Bases de Données Avancées

Khaled Jouini
j.khaled@gmail.com

Institut Supérieur d'Informatique et des Technologies de Communication

Ce document peut comporter des erreurs, à utiliser donc en connaissance de cause!

Plan du cours (1/3)

Plan du cours (2/3)

Plan du cours (3/3)

Syllabus du cours

Pré-requis

- Bases de données relationnelles (modèle relationnelle, SQL, procédures stockées)
- Modèle objet (conception OO, programmation OO, Java, etc.)
- Architecture n-tiers, Services Web (SOAP, etc.) - Connaissances utiles, mais pas indispensables

Objectifs

- Maîtrise des aspects théoriques et techniques du développement d'applications d'entreprise
  - Théoriques : mapping objet-relationnel, fournisseurs de persistance, architecture n-tiers, serveur d'applications, etc.
  - Techniques : Java Persistence API, Enterprise Java Beans, GlassFish, EclipseLink, etc.

Syllabus du cours

JEE à la maison...

- JEE SDK (dernière version)
  http://www.oracle.com/technetwork/java/javaee/downloads/.
- NetBeans IDE ($\geq$ 7.2) http://netbeans.org/downloads/.
- Galssfish ($\geq$ 3) http://glassfish.java.net/downloads/
- Oracle Database 11g Express Edition.
  http://www.oracle.com/technetwork/indexes/downloads/index.html.
  Alternativement, PostGreSQL.
- Tutoriels JEE http://netbeans.org/kb/trails/java-ee.html

Syllabus du cours

Pour en savoir plus...

- *Développements n-tiers avec JAVA EE*. J. Lafosse, Eni Edition. 2010.
- *Java EE 6 et Glassfish*. A. Goncalves. Pearson Education. 2010.
- *Patterns of Enterprise Application Architecture*. M. Fowler & Al. Addison Wesley. 2002.
- *EJB 3 in Action (Second Edition)*. D. Panda & Al. Manning Publications. 2012.
- Etc.

Chapitre 1 - Introduction générale

- Qu'est ce qu'une application d'entreprise?
- En quoi le développement d'applications d'entreprise est il différent?
- Architecture n-tiers

Qu'est ce qu'une application d'entreprise?

- Pas de définition précise, mais quelques points communs
  - Gestion d'une volumétrie importante de données
  - Répartition sur plusieurs machines
  - Utilisation de services non fonctionnels : persistance des données, sécurité d'accès, transactionnel, gestion de la concurrence, reprise sur panne, etc.
  - Règles de gestion (souvent complexes)
  - "Vues" et interfaces homme-machine différentes selon les acteurs
  - Peut soutenir une charge importante et s'adapter à une augmentation de cette charge
- Exemples : application bancaire, gestion de la relation client, vente en ligne,

Chapitre 1 - Introduction générale

- Qu'est ce qu'une application d'entreprise?
- En quoi le développement d'applications d'entreprise est il différent?
- Architecture n-tiers

En quoi le développement d'applications d'entreprise est il différent?

- Contre exemple
  - mélange de la présentation (ex. HTML) avec la logique métier (ex. PHP)
  - mélange de la logique métier avec l'accès aux données

  $\Rightarrow$ la modification de l'un des aspects a des répercussions sur les autres

  $\Rightarrow$ code d'accès à la base non réutilisable
  - le développeur prend en charge les aspects non fonctionnels (accès concurrents, transactions ACID, sécurité)

  $\Rightarrow$ pas toujours simple!!
  - Si on souhaite développer un autre type d'interface (Ex. Application Android), tout le code est à ré-écrire!

En quoi le développement d'applications d'entreprise est il différent?

- Développement d'une application d'entreprise :
  - complexe (besoins fonctionnels et non fonctionnels)
  - implique différentes technologies et équipes de développement
- Besoin :
  - de suivre des patterns de conception éprouvés (Model-View-Controller, Data Access Object, etc.)
  - d'outils qui prennent en charge les services non fonctionnels (conteneurs et serveurs d'applications)
  - de séparer les différents aspects de l'application pour faciliter le développement, l'évolutivité et la maintenance de l'application (architecture n-tiers)

Chapitre 1 - Introduction générale

- Qu'est ce qu'une application d'entreprise?
- En quoi le développement d'applications d'entreprise est il différent?
- Architecture n-tiers

