:

Une colonne id_contrat va persister au niveau de la table Employe

RELATION 1:1 CAS UNIDIRECTIONNELLE(2)

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name="Emp")
public class Employe {

    private int id_Employe;
    private String nom;
    private Contrat contrat;

    @OneToOne
    @JoinColumn(name="id_contrat")
    public Contrat getContrat() {
        return contrat;
    }

    public void setContrat(Contrat contrat) {
        this.contrat = contrat;
    }

    @Id
    @GeneratedValue
    public int getId_Employe() {
        return id_Employe;
    }

    public void setId_Employe(int id_Employe) {
        this.id_Employe = id_Employe;
    }

    public String getNom() {
```



RELATION 1:1 CAS BIDIRECTIONNELLE(1)

On choisit une entité maitre et une esclave:

- L'entité maitre :vue précédemment cas unidirectionnelle
- L'entité esclave: doit préciser un champ retour par une annotation:
 - @OneToOne
 - un attribut mappedBy, qui doit référencer le champ qui porte la relation côté maître .Il est est défini sur l'entité esclave de la relation.

RELATION 1:1 CAS BIDIRECTIONNELLE(2)

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;
import javax.persistence.OneToOne;
import javax.persistence.Table;

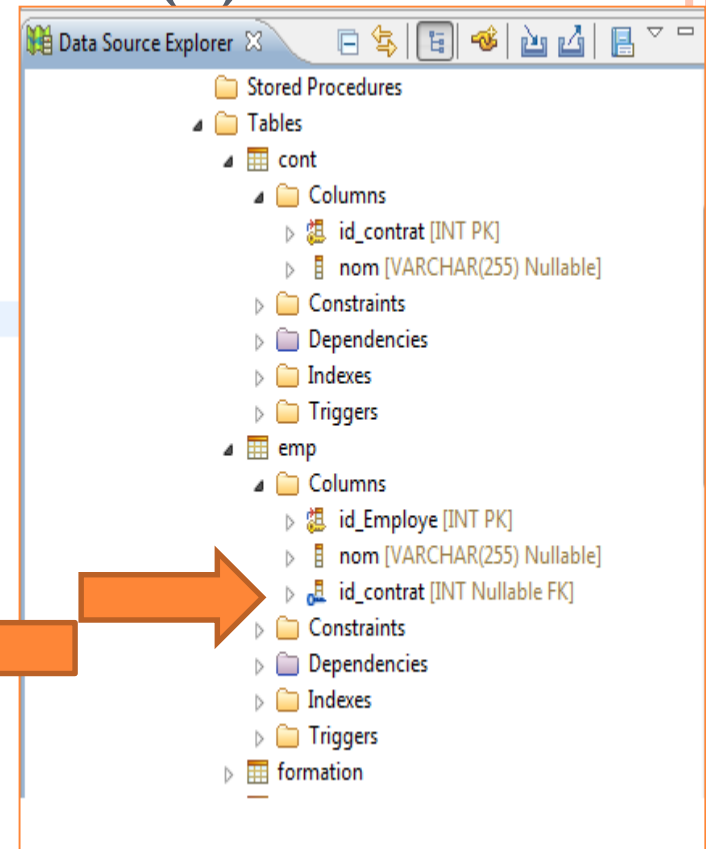
@Entity
@Table(name="cont")
public class Contrat {

    private int id_contrat;
    private String nom;
    private Employee employee;

    @OneToOne(mappedBy="contrat")// référence la relation dans la classe Employee
    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }

    @Id
    @GeneratedValue
    public int getId_contrat() {
        return id_contrat;
    }
}
```



RELATION 1:N CAS UNIDIRECTIONNELLE(1)



Une relation 1:p est caractérisée par :

- Un champ de type **Collection** dans la classe maître
- La classe esclave ne porte pas de relation retour

Dans ces deux cas, JPA crée **une table de jointure** entre les deux tables associées aux deux entités. Cette table de jointure porte une clé étrangère vers la clé primaire de la première table, et une clé étrangère vers la clé primaire de la deuxième table.

Question: pourquoi la spécification JPA prévoit-elle la création d'une table de jointure alors que la relation est de type 1:p ?

La réponse est simple : parce qu'aucune information n'existe sur le champ retour, et que JPA n'a donc aucune information sur la colonne de jointure à utiliser dans la table destination.

RELATION 1:N CAS UNIDIRECTIONNELLE(2)

```
+import java.util.Collection;□
```

```
@Entity
```

```
@Table(name="formation")
```

```
public class Formation {
```

```
    private Long id; // Identifiant formation (Clé primaire)
```

```
    private String theme;
```

```
    private Collection <Employee>employee;
```

```
⊖ @OneToMany
```

```
    public Collection<Employee> getEmployee() {
```

```
        return employee;
```

```
    }
```

```
⊖ public void setEmployee(Collection<Employee> employee) {
```

```
    this.employee = employee;
```

```
    }
```

