Persistance des données en Java

Plan

Plan

Plan

Correspondance Objet/Relationnel

- Ce cours explique les problèmes de base qui se posent quand on veut faire correspondre :
  les données contenues dans un modèle objet avec les données contenues dans une base de données relationnelle
- Dans le TP1 vous avez vu en partie comment faire correspondre manuellement un modèle relationnel avec un modèle objet.
- Dès qu'un modèle objet est complexe (de l'héritage et beaucoup d'associations), il n'est pas simple de lui faire correspondre un modèle relationnel.
- Nécessité d'un outil pour définir le plus simplement possible cette correspondance : JPA.

Quelques problèmes du passage Relationnel $\leftrightarrow$ Objet

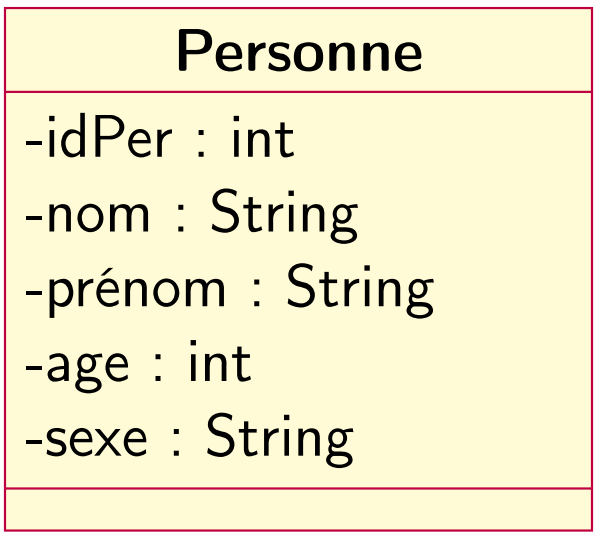
- Identité des objets (pas de notion de clef)
- Traduction des différents types d'associations (hiérarchiques, non hiérarchiques, ...)
- Navigation entre les objets (restriction de la navigabilité)
- Traduction de l'héritage

Problème de base

- Un objet a une structure complexe qui peut être représentée par un graphe.
- Le plus souvent ce graphe est un arbre dont la racine correspond à l'objet et les fils correspondent aux valeurs de ses variables d'instance.
- Il faut « aplatir » ce graphe pour le ranger dans la base de données relationnelle sous une forme tabulaire.
- Le chargement d'un seul objet peut demander de récupérer un grand nombre de tuples situés dans des relations différentes.

Traduction d'une classe en une relation

- Dans les cas les plus simples (lorsque toutes les données membres ont un type simple) une classe est traduite en une relation.
- Chaque objet est conservé dans un tuple de la relation et les données membres sont traduites en attributs.
- Exemple : la classe Personne est traduite par la relation : `Personne(IdPer, Nom, Prénom, Age, Sexe)`



Identification dans le monde Objet

- Tout objet est identifiable simplement dans les langages objets par l'adresse de son emplacement mémoire, indépendamment des données qu'il contient.
- Deux objets distincts peuvent avoir exactement les mêmes valeurs pour leurs propriétés.
- Il y a donc une différence conceptuelle entre deux objets qui sont égaux (`o1.equals(o2)`) et deux objets qui sont identiques (`o1==o2`)

Identification dans le monde relationnel

- Seules les valeurs des attributs peuvent servir à identifier un tuple.
- Si deux tuples ont les mêmes valeurs, il est impossible de les différencier. Ils sont donc considérés identiques.
- Une clef primaire est le plus petit ensemble d'attributs permettant d'identifier de manière non équivoque tous les tuples d'une relation.
- Toute relation doit donc comporter une clé primaire pour identifier facilement un tuple parmi tous les autres tuples de la même relation.
- Deux tuples égaux étant considéré identique, une relation ne contiendra pas de doublon.

Clef primaire pour les objets

- Pour des raisons d'efficacité, on fait en sorte que les relations aient une clef simple(un Id numérique).
- Cet attribut devra être représenté dans une donnée membre de chaque classe.
- En effet, il pourra être utilisé par le code pour un accès rapide aux données ou pour aider à gérer le mapping objet-relationnel.
- Cette propriété servant à l'identification d'un objet, on ne doit pas pouvoir la modifier après la création de l'objet (pas de méthode `setId`() publique).

