

OR-Mapping with JPA

Steve Hostettler

Software Modeling and Verification Group
University of Geneva



Persistence des données



Persistence des données?

- A la fin d'une session utilisateurs les objets en mémoire sont perdus.
- Rendre un objet ou une donnée persistente c'est la sauvegarder sur un support non-volatile pour le réutiliser lors d'une session ultérieure.



Comment persister des données?

- Sur disque durs sous forme de fichiers
- Sur le cloud
- Dans une base de données (SGBD)



Qu'est qu'un SGBD?

- Système de Gestion de Base de Données
- Modèle d'organisation des données
- Objets
- Relationnelles
- Distribution
- Embarquées, distribuées, centralisées



Base de Données Relationnelles



SGBD relationnelles vs SGBD objet

- SGBD Relationnelles
- Très populaires / beaucoup d'existant
- Très performantes pour l'OLTP
- Théorie solide, norme reconnues
- Moins riche (pas d'héritage, références...)



SGBD relationnelles vs SGBD objet

- SGBD Objets
- Très adaptés au monde objet
- Peu usitées
- Manque de normes

SGBD Relationnel

- Paradigmes entités-relations
- Entités = Tables
- Relations = Many 2 One, One 2 Many,



JDBC



JDBC

1. Java Data Base Connectivity

1. Standard library to access a database

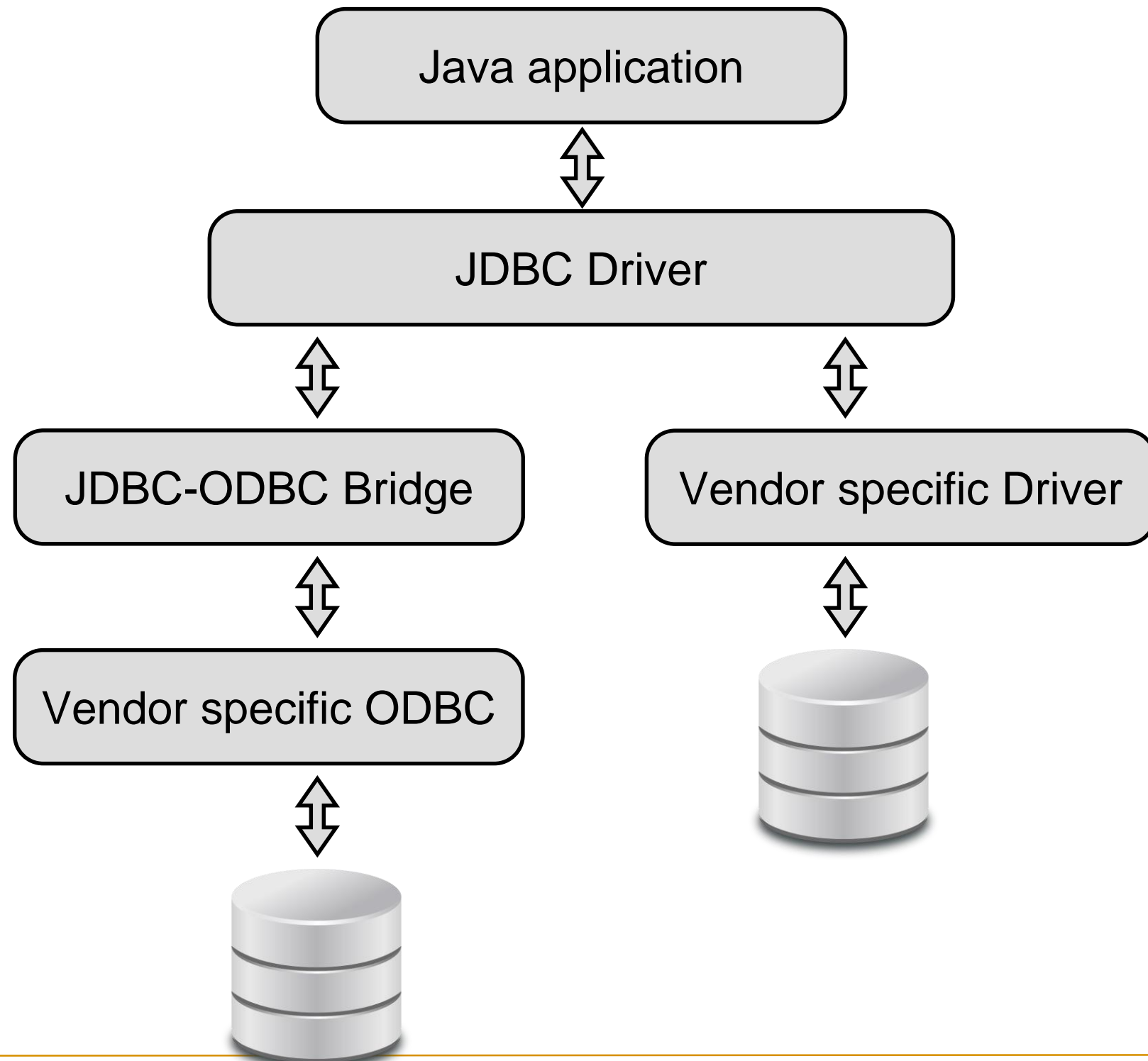
1. Connect a DB

2. Creation of the queries

3. `java.sql.*` packages



JDBC



JDBC How to

1. Charger le driver
2. Définir l'URL de la base de données
3. Etablir la connexion
4. Créer un Statement
5. Executer une "Query"
6. Traiter les résultats
7. Fermer la connexion



Charger le driver

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
} catch (ClassNotFoundException cnfe) {  
    System.out.println("Error loading driver: " cnfe);  
}
```

Pas nécessaire en Java 1.6 uniquement Java
1.5 et précédent



Définir l'URL

Format:

`jdbc:vendor://dbhost:port/database`

`jdbc:mysql://localhost:3008/test`



Etablir la connexion

```
String JDBC_DRIVER = "com.mysql.jdbc.Driver";  
String DB_URL = "jdbc:mysql://localhost:3306/Students_DB";  
  
String USER = "root";  
String PASS = "";  
Connection conn = null;  
Statement stmt = null;  
  
Class.forName(JDBC_DRIVER);  
  
System.out.println("Connecting to database...");  
conn = DriverManager.getConnection(DB_URL, USER, PASS);
```



Charger des informations sur la DB

```
DatabaseMetaData dbMetaData = conn.getMetaData();
```

```
String productName =  
dbMetaData.getDatabaseProductName();
```

```
System.out.println("Database: " + productName);
```

```
String productVersion =  
dbMetaData.getDatabaseProductVersion();
```

```
System.out.println("Version: " + productVersion);
```



Etablir un statement

Un “Statement” envoi une commande ou une requête au SGBD

```
Statement statement =  
    connection.createStatement()
```



Executer une requête

```
statement.executeQuery("SELECT ... FROM ...");
```

```
statement.executeUpdate("UPDATE ...");
```

```
statement.executeUpdate("INSERT ...");
```

```
statement.executeUpdate("DELETE...");
```

```
statement.execute("CREATE TABLE...");
```

```
statement.execute("DROP TABLE ...")
```



Traiter le résultat

```
ResultSet rs = stmt.executeQuery(sql);

while (rs.next()) {

    int id = rs.getInt("id");
    Date birthdate = rs.getDate("birth_date");

    System.out.print("ID: " + id);
    System.out.println(", Birthdate: " + birthdate);
}
```

Gestion des exceptions

Il FAUT toujours fermer les “statements” et les “connections” après utilisation. Sinon on risque les “connection leaks”

Mais comment le faire proprement dans le cas d'une exceptions?



Gestion des exceptions

- Dans le cas d'une exception il faut fermer les connexions mais...
- La fermeture d'une connexions peut entrainer ... une exception
- C'est aussi vrai pour la fermeture d'un statement ou d'un resultSet
- Ce qui nous donne une gestion des erreurs très lourde



Problème majeur n°1 : la gestion des erreurs

```
String JDBC_DRIVER = "com.mysql.jdbc.Driver";  
String DB_URL = "jdbc:mysql://localhost:3306/Students_DB";
```

```
String USER = "root";  
String PASS = "";  
Connection conn = null;  
Statement stmt = null;
```

```
try {  
    // STEP 2: Register JDBC driver  
    Class.forName("com.mysql.jdbc.Driver");  
    // STEP 3: Open a connection  
    System.out.println("Connecting to database...");  
    conn = DriverManager.getConnection(DB_URL, USER, PASS);  
    // STEP 4: Execute a query  
    System.out.println("Creating statement...");  
    stmt = conn.createStatement();  
    String sql;  
    sql = "SELECT id, first_name, last_name, birth_date FROM students";  
    ResultSet rs = stmt.executeQuery(sql);
```



Problème majeur n°1 : la gestion des erreurs

```
while (rs.next()) {  
    // Retrieve by column name  
  
    int id = rs.getInt("id");  
    Date birthdate = rs.getDate("birth_date");  
    String firstname = rs.getString("first_name");  
    String lastname = rs.getString("last_name");  
    // Display values  
    System.out.print("ID: " + id);  
    System.out.print(", First Name: " + firstname);  
    System.out.print(", Last Name: " + lastname);  
    System.out.println(", Birthdate: " + birthdate);  
}  
// STEP 6: Clean-up environment  
rs.close();  
stmt.close();  
conn.close();
```



Problème majeur n°1 : la gestion des erreurs

```
} catch (SQLException se) {  
    // Handle errors for JDBC  
    se.printStackTrace();  
} catch (Exception e) {  
    // Handle errors for Class.forName  
    e.printStackTrace();  
} finally {  
    // finally block used to close resources  
    try {  
        if (stmt != null) { stmt.close(); }  
    } catch (SQLException se2) {  
        se2.printStackTrace();  
    } // nothing we can do  
    try {  
        if (conn != null) { conn.close(); }  
    } catch (SQLException se) {  
        se.printStackTrace();  
    } // end finally try  
} // end try
```



Problème majeur n°2

JDBC repose sur la création de chaînes de caractères:

Pas de vérification de syntaxe

Pas de vérification de type

Très répétitif



Problème majeur n°3

Pas de méta-données sur la structure dans le code:

Le programme et la structure de DB
peuvent être désynchronisés



Transactions

ACID

ATOMIC: tout ou rien

COHERENCE: pas de contraintes non respectées

ISOLATION: la transaction est isolée des autres

DURABILITY: une fois que c'est fait... c'est fait



Transactions

Désactivation de l'auto-commit