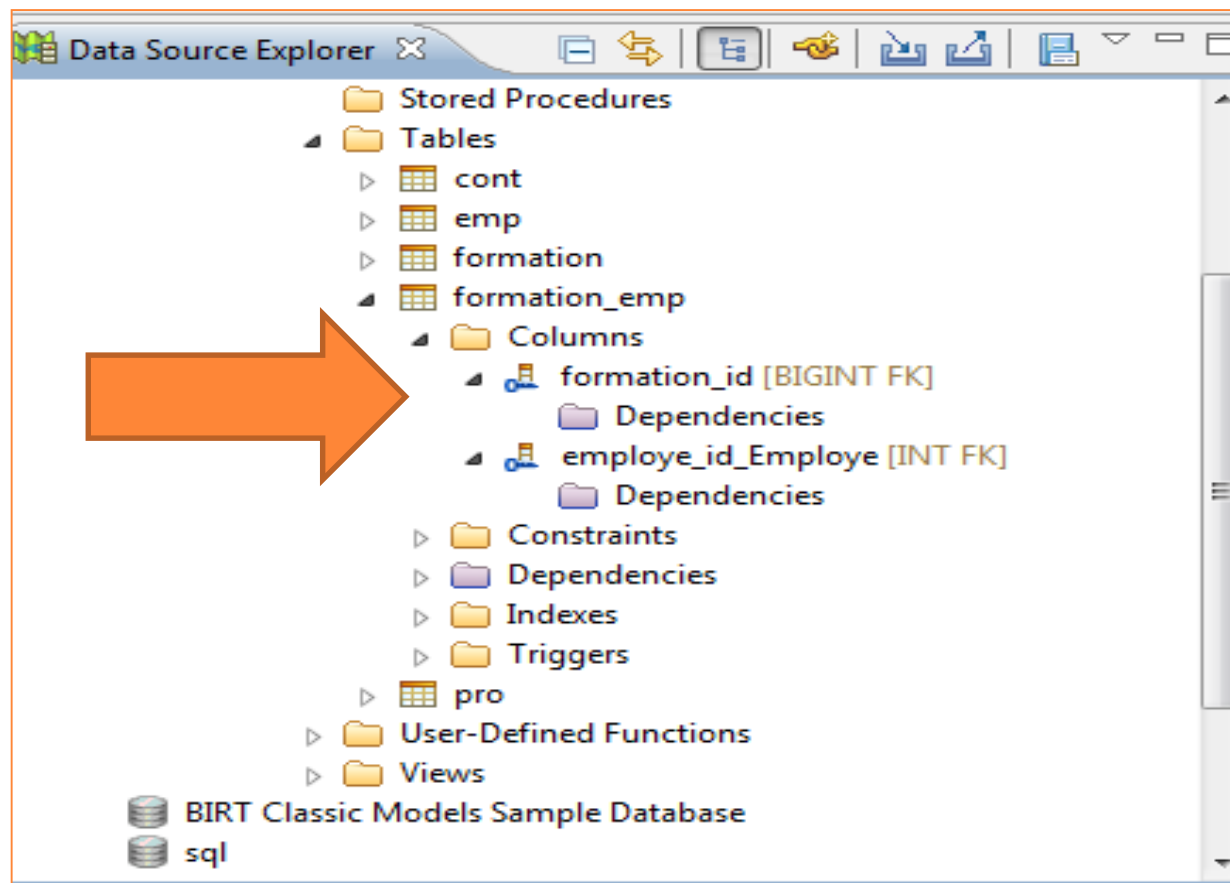
```
⊖ @Id
```

```
@GeneratedValue
```

```
public Long getId() {
```

RELATION 1:N CAS UNIDIRECTIONNELLE(2)

- Au niveau Sql : avec la plateforme eclipseLink: on a le résultat suivant



RELATION 1:N CAS BIDIRECTIONNELLE(1)



Une relation 1:n bidirectionnelle doit correspondre à une relation n:1 dans la classe destination de la relation

Comme pour le cas des relations 1:1, le caractère bidirectionnel d'une relation 1:n est marqué en définissant l'attribut mappedBy sur la relation.

↓
Contrainte JPA:

-l'attribut mappedBy est défini pour l'annotation @OneToMany,
mais pas pour l'annotation @ManyToOne.

-comme nous l'avons vu dans le cas de l'annotation @OneToOne,
mappedBy doit être précisé sur le côté esclave d'une relation.

JPA ne nous laisse donc pas le choix de l'entité maître et de l'entité esclave.

RELATION 1:N CAS BIDIRECTIONNELLE(2)

<one-to-many> coté Formation.java suivie de l'attribut mappedBy

```
+import java.util.Collection;

@Entity
@Table(name="formation")
public class Formation {

    private Long id; // Identifiant formation (Clé prin
    private String theme;
    private Collection <Employee>employee;

    @OneToMany(mappedBy="form")
    public Collection<Employee> getEmployee() {
        return employee;
    }

    public void setEmployee(Collection<Employee> employee)
        this.employee = employee;
    }

    @Id
    @GeneratedValue
    public Long getId() {
```

RELATION 1:N CAS BIDIRECTIONNELLE(3)

<many-to-one>coté Employe.java

```
@Entity
@Table(name="Emp")
public class Employe {

    private int id_Employe;
    private String nom;
    private Contrat contrat;
    private Formation form;

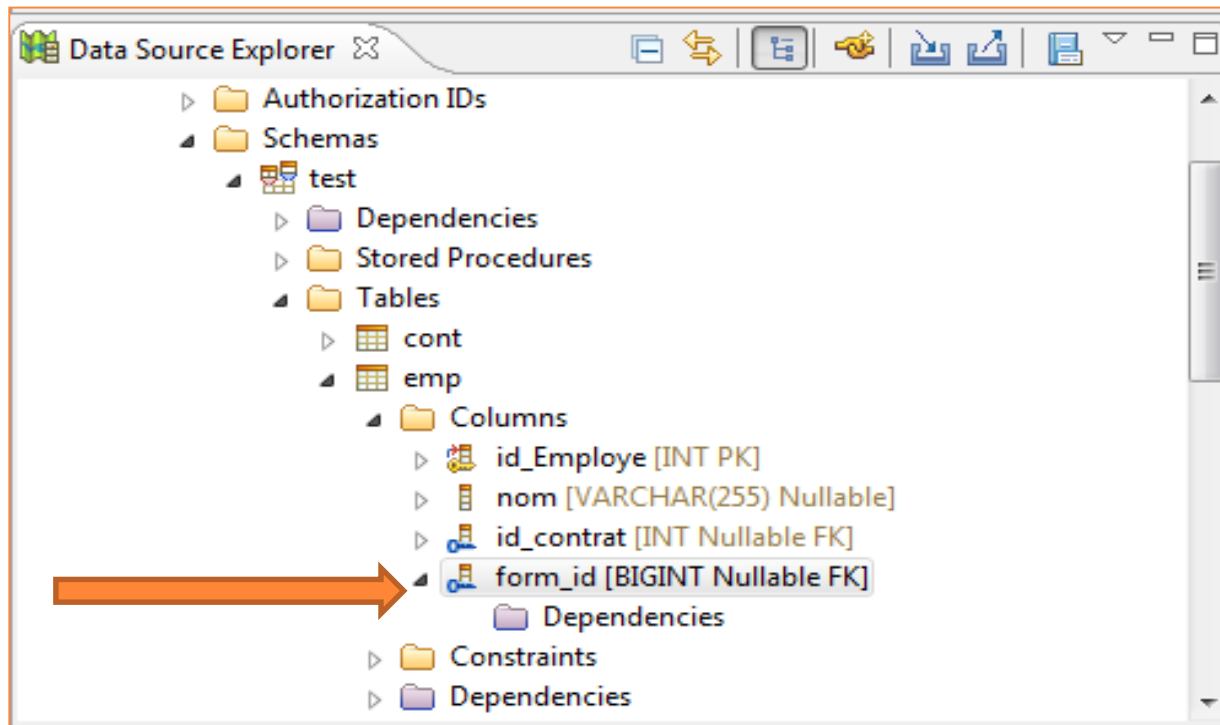
    @ManyToOne
    public Formation getForm() {
        return form;
    }

    public void setForm(Formation form) {
        this.form = form;
    }
}
```


RELATION 1:N CAS BIDIRECTIONNELLE(3)

Au niveau Sql : avec la plateforme eclipseLink: on a le résultat suivant:

JPA n'a plus besoin dans ce cas d'une table de jointure pour coder cette relation. On retombe donc dans le cas nominal d'une relation 1:n avec colonne de jointure dans la table cible de la relation.