Problème des duplications

- Une même ligne de la base de données ne doit pas être représentée par plusieurs objets en mémoire (ou alors le code doit le savoir et en tenir compte).
- Si elle n'est pas gérée, cette situation peut conduire à des pertes ou des incohérences de données.
- Par exemple, Un objet p_1 de la classe `Produit` de code 1003 est chargé en mémoire à l'occasion d'une navigation à partir d'une ligne de `Facture`.
- On peut retrouver le même produit en navigant depuis une autre facture, ou en cherchant tous les produits qui vérifient un certain critère.

Problème des duplications (suite)

- Si l'on crée en mémoire centrale un autre objet p_2 indépendant du premier objet p_1 déjà créé, il peut arriver que les deux objets soient modifiés en même temps par deux parties du code. Dans ce cas on ne peut plus savoir lequel des objets est dans l'état correct.
- Pour éviter ce problème, on doit généralement conserver en mémoire un "cache" des objets déjà chargés.
- Ainsi si l'on à besoin d'un objet, on vérifie toujours s'il en existe pas déjà un qui possède la même identité dans le cache.
- C'est le pattern "Unit of Work" de Martin Fowler (Oncle Bob).

Objets « intégrés »

- Les instances de certaines classes peuvent être sauvegardées dans la même relation qu'une autre classe.
- Ces instances sont appelées des objets intégrés (embedded en anglais, qui signifie intégré, imbriqué) et ne nécessitent pas d'identificateur « clé primaire ».
- Par exemple, une classe `Adresse` peut ne pas avoir de correspondance sous la forme d'une relation séparée dans le modèle relationnel.
- Les attributs de la classe `Adresse` sont intégrés dans la relation qui représente la classe `Client`
- Les objets de la classe `Adresse` n'ont pas d'identification liée à la base de données.

Les associations en objet

Dans le code objet une association peut être représentée par :

- Une variable d'instance référençant l'objet associé (association 1 :1 ou N :1)
- Une variable d'instance de type collection ou map contenant les objets associés (association 1 :N ou M :N)
- Une classe « association » (M :N)

Par exemple pour une association 1 :N :

```
class Departement {
  private Collection<Employe> employes;
   ...
}
```

```
class Employe {
  private Departement departement;
   ...
}
```

Association en relationnel

Dans le monde relationnel, une association peut être représentée par :

- une ou plusieurs clés étrangères (associations hiérarchique 1 :1, N :1 ou 1 :N).
- une table association (associations non hiérarchique M :N le plus souvent)

Navigabilité des associations

- Dans un modèle objet, une association peut être bi ou unidirectionnelle.
- Exemple d'association unidirectionnelle : la classe `Employé` a une variable d'instance donnant son département mais la classe `Département` n'a pas de collection d'employés.
- En partant d'un département, on n'a alors pas de moyen simple de retrouver ses employés.
- Dans le modèle relationnel, toutes les associations sont bidirectionnelles.
- La navigation se fait grâce à des jointures.

Objet dépendant

- Un objet dont le cycle de vie dépend du cycle de vie d'un autre objet auquel il est associé, appelé objet propriétaire.
- Aucun autre objet que le propriétaire ne doit avoir de référence directe vers un objet dépendant.
- En ce cas, la suppression de l'objet propriétaire doit déclencher la suppression des objets qui en dépendent (déclenchement « en cascade »).
- Par exemple, une ligne de facture ne peut exister qu'associée à une (en-tête de) facture.
- Si on supprime une facture, toutes les lignes de la facture doivent être supprimées.

Framework de mapping objet/relationnel I

La correspondance entre le modèle objet et le modèle relationnel n'est pas une tâche facile. Il est nécessaire de posséder des outils pour maîtriser cette tâche. L'outil aura les qualités suivantes :

- **Objet et pas relationnel** : L'application doit être écrite en terme de domaine métier en ne pas se borner à une modélisation relationnelle. Donc on doit pouvoir développer en faisant abstraction des concepts relationnels comme les clefs, les relations ou les tuples.
- **Facile mais pas inculte** : L'outil de mapping doit aider le développeur à interagir avec un SGBD-R pas le rendre ignorant des mécanismes sous-jacents. Pour bien utiliser la persistance, il faut comprendre le fonctionnement des BD.