```
conn.setAutoCommit(false);
```

Persister les changements : fin de transaction

```
conn.commit();
```

Annuler les changements : fin de transaction

```
conn.rollback();
```



Requête paramétrées

Prépare une partie du travail: pas de compilation

Gain de performances:

Très utile pour les boucles de recherche



Requête paramétrées

```
sql = "SELECT id, first_name, last_name, birth_date "  
      + "FROM students WHERE last_name = ?";  
PreparedStatement stmt = conn.prepareStatement(sql);  
  
stmt.setString(1, "Barrett");  
ResultSet rs = stmt.executeQuery();
```



JDBC : Conclusion

- Très verbeux
- Difficile de tout faire correctement
- Gestion des exceptions
- fermeture des connexions, statement, ...
- Comment faire cohabiter le monde objet et le monde relationnel?



Patron de conception

Directeur-Monteur

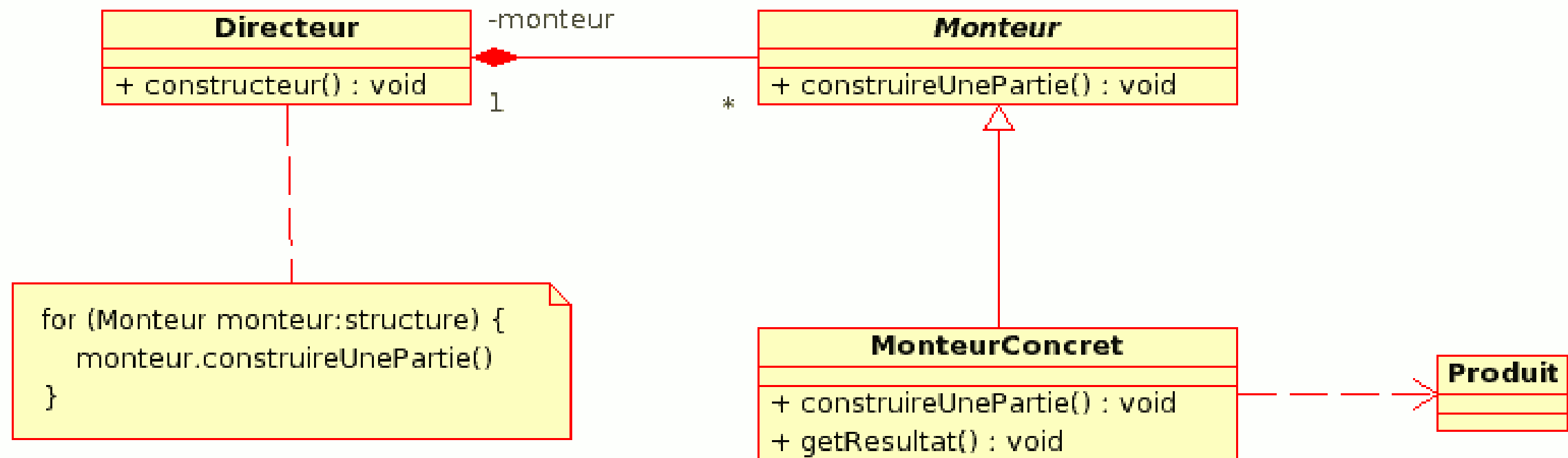


Directeur-Monteur

- Patron de conception
- Création d'objets à partir d'une "source"
- Séparer la construction de l'objet de sa représentation



Directeur-Monteur



source:wikipedia

Bibliographie

http://www.tutorialspoint.com/jdbc/jdbc_tutorial.pdf

<http://java.sun.com/developer/Books/JDBCTutorial/>



JPA



JPA : un peu d'histoire

- 1998 EJB 1.0 : Container Management Persistence
- Ne fonctionnait que dans un container
- Très difficile à mettre en oeuvre
- Très difficile à tester
- Mapping très limité
- 2001 - EJB 2.0 / JDO 1.0
- 2002 - EJB 2.1
- Hibernate propose une véritable alternative au EJB CMP 2.x
- 2005 JPA 1.0 (intégré à EJB 3.0 - JEE 1.5)
- Basé sur l'expérience d'Hibernate
- JPA 2.0 (JEE 1.6)
- Eclipse Link est l'implémentation de référence



Concepts de base



Configuration par l'exception

- Différent de l'approche pré-JEE5
- Toute la configuration est assumée connue par conventions.
- Seules les exceptions aux conventions sont configurées.
- La configuration peut être exprimée soit:
 - par des fichiers XML
 - par des annotations
- En cas de conflit XML-Annotations c'est le XML qui a la priorité.



XML vs Annotations

- Annotations
- + Peu verbeux
- + Intuitif
- - Modèle “pollué” par des informations de persistences
- Descripteurs XML
- + Séparation des responsabilités
- - Très verbeux
- Nous utiliserons la configuration par annotations



O-R Mapping

- Object-Relational Mapping
- Evite l'implémentation du patron de conception Directeur-Monteur
- Offre une vue OO transparent de la base de données
- Meta-données associées aux objets du modèle pour faire l'association avec la base de données
- Nom de table
- Nom de colonne
- Taille de colonne
- Nom des clés étrangères



Dépendances MAVEN

```
<dependency>  
  <groupId>org.eclipse.persistence</groupId>  
  <artifactId>javax.persistence</artifactId>  
  <version>2.0.0</version>  
</dependency>
```

API JPA

```
<dependency>  
  <groupId>org.eclipse.persistence</groupId>  
  <artifactId>eclipselink</artifactId>  
  <version>2.0.0</version>  
  <scope>compile</scope>  
</dependency>  
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>5.1.18</version>  
  <scope>runtime</scope>  
</dependency>
```

Dépendances MAVEN

```
<dependency>  
  <groupId>org.eclipse.persistence</groupId>  
  <artifactId>javax.persistence</artifactId>  
  <version>2.0.0</version>  
</dependency>
```

```
<dependency>  
  <groupId>org.eclipse.persistence</groupId>  
  <artifactId>eclipselink</artifactId>  
  <version>2.0.0</version>  
  <scope>compile</scope>  
</dependency>
```

```
<dependency>
```

```
  <groupId>mysql</groupId>
```

```
  <artifactId>mysql-connector-java</artifactId>
```

```
  <version>5.1.18</version>
```

```
  <scope>runtime</scope>
```

```
</dependency>
```

Implémentation JPA



Dépendances MAVEN

```
<dependency>  
  <groupId>org.eclipse.persistence</groupId>  
  <artifactId>javax.persistence</artifactId>  
  <version>2.0.0</version>  
</dependency>
```

```
<dependency>  
  <groupId>org.eclipse.persistence</groupId>  
  <artifactId>eclipselink</artifactId>  
  <version>2.0.0</version>  
  <scope>compile</scope>  
</dependency>
```

```
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>5.1.18</version>  
  <scope>runtime</scope>  
</dependency>
```

Driver base de données



Utilisation de JPA

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY" />
      <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:memory:StudentsDB;create=true" />
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```



Utilisation de JPA

META-INF/persistence.xml

Nom de l'unité de persistence: entity manager

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY" />
      <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:memory:StudentsDB;create=true" />
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```



Utilisation de JPA

META-INF/persistence.xml

Classes gérés par JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY" />
      <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:memory:StudentsDB;create=true" />
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```



Utilisation de JPA

META-INF/persistence.xml

Nom de la DB : utile
pour les optimisations

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY" />
      <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:memory:StudentsDB;create=true" />
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```



Utilisation de JPA

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  <persistence-unit name="Persistence" transaction-type="LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY" />
      <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:memory:StudentsDB;create=true" />
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```

Classe du pilote de DB

Utilisation de JPA

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database" value="Derby" />
      <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:memory:StudentsDB;create=true" />
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```

Base de données en mémoire
(utile pour tester)



Utilisation de JPA

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database">
        <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver" />
        <property name="javax.persistence.jdbc.url" value="jdbc:derby:memory:StudentsDB;create=true" />
        <property name="javax.persistence.jdbc.user" value="" />
        <property name="javax.persistence.jdbc.password" value="" />
      </properties>
    </persistence-unit>
  </persistence>
```

URL de la DB

Utilisation de JPA

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Persistence" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <properties>
      <property name="eclipselink.target-database">
        <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver" />
        <property name="javax.persistence.jdbc.url" value="jdbc:derby:memory:StudentsDB;create=true" />
        <property name="javax.persistence.jdbc.user" value="" />
        <property name="javax.persistence.jdbc.password" value="" />
      </properties>
    </persistence-unit>
  </persistence>
```

Sécurité de la DB



Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSerializableField;

    @Transient
    private Long mComputedField;

    ...
}
```

Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class ... Serializable {

    /** The serial number */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSerializableField;

    @Transient
    private Long mComputedField;

    ...
}
```

Géré par JPA

Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = 1L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSerializableField;

    @Transient
    private Long mComputedField;

    ...
}
```

Table qui contient les données. Si non indiquée alors le nom de la table est "STUDENT" par défaut.

Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {


    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSerializableField;

    @Transient
    private Long mComputedField;

    ...
}
```



Champs identité (unique)

Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSer

    @Transient
    private Long mComputedField;

    ...
}
```

Nom de la colonne.
Par défaut "MID".



Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSer

    @Transient
    private Long mComputedField;

    ...
}
```

Colonne dont le contenu est générée
automatiquement par la DB



Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSerializableField;

    @Transient
    private Long mComputedField;

    ...
}
```

Non sérialisé et
non persisté dans
la DB

Utilisation de JPA

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    private transient Long mNonSerializedField;

    @Transient
    private Long mComputedField;

    ...
}
```

Sérialisé mais non
persisté dans la DB

Utilisation de JPA

```
// Get the entity manager for persistence
EntityManagerFactory mEmf = Persistence.createEntityManagerFactory("Persistence");
EntityManager mEntityManager = mEmf.createEntityManager();
mEntityManager.getTransaction().begin();
mEntityManager.persist(student);
mEntityManager.getTransaction().commit();
mEntityManager.close();
mEmf.close();
```





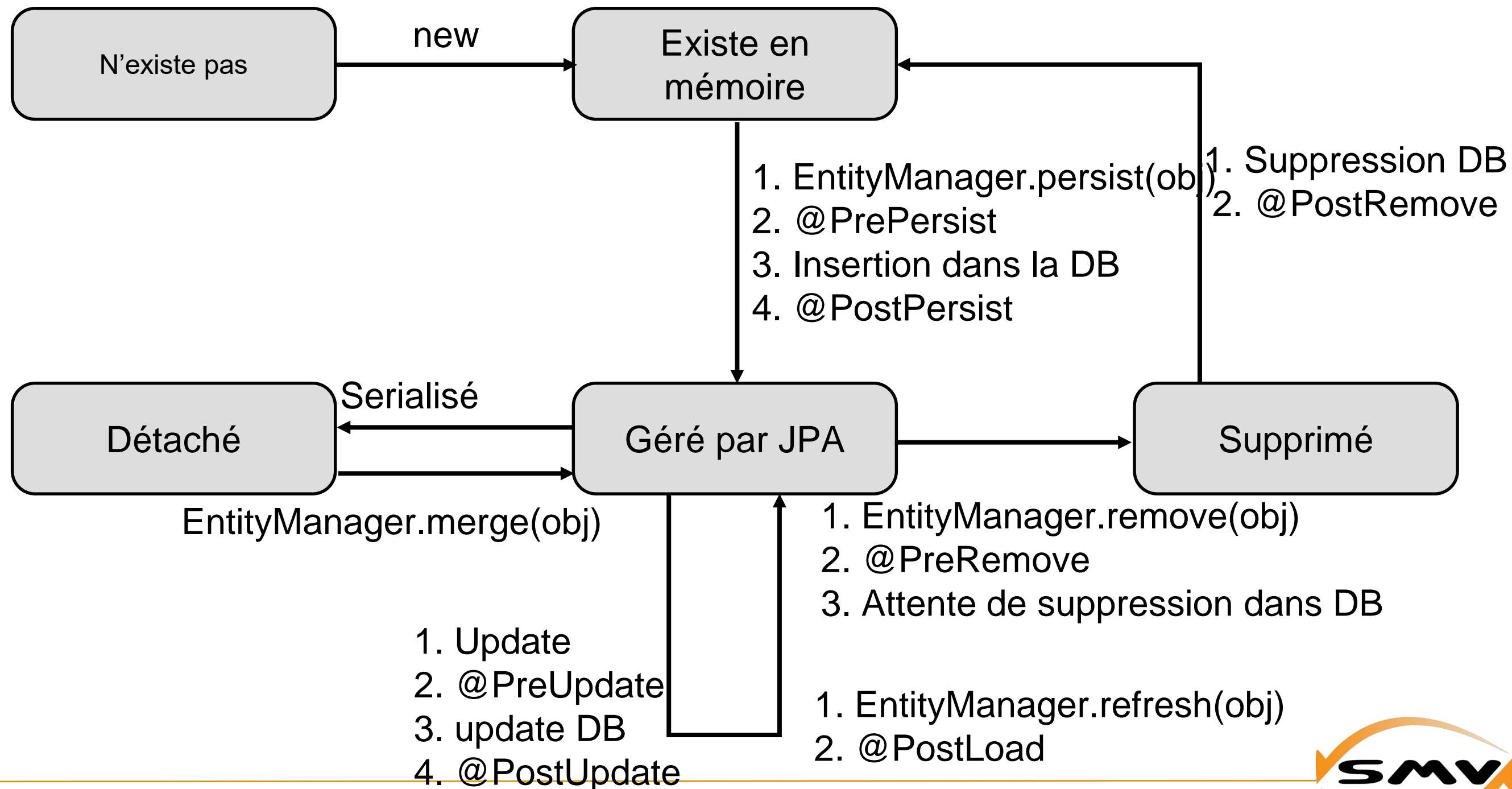
3) Ecrire un programme qui utilise JPA pour faire la même chose que 1)

Méthodes de rappel et écouteurs

- Un objet du modèle est un simple objet java tant qu'il n'est pas géré par le conteneur de persistance.
- La gestion d'un objet par JPA donne lieu à un cycle de vie avec lequel il est possible d'interagir
- Equivalent à des "triggers"



Méthodes de rappel et écouteurs



Méthodes de rappel et écouteurs

Un objet du modèle est un simple objet java tant qu'il n'est pas géré par le conteneur de persistance

La gestion d'un objet par JPA donne lieu à un cycle de vie avec lequel il est possible d'interagir

Equivalent à des “triggers”



Gestion des conflits de transactions

-

Risques de manque d'isolation

BAD_GUYS

ID	FIRST_NAME	LAST_NAME
1	Joe	Dalton
2	Averell	Dalton
3	William	Dalton
4	Jack	Dalton

Gestion des conflits de transactions

-

Risques de manque d'isolation

select LAST_NAME
from BAD_GUYS
where id = 1

select LAST_NAME
from BAD_GUYS
where id = 1

#Trx 1

LAST_NAME
=
DALTON

update BAD_GUYS
set LAST_NAME='CAPONE'
where id = 1

LAST_NAME
=
CAPONE

rollback

#Trx 2

DIRTY READS



Gestion des conflits de transactions

-

Risques de manque d'isolation

select LAST_NAME
from BAD_GUYS
where id = 1

select LAST_NAME
from BAD_GUYS
where id = 1

#Trx 1

LAST_NAME
=
DALTON

update BAD_GUYS
set LAST_NAME='CAPONE'
where id = 1;
commit;

LAST_NAME
=
CAPONE

#Trx 2

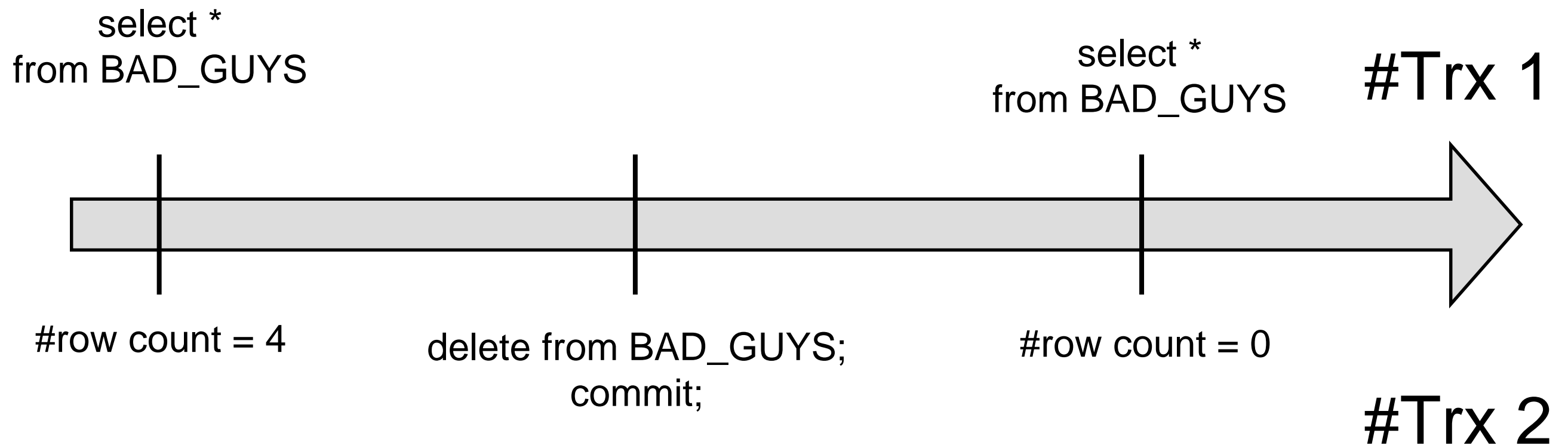
Non-repeatable READS



Gestion des conflits de transactions

-

Risques de manque d'isolation



Phantom READS



Gestion des conflits de transactions

-

Risques de manque d'isolation

- Lecture sale (dirty read), lecture pendant la mise à jour d'une donnée par une transaction concurrente avant la garantie de cohérence. Pire il y a risque de rollback
- Lecture non-reproductible, lecture de deux valeurs différentes pendant la même requête. Par exemple si une transaction concurrente à mis à jour les données pendant ce temps.
- Lecture fantôme, lecture de donnée en évolution



Niveaux d'isolation

- Serializable: soit un principe de verrous sur les données lues et modifiées. ATTENTION les “select *” provoque des “range locks” pour éviter les lectures fantômes; soit un principe de détection des collisions
- Repeatable Reads: idem que précédemment mais sans les “range locks”.
- Read committed: uniquement des verrous d'écriture
- Read uncommitted: aucune sécurité



Niveaux d'isolation

Isolation level	Dirty Reads	Non-repeatable reads	Phantoms
Read Uncommitted			
Read committed			
Repeatable Reads			
Serializable			

Niveaux d'isolation

- Serializable: 100% ACID mais problème de performance
- Repeatable Reads: Ecriture et relecture des mêmes informations au cours d'une même transaction
- Read committed: Niveau d'isolation par défaut de JPA. Très bien pour la production de rapport : lecture seule
- Read uncommitted: performant mais dangereux



Locking

- Gestion des conflits entre les transactions:
- Optimistic Locking
- Pessimist Locking
- Garantir la cohérence de la base de données
- Peut être très gourmand en ressources

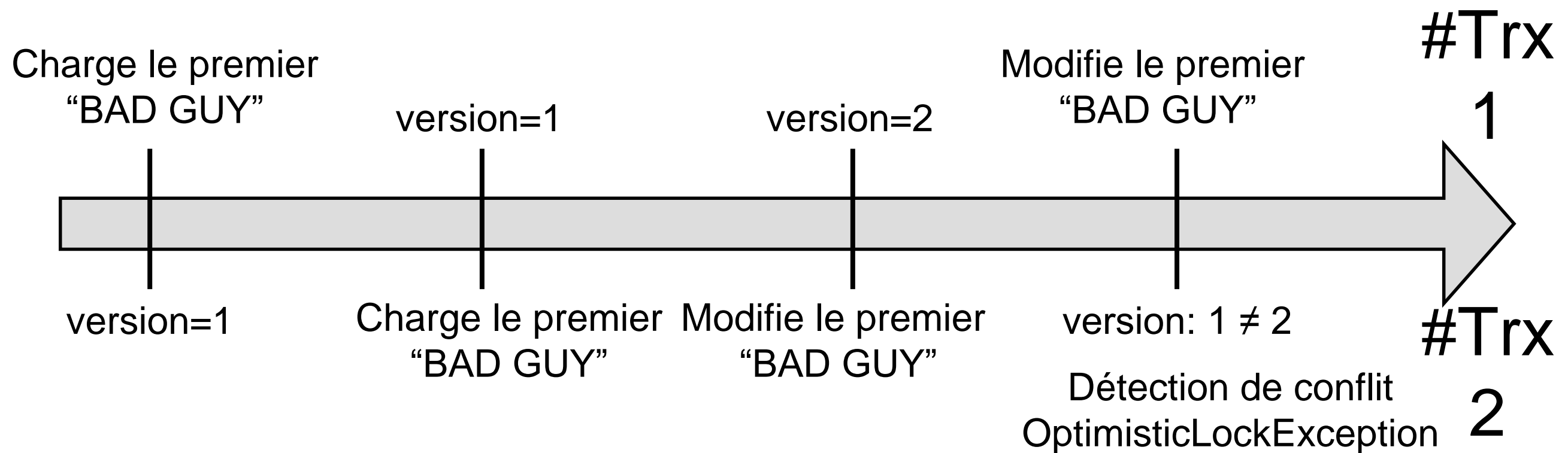


Optimistic Locking

- Optimistic Locking
- a.k.a. Optimistic concurrency checking
- a.k.a. Optimistic Concurrency Control
- Assume que la grande partie des transactions ne seront pas sur les même données au même moment
- Peu couteux en ressources de la base de données si le nombre de conflits restent restraints.
- Permet de ne pas devoir garder la même transaction



Optimistic Locking



Optimistic Locking

- Nécessite un numéro de version
- Avant le commit, le numéro de version est vérifié pour être certain qu'aucune autre transaction n'a eu lieu.

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {
    ...
    /** The version id. */
    @Version
    private Long mVersion;
    ...
}
```