RELATION N:1 CAS UNIDIRECTIONNELLE



Selon le même principe de l'entité maitre et l'entité esclave

L'entité maitre est Employe: cette entité porte la relation <many-to-one>

-L'entité esclave est Département

RELATION N:1 CAS UNIDIRECTIONNELLE(2)

```
import javax.persistence.Table;

@Entity
@Table(name="Emp")
public class Employe {

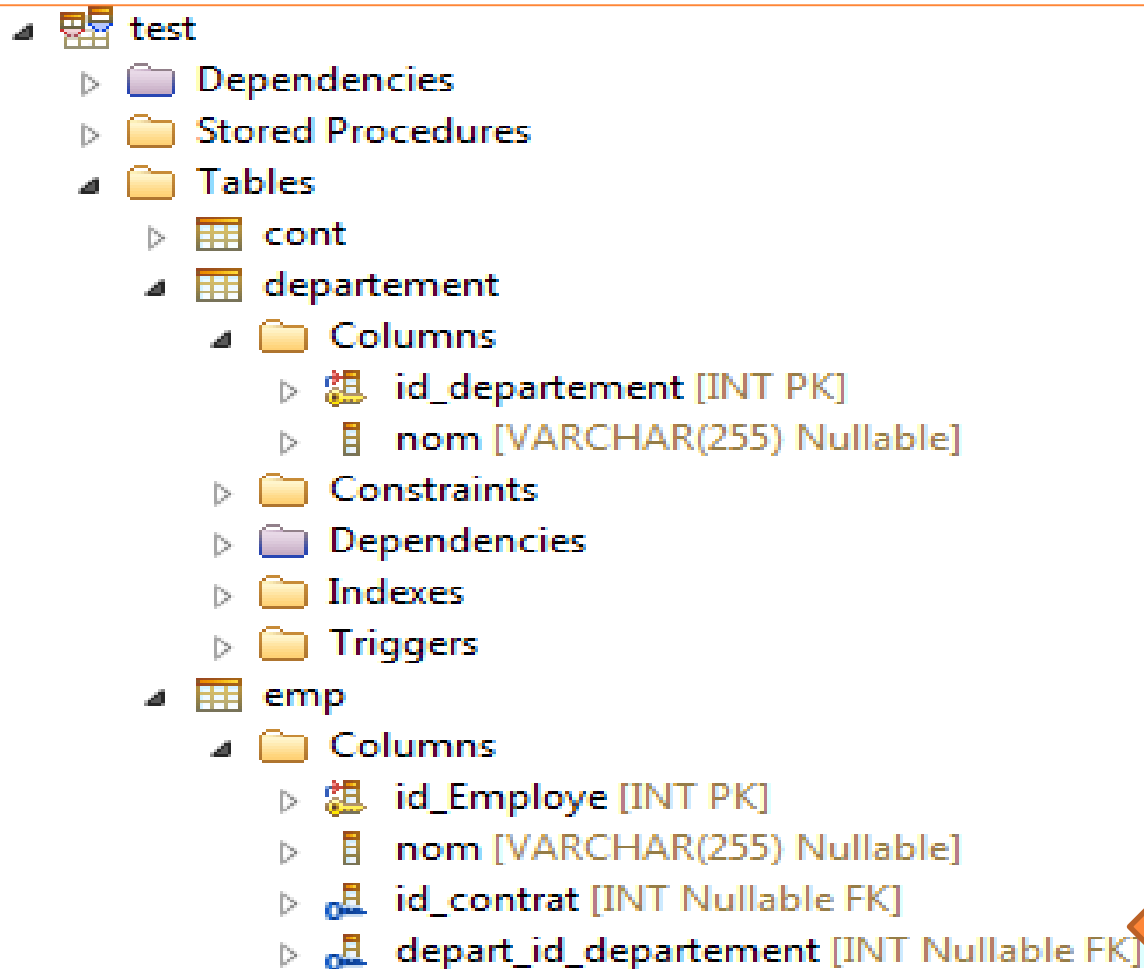
    private int id_Employe;
    private String nom;
    private Contrat contrat;
    private Formation form;
    private Departement depart;

    @ManyToOne
    public Departement getDepart() {
        return depart;
    }

    public void setDepart(Departement depart) {
        this.depart = depart;
    }
}
```

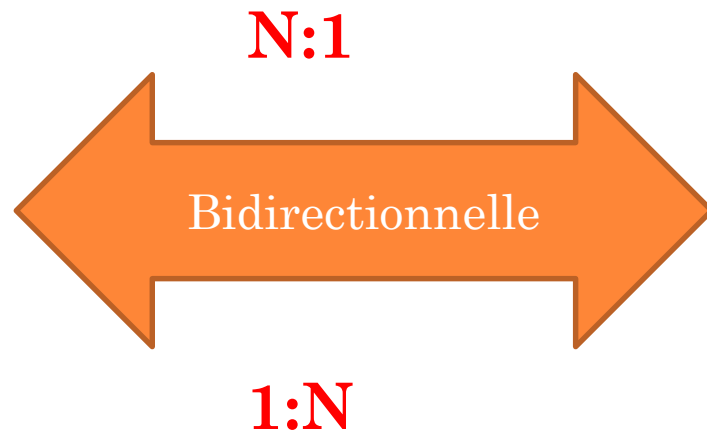
RELATION N:1 CAS UNIDIRECTIONNELLE(3)

Au niveau Sql : avec la plateforme eclipseLink: on a le résultat suivant:



RELATION N:1 CAS BIDIRECTIONNELLE

Le cas bidirectionnel a déjà été traité, puisqu'une relation bidirectionnelle 1:n en JPA est identique à une relation n:1 bidirectionnelle.



RELATION N:M CAS UNIDIRECTIONNELLE(1)



Une relation n:m est une relation multivaluée des deux côtés de la relation. La façon classique d'enregistrer ce modèle en base consiste à créer une table de jointure.

Une table de jointure est créé entre les deux tables qui portent les entités. Cette table référence les deux clés primaires des deux entités au travers de clés étrangères.