Framework de mapping objet/relationnel II

- **Non intrusif mais pas transparent** : Il n'est pas raisonnable de demander à la persistance d'être totalement transparente (automatique) car l'application doit toujours garder le contrôle du cycle de vie de ses objets. Néanmoins, il ne faut pas non plus que le code soit trop profondément dépendant de l'outil de persistance car le code métier (principal) serait noyé au milieu du code de persistance (juste utilitaire).
- **Suffisant mais pas excessif** : Le développeur d'une application à des problèmes à résoudre et il souhaite un outil assez complet pour remplir tous ses besoins même les plus complexes. Mais il ne souhaite pas devoir configurer lourdement un outil pour ses besoins les plus simples.
- **Local mais mobile** : Les objets persistants ne doivent pas être des vrais objets distribués mais il faut pouvoir en récupérer une copie peu importe l'endroit ou l'on se trouve.

JPA2

- JPA2 (Java Persistence API version 2.0) est un standard pour la persistance des objets en Java.
- JPA2 est le plus souvent utilisé dans le contexte d'un serveur d'applications (Java EE).
- Ce cours étudie l'utilisation de JPA2 par une application classique (Java SE).
- Comme pour JDBC, l'utilisation de JPA2 nécessite un fournisseur de persistance qui implémente les classes et méthodes de l'API.
- Nombreuses implémentations commerciale et libre de cette spécification : Hibernate, TopLink, OpenJPA, EclipseLink, ...

Qualités de JPA I

Le modèle de JPA est simple et élégant, puissant et flexible. Les points forts de cette API sont les suivants :

- Les opérations principales de l'API sont contenues dans un petit nombre de classes faciles à apprendre et à comprendre.
- La persistance est basée uniquement sur des POJO (bons vieux objets java) donc il n'est pas nécessaire d'étendre ou implémenter quoi que ce soit pour gérer la persistance. Globalement tout objet java avec un constructeur par défaut peut devenir persistant.
- JPA est non intrusive, les objets métier n'ont pas besoin d'être modifié pour devenir persistant.
- Un langage de requête Objet proche du SQL permet de facilement récupérer les objets correspondant à des critères donnés.

Qualités de JPA II

- La configuration est relativement simple, par défaut JPA propose une convention pouvant convenir à la majorité des projets. On ne doit configurer l'outil que si la convention ne s'adapte pas à notre cas.
- JPA n'utilisant que des spécifications disponibles dans Java SE, il est relativement simple à tester.

Entités

Pour JPA, une entité est une classe dont les instances peuvent être rendues persistantes.

Entités

Pour JPA, une entité est une classe dont les instances peuvent être rendues persistantes.

```
public class Employe {
    private int id;
    private String nom;
    private long salaire;
    private Departement departement;
    public Employe() {}
    public Employe(int id) { this.id = id; }
    public int getId() { return this.id; }
    private void setId(int id) { this.id = id; }
    public String getNom() { return this.nom; }
    public void setNom(String nom) { this.nom = nom; }
    public int getSalaire() { return this.salaire; }
    public void setSalaire(long salaire) { this.salaire =salaire; }
    public Departement getDepartement() { return this.departement; }
    public void setDepartement(Departement dept) { this.departement = dept; }
}
```

Entités

Pour transformer `Employe` en entité il suffit de lui rajouter les annotations suivantes :

Entités

Pour transformer `Employe` en entité il suffit de lui rajouter les annotations suivantes :

```
@Entity
public class Employe {
    @Id private int id;
    private String nom;
    private long salaire;
    @ManyToOne private Departement departement;
    public Employe() {}
    public Employe(int id) { this.id = id; }
    public int getId() { return this.id; }
    private void setId(int id) { this.id = id; }
    public String getNom() { return this.nom; }
    public void setNom(String nom) { this.nom = nom; }
    public int getSalaire() { return this.salaire; }
    public void setSalaire(long salaire) { this.salaire =salaire; }
    public Departement getDepartement() { return this.departement; }
    public void setDepartement(Departement dept) { this.departement = dept; }
}
```

Entité

- Par convention cette entité sera associée à la relation `EMPLOYE`.
- les propriétés seront stockés dans des attributs de même nom avec le type SQL adapté.
- L'annotation `@Entity` permet d'indiquer à JPA que la classe pourra être rendue persistante.
- L'annotation `@Id` précise la donnée membre qui sera utilisée comme identifiant (clef primaire).
- L'annotation `@ManyToOne` précise que la donnée membre `departement` est une association hiérarchique de type N :1.
- Pour sauvegarder l'état d'une instance de `Employe`, il faudra faire un appel à l'API JPA.