Architecture n-tiers

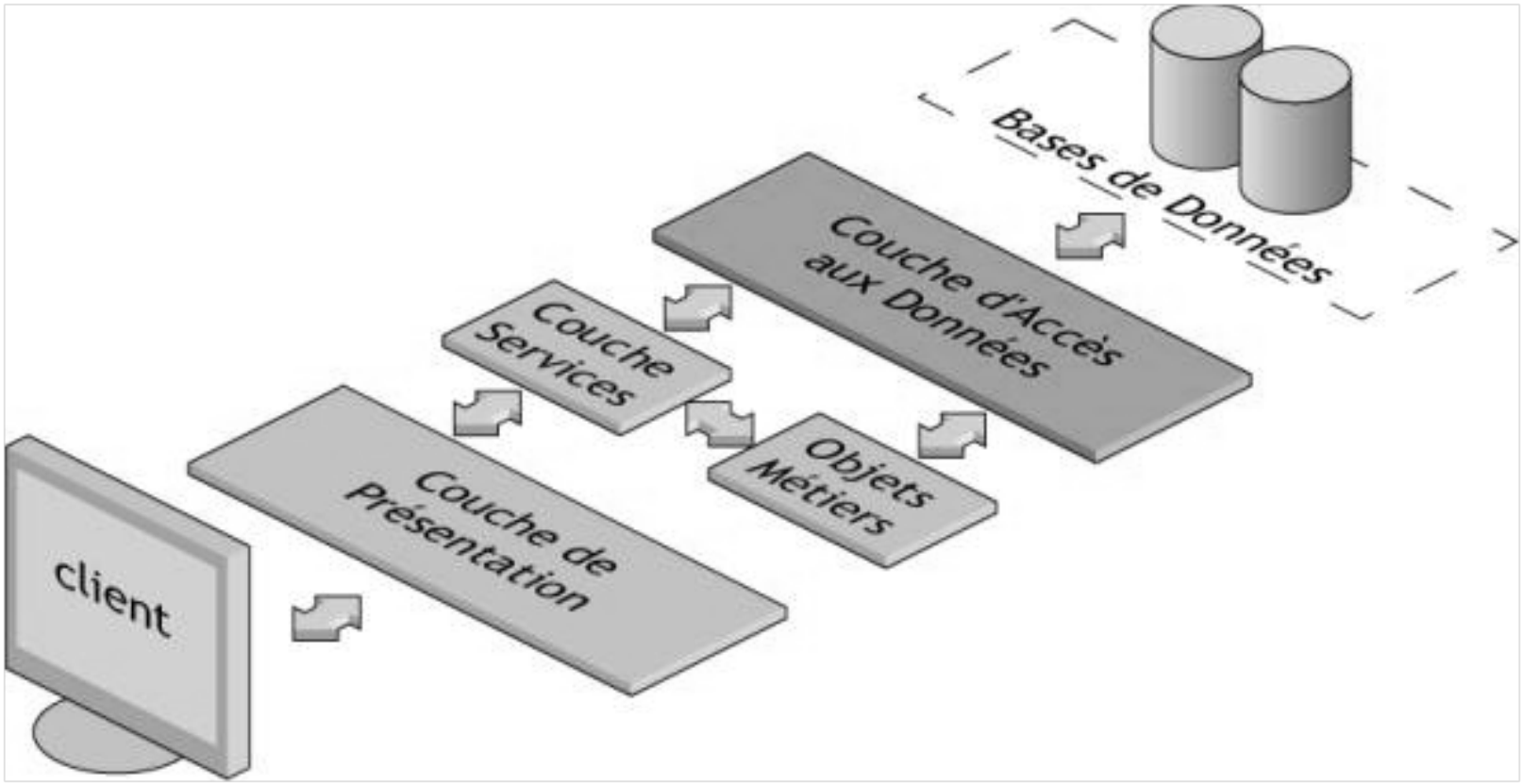


Figure: Architecture n-tiers. ["Objet-relationnel". Notes de cours. R. Grin. 2007]

- Plateforme de référence : Java Entreprise Edition, étudiée en partie dans ce cours.

Les traitements "métier"

- Ce sont les "vrais" traitements : ils font le travail essentiel liés au domaine de l'application (comptabilité, facturation, calculs scientifiques, etc.)
- Ils nécessitent des traitement techniques, non fonctionnels, pour gérer le transactionnel, la concurrence, etc.
- Ces traitements techniques souvent réunis sous le nom de couche "services" et pris en charge par un framework de développement (*container EJB* par exemple)

La couche persistance

- Elle est composée de la base de données, souvent relationnelle
- Le passage du monde objet (Objet Java) au monde relationnel (tuples de la BD) peut nécessiter des traitements complexes (étudiés dans ce cours)

$\Rightarrow$ Le plus souvent on ajoute une couche qui effectue la correspondance (*mapping*) entre les objets et la base de données (ORM, étudiés dans la suite du cours)

- Cette couche sert aussi de tampon pour les objets récupérés dans la BD et améliore donc les performances

Indépendance des couches

- La couche présentation est souvent modifiée (selon les remarques des utilisateurs) et cela ne doit pas induire des modifications dans les autres couches
- Le plus les données (du moins leur structure) et les processus métier restent stables durant toute la vie de l'application
- Les données et les processus métier ne doivent être modifiés que pour les améliorer ou les corriger mais pas pour tenir compte d'une modification (le plus souvent technique)
- c'est le principe d'indépendance des couches
- Indépendance totale : irréalisable, mais c'est à cela qu'il faut tendre

Chapitre 2 - Java DataBase Connectivity : opérations basiques

Présentation

- Avant JDBC, il était difficile d'accéder à des BD SQL à partir d'un programme
  - utilisation de bibliothèques C/C++
  - utilisation d'API native comme ODBC (ou équivalent unix)
- Problèmes
  - dépendance totale au SGBD utilisé
  - dépendance au système d'exploitation

  $\Rightarrow$ code non portable

Plan

- JDBC (Java DataBase Connectivity) : API permettant un accès homogène aux BD SQL à partir d'un programme Java. Atouts :
  - portabilité
  - liberté vis-à-vis des constructeurs
  - simplicité

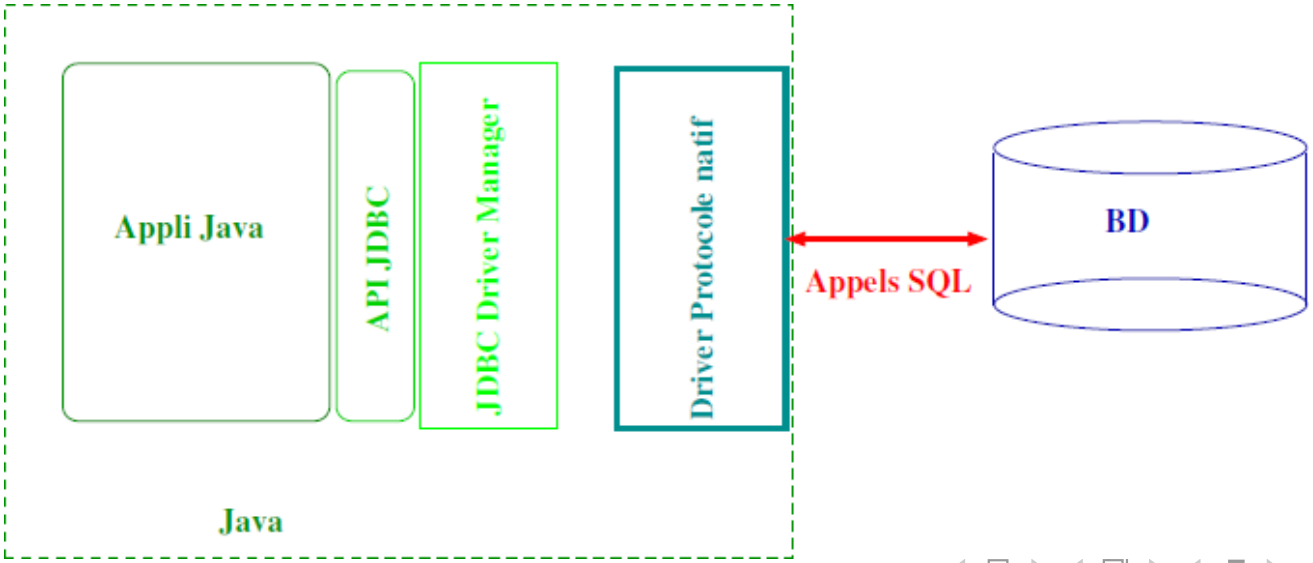
Ce chapitre...

1. Tour d'horizon
2. Travailler avec JDBC
3. Requêtes paramétrées et procédures stockées
4. Alertes et exceptions
5. Métadonnées

Chapitre 2 - Java DataBase Connectivity : opérations basiques

Qu'est ce que JDBC?