Optimistic Locking

- Pièges à éviter
- Ne pas gérer automatiquement les “OptimisticLockException”
- Ne pas être paranoïac, une erreur peut être toléré par l'utilisateur si ses données ne sont pas compromises et si il en est informé
- Données modifiées par une application qui ignore le mécanisme
- Cela ne marche que si vous pouvez rajouter le champs version : problème avec le “legacy”



Mise en place de JPA



Comment démarrer JPA dans un test unitaire?

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>2.0.0</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>javax.persistence</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.18</version>
  <scope>runtime</scope>
</dependency>
```



Comment démarrer JPA dans un test unitaire?

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="JEE6Demo-Persistence"
    transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.demo.dom.Student</class>
    <class>ch.demo.dom.Grade</class>
    <properties>
      <property name="eclipselink.target-database" value="MYSQL" />
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/Students_DE
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
      <property name="eclipselink.ddl-generation" value="create-tables" />
      <property name="eclipselink.ddl-generation.output-mode" value="both"/>
      <property name="eclipselink.create-ddl-jdbc-file-name" value="createStudentsDB.sql"/>
      <property name="eclipselink.drop-ddl-jdbc-file-name" value="dropStudentsDB.sql"/>
      <property name="eclipselink.logging.level" value="INFO" />
    </properties>
  </persistence-unit>
</persistence>
```



Comment démarrer JPA dans un test unitaire?

```
// Get the entity manager for the tests.  
mEmf = Persistence.createEntityManagerFactory("JEE6Demo-Persistence");  
mEntityManager = mEmf.createEntityManager();  
mTrx = mEntityManager.getTransaction();  
  
mTrx.begin();  
getEntityManager().persist(student);  
mTrx.commit(); //ou mTrx.rollback();  
  
mEntityManager.close();  
mEmf.close();
```



Comment démarrer JPA dans un le serveur d'application?

Transaction gérée par le serveur

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="JEE6Demo-Persistence" transaction-type="JTA">

    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <jta-data-source>jdbc/StudentsDS</jta-data-source>

    <class>ch.demo.dom.Student</class>
    <class>ch.demo.dom.Grade</class>
    <class>ch.demo.dom.Badge</class>

    <properties>
      <property name="eclipselink.target-database" value="DERBY" />
      <property name="eclipselink.logging.level" value="INFO" />
    </properties>
  </persistence-unit>
</persistence>
```

Datasource du serveur



Comment démarrer JPA dans un le serveur d'application?

@Stateless

```
public class StudentServiceJPAImpl implements StudentService, StudentServiceRemote {
```

```
/** The entity manager that manages the persistence. As there is only one persistence unit,  
 * it takes it by default. */
```

```
@PersistenceContext
```

```
private EntityManager entityManager;
```

```
/**
```

```
 * The following service is allowed for all connected users.
```

```
 * Furthermore, the @Benchmarkable annotation triggers an integrated
```

```
 * that will measure the time consumed by this method.
```

```
 */
```

```
@Override
```

```
@RolesAllowed({ "user" })
```

```
public List<Student> getAll() {
```

```
    numberOfAccess++;
```

```
    CriteriaBuilder qb = entityManager.getCriteriaBuilder();
```

```
    CriteriaQuery<Student> c = qb.createQuery(Student.class);
```

```
    TypedQuery<Student> query = entityManager.createQuery(c);
```

```
    return query.getResultList();
```

```
}
```

```
...
```

```
}
```

Injection de l'em par le serveur d'app

Par défaut pas besoin de gérer les transactions (JTA voir EJB)

O-R Mapping



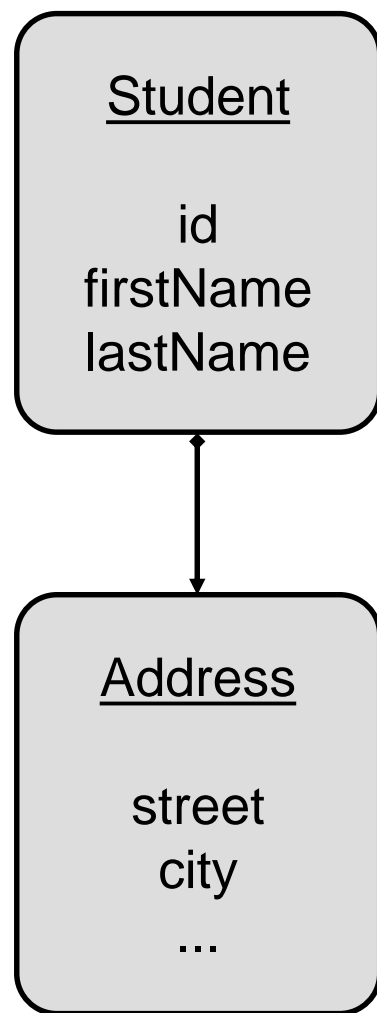
O-R Mapping

- Les relations entre objets doivent être projetées sur des relations entre tables
- Les références entre objets deviennent des clés étrangères
- Comment projeter les relations suivantes
 - Classes embarquées (one to one)
 - Héritage
 - List<Student>
 - Map<String, Student>
 - ...

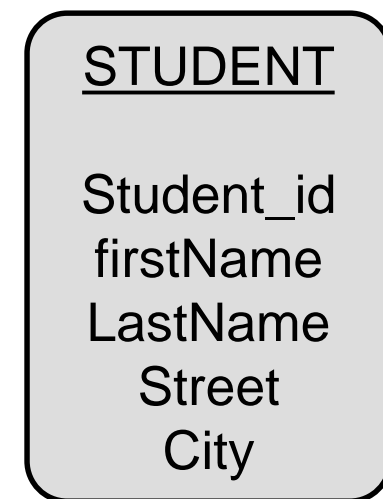
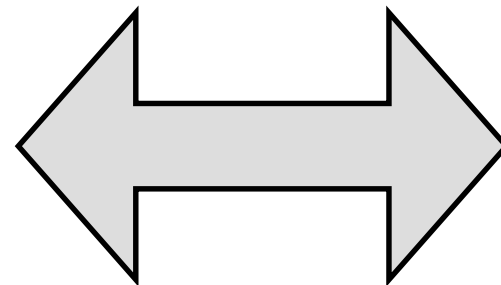
Simple associations



Objet embarqués



Monde objet



Monde relationnel

Objet embarqués

```
@Embeddable  
public class Adresse
```

```
/** The serial id.  
private static final
```

Cette classe peut-être embarquée et n'a donc pas de table dédiée

```
/** house number. */  
@Column(name = "NUMBER")  
private String mNumber;
```

```
/** the name of the street. */  
@Column(name = "STREET")  
private String mStreet;
```