`EntityManager`

Pour rendre persistant des objets, il faut utiliser les classes fournies par JPA. Parmi toutes ces classes la plus importante est le Gestionnaire d'entité `EntityManager`.

- Le gestionnaire d'entités (GE) est l'interlocuteur principal pour le développeur.
- Il fournit les méthodes pour gérer les entités : les rendre persistantes, les supprimer de la base de données, retrouver leurs valeurs dans la base, etc.
- Le GE est créé par la classe `EntityManagerFactory` en utilisant l'unité de persistance définie dans le fichier `persistence.xml`
- On peut configurer l'unité de persistance pour que les relations soient automatiquement générées au premier lancement de l'application.

Récupération d'un gestionnaire d'entités

Avant de récupérer le gestionnaire d'entité, il faut construire la fabrique de gestionnaire d'entité associée à notre unité de persistance (équivalent d'une connexion en JDBC).

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("persUnit");
```

Le nom passé en paramètre est celui de l'unité de persistance définie dans le fichier `persistence.xml`. À partir de cette fabrique on peut donc construire notre GE :

```
EntityManager em = emf.createEntityManager();
```

Grâce à cet objet nous allons pouvoir commencer à travailler avec les entités persistantes.

Persistance d'une entité

- Par défaut un objet de la classe `Employe` n'est pas persistant. C'est une entité dite `transient`.
- Pour rendre une entité `transient` persistante, il faut l'enregistrer au près du GE avec la méthode `persist()`.
- Une fois persistant, l'objet est transformé en un tuple de la relation `EMPLOYE`.

```
Employe emp = new Employe(123);
em.persist(emp);
```

- Cette méthode peut lever une exception non contrôlée du type `PersistenceException`.

Persistance d'une entité

- L'exemple ci-dessous incorpore ces éléments dans une méthode permettant de créer directement des entités persistantes. On supposera que cette méthode appartient à une classe ayant le GE en donnée membre.

```
public Employe createEmploye(int id,
                             String nom, long salaire) {
    Employe emp = new Employe(id);
    emp.setNom(nom);
    emp.setSalaire(salaire);
    em.persist(emp);
    return emp;
}
```

Rechercher des entités

- Une fois les entités dans la base de données, l'étape suivante est de pouvoir la retrouver ultérieurement.
- Pour retrouver une entité à partir de son identifiant, il suffit d'appeler la méthode `find()` du GE.

```
Employe emp = em.find(Employe.class, 123);
```

- L'entité retourné par le GE est une entité gérée (managed).
- Si le GE ne trouve aucune entité correspondant au paramètre passé à la méthode `find()`, il retourne simplement la valeur `null`.
- La méthode de recherche d'un employé :

```
public Employe findEmploye(int id) {
    return em.find(Employe.class, id);
}
```

Supression d'une entité

- Généralement les données ne sont que rarement totalement effacées de la BD. Le stockage coûtant peu, les entités seront archivé au lieu d'être supprimées.
- Pour qu'un objet puisse être supprimé, il faut que ce soit une entité gerée (managed).

```
Employe emp = em.find(Employe.class, 123);
em.remove(emp);
```

- La methode suivante supprime un employé à partir de son Id :

```
public void removeEmploye(int id) {
    Employe emp = em.find(Employe.class, id);
    if (emp != null)
        em.remove(emp);
}
```

Mise à jour d'une entité I

- Il existe plusieurs manière de mettre à jour une entité avec JPA, nous allons voir la plus commune : nous avons une entité gerée (managed) et nous voulons la modifier.