- L'entité maitre :Formation.java porteuse de la relation
- L'entité esclave:Module.java

RELATION N:M CAS UNIDIRECTIONNELLE(2)

```
⚠ ⊕ import java.util.Collection;

@Entity
@Table(name="formation")
public class Formation {

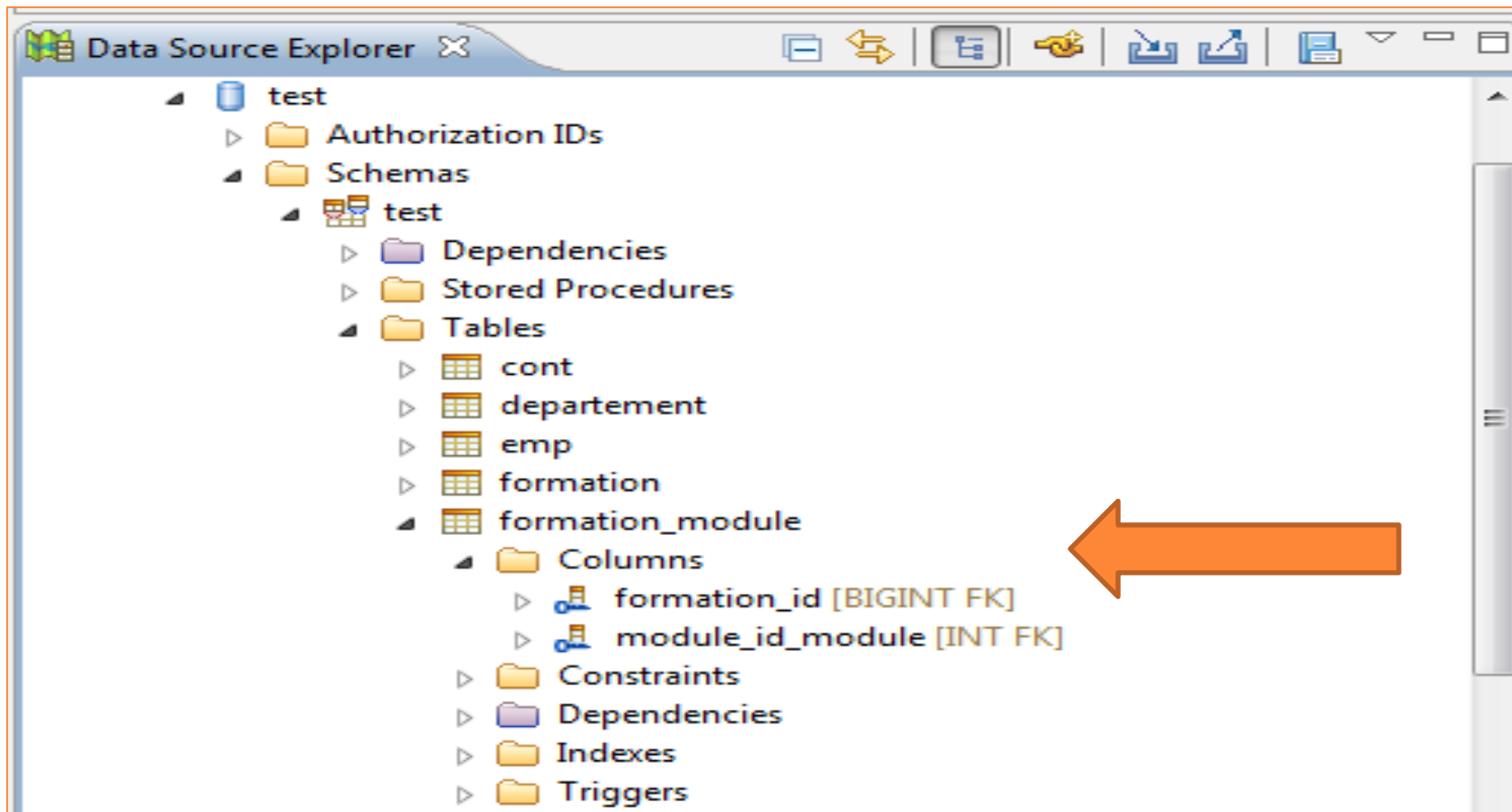
    private Long id; // Identifiant formation (Clé primaire)
    private String theme;
    private Collection <Employee>employee;
    private Collection<Module>module;

    @ManyToMany
    public Collection<Module> getModule() {
        return module;
    }

    public void setModule(Collection<Module> module) {
        this.module = module;
    }
}
```

RELATION N:M CAS UNIDIRECTIONNELLE(3)

Au niveau Sql : avec la plateforme eclipseLink: on a le résultat suivant:



RELATION N:M CAS BIDIRECTIONNELLE(1)



- l'entité cible porte également une relation @ManyToMany vers l'entité maître.
- Cette relation doit comporter un attribut mappedBy, qui indique le nom de la relation correspondante dans l'entité maître.
- L'entité maître porte une relation @ManyToMany vers l'entité cible
- la structure de tables de jointure générée est la même que dans le cas unidirectionnelle.
- Notons encore une fois que c'est la présence de l'attribut mappedBy qui crée le caractère bidirectionnel de la relation. Si l'on ne le met pas, alors JPA créera une seconde table de jointure.

RELATION N:M CAS BIDIRECTIONNELLE(2)

```
package com.esprit.jpa.testJPA.Entity;

import java.util.Collection;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Module {

    private int id_module;
    private String nom;
    private Collection<Formation> Formation;

    @ManyToMany(mappedBy="module")
    public Collection<Formation> getFormation() {
        return Formation;
    }

    public void setFormation(Collection<Formation> formation) {
        Formation = formation;
    }
}
```