```
...
```

```
}
```

Objet embarqués

@Entity

@Table(name = "STUDENTS")

public class Student implements Serializable {

/** The serial-id. */

private static final long serialVersionUID = -6146935825517747043L;

/** The unique id. */

@Id

@Column(name = "ID")

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long mId;

...

/** The address of the student. */

@Embedded

private Address mAddress;

...

}

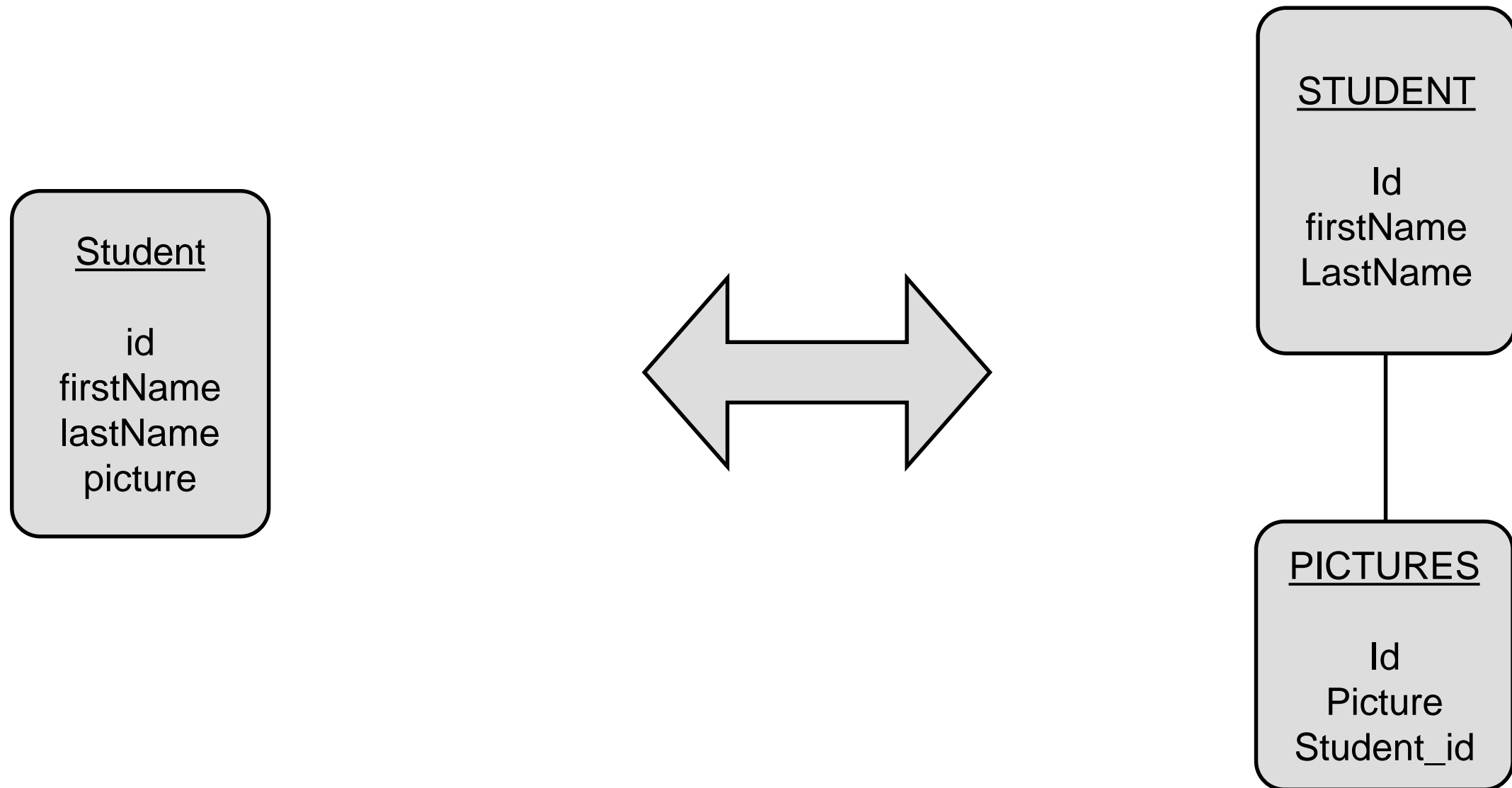
Embarque la classe Address dont ses données sont stockées dans la table STUDENTS

Objet embarqués

```
SELECT  
  ID, FIRST_NAME, PHONE_NUMBER, BIRTH_DATE,  
  LAST_NAME, NUMBER, CITY, STREET,  
  POSTAL_CODE  
FROM  
  STUDENTS
```



One to one (variante 1)



Monde objet

Monde relationnel

One to one (variante 1)

```
@Entity
```

```
@NamedQuery(name = "findAllStudentsByFirstName", query = "SELECT s FROM Student  
WHERE s.mFirstName = :firstname")
```

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

```
@SecondaryTable(name = "PICTURES", pkJoinColumns = @PrimaryKeyJoinColumn(  
    name = "STUDENT_ID", referencedColumnName = "ID"))
```

```
public class Student implements Serializable {
```

```
    /** A picture of the student. */
```

```
    @Lob
```

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

```
    @Column(table = "PICTURES", name = "PICTURE", nullable = false)
```

```
    private byte[] mPicture;
```

```
}
```

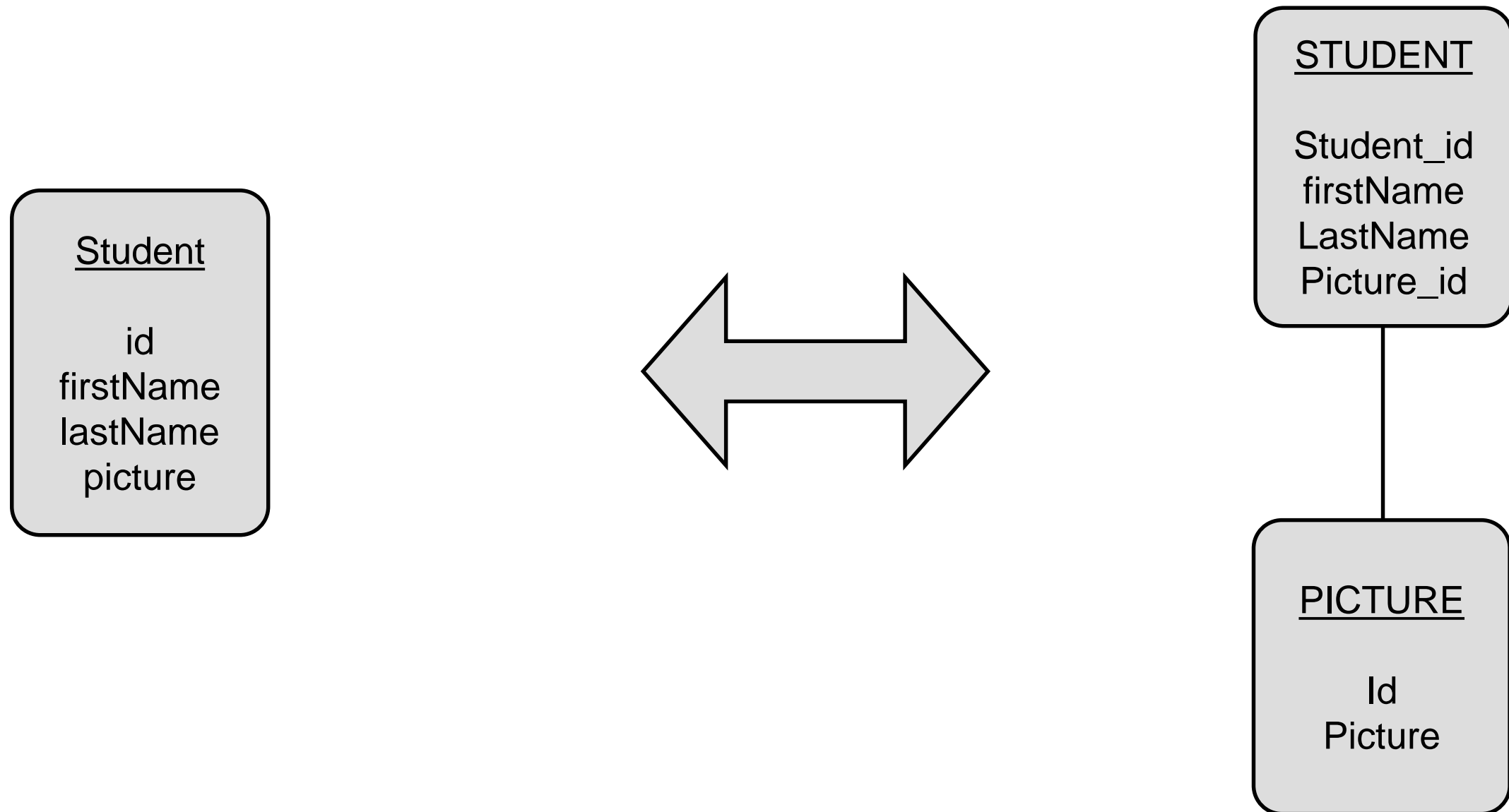


Objet a One to one (variante 1)

```
SELECT
  t0.ID, t1.STUDENT_ID, t0.BIRTH_DATE,
  t0.FIRST_NAME, t0.LAST_NAME, t0.PHONE_NUMBER,
  t1.PICTURE, t0.CITY, t0.NUMBER, t0.POSTAL_CODE,
  t0.STREET
FROM
  STUDENTS t0, PICTURES t1
WHERE
  (t1.STUDENT_ID = t0.ID)
```



One to one (variante 2)



Monde objet

Monde relationnel

One to one (variante 1)

```
@Entity
```

```
@NamedQuery(name = "findAllStudentsByFirstName", query = "SELECT s FROM Student  
WHERE s.mFirstName = :firstname")
```

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

```
@SecondaryTable(name = "PICTURES", pkJoinColumns = @PrimaryKeyJoinColumn(  
    name = "ID", referencedColumnName = "PICTURE_ID"))
```

```
public class Student implements Serializable {
```

```
    /** A picture of the student. */
```

```
    @Lob
```

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

```
    @Column(table = "PICTURES", name = "PICTURE", nullable = false)
```

```
    private byte[] mPicture;
```

```
}
```



Objet a One to one (variante 1)

```
SELECT
  t0.ID, t0.PICTURE_ID, t1.ID, t0.BIRTH_DATE,
  t0.FIRST_NAME, t0.LAST_NAME, t0.PHONE_NUMBER,
  t1.PICTURE, t0.CITY, t0.NUMBER, t0.POSTAL_CODE,
  t0.STREET
FROM
  STUDENTS t0, PICTURES t1
WHERE
  (t1.ID = t0.PICTURE_ID)
```



Mapping avancé



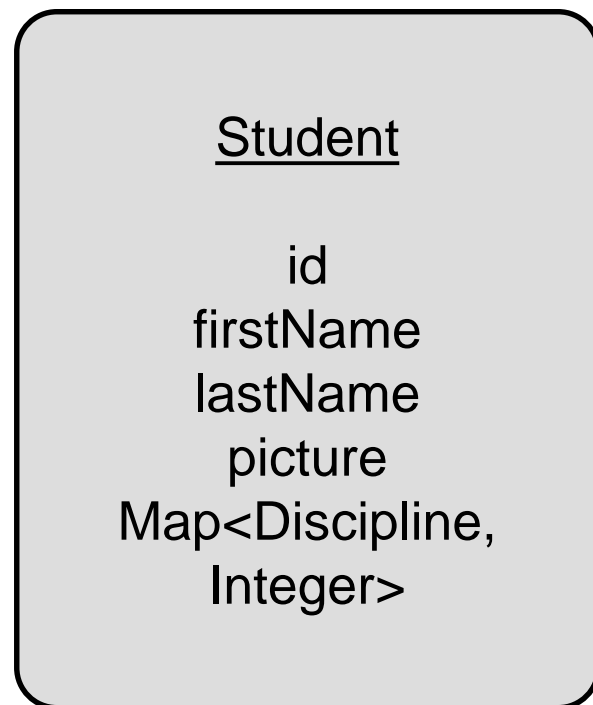
java.lang.Map