- Pour que l'accès à tous les SGBDs puissent se faire de manière uniforme (avec les mêmes méthodes) :
  1. Java spécifie l'interface des classes et la signature des méthodes à implanter (API JDBC)
  2. Chaque SGBD fournit son implémentation de chacune des méthodes. Le tout fait partie du pilote (*Driver*). Tous les SGBDs importants fournissent un pilote JDBC.
- Pilote : les requêtes qui lui sont passées sont converties dans le langage natif du SGBD
- JDBC : fournit des classes pour pouvoir charger un pilote et utiliser ses méthodes



Qu'est ce que JDBC?

Principales interfaces

`Driver`	renvoie une instance de `Connection`. A moins de vouloir écrire son propre pilote, on n'utilise pas directement cette interface
`Connection`	connexion à la base
`Statement`	ordre SQL
`PreparedStatement`	ordre SQL paramétré
`CallableStatement`	procédure stockée
`ResultSet`	tuples récupérés par un ordre SELECT
`ResultSetMetaData`	description des tuples récupérés par un SELECT
`DataBaseMetaData`	informations sur la base de données

`DriverManager` : **classe** qui gère les drivers, lance les connexions aux bases. Utilise `Driver` en interne (de manière transparente à l'utilisateur)

Chapitre 2 - Java DataBase Connectivity : opérations basiques

1. Préliminaires

- Ajouter le chemin des classes du pilote (fichier .jar) dans le classpath, pour que la JVM puisse y accéder à l'exécution
- Importer le paquetage java.sql (`import java.sql.*;`)
- Charger en mémoire la classe représentant le pilote

```
Class.forName("oracle.jdbc.OracleDriver");
```

- Remarque : `Class.forName("str")` permet de
  - charger dynamiquement (à l'exécution) la classe "str"
  - initialiser la classe "str" (méthodes statiques) sans créer d'instance

2. Étapes du travail sur une BD avec JDBC

1. Charger le pilote

```
Class.forName("oracle.jdbc.OracleDriver")
```

2. Établir une connexion avec une source de données

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:thin@
localhost:dbstcm:1521", "khaled","isitc0m")
```

3. Créer des instructions SQL (`Statement`, `PreparedStatement` ou `CallableStatement`) et l'objet de type `ResultSet` devant accueillir les résultats

```
Statement stmt = conn.createStatement();
Resultset rs = stmt.executeQuery("SELECT * FROM emp");
```

4. Parcourir les données et effectuer des traitements (affichage, calcul, etc.)

```
while (rs.next())...
```

5. Fermer la connexion

```
stmt.close();
conn.close();
```

3. Étape 1 : Charger le pilote

- Soit à l'aide de `DriverManager` (méthode la plus simple) soit avec `DataSource` (méthode recommandée pour les applications d'entreprise, on y reviendra)
- `DriverManager` : gère les drivers (instances de `Driver`)
- `Class.forName("nomDriver")` : crée la classe correspondant au pilote (implémentant `Driver`) et l'enregistre auprès de la classe `DriverManager`

4. Étape 2 : établir une connexion

- Objet `Connection` :
  - représente une session client
  - possède des informations associées : l'identificateur de l'utilisateur, l'ensemble des ordres SQL et leurs résultats générés dans la session, ainsi que des informations sur les transactions.
- méthode `DriverManager.getConnection` : renvoie un objet de type Connection (en coulisse, `DriverManager.getConnection` appelle la méthode `Driver.connect()`)
- `DriverManager.getConnection` : prend en entrée trois arguments
  1. URL de la BD
  2. le login de l'utilisateur de la base
  3. son mot de passe

4. Étape 2 : établir une connexion

- URL de la BD : de la forme
  jdbc:nomSGBD:¡sousProtocole:¿Adresse:nomBD:port[;attribut=valeur]*
- Ex. `Connection conn = DriverManager.getConnection ("jdbc:oracle:thin@localhost:dbstcm:1521", "khaled","isitc0m")`
- `DriverManager` essaye tous les pilotes enregistrés (avec `Class.forName()`) jusqu'à ce qu'il trouve un qui lui fournisse une connexion

5. Étape 3 : Création d'un ordre SQL simple

Création de l'ordre

- A l'aide de la méthode `createStatement` de la classe `Connection`

```
Statement stmt = conn.createStatement();
```

- La méthode à invoquer ensuite dépend de l'action à réaliser
  - `stmt.executeQuery()` : recherche (select). Renvoie un `ResultSet` contenant les résultats de la recherche.
  - `stmt.executeUpdate()` : modification (insert/delete/update) ou DDL (create table,..). Renvoie le nombre de lignes modifiées
  - `stmt.execute()` : si la nature de l'ordre exécuté n'est pas connu.

- Exemple :

```
Statement stmt = conn.createStatement();
Resultset rs = stmt.executeQuery("SELECT * FROM emp");
```

5. Étape 3 : Création d'un ordre SQL simple

Exécution de l'ordre

- L'ordre SQL créé n'est pas interprété par Java
  - le pilote le transmet au SGBD tel quel
  - si une requête ne peut pas s'exécuter, l'exception `SQLException` est levée (on y reviendra)
- Le pilote effectue d'abord un accès à la base pour découvrir les types des colonnes impliquées puis un deuxième pour transmettre la requête

5. Étape 3 : Création d'un ordre SQL simple

Optimisation

- Quand le réseau est lent et que l'on veut récupérer de nombreuses lignes, il est parfois possible d'améliorer les performances en modifiant le nombre de lignes récupérées
- Pour cela on utilise la méthode `setFetchSize` de `Statement`
- C'est seulement une indication, le pilote peut ne pas en tenir compte

6. Étape 4 : Parcourir le résultat avec ResutlSet

Parcours d'un `ResultSet`

- `executeQuery()` renvoie les données dans un objet de type `ResultSet`
- Un objet `ResultSet` se parcourt par un curseur, initialement positionné avant la première ligne
- JDBC 1.x
  - Parcours séquentiel ligne par ligne avec la méthode `next()`
  - next() renvoie `false` si dernière ligne lue, et `true` sinon
  - Ex. `while(rs.next())`
  - Impossible de revenir en arrière ou d'avoir un accès aléatoire
- A partir de JDBC 2.0 : possibilité de parcourir un `ResultSet` dans les deux sens (nous y reviendrons)

6. Étape 4 : Parcourir le résultat avec ResultSet

Accès aux valeurs des colonnes d'un `ResultSet`

- Les colonnes sont référencées par leur numéro (commençant à 1) ou par leur nom
- L'accès aux valeurs des colonnes se fait par des méthodes de la forme `getXXX()`
- Exemple 1 : `Long id = rs.getLong(1); //Accès au premier attribut`
- Exemple 2 : `String nom = rs.getString("nomE");`
- Les types Java ne sont pas les mêmes que les types SQL (Exemple : VARCHAR et String)
- Le pilote convertit les types SQL retournés en types Java choisis par le programmeur
- Si mauvais choix, une exception `SQLException` est levée

6. Étape 4 : Parcourir le résultat avec ResulSet

Correspondance type Java/ type SQL

Type SQL	Méthode Java
CHAR, VARCHAR	getString()
LONGVARCHAR	getAsciiStream()
NUMERIC, DECIMAL	getBigDecimal()
BINARY, VARBINARY	getBytes()
BIT	getBoolean()
INTEGER	getIn()
BIGINT	getLong()
SMALLINT	getShort()
TINYINT	getByte()
REAL	getFloat()
DOUBLE, FLOAT	getDouble()
DATE	getDate()
TIME	getTime()
TIMESTAMP	getTimeStamp()
ARRAY	getArray()
BLOB	getBlob()
CLOB	getClob()
REF	getRef()
autre	getObject()

6. Étape 4 : Parcourir le résultat avec ResulSet

Les valeurs NULL

- `wasNull()` : détecter les valeurs NULL de la base
- Les méthodes `getXXX()` convertissent les `NULL` de la BD en une valeur acceptable par le type Java
  - Méthodes retournant un objet (getString(), getDate(), etc.) : retournent un null Java
  - Méthodes numériques : retournent un "0"
  - getBoolean() : retourne "false"

7. Étape 5 : Fermer les ressources

- Pour terminer proprement un traitement, il faut fermer les différentes ressources
- Sinon c'est le ramasse-miettes qui le fait (moins efficace)
- `ResultSet`, `Statement` et `Connection` possède chacune une méthode `close()`

Exercice

Compléter le code ci-dessous, permettant de se connecter à une BD Oracle, d'exécuter une requête et d'afficher les résultats :

