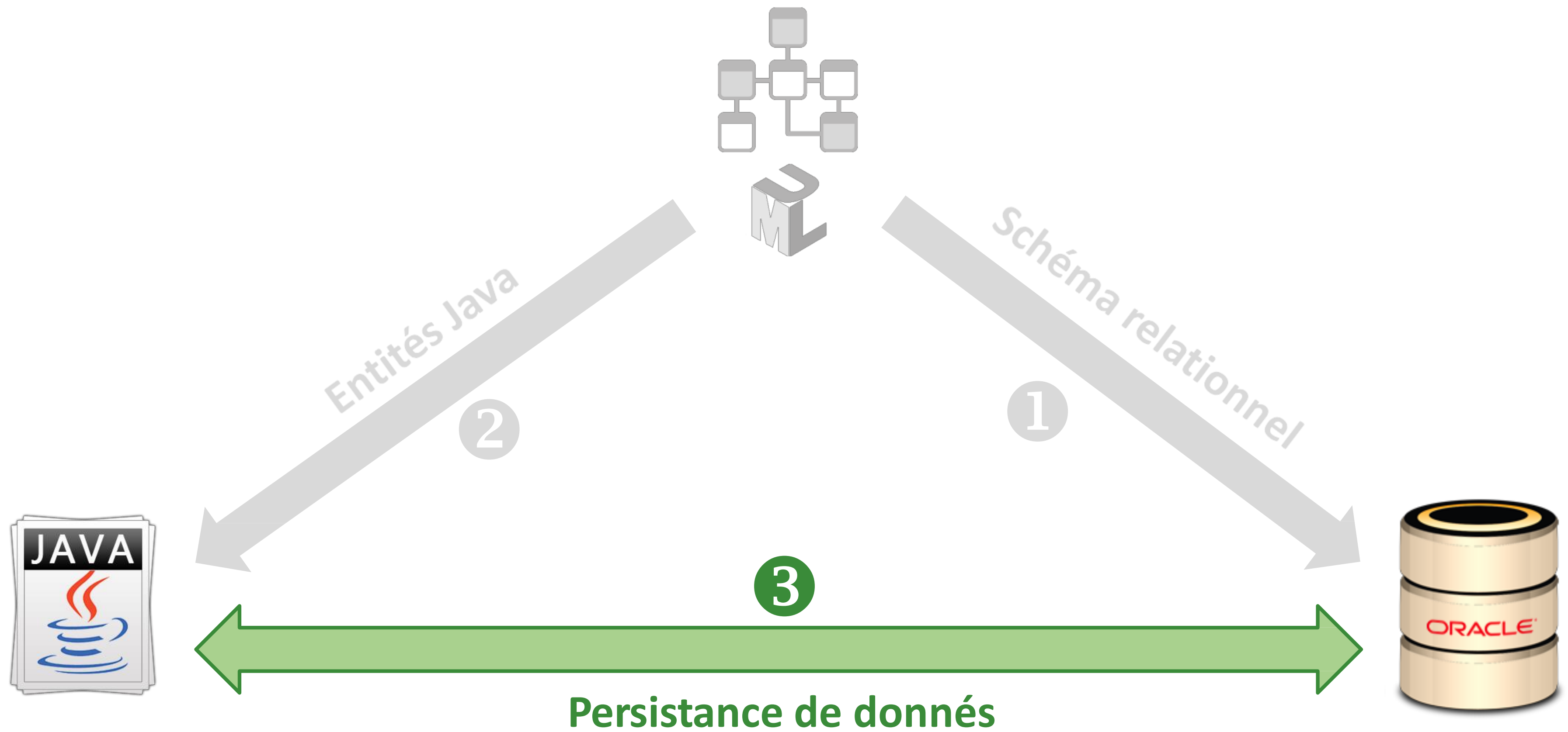
Bases de Données Avancées (BDA)

Chapitre 3-2: Persistance transparente avec Hibernate

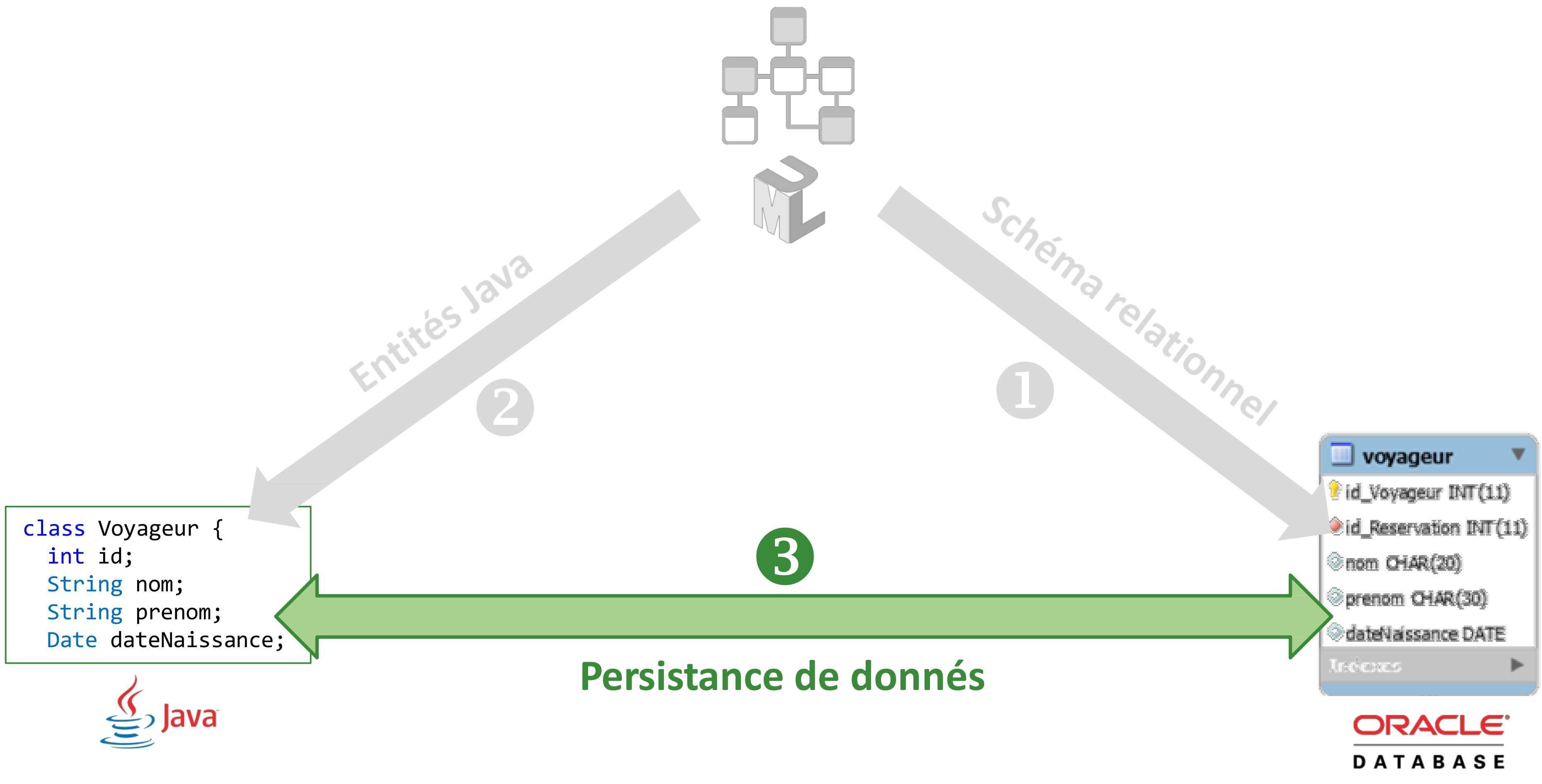
Objet / Relational Mapping (ORM)

Par SELMANE S

Persistance de données (1/2)

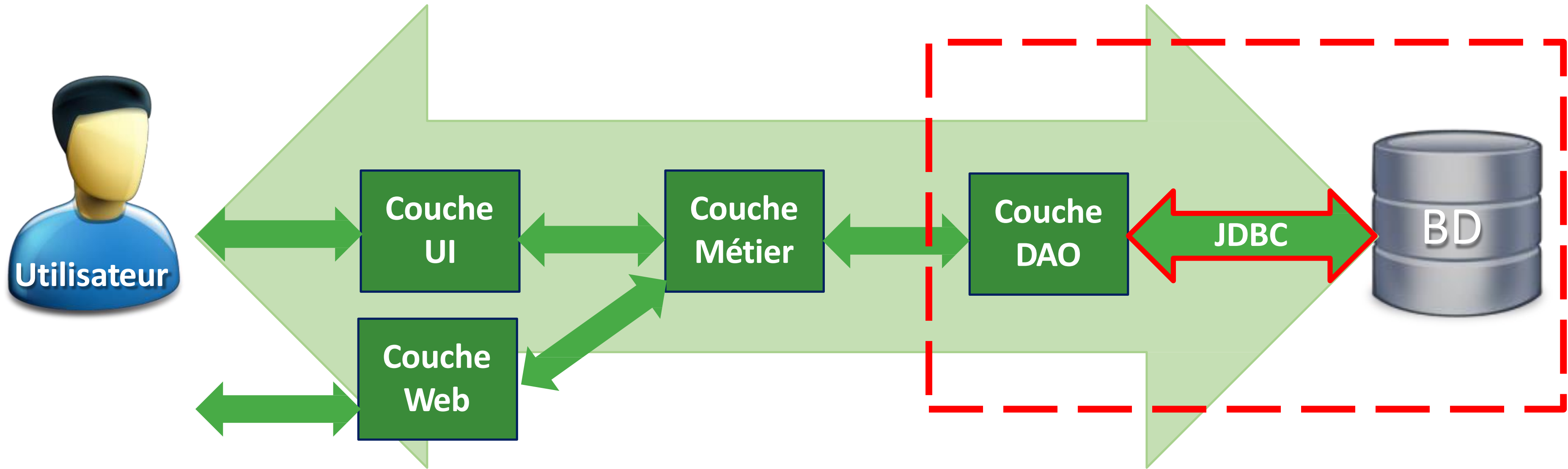


Persistance de données (2/2)



JDBC : Java DataBase Connectivity

JDBC
Java Database Connectivity



Limites de JDBC

Inconvénients de l'API JDBC :