RELATION N:M CAS BIDIRECTIONNELLE(3)


```
⊕ import java.util.Collection;

@Entity
@Table(name="formation")
public class Formation {

    private Long id; // Identifiant formation (Clé primaire)
    private String theme;
    private Collection <Employe> employe;
    private Collection<Module> module;

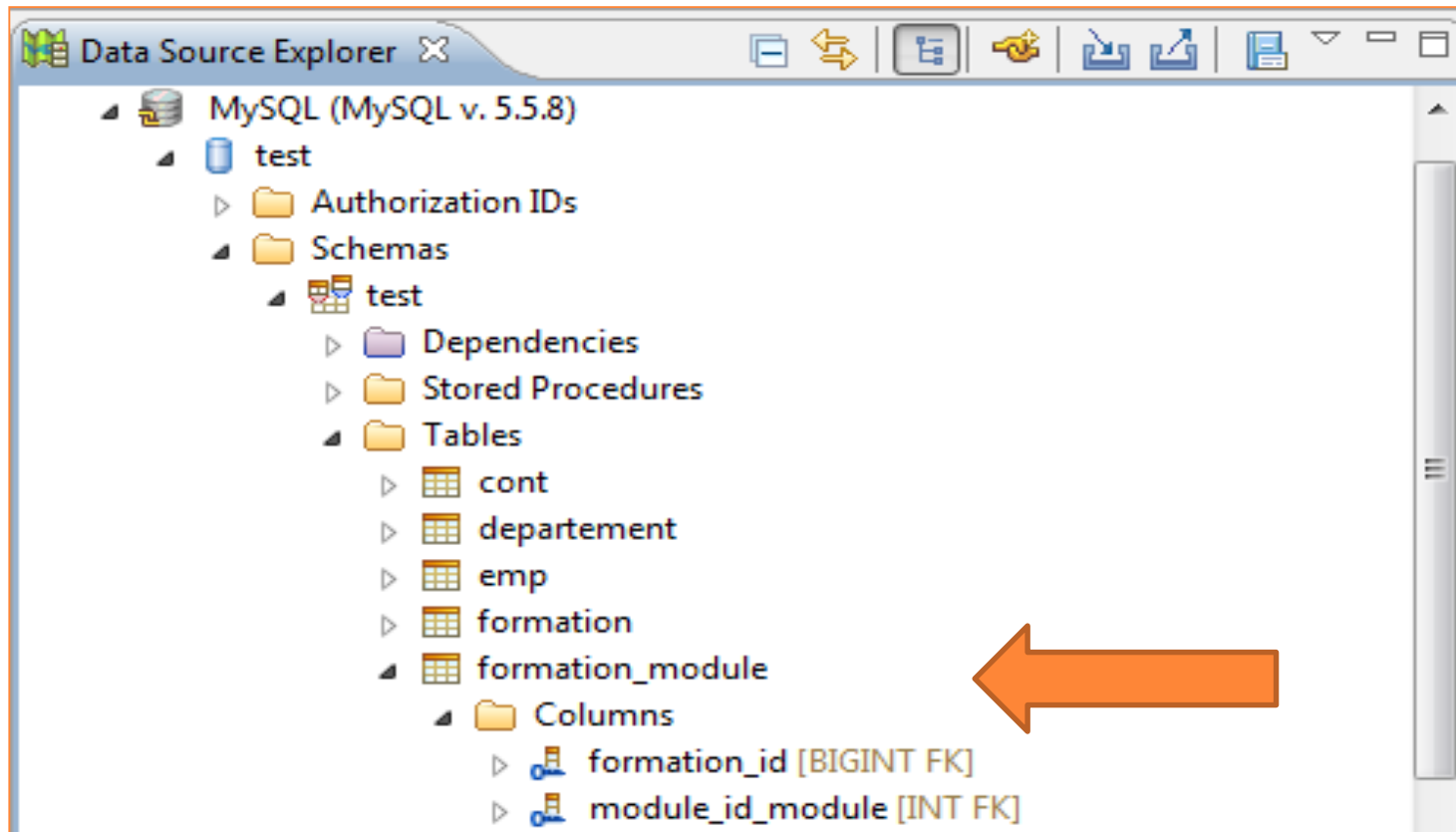
    @ManyToMany
    public Collection<Module> getModule() {
        return module;
    }

    public void setModule(Collection<Module> module) {
        this.module = module;
    }
}
```



RELATION N:M CAS BIDIRECTIONNELLE(3)

- Au niveau Sql : avec la plateforme eclipseLink: on a le résultat suivant:



RELATION N:M CAS BIDIRECTIONNELLE(4) :ENTITÉ ASSOCIATION

@Entity

```
public class Lignebon implements Serializable {  
    private static final long serialVersionUID = 1L;
```

@EmbeddedId

```
private LignebonPK id;
```

```
private int quantiteEndommage;
```

```
private int quantiteSortie;
```

```
private int quatiteRetourne;
```

```
//bi-directional many-to-one association to Article
```

```
@JoinColumn(name = "id_article", referencedColumnName = "idArticle",  
            insertable = false, updatable = false)
```

```
@ManyToOne(optional = false)
```

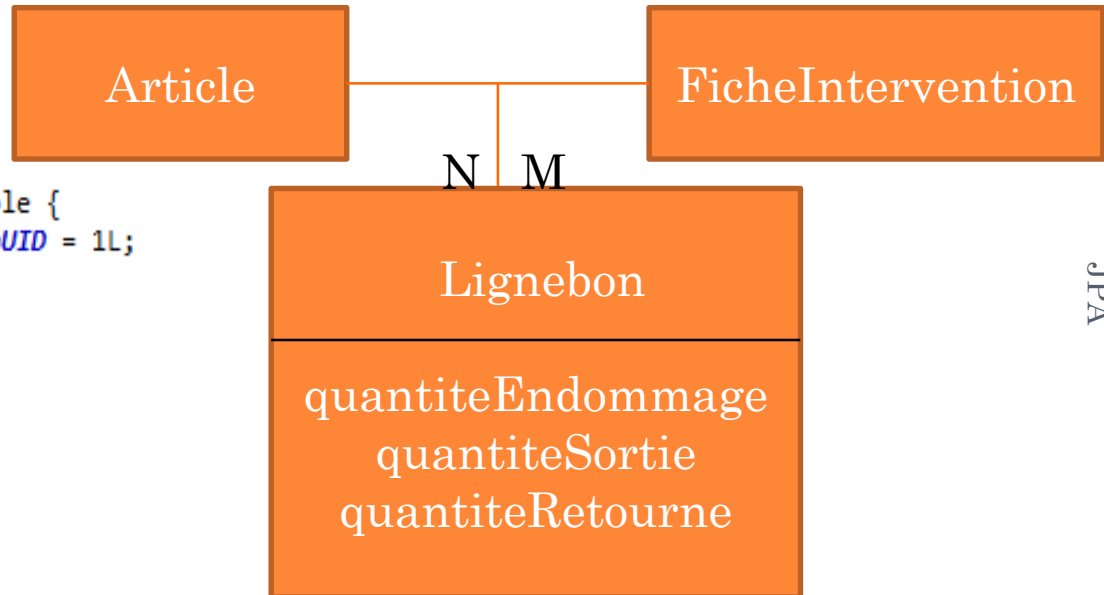
```
private Article article;
```

```
//bi-directional many-to-one association to FicheIntervention
```

```
@JoinColumn(name = "id_fiche", referencedColumnName = "idFiche",  
            insertable = false, updatable = false)
```

```
@ManyToOne(optional = false)
```

```
private FicheIntervention ficheIntervention;
```