```
String X = "jdbc:oracle:thin@localhost:1521:dbstcm";
 String Y = "oracle.jdbc.OracleDriver"
 String user = "foulen";
 String passwd = "pwdfoulen";
 String sql = "SELECT * FROM dept";
 Connection connect;

 Class.forName(...);
 connect = ....getConnection(...);
 Statement stmt = ....createStatement();

 ...  = stmt.executeQuery(sql);

 while (...)  {
      System.out.println("Id "+ ...  + " Nom " + ...);
 }
```

Chapitre 2 - Java DataBase Connectivity : opérations basiques

1. Requêtes paramétrées

- La plupart des SGBD offrent la possibilité de n'analyser qu'une seule fois une requête (Ex. Shared Pool d'Oracle)
- Avec l'interface `PreparedStatement`, JDBC permet de bénéficier de cette fonctionnalité
- Objet `PreparedStatement` : envoie une requête sans paramètres au SGBD pour précompilation et spécifie le moment voulu leurs valeurs
- Plus rapide qu'un `Statement` classique
  - la requête est analysée une seule fois
  - elle peut être exécutée plusieurs fois avec à chaque exécution des valeurs différentes
- `PreparedStatement` : hérite de `Statement`

1. Requêtes paramétrées

Création d'une requête paramétrée

```
PreparedStatement pstmt =
  conn.prepareStatement("UPDATE emp SET sal = ?"
                        + " WHERE nome = ?");
```

- Les "?" indiquent les emplacements des paramètres
- `setXXX(n, valeur)` : indique la valeur du nième paramètre. XXX doit être remplacé par le type Java adéquat. Ex. `pstmt.setString(2,"foulen")`
- Le pilote fait la conversion
- **N.B.** Tous les SGBD n'acceptent pas les requêtes paramétrées (pré-compilées)

1. Requêtes paramétrées

Exemple

```
PreparedStatement pstmt =
  conn.prepareStatement(
    "UPDATE emp SET sal = ? "
    + "WHERE nomE = ?");
for (int i=0; i<10; i++) {
  pstmt.setDouble(1,employe[i].getSalaire());
  pstmt.setString(2, employe[i].getNom());
  pstmt.executeUpdate();
}
```

commence à 1
et pas à 0

2. Procédures stockées

- Procédure stockée : programme nommé, compilé et stocké au niveau de la base (Ex. procédure PL/SQL)
- Peut grouper plusieurs ordres SQL (à ne pas confondre avec une transaction)
- Avantages/inconvénients de l'appel d'une procédure stockée à partir d'un programme Java
  - Amélioration des performances : moins d'accès réseau
  - Le code est moins portable

2. Procédures stockées

Exemple de procédure stockée (PL/SQL Oracle)

```
create or replace procedure augmenter
   (unDept in integer, pourcentage in number,
    cout out number) is
begin
 select sum(sal) * pourcentage / 100
    into cout
    from emp
    where dept = unDept;
  update emp
    set sal = sal * (1 + pourcentage / 100)
    where dept = unDept;
end;
```

2. Procédures stockées

Exemple d'appel d'une procédure stockée

```
CallableStatement csmt = conn.prepareCall(
  "{ call augmenter(?, ?, ?) }");
// 2 chiffres après la virgule pour 3ème paramètre
csmt.registerOutParameter(3, Types.DECIMAL, 2);
// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setDouble(2, 2.5);
csmt.executeQuery(); // ou execute()
double cout = csmt.getDouble(3);
System.out.println("Cout total augmentation : "
                   + cout);
```

2. Procédures stockées

- `CallableStatement` : interface héritant de `PreparedStatement` et permettant l'appel aux procédures stockées
- `prepareCall` de `Connection` : crée une instance de `CallableStatement`
- `prepareCall` : prend en entrée une chaîne de caractère spécifiant comment se fait l'appel de la procédure stockée et si elle renvoie une valeur

2. Procédures stockées

Syntaxe de `prepareCall`

- La syntaxe de l'appel des procédures stockées n'est pas standardisée
- JDBC utilise sa propre syntaxe pour pallier à ce problème
  - `prepareCall("call nomProcédure(?,?,...)")` : les "?" remplace les paramètres
  - `prepareCall("call nomProcédure")` : si nomProcédure ne prend pas d'arguments et ne renvoie pas de valeur
  - Exemple `CallableStatement cstmt = conn.prepareCall("call augmenter(?,?,?)")`

2. Procédures stockées

Exécution

- L'exécution est précédée par l'indication des valeurs des paramètres et de leur types ("in" ou "out")
- `setXXX(n, valeur)` : indique la valeur du nième paramètre
- `registerOutParameter()` : indique si le paramètre est "out"
- `execute` ou `excuteQuery`: exécute l'appel
- `getXXX(n)` : récupère la valeur d'un paramètre "out" ou "in/ou"

Chapitre 2 - Java DataBase Connectivity : opérations basiques

Vue d'ensemble

Trois catégories

- `SQLException` : erreurs SQL
- `SQLWarning` : avertissements SQL (classe fille de SQLException) ;
- `DataTruncation` : avertit quand une valeur est tronquée lors d'un transfert entre Java et le SGBD (classe fille de SQLWarning)

1. SQLException

- Instance de `SQLException` : levée lorsqu'une erreur a lieu lors de l'interaction entre JDBC et une BD
- Elle contient les informations suivantes
  - Le message d'erreur accessible via la méthode `getMessage`
  - Un code d'erreur SQL (standardisé par l'ISO/ANSI), accessible via `getSQLState`. Exemple : code 08003 =¿ connexion fermée.
  - Une exception peut être causée par plus d'une erreur. Appel répétitif de la méthode `getCause()` : accès à ces erreurs.
  - Un ordre SQL peut provoquer plusieurs exceptions. Ces exceptions sont chaînées et accessibles individuellement avec `getNextException()`

1. SQLException