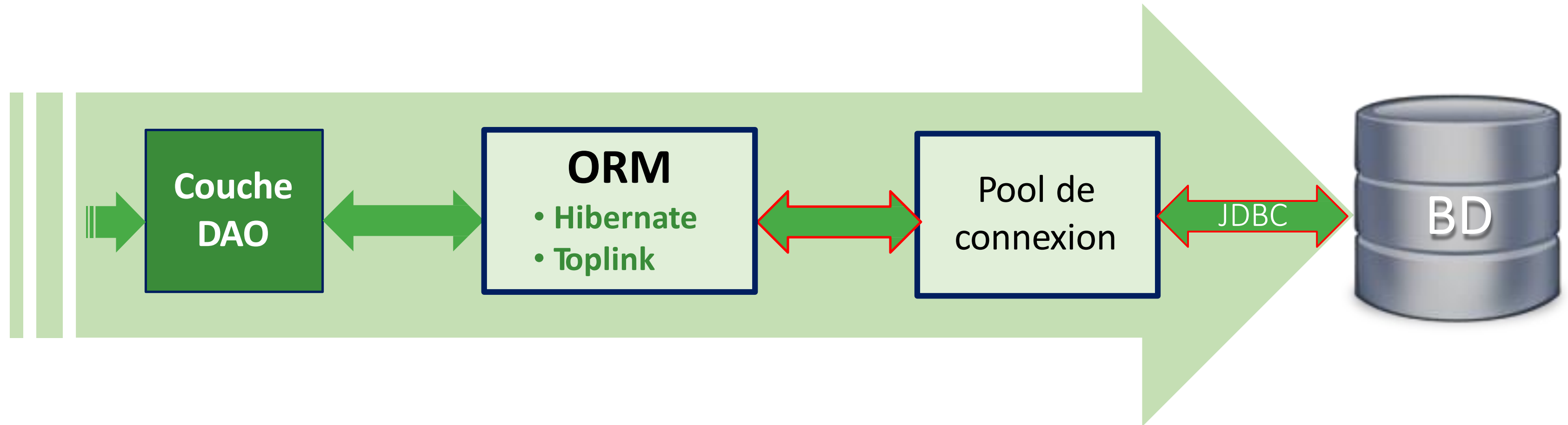
- nécessite l'écriture de nombreuses lignes de codes, souvent répétitives
- le mapping entre les tables et les objets est un travail de bas niveau

Tous ces facteurs réduisent la productivité mais aussi les possibilités d'évolutions et de maintenance.

Différentes solutions :

- Des ORM open source : Hibernate, Toplink, DataNeclius
- Des API Standards Java : JDO, EJB entity, JPA

ORM : Object Relational Mapping



Qu’est-ce qu’un ORM ?

ORM (Object/Relationnal Mapping)

- est une technique qui simule une BD orientée objet à partir d’une BD relationnelle.
- permet de masquer au développeur la persistance de données
- assure le mapping des tables (BD) avec les classes (application Java)
- propose un langage de requêtes indépendant de la base de données cible et assure une traduction en SQL natif
- Aucun code technique ne vient polluer le code des applications.
- assure une gestion des accès concurrents (verrou, dead lock, ...)

Limites des ORM :

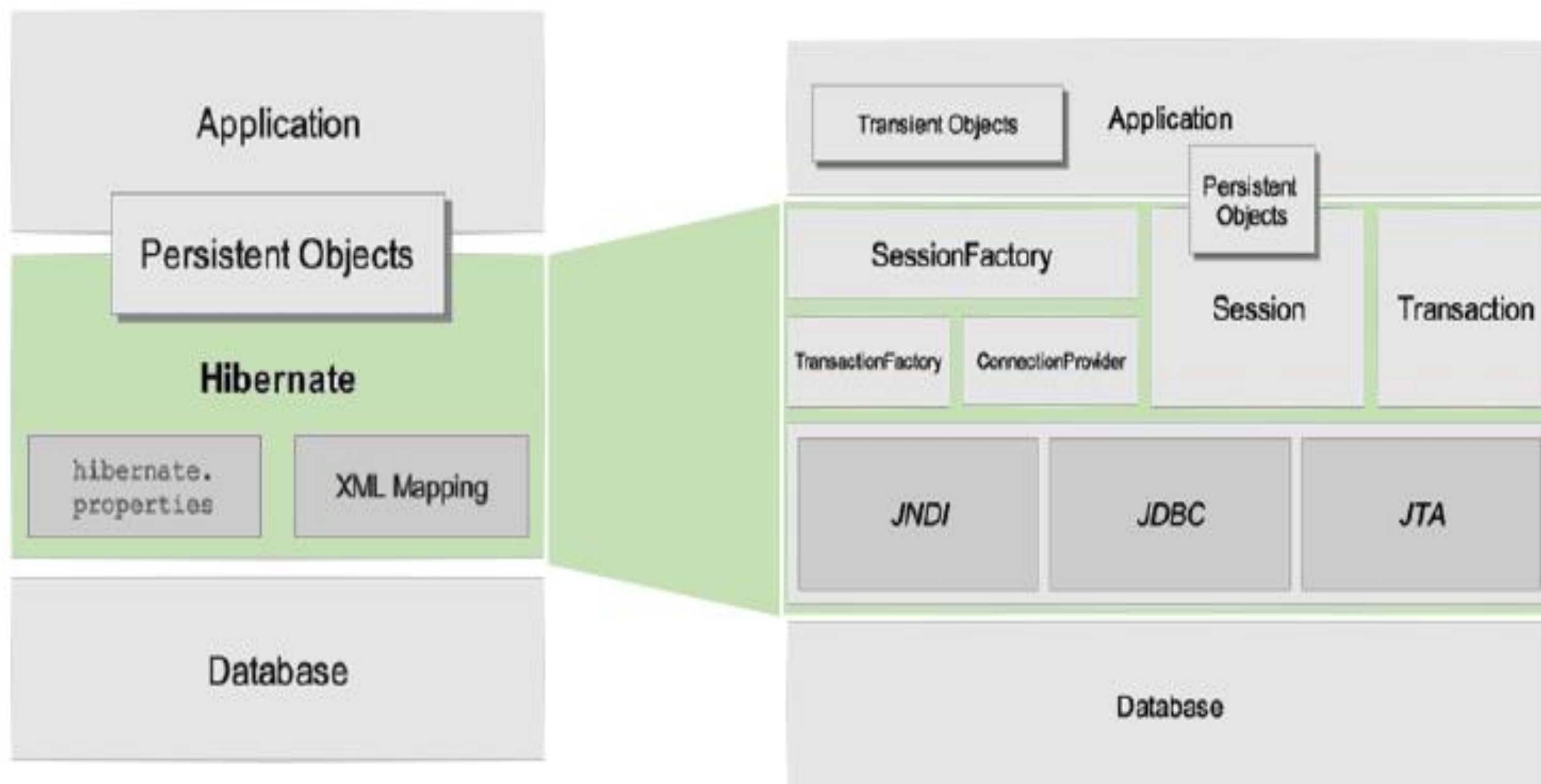
- Beaucoup moins performant que des requêtes SQL optimisées
- Très fortement configurables et difficile à maîtriser

Hibernate

Hibernate

- est un framework d’ORM pour applications JAVA
  - Y a aussi Nhibernate pour les applications .NET
- appelé une solution de gestion de persistance ou couche de persistance
- permet de créer une couche d’accès aux données (DAO) plus modulaire,
- plus maintenable, plus performante en productivité qu’une couche d’accès
- aux données "classique" reposant sur l’API JDBC

Architecture de Hibernate



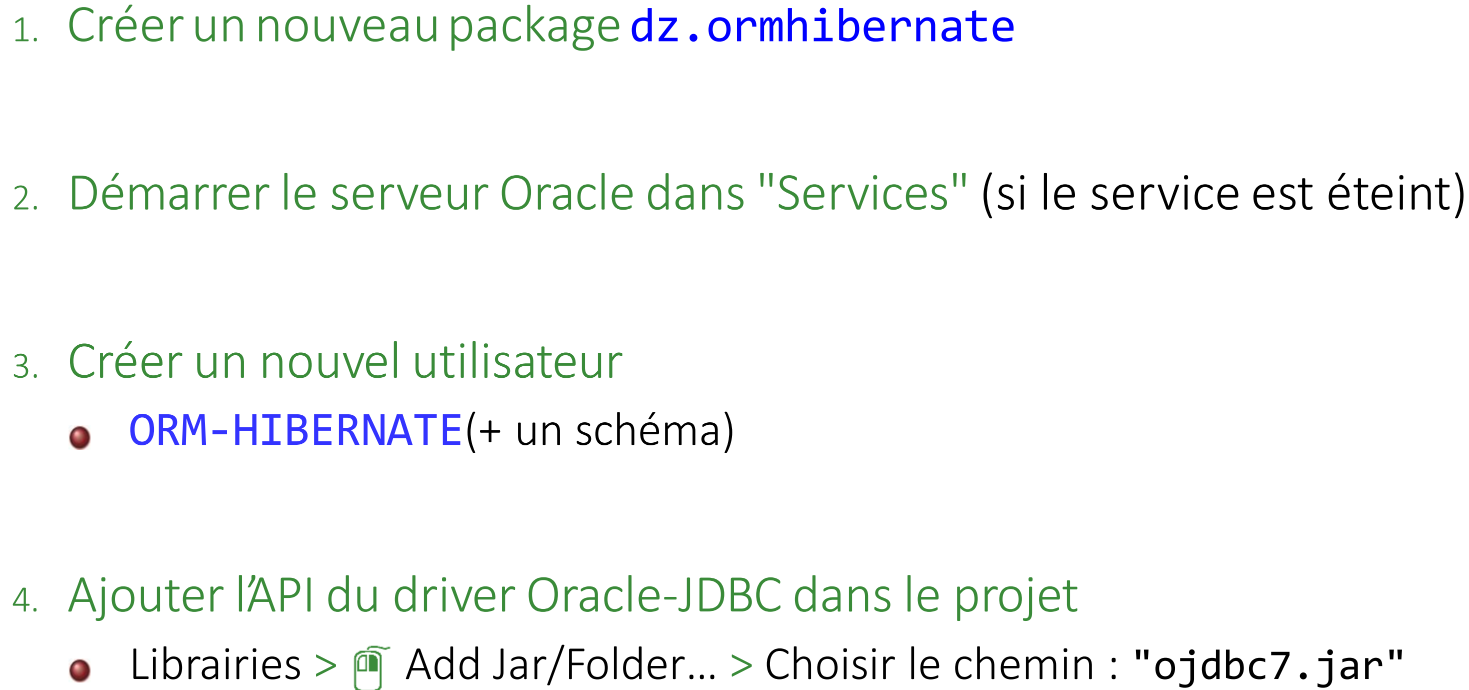
Configuration de la persistance des données

Etapes de développement :

1. Préparer le projet (Exemple NetBeans)
2. Importer l'API de Hibernate
3. Définir les règles de persistance des classes
4. Rendre les classes persistables sous Hibernate
5. Configurer la persistance de Hibernate

Configuration de la persistance des données

1. Préparer le projet (Exemple NetBeans)



Configuration de la persistance des données

2. Importer l'API de Hibernate (Ex: NetBeans)

Ajouter l'API de Hibernate

- Librairies > Add Jar/Folder… > Choisir les fichiers :
  - hibernate-core-5.3.7.Final.jar
  - javassist-3.23.1-GA.jar
  - antlr-2.7.7.jar
  - dom4j-1.6.1.jar
  - jandex-2.0.5.Final.jar
  - hibernate-jpamodelgen-5.3.7.Final.jar
  - hibernate-commons-annotations-5.0.4.Final.jar
  - javax.persistence-api-2.2.jar
  - javax.activation-api-1.2.0.jar
  - byte-buddy-1.8.15.jar
  - jboss-logging-3.3.2.Final.jar
  - jboss-transaction-api_1.2_spec-1.1.1.Final.jar
  - classmate-1.3.4.jar
  - ...