JPA

RELATION N:M CAS BIDIRECTIONNELLE(4) :ENTITÉ ASSOCIATION

```
@Embeddable
public class LignebonPK implements Serializable {
    //default serial version id, required for serializable classes.
    private static final long serialVersionUID = 1L;

    @Column(name="id_fiche")
    private int idFiche;

    @Column(name="id_article")
    private int idArticle;

    public LignebonPK() {
    }
    public int getIdFiche() {
        return this.idFiche;
    }
    public void setIdFiche(int idFiche) {
        this.idFiche = idFiche;
    }
    public int getIdArticle() {
        return this.idArticle;
    }
    public void setIdArticle(int idArticle) {
        this.idArticle = idArticle;
    }
}
```

```
@Entity
@Table(name="articles")
public class Article implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)

    @Column(name="id_article")
    private int idArticle;

    private String codeUnite;

    private String marque;

    @Column(name="nom_article")
    private String nomArticle;

    @Column(name="nu_serie")
    private String nuSerie;

    private int quantiteEnStock;

    @Column(name="seuil_aprovisinnemnet")
    private int seuilAprovisinnemnet;

    //bi-directional many-to-one association to Lignebon
    @OneToMany(mappedBy="article")
    private List<Lignebon> lignebons;
```

RELATION N:M CAS BIDIRECTIONNELLE(4) :ENTITÉ ASSOCIATION

```
@Entity
@Table(name="fiche_intervention")
public class FicheIntervention implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)

    @Column(name="id_fiche")
    private int idFiche;

    private String commentaires;

    @Temporal(TemporalType.DATE)
    @Column(name="date_intervention")
    private Date dateIntervention;

    private String nomdist;

    private String numeroFicheInterv;

    private String typedepanne;
    //bi-directional many-to-one association to Lignebon
    @OneToMany(mappedBy="ficheIntervention")
    private List<Lignebon> lignebons;
```

HÉRITAGE: STRATÉGIE SINGLE_TABLE

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_CPTE",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("C")
```

```
public class Compte implements Serializable {
    @Id
    private String codeCompte;
    private Date dateCreation;
    private double solde;
    @ManyToOne
    @JoinColumn(name="CODE_CLI")
    private Client client;
    @ManyToOne
    @JoinColumn(name="CODE_EMP")
    private Employe employe;
    @OneToMany(mappedBy="compte")
    private Collection<Operation> operations;
```

```
@Entity
@DiscriminatorValue("CE")
public class CompteEpargne extends Compte {
    private double taux;
```

```
@Entity
@DiscriminatorValue("CC")
public class CompteCourant extends Compte {
    private double decouvert;
```

```
mysql> use banque5;
Database changed
mysql> select * from compte;
```

TYPE_CPTE	codeCompte	dateCreation	solde	taux	decouvert	CODE
CLI	CODE_EMP					
CC	CC1	2015-03-16 20:41:22	4000	NULL	8000	
1	2					
CE	CE1	2015-03-16 20:41:23	44000	5.5	NULL	
2	2					

```
2 rows in set (0.00 sec)
```


STRATÉGIE :MAPPEDSUPERCLASS

```
@MappedSuperclass
public abstract class Person {
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "PERSON_NAME")
    private String name;
```

```
@Entity
@Table(name = "TEACHER")
public class Teacher extends Person {

    @Column(name = "DEPT")
    private String department;
```

```
@Entity
@Table(name = "STUDENT")
public class Student extends Person {

    @Column(name = "BRANCH")
    private String branch;
```

```
Student std = new Student();
std.setName("salhi");
std.setId(1221);
std.setBranch("telecom");
Teacher teach= new Teacher();
teach.setId(11);
teach.setDepartment("TIC");
em.persist(std);
em.persist(teach);
```

```
mysql> use heritagepersonne;
Database changed
mysql> select * from student;
+----+-----+-----+
| ID | PERSON_NAME | BRANCH |
+----+-----+-----+
| 122 | salhi      | telecom |
+----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> select * from teacher;
+----+-----+-----+
| ID | PERSON_NAME | DEPT |
+----+-----+-----+
| 1 | NULL        | TIC |
+----+-----+-----+
1 row in set (0.00 sec)
```

STRATÉGIE : *JOINED_TABLE*

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "h2_utilisateur")
public abstract class Utilisateur {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Integer id;
    @Basic(optional = false)
    @Column(unique = true)
    private String login;
    @Basic(optional = false)
    private String password;
    private String email;
    private String nom;
    private String prenom;
```

```
@Entity
@Table(name = "h2_administrateur")
public class Administrateur extends Utilisateur {

    private String visibleEmail;
```

```
mysql> select * from h2_utilisateur;
+-----+-----+-----+-----+-----+-----+
| ID | DTYPE | PASSWORD | EMAIL | LOGIN | NOM |
+-----+-----+-----+-----+-----+-----+
| 2 | Administrateur | telecom2015 | rekik.rojdi@gmail.com | enit2015 | rekik rojdi |
+-----+-----+-----+-----+-----+-----+
```

```
@Entity
@Table(name = "h2_grimpeur")
public class Grimpeur extends Utilisateur {
    private int age;
    private int poids;
    private int taille;
    private String gender;
```

```
mysql> select * from h2_administrateur;
+-----+-----+
| ID | VISIBLEEMAIL |
+-----+-----+
| 2 | rekik.rojdi@topnet.tn |
+-----+-----+
```

AGENDA

- Présentation
- Les *mappings* de base
- Relations
- **EntityManager**
- Queries
- NamedQuery
- Transactions
- Demo
- Support et outils
- Questions

ENTITY MANAGER: LE CŒUR DE LA JPA

- Cet objet est central dans cette API. Ainsi, toute entité JPA persistante possède une référence sur l'*entity manager* qui l'a créée ou qui a permis de la retrouver, et un *entity manager* connaît toutes les entités JPA qu'il gère.
- Sous Java SE, pour obtenir une instance de type EntityManager, il faut utiliser une fabrique de type EntityManagerFactory. Cette fabrique propose la méthode createEntityManager() pour obtenir une instance

```
EntityManagerFactory emf= Persistence.createEntityManagerFactory("testJPA");  
EntityManager em=emf.createEntityManager();
```

ENTITY MANAGER

Toutes les opérations que l'on peut faire sur des entités JPA passent directement ou indirectement par l' *entity manager* . Cet objet est central dans cette API.