```
public static void printSQLException(SQLException ex) {
for (Throwable e : ex) {
    if (e instanceof SQLException) {

            e.printStackTrace(System.err);
            System.err.println("SQLState: " +
                ((SQLException)e).getSQLState());

            System.err.println("Message: " + e.getMessage());

            Throwable t = ex.getCause();
            while(t != null) {
                System.out.println("Cause: " + t);
                t = t.getCause();
            }
        }
    }
}
```

1. SQLException

- `SQLException` : a trois sous-classes (JDBC 4)
  - `SQLNonTransientException` : le problème ne peut être résolu sans une action externe; inutile de réessayer sans ne rien modifier
  - `SQLTransientException` : le problème peut être résolu si on attend un peu avant de réessayer
  - `SQLRecoverableException` : l'application peut résoudre le problème en exécutant une certaine action

2. SQLWarning

- Alerte `SQLWarning` : n'arrête pas l'exécution, mais avertit de l'occurrence d'un événement non planifié (Ex. révocation infructueuse d'un privilège)
- Reportée par un objet `Connection`, `Statement` ou `ResultSet` et accessible par la méthode `getWarnings`
- `getWarnings` retourne la première alerte rencontrée. `getNextWarning` permet d'accéder aux suivantes.

Chapitre 2 - Java DataBase Connectivity : opérations basiques

Métadonnées

- Métadonnées : informations sur les données retournées dans un `ResultSet` ou sur la base de données elle-même.
- Informations sur les résultats : nombre de colonnes, types, noms, etc.
- Informations sur la BD
  - dépendent du SGBD
  - généralement : nom de l'utilisateur connecté, nom de la BD, etc.

1. Métadonnées d'un ResultSet

Accès aux métadonnées `ResultSet`

- Méthode `getMetaData()` de `ResultSet` : renvoie un objet `ResultSetMetaData`.
- Objet `ResultSetMetaData` : contient les métadonnées d'un `ResultSet`
- Méthodes de `ResultSetMetaData` :
  - `getColumnCount()` : Nombre de colonnes
  - `getColumnName(col)` : nom de la colonne d'ordre col
  - `getTableName(col)` : nom de la table à laquelle appartient la colonne d'ordre col dans le résultat
  - etc.

2. Métadonnées d'une BD

- Méthode `getMetaData()` de `Connection` : renvoie un objet `DataBaseMetaData`.
- Objet `DataBaseMetaData` : on y trouve les méta-données sur une BD
- Métadonnées :
  - `getTables()` : Nom des tables
  - `getUserName()` : nom de l'utilisateur connecté
  - etc.

Exercice

Écrire une méthode `void outputResultSet(ResultSet rs)` permettant d'afficher le contenu d'un ResultSet (nom des colonnes, puis chacune des lignes)

Exercice

Écrire une méthode `void outputResultSet(ResultSet rs)` permettant d'afficher le contenu d'un ResultSet

```
private static void outputResultSet(ResultSet rs) throws Exception {
  ResultSetMetaData rsMetaData = rs.getMetaData();
  int numberOfColumns = rsMetaData.getColumnCount();
  for (int i = 1; i < numberOfColumns + 1; i++) {
    String columnName = rsMetaData.getColumnName(i);
    System.out.print(columnName + "    ");
  }
  System.out.println();
  System.out.println("----------------------");

  while (rs.next()) {
    for (int i = 1; i < numberOfColumns + 1; i++) {
      System.out.print(rs.getString(i) + "    ");
    }
    System.out.println();
  }
}
```

Chapitre 3 - Java DataBase Connectivity : aspects avancés

1.a. Le mode Auto-commit

- Par défaut les connexions sont en mode `auto-commit` : chaque ordre SQL est traité comme une transaction validée lorsqu'il s'exécute correctement.
- Pour désactiver le mode par défaut :

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

- Le programmeur gère alors la validation et l'annulation des transaction :
  - Validation : `conn.commit();`
  - Annulation : `conn.rollback();`
- `commit()` et `rollback()` : terminent une transaction et ouvrent implicitement une nouvelle.

1.b. Transactions et exceptions

- Que l'on soit en mode `auto-commit` ou pas, une exception ne provoque pas une annulation
- Si une transaction comportant plusieurs ordres SQL provoque une exception, il est nécessaire de tout annuler avec un `rollback()`. L'appel du `rollback()` est à la charge du code java