Configuration de la persistance des données

3. Définir les règles de persistance des classes

Règles des classes persistantes :

- Basée sur des classes **POJO (Plain Old Java Object)**
- Implémenter un **constructeur vide** (sans arguments)
- Fournir une propriété d'**identifiant**
- Déclarer les **getters et setters** des attributs persistants
- Implémenter **equals()** ou **hashCode()** :
  - en utilisant **l'égalité des identifiants** (clés primaires)

Configuration de la persistance des données

4. Rendre les classes persistables

Décrit la correspondance entre un schéma de la BD et un modèle de classes pour assurer la persistance de l'état des objets.

Définir la persistence des classes en utilisant :

- Des fichiers de mapping XML
  - un fichier par classe/table
  - par convention l'extension est `.hbm.xml` (Ex : `NomClasse.hbm.xml`)
  - le fichier de mapping dans le même répertoire que le fichier de classe
- Des annotations de Java Persistence (JPA)

Mapping avec XML

Une classe

bda2/model/A.hbm.xml

```
<hibernate-mapping>
   <class name="A" table="a">
      <id name="id" type="integer">
           <column name="id" />
      </id>
      <property name="a1" type="string">
           <column name="a1" />
      </property>
      <property name="a2" type="date">
           <column name="a2" />
      </property>
   </class>
</hibernate-mapping>
```

A
id: Integer a1: String a2: Date

Types de Hibernate : *integer, string, character, date, timestamp, float, binary, serializable, object, blob*

Types de Java : *int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob*

Mapping avec XML

Stratégies d'héritage (1/2)

Hibernate supporte les 3 stratégies d'héritage de base :

Une table par hiérarchie de classe :

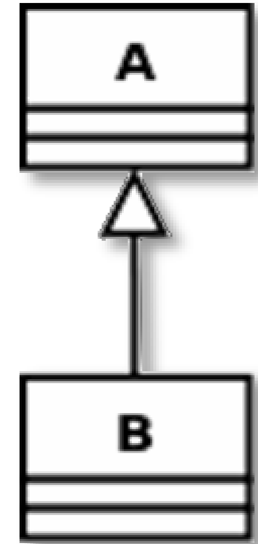
- Une seule table englobe toutes les instances de l'hiérarchie (`<subclass>`)

Une table par classe :

- Chaque table d'une classe ne persiste que les propriétés spécifiques à cette classe (`<joined-subclass>`)

Une table par classe concrète :

- Chaque table persiste les propriétés de la classe et les propriétés de ses super-classes abstraites (`<union-subclass>`)



Documentation :

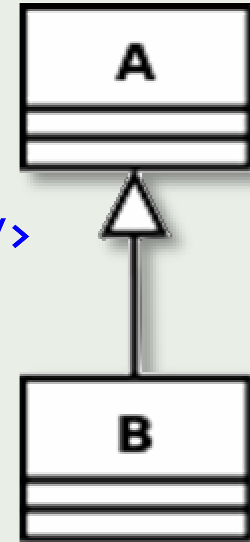
- https://docs.jboss.org/hibernate/orm/3.5/reference/fr/html/inheritance.html

Mapping avec XML

Stratégies d'héritage (2/2) Une table par classe avec discriminateur

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a" discriminator-value="a">
        <id name="id" type="integer">
            <column name="id" />
        </id>
        <discriminator column="discriminator" type="string"/>
        <subclass name="B" discriminator-value="b">
            <join table="b">
                <key column="a_id"/>
                ...
            </join>
        </subclass>
    </class>
</hibernate-mapping>
```



Pas de fichier de mapping **"B.hbm.xml"**

Mapping avec XML

Persistance des associations

C'est la partie la plus complexe de la configuration de la persistence

Types d'associations :

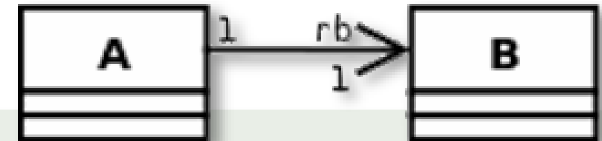
- **Cardinalités des associations :** 1-1, 1-N, N-1, M-N
  - **<one-to-one>, <one-to-many>, <many-to-one>, <many-to-many>**
- **Navigabilité :**
  - **Unidirectionnelle ou bidirectionnelle**
- **Collections :**
  - **<set>, <list>, <map>, <bag>, <array>, <primitive-array>**

Mapping avec XML

Association unidirectionnelle one-to-one

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <one-to-one name="rb" class="B" foreign-key="b_id"
                    constrained="true" />
    </class>
</hibernate-mapping>
```



bda2/model/B.hbm.xml

```
<hibernate-mapping>
    <class name="B" table="b">
        ...
    </class>
</hibernate-mapping>
```

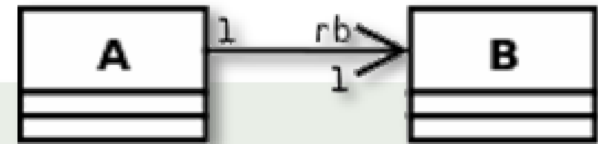
Mapping avec XML

Types de cascade (1/2)

- Hibernate permet d'**ajouter** ou de **supprimer** un objet **en cascade**
- Quand le mot clé **cascade** est spécifié dans une proriété (de type association), alors le changement appliqué à l'entité **sera appliqué aussi à la propriété (l'association)**.

Exemple :

bda2/model/A.hbm.xml




```
<class name="A" table="a">
    ...
    <one-to-one name="rb" class="B" cascade="delete" ... />
</class>
```

- Quand un objet de type A **est supprimé**, alors l'objet de type B associé **est aussi supprimé en cascade.**

Mapping avec XML

Types de cascade (2/2)

- **Types de cascade supportés par Hibernate :**
- **cascade="XXX":**
  - "none" = no cascading
  - "persist" = persist
  - "save-update" = save, update
  - "delete", "remove" = delete
  - "merge" = merge
  - "lock" = lock
  - "refresh" = refresh
  - "replicate" = replicate
  - "evict" = evict
  - "delete-orphan" = delete + delete orphans
  - "all" = save,delete,update,evict,lock,replicate,merge,persist
  - "all-delete-orphan" = all, delete-orphan

Mapping avec XML

Stratégies de chargement (Fetching) (1/5)

- Avec Hibernate, il faut récupérez les données **avec 2 objectifs :**
  1. **Minimiser le nombre de requêtes SQL générées** par Hibernate
  2. **Simplifier les requêtes SQL générées**
- **Le fetching** concerne le chargement des objets **liés par des associations**
- Choisir une stratégie de chargement avec Hibernate,
  c'est choisir **QUAND** et **COMMENT** vos données seront récupérées.
  - **Paramétrage statique** (fichier de mapping) **et**
  - **dynamique** (programmatiquement)

Mapping avec XML

Stratégies de chargement (Fetching) (2/5)

Il y a 6 stratégies **QUAND** **charger les données :**

1. **Chargement différé (Lazy) :** (`lazy="true"`) [par défaut]
   - Une collection est chargée lorsque l'application invoque une méthode sur cette collection (à la demande)
2. **Chargement immediat (Eager) :** (`lazy="false"`)
   - Une association, une collection ou un attribut est chargé immédiatement lorsque l'objet auquel appartient cet élément est chargé.
3. **Chargement super différé (extra) :** (`lazy="extra"`)
   - Plus intelligent que le chargement différé
   - Certaines fonctions comme **`size()`, `contains()`, `get()`**, … ne déclencheront pas de requête SQL supplémentaire.

…

Mapping avec XML

Stratégies de chargement (Fetching) (3/5)

Il y a 6 stratégies **QUAND** **charger les données :**

. . .

4. **Chargement par proxy :** (`lazy="proxy"`)
    - Une association vers un seul objet est chargée lorsqu'une méthode autre que le **getter sur l'identifiant** est appelée sur l'objet associé
5. **Chargement sans proxy :** (`lazy="no-proxy"`)
    - Moins différée par rapport au chargement par proxy
    - l'association est quand même chargée **même si on n'accède qu'à l'identifiant**
6. **Chargement différé des attributs :** (`<property ... lazy="true"/>`)
    - **Un attribut ou un objet associé seul** est chargé lorsque l'on accède à la variable d'instance.