- Les spécifications JPA définissent cinq opérations sur une entité :

PERSIST

REMOVE

REFRESH

DETACH

MERGE.

Ces opérations correspondent à autant de méthodes sur l'objet EntityManager.

ENTITY MANAGER: OPERATIONS

- **Persist**: a pour effet de rendre une entité persistante
`em.persist(Object);`
- **Remove**: a pour effet de rendre une entité non persistante
`em.remove(Object)`
- **Refresh**: ne s'applique qu'aux entités persistantes. Si l'entité passée en paramètre n'est pas persistante, alors une exception de type `IllegalArgumentException` est générée.
`em.refresh(Object)`
- **Detach**: a pour effet de la détacher de l' entity manager qui la gère. Cette entité persistante ne sera donc pas prise en compte lors du prochain commit
`em.detach(Object)`
- **Merge**: attache l'entité à l' entity manager courant. On utilise cette opération pour associer une entité à un autre entity manager que celui qui a été utilisé pour la créer ou la lire.
`em.merge(Object)`

ENTITY MANAGER: COMPORTEMENT CASCADE

- Le comportement *cascade* consiste à spécifier ce qui se passe pour une entité en relation d'une entité père (que cette relation soit monovaluée ou multivaluée), lorsque cette entité père subit une des opérations de l'EntityManager(Persist,...)
- Le comportement *cascade* est précisé par l'attribut **cascade**, disponible sur les annotations : **@OneToOne**, **@OneToMany** et **@ManyToMany**.
- La valeur de cet attribut est une énumération de type CascadeType. En plus des valeurs DETACH, MERGE, PERSIST, REMOVE, REFRESH, cette énumération définit la valeur **ALL**, qui correspond à toutes les valeurs à la fois.

```
...  
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})  
@JoinColumn(name="id_contrat")  
public Contrat getContrat() {  
    return contrat;  
}
```

ENTITY MANAGER: COMPORTEMENT FETCH(1)

- L'attribut fetch d'une association permet d'indiquer une récupération immédiate des entités associées (**FetchType.EAGER**) ou une récupération retardée (**FetchType.LAZY**) si le comportement par défaut ne convient pas
- Pour choisir entre le chargement de type outer-join ou le chargement par select successifs - optionnel par défaut à select.

```
@OneToMany(mappedBy="form", fetch=FetchType.EAGER)  
public Collection<Employee> getEmployee() {  
    return employee;  
}
```

```
$ EAGER: FetchType - FetchType  
$ LAZY: FetchType - FetchType  
$ class: Class<javax.persistence.FetchType>
```


ENTITY MANAGER: COMPORTEMENT FETCH(2)

- Le mode de récupération par défaut est le mode EAGER pour les attributs (ils sont chargés en même temps que l'entité)
- Si un attribut est d'un type de grande dimension (LOB), il peut aussi être marqué par `@Basic(fetch=FetchType.LAZY)` (à utiliser avec parcimonie) Cette annotation n'est qu'une suggestion au GE, qu'il peut ne pas suivre.

ENTITY MANAGER: CHERCHER PAR IDENTITÉ

Find et getReference (de EntityManager) permettent de retrouver une entité en donnant son identité dans la BD. Elles prennent 2 paramètres de type:

- – Class<T> pour indiquer le type de l'entité recherchée (le résultat renvoyé sera de cette classe ou d'une sous-classe)
- – Object pour indiquer la clé primaire.

getReference renvoie une référence vers une entité, sans que cette entité ne soit nécessairement initialisée.



Le plus souvent il vaut mieux utiliser **find**.

ENTITY MANAGER:GETREFERENCE

- `getReference` peut être (rarement) utilisée pour améliorer les performances quand une entité non initialisée peut être suffisante, sans que l'entité entière soit retrouvée dans la base de données.
- *Par exemple*, pour indiquer une association dont le but est unique (OneToOne ou ManyToOne) :

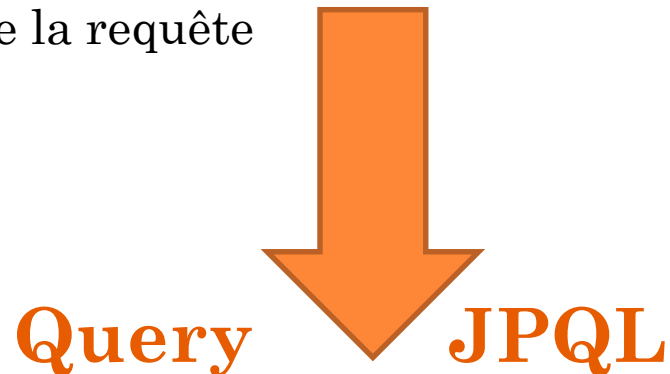
```
Departement dept =  
em.getReference(Departement.class, 10);  
Employe emp.setDepartement(dept);
```

ENTITYMANGAER:OPERATIONS-PRÉDÉFINI

Souvent il est nécessaire de rechercher des données sur des critères plus complexes que la simple identité.

Les étapes sont alors les suivantes :

- 1. Décrire ce qui est recherché (langage JPQL)
- 2. Créer une instance de type Query
- 3. Initialiser la requête (paramètres, pagination)
- 4. Lancer l'exécution de la requête



AGENDA

- Présentation
- Configuration
- Les mappings de base
- Relations
- EntityManager
- **Queries**
- NamedQuery
- Transactions
- Demo
- Support et outils
- Questions

QUERIES: LANGUAGE JPQL

- La JPA introduit le JPA-QL, qui est, tout comme le EJBQL ou encore le HQL, un langage de requête du modèle objet, basé sur SQL.
- Le langage JPQL (Java Persistence QueryLanguage) permet de décrire ce que l'application recherche
- Il ressemble beaucoup à SQL.

Exemple de Requête:

```
select e from Employe as e.
```

```
select e.nom, e.salaire from Employe e.
```

```
select e from Employe e where e.departement.nom = 'Direction'
```

REQUÊTES SUR LES ENTITÉS « OBJET »

- Les requêtes travaillent avec le modèle objet et pas avec le modèle relationnel
- Les identificateurs désignent les classes et leurs propriétés et pas les tables et leurs colonnes
- Les seules classes qui peuvent être explicitement désignées dans une requête (clause from) sont les entités.
- Le texte des requêtes utilise beaucoup les alias de classe.
- Les propriétés des classes doivent être préfixées par les alias.
- Une erreur fréquente du débutant est d'oublier les alias en préfixe

TYPE DU RÉSULTAT(1)

Un select peut renvoyer:

- une (ou plusieurs) expression « entité », par exemple un employé
- une (ou plusieurs) expression « valeur », par exemple le nom et le salaire d'un employé.