```
Connection con = ... ;
try {
    ...
    con.commit();
    }
catch(SQLException e) {
// La situation ne permet pas de corriger le problème
con.rollback();
}
```

2. Niveaux d'isolation - a. Présentation

- Isolation : les modifications faites par une transaction T ne sont rendues visibles pour les transactions concurrentes que lorsque T est validée.
- *Long-running transactions* : Transactions mettant beaucoup de temps à s'exécuter
  - Peuvent verrouiller les ressources pour un temps relativement long
  - Pour cette raison (et pour d'autres), il est parfois utile d'utiliser des règles d'isolation moins strictes
- JDBC prévoit quatre niveaux d'isolation
- A mesure que les règles sont assouplies des anomalies peuvent apparaître

2. Niveaux d'isolation - b. Anomalies (Lectures impropres)

Lectures impropres (*Dirty Reads*):

- Une transaction manipule des valeurs qui n'existent plus dans la base
- Peut avoir lieu lorsqu'une transaction T_B a la possibilité de lire des données en cours de modification par une transaction concurrente T_A non encore validée.
- Exemple : T_A modifie un tuple (sans le verrouiller), T_B sélectionne le tuple, T_A est annulée (`ROLLBACK`)

  $\Rightarrow$ la valeur du tuple utilisée par T_B n'existe pas
- Solution : verrouiller tous les tuples modifiés par T_A du début de T_A jusqu'à ce qu'elle se termine.

2. Niveaux d'isolation - b. Anomalies (Lectures non reproductibles)

Lectures non-reproductibles (*Non-Repeatable Reads*):

- Une transaction lit deux valeurs différentes d'une même donnée (incohérence)
- Peut avoir lieu lorsqu'une transaction T_A relit des données qu'elle a lues précédemment et trouve que les données ont été modifiées par une autre transaction T_B (validée depuis la lecture initiale).
- Exemple : T_A lit un tuple, T_B modifie le tuple, T_B est validée, T_A relit le tuple
  $\Rightarrow$ les deux valeurs lues par T_A sont différentes
- Solution : verrouiller tous les tuples lus (ou modifiés) par T_A du début de T_A jusqu'à ce qu'elle se termine.

2. Niveaux d'isolation - b. Anomalies (Lectures fantômes)

Lectures fantômes (*Phantom reads*):

- Cas spécial des lectures non reproductibles
- Une transaction T_A ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction T_B récemment validée.
- Exemple : soit la requête q : `SELECT * from emp Where Salaire BETWEEN 1000 AND 2000;`

  T_A exécute q, T_B ajoute un employé avec un salaire $\in [1000, 2000]$, T_B est validée, T_A ré-exécute q

  $\Rightarrow$ l'ensemble résultat de q à la première exécution diffère de celui de la deuxième exécution
- Solution : verrouiller l'intervalle $[1000, 2000]$ tant que T_A ne se termine pas (*Range-locks*).

2. Niveaux d'isolation - c. Niveaux prédéfinis

- `TRANSACTION_SERIALIZABLE`
  - Règles d'isolation les plus strictes (plus haut niveau d'isolation).
  - Verrouillage en écriture et en lecture ainsi que sur les intervalles jusqu'à ce que la transaction se termine
- `TRANSACTION_REPEATABLE_READ`
  - Verrouillage en écriture et en lecture jusqu'à ce que la transaction se termine
- `TRANSACTION_READ_COMMITTED`
  - Verrouillage en écriture jusqu'à ce que la transaction se termine
  - En lecture uniquement jusqu'à ce que la lecture se termine
- `TRANSACTION_READ_UNCOMMITTED`
  - Niveau d'isolation le plus bas.
  - Une transaction peut voir les données modifiées par une autre transaction avant que celle-ci ne se termine (*Dirty Reads*)

2. Niveaux d'isolation - c. Niveaux prédéfinis

Niveau Anomalie	Lecture impropre	Lecture non-reproductible	Lecture fantôme
TRANSACTION_READ_UNCOMMITTED	possible	possible	possible
TRANSACTION_READ_COMMITTED	-	possible	possible
TRANSACTION_REPEATABLE_READ	-	-	possible
TRANSACTION_SERIALIZABLE	-	-	-

- Certains pilotes ne prennent pas en charge tous les niveaux d'isolation. la méthode `supportsTransactionIsolationLevel` de `DatabaseMetaData` permet de savoir si un pilote prend en charge un niveau donné
- `getTransactionIsolation` de `Connection` : permet de savoir le degré d'isolation utilisé
- `setTransactionIsolation` de `Connection` : permet de modifier le degré d'isolation

2. Niveaux d'isolation - d. Exemple

Exemple

```
Connection con = ... ;
DatabaseMetaData dbMd = con.getMetaData();
if (dbMd.supportsTransactionIsolationLevel(Connection.TRANSACTION_READ_
{
  System.out.println("Transaction Isolation level "
      + "TRANSACTION_READ_COMMITTED is supported.");
  con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
}
```

3. Points de sauvegarde

- Points de sauvegarde : permettent une annulation partielle des opérations d'une transaction
- Exemple d'utilisation : réessayer à partir de là où la transaction a échouée et non depuis le début
- Création avec la méthode `setSavepoint` (paquetage `java.sql.Savepoint`.)

```
Connection con = ... ;
con.setAutoCommit(false);
//une première instruction
...
Savepoint s1 = con.setSavepoint("Premier point de sauvegarde");
//une deuxième instruction
...
Savepoint s2 = con.setSavepoint(); // point de sauvegarde anonyme
con.releaseSavepoint(s2); // on enlève le second savepoint
con.rollback(s1); // toutes les modifications après s1 sont ignorées
con.commit(); // finalement seulement la première instruction est validée
```

Chapitre 3 - Java DataBase Connectivity : aspects avancés

1. Génération des clés

- La génération automatique des clés n'est pas standardisée, elle varie d'un SGBD à un autre
  - Oracle, postGreSql, DB2 : utilise des séquences
  - MySQL : attribut de type `AUTO_INCREMENT`
  - SQL Server : attribut de type `IDENTITY`

1. Génération des clés

- Dans certaines situations il est nécessaire de connaître la valeur d'une clé générée automatiquement
- Exemple :
  - les valeurs de la clé primaire de la table Dept sont générées automatiquement
  - Pour pouvoir affecter des employés à un nouveau département, il est nécessaire de connaître la valeur de sa clé primaire
- L'interface `Statement` permet de récupérer la valeur d'une clé générée automatiquement avec la méthode `ResultSet getGeneratedKeys()`
- Pour pouvoir utiliser `getGeneratedKeys()` il est nécessaire d'ajouter à `executeUpdate` dans un 2ème paramètre la valeur `Statement.RETURN_GENERATED_KEYS`

1. Génération des clés