```
Employe emp = em.find(Employe.class, 123);
emp.setSalaire(emp.getSalaire()+1000);
```

- Contrairement aux autres modifications, la mise à jour ne passe pas par le GE. L'objet est modifié et ses changements sont directement répercutés dans la BD.
- Ceci est possible car `emp` n'est pas une simple instance de `Employe` mais une entité gérée. Le GE est donc informé dès qu'une modification est faire à l'entité et il exécute directement l'ordre SQL `Update`.

Mise à jour d'une entité II

- La methode suivante augmente le salaire de l'employé d'identifiant `id` d'un montant `raise` :

```
public void raiseEmployeSalaire(int id, long raise) {
    Employe emp = em.find(Employe.class, id);
    if (emp != null) {
        emp.setSalaire(emp.getSalaire() + raise);
    }
    return emp;
}
```

Transaction

- Pour le moment tous nos exemples, ne se sont pas préoccupés des transactions.
- Par hypothèse pourtant toutes nos méthodes d'exemple mis à part `findEmploye()` devraient se trouver dans une transaction pour garantir l'intégrité des données modifiées.
- Le GE nous permet grâce à la méthode `getTransaction()` de récupérer une transaction.
- La méthode `begin()` permet de démarrer la transaction, et la méthode `commit()` permet de la valider.

```
em.getTransaction().begin();
Employe emp = CreateEmploye(123,"Anne Onyme", 45000);
em.getTransaction().commit();
```

Requête JPQL I

- Généralement les développeurs connaissent bien le SQL. Pour interroger la BD d'une manière plus "objet", JPA met à disposition un langage de requête proche de SQL, le JPQL (Java Persistence Query Language).
- Une requête sera matérialisée par un objet du type `Query` ou `TypedQuery`. Elle sera construit à partir du GE.
- Une requête statique est définie par des annotations directement dans la classe. Elles sont vérifiées à la compilation.
- Une requête dynamique est construite à l'exécution dans une chaîne de caractère. Elles ne sont vérifiées qu'à l'exécution, donc elles peuvent échouer si elle sont incorrectes.

```
TypedQuery<Employe> q = em.createQuery(
            "SELECT e FROM Employe e", Employe.class);
List<Employe> listEmp = q.getResultList();
```

Requête JPQL II

- La methode suivante revoie la liste de tous les employés :

```
public List<Employe> findAllEmploye() {
    TypedQuery<Employe> q = em.createQuery(
            "SELECT e FROM Employe e", Employe.class);
    return q.getResultList();
}
```

Unité de persistance : persistence.xml

- Les paramètres de la connexion à la base de données sont définies dans le fichier `persistence.xml`. Ce fichier doit être situé dans le dossier `META-INF` du jar de l'application.

```
<?xml version="1.0" encoding="UTF-8" ?>

<persistence>
  <persistence-unit name="persUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>Employe</class>
    <properties>
      <property name="javax.persistence.jdbc.url"  value="jdbc:mysql://localhost:3306/maBD"
      <property name="javax.persistence.jdbc.driver"  value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user"  value="root"/>
      <property name="javax.persistence.jdbc.password"  value="mysql"/>
      <property name="eclipselink.ddl-generation"  value="create-tables"/>
      <property name="javax.persistence.level"  value="SEVERE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Le paramètre `" eclipselink.ddl-generation"` demande à notre implémentation de JPA de générer à notre place le code sql.

Construction d'un DAO avec JPA

En réunissant toute les méthodes écrite précédemment, on constate la facilité d'écriture d'un DAO à partir de JPA.

```
public class DAOEmploye {
private EntityManager em;
private DAOEmploye(String persistUnit);
public Employe createEmploye(int id,
                    String nom, long salaire);
public Employe findEmploye(int id);
public List<Employe> findAllEmploye();
public void removeEmploye(int id);
public void raiseEmployeSalaire(int id, long raise);
public static DAOEmploye createDAOEmploye();
}
```

Conclusion

- Ce cours est une introduction rapide au monde des ORM. Cette problématique est complexe et elle mériterait beaucoup plus de temps pour être abordée complètement.
- En TP nous allons, si le temps nous le permet, découvrir plusieurs outils Java supplémentaires pour mieux gérer nos projets :
  - Maven pour gérer le cycle de construction d'un projet (compilation, gestion des dépendance, packaging automatique,...)
  - JUnit pour écrire le code testant tous les modules que l'on développe pour éviter d'introduire de nouveaux bug lors de la modification du code.
  - DBUnit qui est un framework de test unitaire pour tester le code de la couche de persistance.
  - Git pour gérer facilement les évolutions de notre base de code lorsque l'on travaille à plusieurs.