Mapping avec XML

Stratégies de chargement (Fetching) (4/5)

Il y a 4 stratégies **COMMENT charger les données :**

1. **par select** [par défaut]
   - Hibernate récupère les données associée dans un **second** `SELECT`
2. **par jointure** (`fetch="join"`)
   - Hibernate récupère les données associée dans un **même** `SELECT` à l'aide d'un `OUTER JOIN`
3. **par lot** (`batch-size="N"`)
   - Hibernate récupère un lot d'instances en un **seul** `SELECT` en spécifiant une liste de **clé primaire** ou de **clé étrangère**
4. **par sous-select** (`fetch="subselect"`)
   - Hibernate récupère les associations pour toutes les entités récupérées dans un **second** `SELECT`

Mapping avec XML

Stratégies de chargement (Fetching) (5/5)

Exemple :

bda2/model/A.hbm.xml

```
<class name="A" table="a">
    ...
    <one-to-one name="rb" class="B" lazy="false" fetch="join" ... />
</class>
```

`lazy="false"`

- Quand un objet de type A est chargé à partir de la BD, Hibernate charge l'objet de type B associé **même s'il n'est pas utilisé dans le code Java**

`fetch="join"`

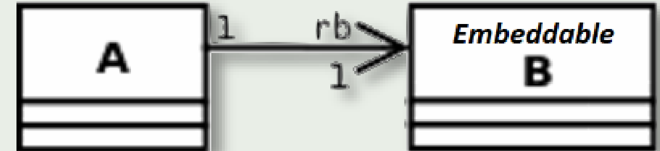
- Quand un objet de type A associé avec un objet de type B sont chargés, Hibernate génèrera implicitement une seule requête de jointure **`SELECT`** en utilisant **`OUTER JOIN`**

Mapping avec XML

Association unidirectionnelle one-to-one "Embedded"

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <component name="rb" class="B" insert="true" update="true" >
            <property name="..." type="...">
                ...
            </property>
        </component>
    </class>
</hibernate-mapping>
```



- Pas de fichier de mapping "B.hbm.xml"

Mapping avec XML

Association bidirectionnelle one-to-one

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <one-to-one name="rb" class="B" property-ref="ra"
            cascade="all" constrained="true" />
    </class>
</hibernate-mapping>
```

bda2/model/B.hbm.xml

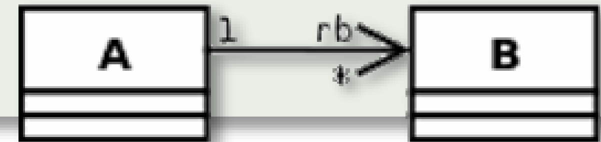
```
<hibernate-mapping>
    <class name="B" table="b">
        ...
        <many-to-one name="ra" column="a_id" unique="true"
                         not-null="true" cascade="all" />
    </class>
</hibernate-mapping>
```

Mapping avec XML

Association unidirectionnelle one-to-many avec `<set>`

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <set name="rb" cascade="delete-orphan">
            <key column="a_id"/>
            <one-to-many class="B" />

        </set>
    </class>
</hibernate-mapping>
```



bda2/model/B.hbm.xml

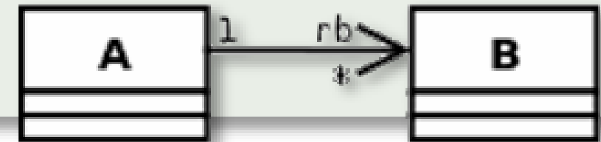
```
<hibernate-mapping>
    <class name="B" table="b"> ... </class>
</hibernate-mapping>
```

Mapping avec XML

Association unidirectionnelle one-to-many avec **<list>**

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <list name="rb" cascade="delete-orphan">
            <key column="a_id"/>
            <one-to-many class="B" />
            <list-index column="idx_rb"/>
        </list>
    </class>
</hibernate-mapping>
```



bda2/model/B.hbm.xml

```
<hibernate-mapping>
    <class name="B" table="b"> ... </class>
</hibernate-mapping>
```

Mapping avec XML

Association bidirectionnelle one-to-many

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a"> ...
        <set name="rb" inverse="true" cascade="delete-orphan">
            <key column="a_id"/>
            <one-to-many class="B" />
        </set>
    </class>
</hibernate-mapping>
```

bda2/model/B.hbm.xml

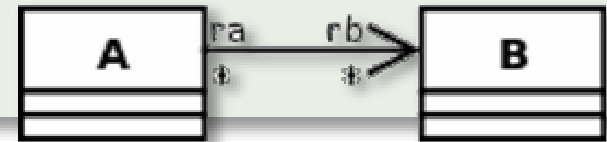
```
<hibernate-mapping>
    <class name="B" table="b"> ...
        <many-to-one name="ra" column="a_id" not-null="true" />
     </class>
</hibernate-mapping>
```

Mapping avec XML

Association unidirectionnelle many-to-many

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <set name="rb" table="a_b" cascade="delete-orphan">
            <key column="a_id"/>
            <many-to-many column="b_id" class="B" />
        </set>
    </class>
</hibernate-mapping>
```



bda2/model/B.hbm.xml

```
<hibernate-mapping>
    <class name="B" table="b"> ... </class>
</hibernate-mapping>
```

Mapping avec XML

Association bidirectionnelle many-to-many (1/2)

bda2/model/A.hbm.xml

```
<hibernate-mapping>
    <class name="A" table="a">
        ...
        <set name="rb" table="a_b" inverse="true"
                            cascade="save-update,delete">
            <key column="a_id"/>
            <many-to-many column="b_id" class="B" />
        </set>
    </class>
</hibernate-mapping>
```

bda2/model/B.hbm.xml

```
...
```

Mapping avec XML

Association bidirectionnelle many-to-many (2/2)

bda2/model/A.hbm.xml

```
...
```

bda2/model/B.hbm.xml

```
<hibernate-mapping>
    <class name="B" table="b">
        ...
        <set name="ra" table="a_b" inverse="true"
                        cascade="save-update,delete">
            <key column="b_id"/>
            <many-to-many column="a_id" class="A" />
        </set>
    </class>
</hibernate-mapping>
```

Configuration de la persistance des données

5. Configurer la persistance de Hibernate (1/2)

Créer le fichier de configuration **dans la racine de** `/src` : `hibernate.cfg.xml`

- définit où trouver les fichiers de mapping
- définit les paramètres d'accès à la BD

`/src/hibernate.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>

        <mapping resource="bda2/model/xxxx.hbm.xml"/>
        ...

    </session-factory>
</hibernate-configuration>
```

Configuration de la persistance des données

5. Configurer la persistance de Hibernate (2/2)

/src/hibernate.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>
        <property name="hibernate.connection.driver_class">
                    oracle.jdbc.driver.OracleDriver</property>
        <property name="hibernate.connection.url">
                    jdbc:oracle:thin:@localhost:1521:XE </property>
        <property name="hibernate.connection.username">ORM_HIBERNATE</property>
        <property name="hibernate.connection.password"> ***** </property>
        <property name="hibernate.dialect">
                    org.hibernate.dialect.Oracle10gDialect </property>
        <property name="current_session_context_class"> thread </property>
        <property name="connection.pool_size"> 1 </property>
        <property name="hbm2ddl.auto"> update </property>
        <property name="hibernate.show_sql"> true </property>
        <property name="hibernate.format_sql"> true </property>

        <mapping resource="bda2/model/xxxx.hbm.xml"/>
        ...
    </session-factory>
</hibernate-configuration>
```

Persistance des données via Hibernate

Etapes de développement :

1. Créer et ouvrir une session
2. Créer une transaction
3. Persister un objet
4. Récupérer des objets persistés
5. Fermer la session

Persistance des données via Hibernate

1. Créer et ouvrir une session

Main.java

```
...
SessionFactory sessionFactory =
                  new Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
...
```

Persistance des données via Hibernate

2. Créer une transaction

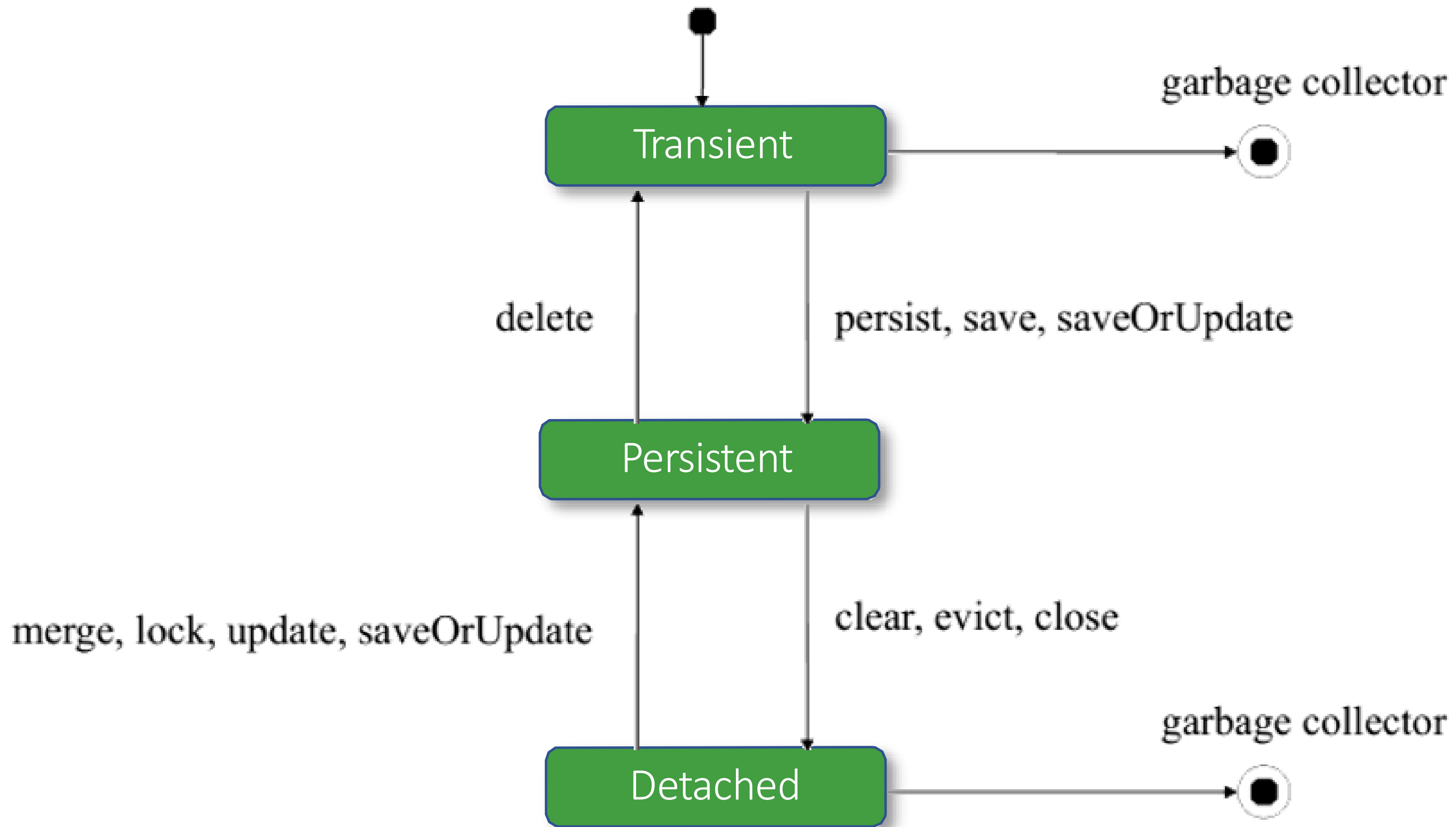
Main.java

```
...
Try {
    Transaction tx = session.beginTransaction();
    ...
    tx.commit();
} catch (HibernateException e) {
    if (tx != null)
        tx.rollback();
}
...
```