```
Statement stmt = conn.createStatement();
stmt.executeUpdate( "INSERT INTO dept VALUES(...)",
Statement.RETURN_GENERATED_KEYS);
ResultSet rsCles = stmt.getGeneratedKeys();
if ( rsCles.next() ) {
int cle = rsCles.getInt(1);
}
```

2. Cas des séquences

- Peu de pilotes implantent la méthode `getGeneratedKeys` (exemple le pilote d'Oracle)
- Dans le cas d'Oracle, il est nécessaire de créer une séquence et de récupérer la valeur du dernier numéro de séquence par une requête
  1. Création d'une séquence
     ```
     CREATE SEQUENCE seq_deptid START WITH 1;
     ```
  2. Insertion
     ```
     INSERT INTO Dept VALUES (seq_deptid.nextval, 'Info');
     ```
  3. Récupération de la dernière valeur générée
     ```
     SELECT seq_deptid.currval FROM dual;
     ```

2. Cas des séquences

- Solution plus élégante pour l'insertion
  1. Création d'un trigger en plus de la séquence
     ```
     CREATE TRIGGER dept_genKey
     BEFORE INSERT ON dept
     FOR EACH ROW
     BEGIN
     SELECT seq_deptid.nextval INTO :new.deptid FROM dual;
     END;
     /
     ```
  2. Insertion
     ```
     INSERT INTO Dept (deptnom) VALUES ('Info');
     ```
  3. Récupération de la dernière valeur générée
     ```
     SELECT seq_deptid.currval FROM dual;
     ```

Chapitre 3 - Java DataBase Connectivity : aspects avancés

1. Présentation

- Objectifs : réduire le nombre d'accès distants à la BD pour améliorer les performances
- Procédures stockées : le permettent mais posent des problèmes de portabilité
- Alternative, traitement par lot (*Batch*) : regrouper plusieurs ordres SQL DML (insert, delete, update) pour les envoyer en une seule fois au SGBD
- `supportsBatchUpdates()` de `DataBaseMetaData`: permet de tester si le pilote implante les méthodes d'un traitement par lot

2. Méthodes

- 3 méthodes de l'interface `Statement` (et donc de ses sous-interfaces) pour manipuler les regroupements d'ordres SQL
- `void addBatch(String sql)` : ajoute un ordre SQL à la liste des ordres à exécuter
- `void clearBatch()` : vide la liste des ordres
- `int[] executeBatch()` :
  - Exécute les ordres et retourne le nombre de lignes modifiées par ordre
  - La ième case contient le nombre de lignes modifiées par le ième ordre.
  - Susceptible de lever une `BatchUpdateException` si un des ordres du lot lève une `SQLException` ou retourne un `ResultSet`.
  - Si le ième ordre SQL lève une exception, la ième case du tableau renvoyé contiendra une valeur négative. Les ordres qui suivent peuvent s'exécuter ou non selon le pilote

2. Méthodes

```
Connection connection = ... ;
Statement stmt = connection.createStatement();
if(connection.getMetaData().supportsBatchUpdates()){
    connection.setAutoCommit(false);
    stmt.clearBatch();
    stmt.addBatch("INSERT ....");
    stmt.addBatch("UPDATE ...");
    stmt.addBatch("...");
    int[] resultat = stmt.executeBatch();
    //voir les différents types de retour possibles
    connection.commit();
    connection.setAutoCommit(false);
}
```

Chapitre 3 - Java DataBase Connectivity : aspects avancés

1. Présentation

- A partir de JDBC 2.0, il est possible :
  - d'avoir un accès aléatoire aux lignes d'un `ResultSet`
  - de modifier les lignes d'un `ResultSet` et de répercuter les modifications automatiquement dans la base (sans utiliser `executeUpadate()`)
- Certains pilotes n'implantent pas les `ResultSet` scrollables et modifiables

2. ResultSet scrollable

- A partir de JDBC 2.0, il y a 3 types de parcours possibles pour un `ResulSet`
  - `TYPE_FORWARD_ONLY` : ne peut pas être parcouru que dans un sens. Valeur par défaut.
  - `TYPE_SCROLL_INSENSITIVE` : peut être parcouru dans les 2 sens, mais ne reflète pas les modifications faites dans la base après la récupération du `ResultSet`
  - `TYPE_SCROLL_SENSITIVE` : peut être parcouru dans les 2 sens, et reflète les modifications faites dans la base après la récupération du `ResultSet`

- Exemple de création

```
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE);
ResultSet srs = stmt.executeQuery(SELECT nomE, salaire FROM emp);
```

2. ResultSet scrollable

Parcours d'un `ResultSet` scrollable

- Parcours dans les deux sens avec `next()` et `previous()`
- Accès direct avec `absolute(int row)`, `first()`, `last()`,
- Déplacement relatif avec `relative(int row)`, `afterLast()`, `beforeFirst()`
- `getRow()` : position courante du curseur

2. ResultSet scrollable

Parcours en arrière d'un `ResultSet` scrollable

```
srs.afterLast();
while (srs.previous()) {
  String nomE = srs.getString("nomE");
  double salaire = srs.getFloat("salaire");
  System.out.println(nomE + " ; " + salaire);
}
```

2. ResultSet scrollable

Positionnement absolu et relatif dans un `ResultSet` scrollable

```
srs.afterLast()
```

```
srs.absolute(-2); // avant-dernière ligne
srs.absolute(4);
int numLigne = srs.getRow(); // rowNum = 4
srs.relative(-3);
int numLigne = srs.getRow(); // rowNum = 1
srs.relative(2);
int numLigne = srs.getRow();  // rowNum = 3
```

3. ResultSet scrollable et modifiable

- Variable statique `CONCUR_UPDATABLE` : permet de rendre un `ResultSet` modifiable (les modifications faites sur les tuples du `ResultSet` sont répercutées automatiquement dans la BD)

```
Statement stmt = con.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
      "SELECT nomE, salaire FROM emp");
```

- Par défaut un `ResultSet` est `CONCUR_READONLY` : les données sont consultables en lecture seule

3. ResultSet scrollable et modifiable

- La requête d'un `ResultSet` modifiable doit
  - ne pas contenir de jointure ou de clause `group by`
  - contenir la clé primaire de la table
- Mise à jour
  - `updateXXX("`$nomColonne$`",` $valeur$`)` : mise à jour
  - `cancelRowUpdates()` : annulation des modifications
  - `updateRow()` : validation et enregistrement des modifications
- Suppression : `deleteRow()`
- Insertion
  - `moveToInsertRow()` : création d'un nouveau tuple en mémoire
  - `insertRow()` : enregistrement dans la BD

3. ResultSet scrollable et modifiable

Exemple : mise-à-jour

```
uprs.last();
uprs.updateDouble("salaire", 10000);
uprs.cancelRowUpdates(); // annule
uprs.updateDouble(2, 12000);
uprs.updateRow(); // enregistre modifs dans BD
```

Exemple : suppression

```
uprs.absolute(4);
uprs.deleteRow();
```

3. ResultSet scrollable et modifiable

Exemple : insertion