```
/** Alternative representation of the set of grades of the student. */  
@ElementCollection  
@CollectionTable(name = "GRADES",  
    joinColumns = @JoinColumn(name = "STUDENT_ID"))  
@MapKeyColumn(name = "Discipline")  
@Column(name = "GRADE")  
private Map<Discipline, Integer> mAlternativeGrades;
```

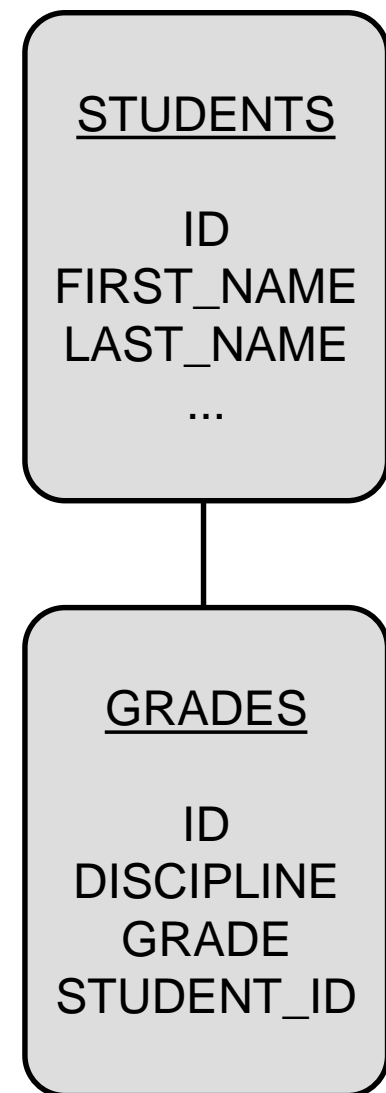
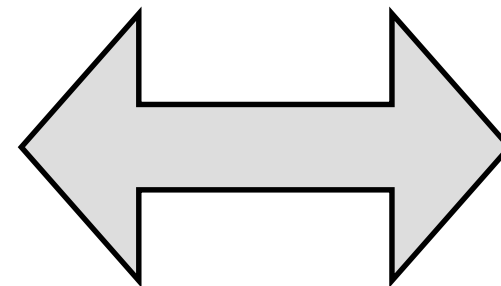
Par comparaison, regarder le mapping de mGrades...



Map



Monde objet



Monde relationnel

Glouton/fainéant

```
/** The set of grades of the student. */  
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
@JoinColumn(name = "STUDENT_ID", nullable = true)  
private List<Grade> mGrades;
```

La liste ne sera chargée que si l'utilisateur utilise le getter `getGrades()`.

```
/** The set of grades of the student. */  
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)  
@JoinColumn(name = "STUDENT_ID", nullable = true)  
private List<Grade> mGrades;
```

La liste sera toujours chargée.



Tri des relations

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name = "STUDENT_ID", nullable = true)
@OrderBy("mDiscipline DSC")
private List<Grade> mGrades;
```

La liste sera ordonnée par la propriété discipline de façon descendante

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name = "STUDENT_ID", nullable = true)
@OrderBy
private List<Grade> mGrades;
```

La liste sera ordonnée par la clé primaire de façon ascendante



Tri des relations

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name = "STUDENT_ID", nullable = true)
@OrderColumn("DISCIPLINE")
private List<Grade> mGrades;
```

Contrairement à `@OrderBy` qui ne modifie que la façon dont les données sont chargées de la DB, avec `@OrderColumn`, l'ordre sera persisté lors de la sauvegarde de la liste.



One to Many



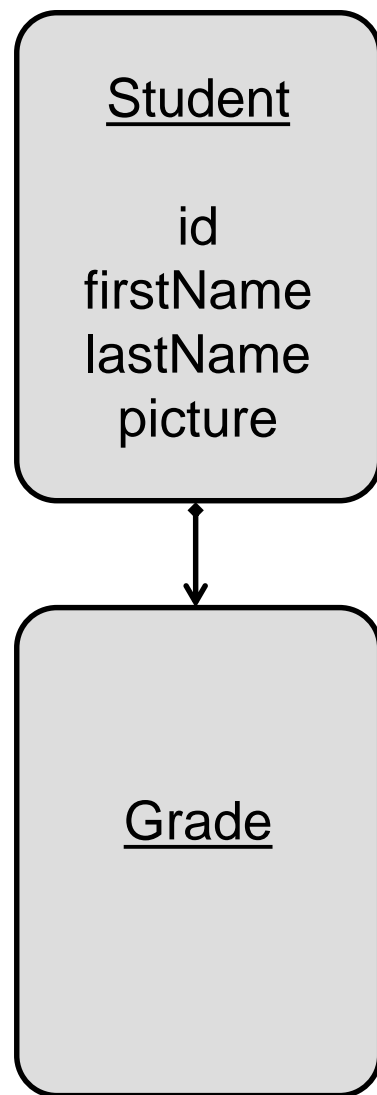
One 2 Many uni-directionnelle

Modélise une association de cardinalité 0..1 à N

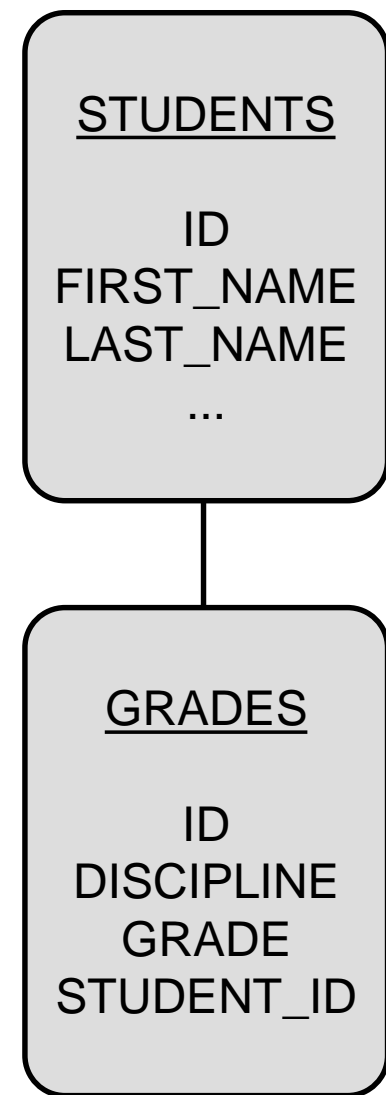
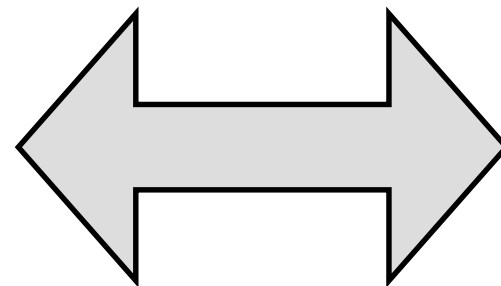
Seul un objet a connaissance de l'association



One 2 Many uni-directionnelle



Monde objet



Monde relationnel

One 2 Many uni-directionnelle

```
@Entity
@Table(name = "GRADES")
public class Grade implements Serializable {

    /** The unique identifier of the grade. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long mId;

    /** The discipline of this grade. */
    @Column(name = "DISCIPLINE", nullable = false)
    private Discipline mDiscipline;

    /** The actual grade. */
    @Column(name = "GRADE", nullable = true)
    private Integer mGrade;

    ...
}
```

Les données de cette classe sont stockées
dans la table GRADES



One 2 Many uni-directionnelle

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy =
private Long mId;

...
    /** The set of grades of the student. */
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "STUDENTS_ID", nullable = true)
private List<Grade> mGrades;

...
}
```

La liste des notes provient d'une autre entité,
le lien est fait par la colonne STUDENTS_ID



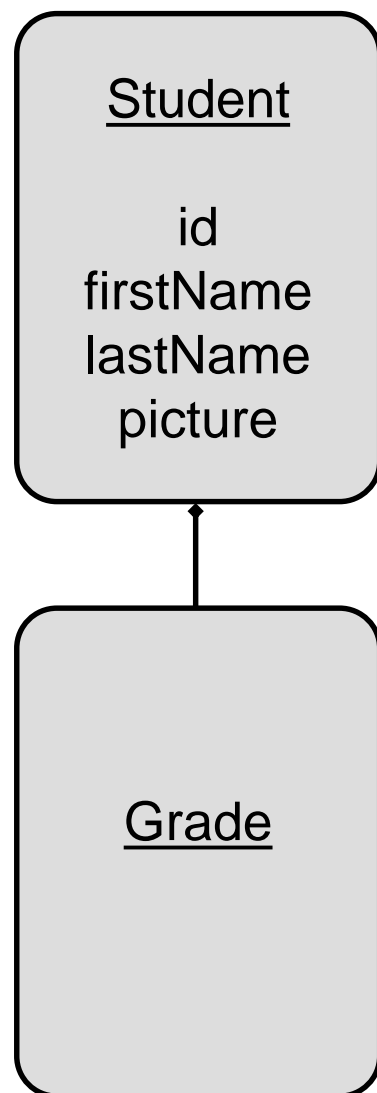
One 2 Many bi-directionnelle

Modélise une association de
cardinalité 0..1 à N

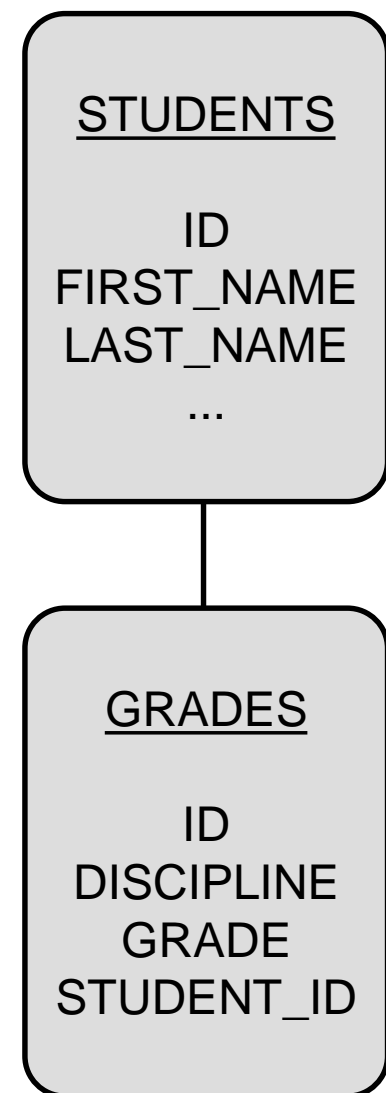
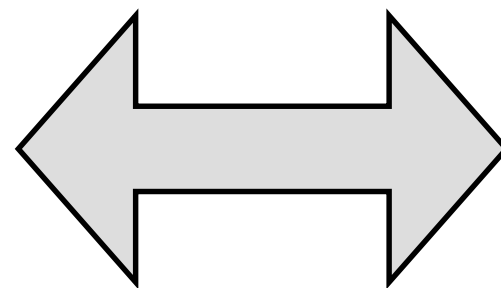
Les deux objets ont connaissance de
l'association



One 2 Many bi-directionnelle



Monde objet



Monde relationnel

One 2 Many uni-directionnelle

```
@Entity
@Table(name = "GRADES")
public class Grade implements Serializable {
    ...
    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long mId;

    ...