Persistance des données via Hibernate

3. Persister un objet (1/4)

- Cycle de vie d'un objet persistant

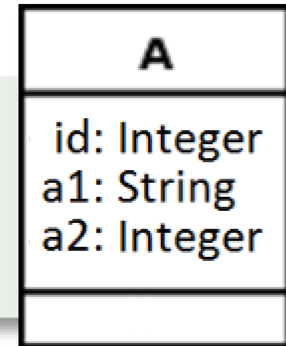


Persistance des données via Hibernate

3. Persister un objet (2/4)

- Insertion :

```
...
A a = new A(1,"attr A1", 100);    // a : Transient
session.persist(a);               // a : Persistent
...
```



- Mise à jour :

```
...
a.setA1("attr A1 updated");       // a : Persistent
...
```

- Suppression :

```
...                               // a : Persistent
session.delete(a);                // a : Transient
...
```

Persistance des données via Hibernate

3. Persister un objet (3/4)

Persistance explicite des objets :

- Persister chaque instance individuellement

Main.java

```
...
A a = new A(1, "attr A1", 100);              // a : Transient
B b = new B(1,"attr B1", "attr B2");         // b : Transient
a.addB(b);

session.persist(a);                          // a.: Persistent
session.persist(b);                          // b.: Persistent
tx.commit();
...
```

Persistance des données via Hibernate

3. Persister un objet (4/4)

Persistance en cascade :

- Propager la persistance à toutes les objets associés

bda2/model/A.hbm.xml

```
<class name="A" table="a">
   ...
   <one-to-one name="rb" class="B" cascade="save-update" ... />
</class>
```

Main.java

```
A.a = new A(1, "attr A1", 100);             // a : Transient
B.b = new B(1,"attr B1", "attr B2");        // b : Transient
a.addB(b);

session.persist(a);                         // a et b : Persistent
tx.commit();
...
```

Persistance des données via Hibernate

4. Récupérer des objets persistés (1/4)

Hibernate génère automatiquement le code SQL

Hibernate fournit 3 stratégies pour interroger la BD :

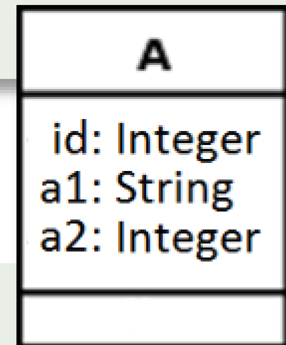
- Langage SQL natif
- Langage HQL
- API Criteria
    - avec des options de fetching et de mise en cache sophistiquées.

Persistance des données via Hibernate

4. Récupérer des objets persistés (2/4)

1 Requête SQL :

```
Query q = session.createSQLQuery("SELECT * FROM a
                                  WHERE a2 < 200");
q.addEntity(A.class);
List<A> result = (List<A>) q.list();
```



2 Requête SQL paramétrée :

```
Query q = session.createSQLQuery("SELECT FROM a
                                  WHERE a2 < ?")
                 .setString(0, "200");
q.addEntity(A.class);
List<A> result = (List<A>) q.list();
```

Persistance des données via Hibernate

4. Récupérer des objets persistés (3/4)

3. Requête HQL : (HQL est la version objet de SQL)

```
Query q = session.createQuery("FROM A obj
                               WHERE obj.a2 < 200");
List<A> result = (List<A>) q.list();
```

A
id: Integer a1: String a2: Integer

4. Requête HQL paramétrée :

```
Query q = session.createQuery("FROM A obj
                               WHERE obj.a2 < :X")
                  .setString("X", "200");
List<A> result = (List<A>) q.list();
```

Persistance des données via Hibernate

4. Récupérer des objets persistés (4/4)

5. API Creteria : (Criteria est une alternative plus élégante à HQL)

```
Criteria cr = session.createCriteria(A.class);
cr.add(Restrictions.gt("a2", 200));

List<A> result = (List<A>) cr.list();
```

A
id: Integer a1: String a2: Integer

```
cr.add(Restrictions.between("a2", 100, 200));
cr.add(Restrictions.like("a1", "%bda%"));
cr.add(Restrictions.isNotNull("a2"));
cr.addOrder(Order.asc("a2") );     // trier le résultat
cr.setFirstResult(20);      // commencer après le 20ème tuple
cr.setMaxResults(10);      // récupérer les 10 prochains tuples
```

Persistance des données via Hibernate

5. Fermer la session

Main.java

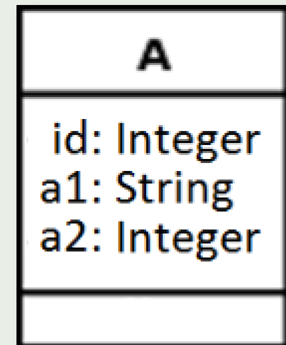
```
...
session.close();
sessionFactory.close();
...
```

Persistance des données via Hibernate

Récapitulatif

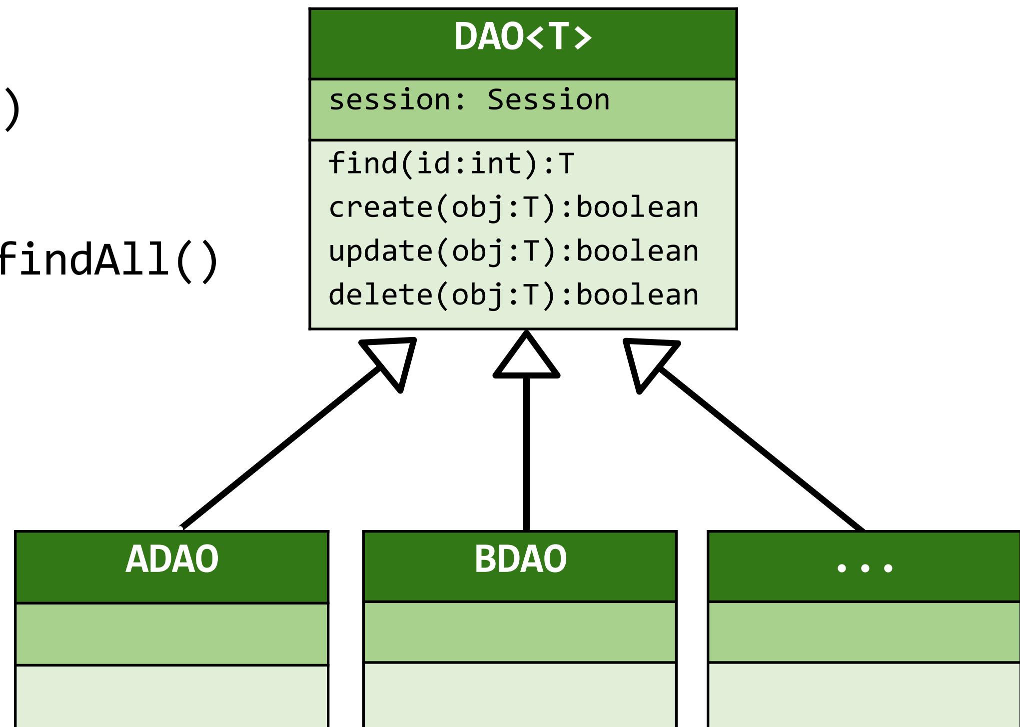
Main.java

```
1 SessionFactory sessionFactory =
              new Configuration().configure().buildSessionFactory();
  Session session = sessionFactory.openSession();

2 Transaction tx = session.beginTransaction();

3 session.persist(a);

  tx.commit();

4 Query q = session.createQuery("FROM A obj
                                  WHERE obj.a2 < 200");
  List<A> result = (List<A>) q.list();

5 session.close(); sessionFactory.close();
```



Design pattern DAO (1/3)

- Chaque classe persistante doit posséder sa classe DAO
- DAO fournit les méthodes de
  - Création : `create(obj)`
  - Suppression : `delete(obj)`
  - Mise à jour : `update(obj)`
  - Interrogation : `find(id)`, `findAll()`



Design pattern DAO (2/3)

Classe **DAO<T>**

```java
public abstract class DAO<T> {
  protected Session session = null;

  public DAO(Session session){
    this.session = session;
  }

  public abstract boolean create(T obj);
  public abstract boolean delete(T obj);
  public abstract boolean update(T obj);
  public abstract T find(int id);
}
```

DAO<T>
session: Session
find(id:int):T create(obj:T):boolean update(obj:T):boolean delete(obj:T):boolean

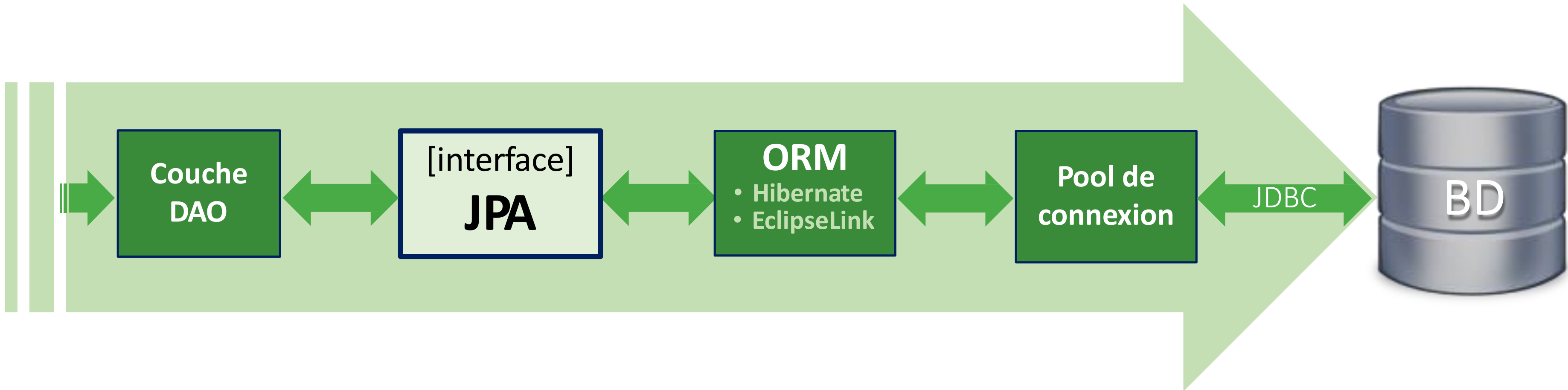
Design pattern DAO (3/3)

Exemple : Classe **ADAO**

```
public class ADAO extends DAO<A>{

  public ADAO(Session session) {
    super(session);
  }

  public boolean create(A obj) { ... }
  public boolean delete(A obj) { ... }
  public boolean update(A obj) { ... }
  public A find(int id) { ... }
  public List<A> findAll() { ... }
}
```

JPA : Java Persistence API (1/4)

- La spécification JPA est **un ensemble d'interface** du package `javax.persistence`



JPA : Java Persistence API (2/4)

JPA

- est un standard Java
- une synthèse standardisée des meilleures technologies de persistance

Devant le succès des Frameworks ORM (Hibernate, EclipseLink, ...)