Nombre d'éléments du résultats:

-Single: On utilise la méthode (Object getSingleResult()), Pour le cas où une seule valeur ou entité est renvoyée.

-Plusieurs: On utilise la méthode (java.util.List getResultList()), Pour le cas où une plusieurs valeurs ou entités peuvent être renvoyées

Attention:

Un message d'avertissement sera affiché durant la compilation si le résultat est rangé dans une liste générique (Liste<Employe> par exemple

TYPE DU RÉSULTAT(2)

- Le type des éléments de la liste est `Object` si la clause `select` ne comporte qu'une seule expression, ou `Object[]` si elle comporte plusieurs expressions.

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

INTERFACE QUERY

- Représente une requête.
- Une instance de Query est obtenue par les méthodes:
- **createQuery,**
- **createNativeQuery**
- **ou createNamedQuery de l'interface EntityManager**

INTERFACE QUERY:MÉTHODES DE QUERY

- List getResultList()
- Object getSingleResult()
- int executeUpdate()
- Query setMaxResults(int nbResultats)
- Query setFirstResult(int positionDepart)

INTERFACE QUERY: NATIVEQUERIES

Une façon de faire des requête en SQL natif. Sert principalement à avoir plus de contrôle sur les requêtes à la base de donnée.

```
public class MyService {  
    public void myMethod() {  
        ...  
        List results  
            = em.createNativeQuery("SELECT * FROM MyPojo", MyPojo.class)  
                .getResultList();  
        ...  
    }  
}
```

AGENDA

- Présentation
- Configuration
- Les mappings de base
- Relations
- EntityManager
- Queries
- **NamedQuery**
- Transactions
- Demo
- Support et outils
- Questions

NAMEDQUERIES

On peut sauvegarder des gabarits de requête dans nos entités. C'est ce qu'on appelle une *NamedQuery*. Ceci permet :

- La réutilisation de la requête
- D'externaliser les requête du code.

```
@Entity
@NamedQuery(name="myQuery", query="Select o from MyPojo o")
public class MyPojo { ... }

public class MyService {
    public void myMethod() {
        ...
        List results = em.createNamedQuery("myQuery").getResultList();
        ...
    }
}
```

AGENDA

- Présentation
- Configuration
- Les mappings de base
- Relations
- EntityManager
- Queries
- NamedQuery
- **Transactions**
- Demo
- Support et outils
- Questions

TRANSACTIONS


2 façons de mettre en place les transactions:

- JTA
 - En utilisant la Java Transaction API, typiquement *in-container*
- Resource-local
 - En utilisant le modèle de transaction du persistence manager

TRANSACTIONS: RESOURCE-LOCAL

- Toutes les opérations de création, effacement ou modification doivent nécessairement se faire dans une transaction. Une transaction démarre sur appel à la méthode `begin()` de la transaction dans laquelle on se trouve, et est validée sur appel à sa méthode `commit()`. On peut annuler une transaction par appel à sa méthode `rollback()`.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">  
  <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">  
    <non-jta-data-source>amce</non-jta-data-source>  
  </persistence-unit>  
</persistence>
```



```
EntityManager em = createEntityManager();  
em.getTransaction().begin();  
Employee employee = em.find(Employee.class, id);  
employee.setSalary(employee.getSalary() + 1000);  
em.getTransaction().commit();  
em.close();
```

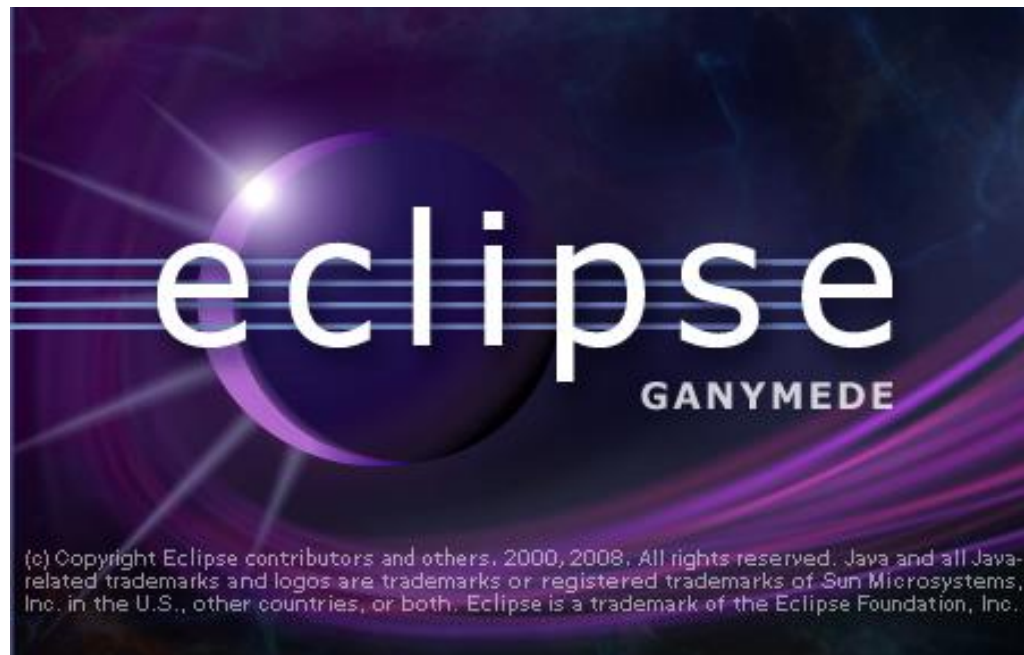
AGENDA

- Présentation
- Configuration
- Les mappings de base
- Relations
- EntityManager
- Queries
- NamedQuery
- Transactions
- **Demo**
- Support et outils
- Questions

DEMO

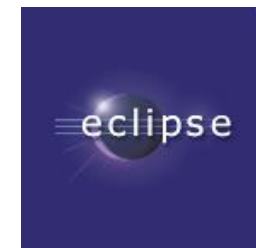
Amusons-nous tous ensemble avec:

- Eclipse
- MySQL (5.0)
- Application: Mise en place d'un Logiciel d'integration d'un nouveau Employé dans une société



SUPPORT DE L'INDUSTRIE

La JPA a fait consensus à JavaOne2006, tous les grands acteurs y trouvent leur compte, et jusqu'à preuve du contraire, vont supporter et endosser cette technologie.





Questions?