    @ManyToOne
    @JoinColumn(name = "STUDENT_ID", nullable = true)
    private Student mStudent;
}
```


One 2 Many uni-directionnelle

```
@Entity
@Table(name = "STUDENTS")
public class Student implements Serializable {

    /** The serial-id. */
    private static final long serialVersionUID = -6146935825517747043L;

    /** The unique id. */
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    ...
    /** The set of grades of the student. */
    @OneToMany(cascade = CascadeType.ALL,
        fetch = FetchType.LAZY, mappedBy = "mStudent")
    private List<Grade> mGrades;

    ...
}
```

Many to Many



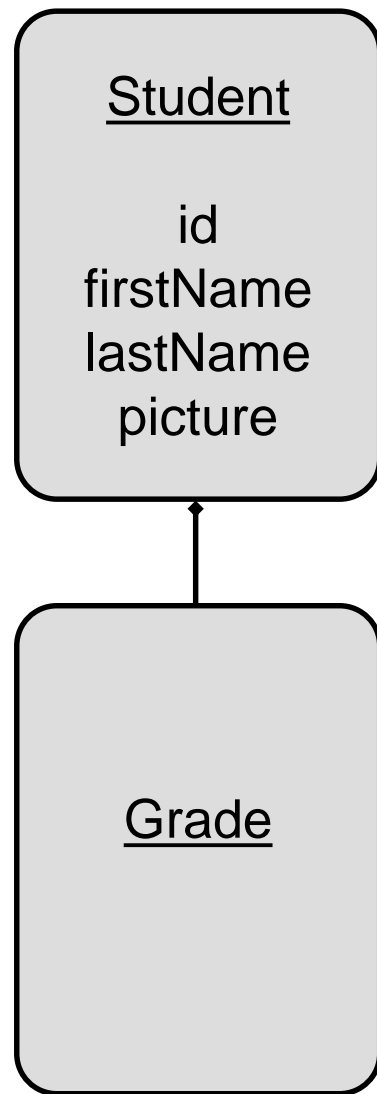
Many 2 Many

Modélise une association de cardinalité
 $0..N$ à $0..M$

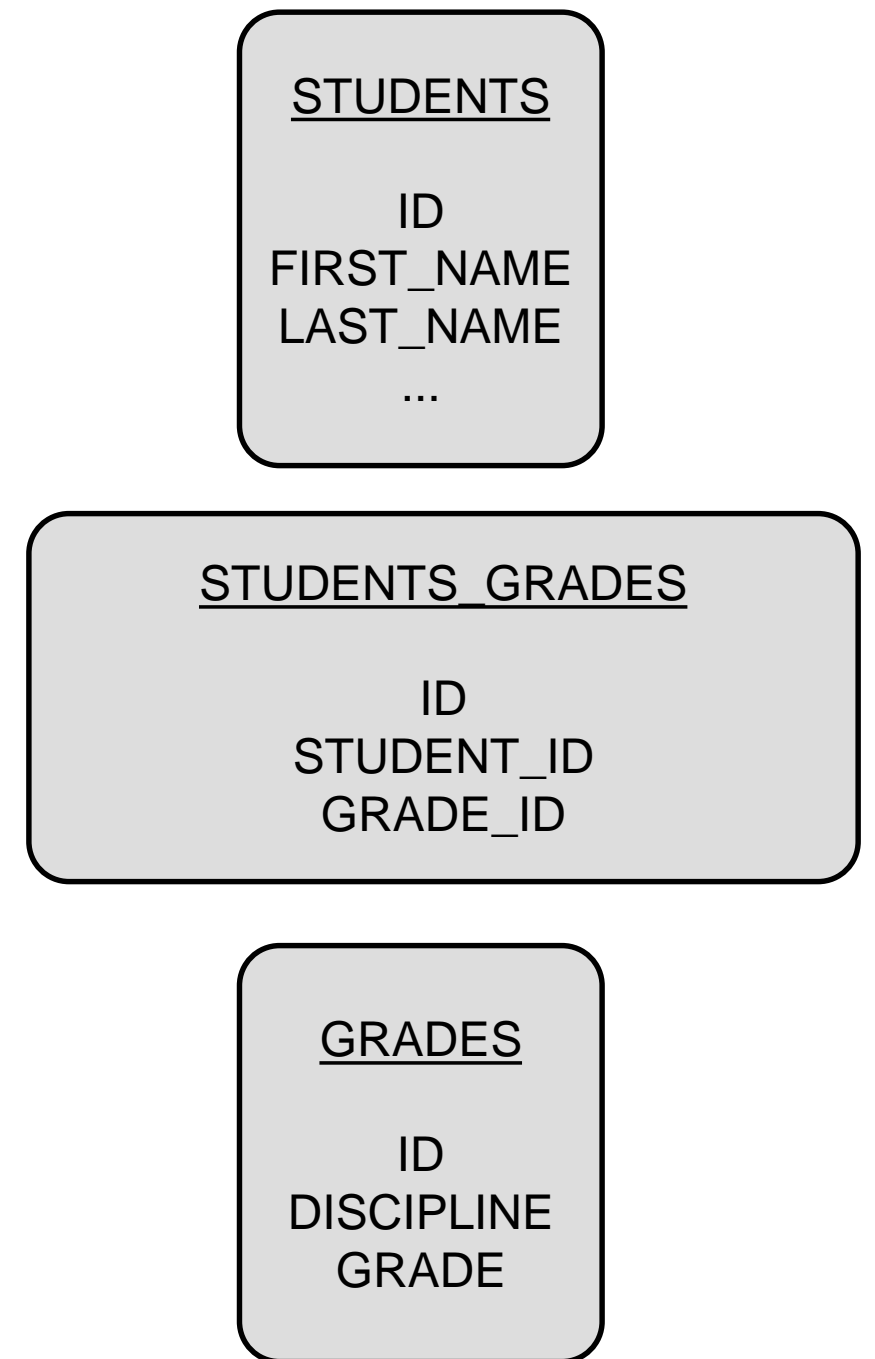
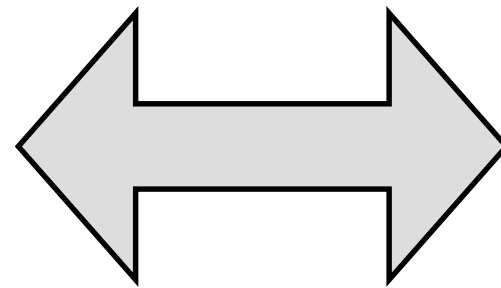
Similaire dans le fonctionnement à $0..1$ à N



Many 2 Many



Monde objet



Monde relationnel

Many 2 Many

```
@Entity
```

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

```
public class Student implements Serializable {
```

```
    /** The serial-id. */
```

```
    private static final long serialVersionUID = -6146935825517747043L;
```

```
    ...
```

```
    /** The set of grades of the student. */
```

```
@ManyToMany(cascade = CascadeType.ALL)
```

```
@JoinTable(name = "STUDENTS_GRADES",
```

```
    joinColumns={@JoinColumn(name="ID", referencedColumnName="STUDENT_ID")},
```

```
    inverseJoinColumns={@JoinColumn(name="ID", referencedColumnName="GRADE_ID")})
```

```
private List<Grade> mGrades;
```

```
    ...
```

```
}
```



Many 2 Many

```
@Entity
@Table(name = "GRADES")
public class Grade implements Serializable {
    ...
    /** The unique id. */

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long mId;

    ...