- Sun a décidé de standardiser une couche ORM via la spécification JPA apparue en même temps que Java 5

Objectifs

- Eliminer le couplage fort entre l'ORM et l'application
- Unifier la configuration du mapping des classes persistances²

JPA : Java Persistence API (3/4)

Implémentations de JPA

- **JPA 1.0 :** Mai 2006
  - Hibernate 3.2, TopLink
- **JPA 2.0 :** Décembre 2009 (collections, criteria query API, …)
  - EclipseLink (l'implémentation référence)
  - Batoo JPA, Hibernate, DataNucleus, ObjectDB, ...
- **JPA 2.1 :** Avril 2013 (Converters, Stored Procedures, JPQL/Criteria, …)
  - EclipseLink
  - DataNucleus AccessPlatform
  - Hibernate
- **JPA 2.2 :** Janvier 2018 (Java 8, Stream, LocalDate, prepare Java 9, …)
  - EclipseLink 2.7.3 (ancienement TopLink) DataNucleus
  - AccessPlatform 5.2 (ancienement JPOX) **Hibernate**
  - **5.3.4**

JPA : Java Persistence API (4/4)

JPA repose sur :

- Des **classes persistantes** (`Entity`) sous la forme de POJOs
- Des **annotations** (ou un fichier xml) pour réaliser le mapping O/R
- Un **gestionnaire de persistance** (`EntityManager`)
- Un **langage de requêtes JPQL** (Java Persistence Query Language)

Mapping Objet Relationnel

JPA vs. Hibernate

	JPA	Hibernate
Classes	Entity	Persistent
Mapping	Annotations/Fichiers .orm.xml	Fichiers .hbm.xml
Configuration	persistence.xml	hibernate.cfg.xml
Factory	EntityManagerFactory	SessionFactory
Manager	EntityManager	Session
Transaction	EntityTransaction	Transaction
Query	Query	Query

Configuration de la persistance des données

1. Préparer le projet
2. Importer l'API de Hibernate **en intégrant Hibernate-JPA**
3. Définir les règles de persistance des classes
4. Rendre les classes persistables via des annotations JPA
5. Configurer la persistance avec JPA

Configuration de la persistance des données

1. Préparer le projet

1. **Créer un nouveau package** `bda.tp.jpa`
2. **Démarrer le serveur Oracle dans "Services"** (si le service est éteint)
3. **Créer un nouvel utilisateur**
   - `ORM_JPA`(+ un schéma)
4. **Ajouter l'API du driver Oracle-JDBC dans le projet**
   - **(NetBeans)** Librairies > Add Jar/Folder… > Choisir le chemin : "`ojdbc7.jar`"

Configuration de la persistance des données

2. Importer l'API de Hibernate

Ajouter l'API de Hibernate

- Librairies > Add Jar/Folder… > Choisir les fichiers :
  - hibernate-core-5.3.4.Final.jar
  - javassist-3.23.1-GA.jar
  - antlr-2.7.7.jar
  - dom4j-1.6.1.jar
  - jandex-2.0.5.Final.jar
  - hibernate-jpamodelgen-5.3.4.Final.jar
  - hibernate-commons-annotations-5.0.4.Final.jar
  - javax.persistence-api-2.2.jar
  - javax.activation-api-1.2.0.jar
  - byte-buddy-1.8.15.jar
  - jboss-logging-3.3.2.Final.jar
  - jboss-transaction-api_1.2_spec-1.1.1.Final.jar
  - classmate-1.3.4.jar
  - ...

Configuration de la persistance des données

3. Définir les règles de persistance des classes

Règles des classes persistantes :

- Basée sur des classes **POJO (Plain Old Java Object)**
- Implémenter un **constructeur vide** (sans arguments)
- Si un attribut n'est pas persistable, alors il est marqué par **`@Transient`**
- Fournir une propriété d'**identifiant**
- Il n'est pas possible de combiner **champs** et **propriétés** dans la **même entité**
  - **Propriété (Property) :** Attribut <u>privé</u> manipulé avec un getter et un setter
  - **Champ (Field) :** Attribut <u>publique</u>
- Implémenter **`equals()`** ou **`hashCode()`** :
  - en utilisant **l'égalité des identifiants** (clés primaires)

Configuration de la persistance des données

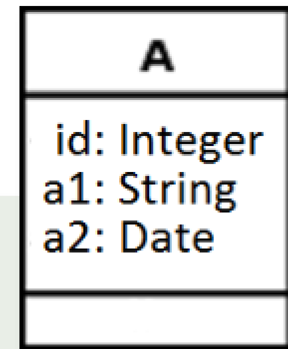
4. Rendre les classes persistables

Dans JPA, il y a **2 méthodes pour persister des entités**, en utilisant :

1. **Des annotations**, ou
2. **Un fichier de mapping XML** (`NomClasse.orm.xml`)
   - Similaire aux fichiers de mapping de Hibernate (**`NomClasse.hbm.xml`**)

Mapping avec des annotations JPA

Une classe



bda2/model/A.java

```
import javax.persistence.*;

@Entity
@Table(name="a")
public class A {

    @Id
    @Column(name="id")
    int id;
    String a1;
    Date a2;
}
```

Mapping avec des annotations JPA

Stratégies d’héritage (1/2)

JPA supporte les **4 stratégies d'héritage** de base :

Une table par hiérarchie de classe (Single Table) :

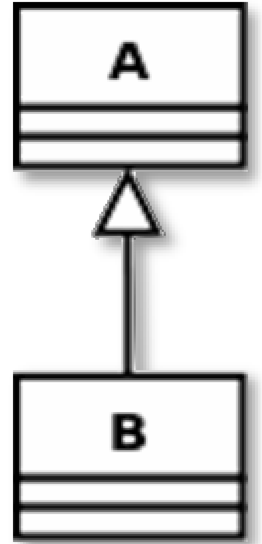
- Une seule table englobe toutes les instances de l’hiérarchie

Une table par sous-classe (Joined, Multiple Table) :

- Chaque table d’une sous-classe ne persiste que les propriétés spécifiques à cette sous-classe

Une table par classe concrète :

- **Table Per Class :** Chaque table persiste les propriétés de la classe et les propriétés de ses super-classes
- **Mapped Superclasses :** Le manager ne gère pas les super-classes (entités non-persistantes)



Documentation :

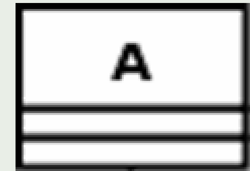
- https://www.thoughts-on-java.org/complete-guide-inheritance-strategies-jpa-hibernate/

Mapping avec des annotations JPA

Stratégies d'héritage (2/2) – Une table par hiérarchie de classes

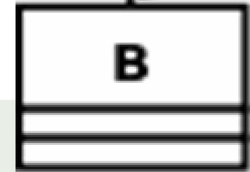
bda2/model/A.java

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="discriminator")
@DiscriminatorValue("a")
public class A {
    ...
}
```



bda2/model/B.java

```
@Entity
@DiscriminatorValue("b")
public class B extends A {
    ...
}
```



Mapping avec des annotations JPA

Persistance des associations

Types d'associations :

- **Cardinalités des associations :** 1-1, 1-N, N-1, M-N
  - `@OneToOne, @OneToMany, @ManyToOne, @ManyToMany`
- **Navigabilité :**
  - **Unidirectionnelle ou bidirectionnelle**
- **Collections :**
  - `collection, set, list, map`

Mapping avec des annotations JPA

Association unidirectionnelle one-to-one

bda2/model/A.java

```
@Entity
public class A {

    @OneToOne
    @JoinColumn(unique=true, name="a_id")
    B rb;
}
```

bda2/model/B.java

```
@Entity
public class B {
    ...
}
```

Mapping avec des annotations JPA

Types de cascade (1/2)

JPA supporte **5 types de cascade** :

`cascade=CascadeType.PERSIST :`

- si **`em.persist()`** est appelée dans le parent, et le fils est aussi nouveau, alors le fils est aussi persisté

`cascade=CascadeType.REMOVE :`

- si **`em.remove()`** est appelée dans le parent, alors le fils est aussi supprimé

`cascade=CascadeType.MERGE :`

- si **`em.merge()`** est appelée dans le parent, alors le fils est aussi fusionné (merged)

`cascade=CascadeType.REFRESH :`

- si **`em.refresh()`** est appelée dans le parent, alors le fils est aussi actualisé (refreshed)

`cascade=CascadeType.ALL :`

- Toutes les opérations précédentes sont prises en compte

Mapping avec des annotations JPA

Types de cascade (2/2)

Exemple :

bda2/model/A.java

```
@Entity
public class A {
    @OneToOne(cascade=CascadeType.REMOVE ...)
    @JoinColumn(unique=true, name="a_id")
    B rb;
}
```

- Quand un objet de type A **est supprimé**, alors l'objet de type B associé **est aussi supprimé en cascade.**

Mapping avec des annotations JPA

Stratégies de chargement (Fetching) (1/3)

JPA permet de paramétrer **QUAND** **charger les données**, (relation par relation) :

- doit-il la charger par défaut ?
- doit-il la laisser vide, et la charger à la demande ?

JPA supporte **2 types de fetching :**

1. `fetch=FetchType.LAZY`
   - indique que la relation doit être chargée à la demande
2. `fetch=FetchType.EAGER` (par défaut) :
   - indique que la relation doit être chargée en même temps que l'entité qui la porte

Mapping avec des annotations JPA

Stratégies de chargement (Fetching) (2/3)

Sous JPA, il y a **3 modes COMMENT charger les données :**

1. `@Fetch(FetchMode.SELECT)` (par défaut)
   - JPA récupère les données associée dans un **second** `SELECT`

2. `@Fetch(FetchMode.SUBSELECT)`
   - JPA récupère les données associée dans un **même** `SELECT` à l'aide d'un `OUTER JOIN`

3. `@Fetch(FetchMode.JOIN)`
   - JPA récupère les associations pour toutes les entité récupérées dans une requête dans un **second** `SELECT`

Mapping avec des annotations JPA

Stratégies de chargement (Fetching) (3/3)

Exemple :

bda2/model/A.java

```
@Entity
public class A {
    @OneToOne(fetch=FetchType.LAZY ...)
    @JoinColumn(name="a_id", unique=true, nullable=false)
    @Fetch(FetchMode.JOIN)
    B rb;   ...
}
```

`fetch=FetchType.LAZY`

- Quand un objet de type A est chargé à partir de la BD, JPA charge l'objet de type B associé à la demande c-à-d. **seulement s'il est utilisé dans le code Java**

`FetchMode.JOIN`

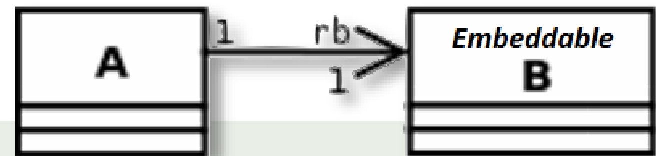
- Quand un objet de type A associé avec un objet de type B sont chargés, JPA génèrera implicitement une seule requête de jointure **`SELECT`** en utilisant **`OUTER JOIN`**

Mapping avec des annotations JPA

Association unidirectionnelle one-to-one "Embedded"

bda2/model/A.java

```
@Entity
public class A {
    @Embedded
    B rb;
}
```



bda2/model/B.java

```
@Embeddable
public class B {
    @Transient
    int id;
    ...
}
```

Mapping avec des annotations JPA

Association bidirectionnelle one-to-one

bda2/model/A.java

```
@Entity
public class A {

   @OneToOne(mappedBy="ra", cascade=CascadeType.ALL)
   B rb;
}
```

bda2/model/B.java

```
@Entity
public class B {

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(unique=true, name="a_id")
    A ra;
}
```

Mapping avec des annotations JPA

Association unidirectionnelle one-to-many <set>

bda2/model/A.java

```
@Entity
public class A {

    @OneToMany
    @JoinColumn(name="a_id")
    Set<B> rb;
}
```

bda2/model/B.java

```
@Entity
public class B {
    ...
}
```

Mapping avec des annotations JPA

Association unidirectionnelle one-to-many `<list>`

bda2/model/A.java

```
@Entity
public class A {
    @OneToMany
    @JoinColumn(name="a_id")
    @IndexColumn(name="idx_rb", base=0, nullable=false)
    List<B> rb;
}
```

bda2/model/B.java

```
@Entity
public class B {
    ...
}
```

Mapping avec des annotations JPA

Association bidirectionnelle one-to-many

bda2/model/A.java

```
@Entity
public class A {

    @OneToMany(mappedBy="ra", cascade=CascadeType.ALL)
    Set<B> rb;
}
```

bda2/model/B.java

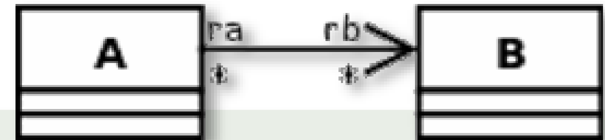
```
@Entity
public class B {

    @ManytoOne(cascade=CascadeType.ALL)
    @JoinColumn(name="a_id")
    A ra;
}
```

Mapping avec des annotations JPA

Association unidirectionnelle many-to-many

bda2/model/A.java

```
@Entity
public class A {
    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="a_b", joinColumns=@JoinColumn(name="a_id"),
               inverseJoinColumns=@JoinColumn(name="b_id"))
    Set<B> rb;
}
```



bda2/model/B.java

```
@Entity
public class B {
    ...
}
```

Mapping avec des annotations JPA

Association bidirectionnelle many-to-many

bda2/model/A.java

```
@Entity
public class A {
    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="a_b", joinColumns=@JoinColumn(name="a_id"),
               inverseJoinColumns=@JoinColumn(name="b_id"))
    Set<B> rb;
}
```

bda2/model/B.java

```
@Entity
public class B {
    @ManyToMany(mappedBy="rb", cascade=CascadeType.ALL)
    Set<A> ra;
}
```

Configuration de la persistance des données

5. Configurer la persistance de JPA (1/2)

1. Créer un nouveau package dans la racine `/src` : **`"META-INF"`**
2. Créer et mettre dedans un fichier xml : **`"persistence.xml"`**

/src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="..." xmlns:xsi="..." xsi:schemaLocation="..." version="2.0">
    <persistence-unit name="...">
        <provider> org.hibernate.ejb.HibernatePersistence </provider>
        <class>...</class>
        ...
        <properties>
            <property name="..." value="..."/>
        </properties>
    </persistence-unit>
</persistence>
```

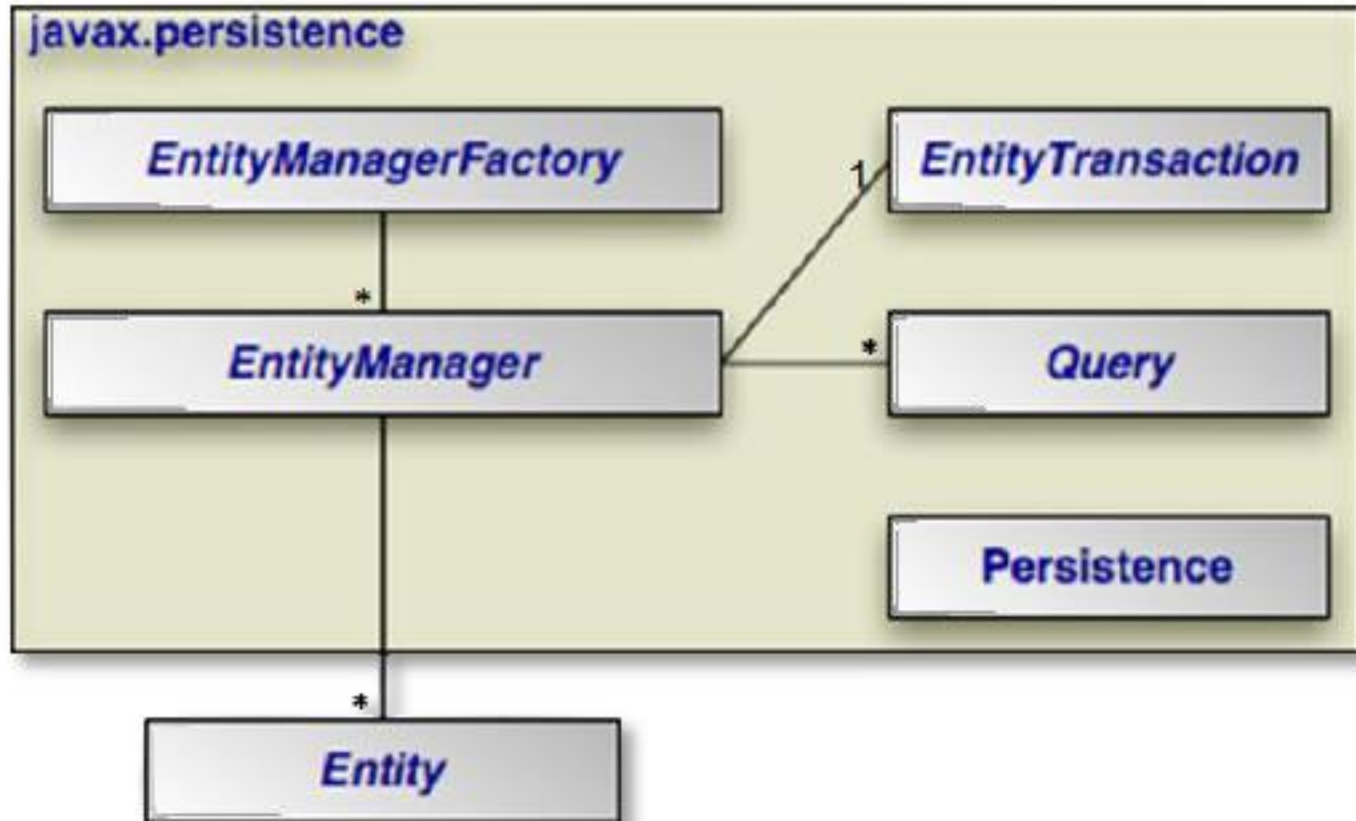
Configuration de la persistance des données

5. Configurer la persistance de JPA (2/2)

/src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"   version="1.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
>
    <persistence-unit name="orm-unit">
        <provider> org.hibernate.ejb.HibernatePersistence </provider>
        <class>bda.model.XXXX</class>
        ...
        <properties>
            <property name="hibernate.connection.driver_class"
                      value="oracle.jdbc.driver.OracleDriver"/
                      >
            <property name="hibernate.connection.url"
                      value="jdbc:oracle:thin:@localhost:1521:XE" />
            <property name="hibernate.connection.username" value="ORM_JPA" />
            <property name="hibernate.connection.password" value="*****"/>
            <property name="hibernate.dialect"
                      value="org.hibernate.dialect.Oracle10gDialect"/>
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>

</persistence>
```

Api de JPA



Persistance de données via JPA

Etapes de la persistance via JPA :

1. Ouvrir une session (`EntityManager`)
2. Créer une transaction
3. Persister un objet
4. Récupérer des objets persistés
5. Fermer la session (`EntityManager`)

Persistance de données via JPA

1. Ouvrir une session

Main.java