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "STUDENTS_GRADES",
        joinColumns={@JoinColumn(name="ID", referencedColumnName="GRADE_ID")},
        inverseJoinColumns={@JoinColumn(name="ID", referencedColumnName="
        STUDENT_ID")})
    private List<Student> mStudents;
}
```



Héritage



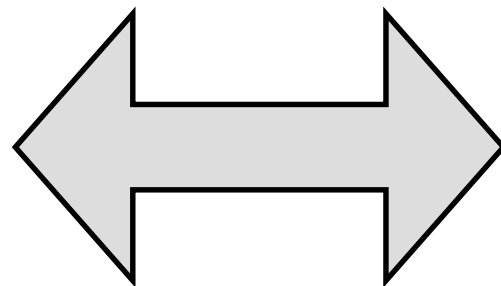
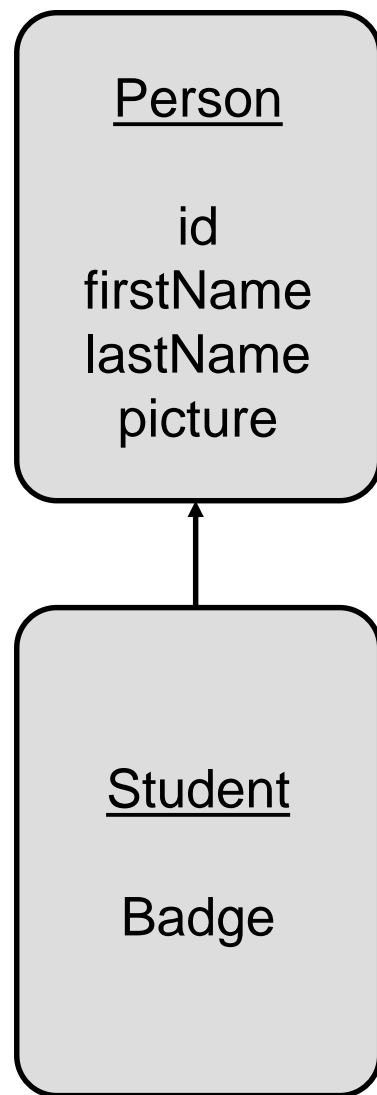
Héritage

Faire correspondre un concept purement objet dans un monde relationnel.

Repose sur une colonne discriminante pour les sous-types.



Héritage



?

Stratégie à une table par hiérarchie

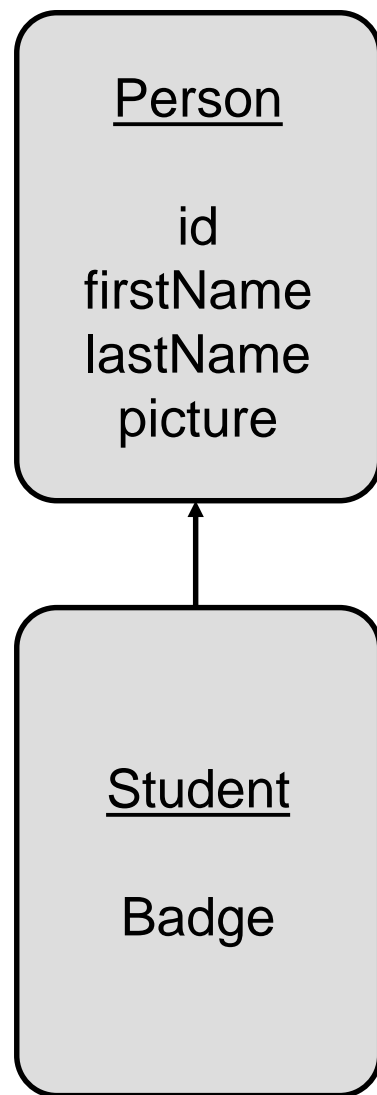
Toutes les classes de la hiérarchie sont représentées par la même table.

Certaines données seront donc NULL selon les cas

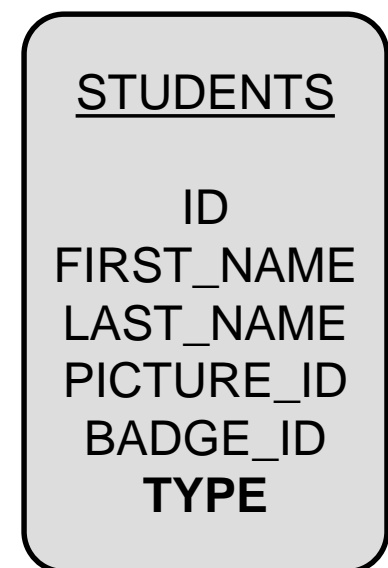
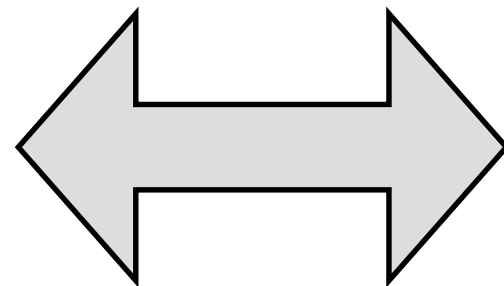
- + Rapide (pas de join)
- Beaucoup de colonnes NULL



Stratégie à une table par hiérarchie



Monde objet



Monde relationnel

Stratégie à une table par hiérarchie

```
@Entity
@Table(name="STUDENTS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE", discriminatorType = DiscriminatorType.STRING,
length = 20)
@DiscriminatorValue("P")
public class Person {

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    /** The student last name. */
    @Column(name = "LAST_NAME", length = 35)
    private String mLastName;
}
```



Stratégie à une table par hiérarchie

```
@Entity  
@DiscriminatorValue("S")  
public class Students extends Person {  
  
    private Badge mBadge;  
    ...  
}
```

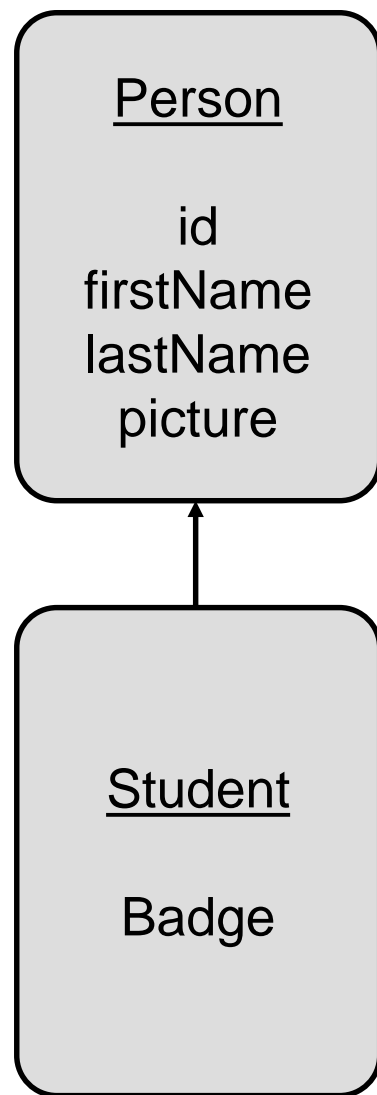
Stratégie par jointure entre sous-classes

Chaque sous classe a sa propre table avec ses colonnes supplémentaires.

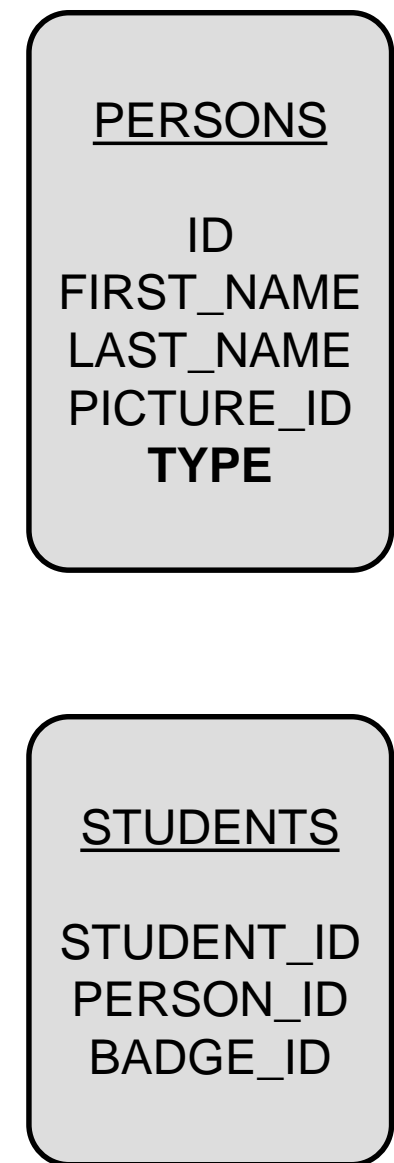
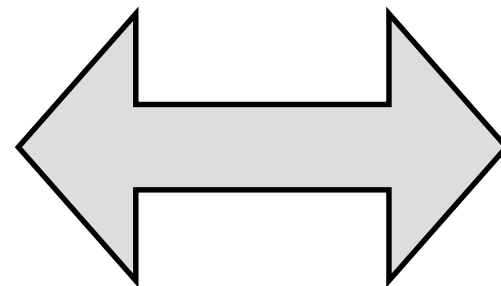
- + Schéma plus normalisé
- + Plus extensible puisqu'il ne faut pas modifier la table mère
- Jointure pouvant ralentir les performances



Stratégie par jointure entre sous-classes



Monde objet



Monde relationnel

Stratégie par jointure entre sous-classes

```
@Entity
@Table(name="PERSONS")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="TYPE", discriminatorType = DiscriminatorType.STRING,
length=20)
@DiscriminatorValue("P")
public class Person {

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long mId;

    /** The student last name. */
    @Column(name = "LAST_NAME", length = 35)
    private String mLastName;
}
```



Stratégie par jointure entre sous-classes

```
@Entity
@Table(name = "STUDENTS")
@DiscriminatorValue("S")
public class Students extends Person {

    private Badge mBadge;
    ...
}
```

JPQL



JPQL

Similaire à SQL mais adapté au monde objet

Peut être utilisé pour des requêtes ou des modifications en masse.



Syntaxe d'une requête

SELECT ... FROM ...[WHERE ...]

[GROUP BY ... [HAVING ...]]

[ORDER BY ...]

```
TypedQuery<Student> query =  
    getEntityManager().createQuery("SELECT s FROM Student s", Student.class);
```

Requêtes sous forme de chaîne de caractères
+ Facile pour les personnes qui connaissent SQL
- Pas typées



Syntaxe d'une requête

```
CriteriaBuilder qb = getEntityManager().getCriteriaBuilder();  
CriteriaQuery<Student> c = qb.createQuery(Student.class);  
TypedQuery<Student> query = getEntityManager().createQuery(c);
```

Requêtes sous forme d'appels API

- Moins lisible
- + En partie validée par le compilateur JAVA



Requêtes nommées

```
@Entity
@NamedQuery(name = "findAllStudentsByFirstName",
    query = "SELECT s FROM Student s WHERE s.firstName = :firstname")
@Table(name = "STUDENTS")
@SecondaryTable(name = "PICTURES", pkJoinColumns = @PrimaryKeyJoinColumn(
    name = "STUDENT_ID", referencedColumnName = "ID"))
public class Student implements Serializable {
```

```
TypedQuery<Student> queryStudentsByFirstName = getEntityManager().
    createNamedQuery("findAllStudentsByFirstName", Student.class);
queryStudentsByFirstName.setParameter("firstname", "Steve");
Collection<Student> students = queryStudentsByFirstName.getResultList();
```



Requêtes natives

```
Query query3 = getEntityManager().createNativeQuery(  
    "select FIRST_NAME, LAST_NAME from STUDENTS");  
@SuppressWarnings("unchecked")  
List<String[]> list = query3.getResultList();
```



Bibliographie

<http://www.vogella.com/articles/JavaPersistenceAPI/article.html>

<http://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html>

<http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html>