```
...
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("orm-unit");
EntityManager em = emf.createEntityManager();
...
```

Persistance de données via JPA

2. Créer une transaction

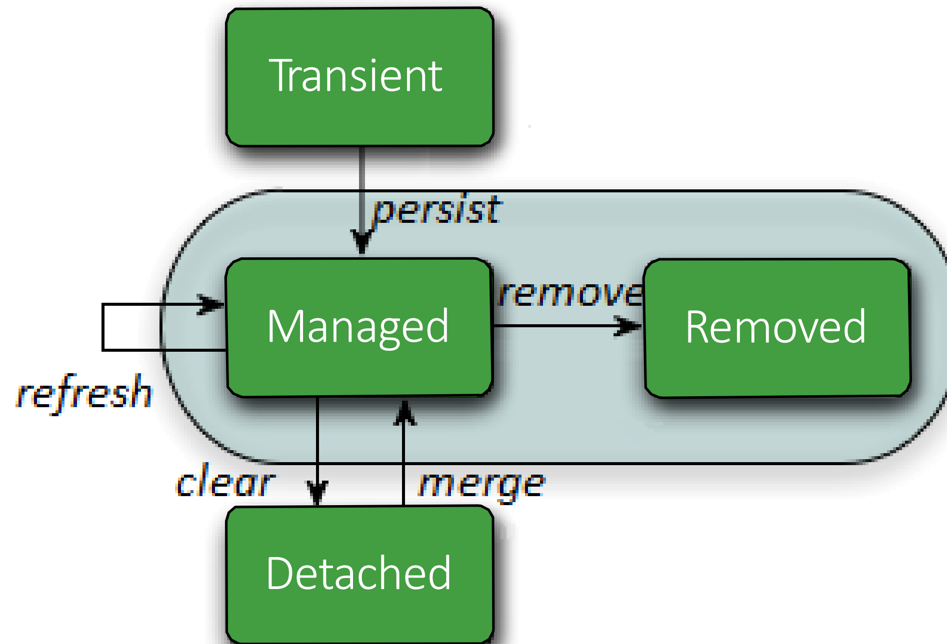
Main.java

```
EntityTransaction tx = em.getTransaction();
try{
    tx.begin();

    ...

    tx.commit();

} catch (Exception e){
    if(tx.isActive())
        tx.rollback();
}
```

Persistance de données via JPA

3. Persister un objet (1/4)

- Cycle de vie d'un objet persistant

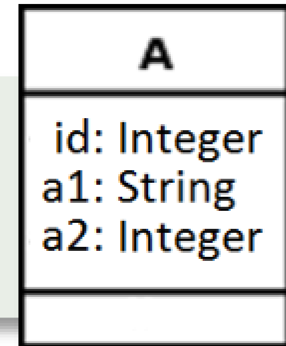


Persistance de données via JPA

3. Persister un objet (2/4)

- Insertion :

```
...                                   100);    // a : Transient
A a = new A(1,"attr A1",
em.persist(a);                                 // a : Managed
...
```



- Mise à jour :

```
                                               // a : Managed
```

- Suppression :

```
...                                            // a : Managed
em.remove(a);                                  // a : Removed
...
```

Persistance de données via JPA

3. Persister un objet (3/4)

Persistance explicite des objets :

- Persister chaque instance individuellement

Main.java

```
...
A a = new A(1, "attr A1", 100);              // a : Transient
B b = new B(1,"attr B1", "attr B2");        // b : Transient
a.addB(b);

em.persist(a);                               // a.: Managed
em.persist(b);                               // b.: Managed
tx.commit();
...
```

Persistance de données via JPA

3. Persister un objet (4/4)

Persistance en cascade :

- Propager la persistance à toutes les objets associés

bda2/model/A.java

```
...
@OneToMany(cascade=CascadeType.PERSIST)
@JoinColumn(name="a_id")
Set<B> rb;
```

Main.java

```
A.a = new A(1, "attr A1", 100);              // a : Transient
B.b = new B(1,"attr B1", "attr B2");         // b : Transient
a.addB(b);

em.persist(a);                               // a et b : Managed
tx.commit();
...
```

Persistance de données via JPA

4. Récupérer des objets persistés (1/4)

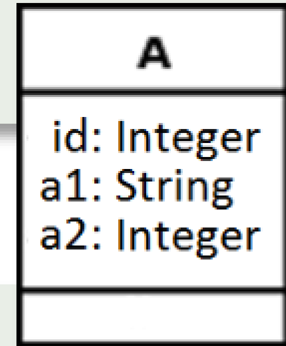
- JPA fournit 3 stratégies pour interroger la BD :
- Langage SQL natif
- Langage JPQL
- API Criteria
  - avec des options de fetching et de mise en cache sophistiquées

Persistance de données via JPA

4. Récupérer des objets persistés (2/4)

1 Requête SQL :

```
Query q = em.createNativeQuery("SELECT * FROM a
                                WHERE a2 < 200");
List<A> result = (List<A>) q.getResultList();
or
Stream<A> result = (List<A>) q.getResultStream();
```



2 Requête SQL paramétrée :

```
Query q = em.createNativeQuery("SELECT FROM a
                                WHERE a2 < ?")
                     .setParameter(0, "200");
List<A> result = (List<A>) q.getResultList();
```

Persistance de données via JPA

4. Récupérer des objets persistés (3/4)

3. Requête JPQL : (JPQL est un langage défini par la spécification JPA)

```
Query q = em.createQuery("FROM A obj
                           WHERE obj.a2 < 200");
List<A> result = (List<A>) q.getResultList();
```

A
id: Integer a1: String a2: Integer

4. Requête JPQL paramétrée :

```
Query q = em.createQuery("FROM A obj
                           WHERE obj.a2 < :X")
                  .setParameter("X", "200");
List<A> result = (List<A>) q.getResultList();
```

Persistance de données via JPA

4. Récupérer des objets persistés (4/4)

5. API Creteria :

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<A> cq = cb.createQuery(A.class);
Root<A> a = cq.from(A.class);
cq.select(a);
cq.where(cb.gt(a.get("a2"), 200));
TypedQuery<A> q = em.createQuery(cq);
List<A> result = q.getResultList();
```

A
id: Integer a1: String a2: Integer

```
cq.where(cb.between(a.get("a2")), 100, 200);
cq.where(cb.like(a.get("a1")), "%bda%");
cq.where(cb.isNotNull(a.get("a2")));
cq.orderBy(cb.asc(a.get("a2")));     // trier le résultat
q.setFirstResult(20);      // commencer après le 20ème tuple
q.setMaxResults(10);      // récupérer les 10 prochains tuples
```

Persistance de données via JPA

5. Fermer la session

Main.java

```
...
em.close();
emf.close();
...
```

Persistance de données via JPA

Récapitulatif

Main.java

1
```java
EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("orm-unit");
EntityManager em = emf.createEntityManager();
```

2
```java
EntityTransaction tx = em.getTransaction();
tx.begin();
```

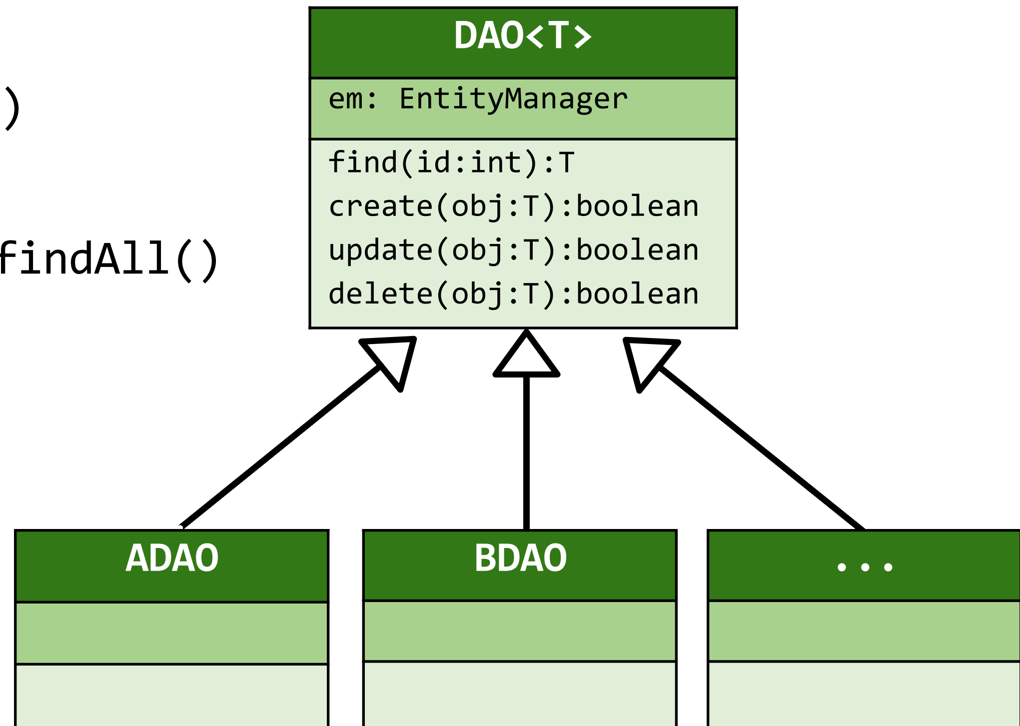
3
```java
em.persist(p);

tx.commit();
```

4
```java
Query q = em.createQuery("FROM A obj
                          WHERE obj.a2 < 200");
List<A> results = (List<A>) q.getResultList();
```

5
```java
em.close(); emf.close();
```

Design pattern DAO (1/3)

- Chaque entité doit posséder sa classe DAO
- DAO fournit les méthodes de
  - Création : `create(obj)`
  - Suppression : `delete(obj)`
  - Mise à jour : `update(obj)`
  - Interrogation : `find(id)`, `findAll()`



Design pattern DAO (2/3)

Classe DAO<T>

```
public abstract class DAO<T> {
  protected EntityManager em = null;

  public DAO(EntityManager em){
    this.em = em;
  }

  public abstract boolean create(T obj);
  public abstract boolean delete(T obj);
  public abstract boolean update(T obj);
  public abstract T find(int id);
}
```

DAO<T>
em: EntityManager
find(id:int):T create(obj:T):boolean update(obj:T):boolean delete(obj:T):boolean

Design pattern DAO (3/3)

Exemple : Classe **ADAO**

```
public class ADAO extends DAO<A>{

  public ADAO(EntityManager em) {
    super(em);
  }

  public boolean create(A obj) { ... }
  public boolean delete(A obj) { ... }
  public boolean update(A obj) { ... }
  public A find(int id) { ... }
  public List<A> findAll() { ... }
}
```