```
uprs.moveToInsertRow();
uprs.updateInt("matr", 150);
uprs.updateString("nomE", "Kleber");
uprs.updateDouble("salaire", 10000);
. . .
uprs.insertRow();
```

Chapitre 3 - Java DataBase Connectivity : aspects avancés

1.a. Présentation

- Définie depuis la version JDBC 2.0 (paquetage `javax.sql.RowSet`).
- Il s'agit d'un `ResultSet` scrollable et modifiablr, ayant l'avantage de se conformer au modèle JavaBeans : sérialisable, avec propriétés et observables par des écouteurs (`Listeners`)
- Propriétés
  - Attributs (variables d'instance) avec des accesseurs `setXXX` et `getXXX`
  - Permettent de facilement configurer un `RowSet` avec des méthodes `setXXX` (url BD, utilisateur, ordre SQL, Connection...)
- Écouteurs (`RowSetListener`)
  - Permettent l'écoute (l'interception) des événements relatifs au `RowSet`
  - Évènements : changement de la position du curseur, modification d'une ligne et modification du contenu du `RowSet`.
- Autre avantage : résultat d'une requête scrollable et modifiable, même si le pilote ne le permet pas

1.b. Catégories de RowSet

- Il existe 2 catégories de `RowSet` :
  - connectés
  - détachés
- `RowSet` connecté
  - Garde sa connexion avec la base ouverte tout au long de son existence
  - Fonctionnement similaire à celui des `ResultSet` scrollables et modifiables
  - Généralement utilisés pour rendre un `ResultSet` scrollable et modifiable lorsque le pilote ne le permet pas

1.b. Catégories de RowSet

- `RowSet` détachés, fonctionnement
  - Se connecter à la base pour récupérer des données
  - Rompre la connexion à la base
  - Faire les modifications sur les données
  - Se reconnecter afin de transmettre les modifications.

1.c. Implémentations proposées

- Cinq interfaces filles de RowSet sont proposées
  - `JdbcRowSet` : `RowSet` connecté. "Enveloppe" autour d'un `ResultSet`, permettant de le faire fonctionner comme un JavaBean
  - `CachedRowSet` : `RowSet` détaché stockant les données en mémoire vive. Adapté pour fournir aux clients légers (téléphones mobiles) des données provenant d'une BD
  - `WebRowSet` : `RowSet` détaché fille de `CachedRowSet`. Permet en plus de se sauvegarder sous format XML
  - `FilteredRowSet` : `RowSet` détaché fille de `CachedRowSet`. Filtre les données présentées à l'utilisateur. Equivalent à une requête de sélection sur le `RowSet`.
  - `JoinRowSet` : `RowSet` détaché fille de `CachedRowSet`. Permet de faire des jointures sans avoir à se connecter à la base.

2. JdbcRowSet - a. Création

- `JdbcRowSetImpl` : implantation de `JdbcRowSet` fournie avec la distribution Java
- Différentes possibilités de création
  - En passant un objet `ResultSet` au constructeur. Le ResultSet doit être scrollable et modifiable
  - En passant un objet `Connect` au constructeur
  - En utilisant le constructeur par défaut (sans paramètres) : nécessite l'instantiation des paramètres ultérieurement par les setters (`setURL`, `setUserName`, etc.)

2. JdbcRowSet - a. Création

Création en utilisant un `ResultSet` existant

```
stmt = con.createStatement(
            ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery("select * from emp");
jdbcRs = new JdbcRowSetImpl(rs);
```

- Si rs n'est pas scrollable et modifiable, jdbcRs, ne le serait pas

2. JdbcRowSet - a. Création

Création en passant un objet `Connection` existant

```
jdbcRs = new JdbcRowSetImpl(con);
jdbcRs.setCommand("select * from emp");
jdbcRs.execute();
```

- jdbcRs ne contient pas de données tant que `execute()` n'exécute pas la requête spécifiée par `setCommand()`
- jdbcRs est scrollable et modifiable

2. JdbcRowSet - a. Création

Création avec le constructeur par défaut

```
jdbcRs = new JdbcRowSetImpl();
jdbcRs.setCommand("select * from emp");
jdbcRs.setUrl("jdbc:oracle:thin@localhost:dbstcm:1521");
jdbcRs.setUsername("khaled");
jdbcRs.setPassword("i51tc0m");
jdbcRs.execute();
```

2. JdbcRowSet - a. Création

Il est possible d'utiliser les requêtes précompilées avec `RowSet`

```
jdbcRs.setCommand("select nom, salaire "
                  + " from emp where dept = ?");
jdbcRs.setInt(1, 20);
jdbcRs.execute();
```

2. JdbcRowSet - b. Manipulation

- Le parcours et la modification des données d'un `JDBCRowSet` se font de la même manière qu'un `ResultSet` scrollable et modifiable
- Les méthodes `commit` et `rollback` de `JDBCRowSet` valident ou annulent les modifications de la transaction en cours
- Elles ne doivent être utilisées que si la transaction n'est pas en autoCommit.
- Les méthodes `getAutoCommit` et `setAutoCommit` permettent de consulter et modifier le mode de validation

2. JdbcRowSet - b. Manipulation

```
jdbcRs.last();
while(jdbcRs.previous()){
System.out.println("Nom = "+jdbcRs.getObject("nom"));
}
//modification de la colonne nom de la première ligne
jdbcRs.first();
jdbcRs.updateString("nom","foulen");
jdbcRs.updateRow();
//positionnement et suppression de la seconde ligne
jdbcRs.relative(1);
jdbcRs.deleteRow();
//insertion d'une nouvelle ligne
jdbcRs.moveToInsertRow();
jdbcRs.updateString("nom", "foulena");
jdbcRs.insertRow();
```

2. JdbcRowSet - c. Mise à l'écoute d'un RowSet

- Écouteur de `RowSet` : implante les méthodes suivantes de l'interface `RowSetListener`
  - `cursorMoved` : détermine ce que l'écouteur doit faire lors du déplacement d'un du curseur
  - `rowChanged` : détermine ce que l'écouteur doit faire lors de la modification des données d'un `RowSet`
  - `rowSetChanged` : détermine ce que l'écouteur doit faire lors de l'alimentation du `RowSet` par de nouvelles données.
- `addListener` de `RowSet` : associe un écouteur à un `RowSet`
- `removeListener` : dissocie un écouteur d'un `RowSet`

2. JdbcRowSet - c. Mise à l'écoute d'un RowSet

```
class ExampleListener implements RowSetListener {

 public void cursorMoved(RowSetEvent event) {
 System.out.println("Changement de position");
 System.out.println(event.toString());
 }

 public void rowChanged(RowSetEvent event) {
 System.out.println("Changement de données");
 System.out.println(event.toString());
 }

 public void rowSetChanged(RowSetEvent event) {
 System.out.println("Changement de contenu");
 System.out.println(event.toString());
 }
}
```

```
ExampleListener lstnr = new ExampleListener();
jdbcRs.addRowSetListener(lstnr);
```

3. CachedRowSet - a. Fonctionnement

- `CachedRowSetImpl` : implantation de `CachedRowSet` fournie avec la distribution Java
- Fonctionnement
  1. Connexion à la base juste le temps de récupérer des données
  2. Déconnexion. Il est alors possible de lire, modifier, supprimer des données du `RowSet` (même syntaxe que `ResultSet`) "en local"
  3. Re-connexion pour répercuter dans la base les modifications faites pendant la déconnexion (méthode `acceptChanges`)

3. CachedRowSet - b. Création

1. La création se fait avec le constructeur par défaut de `CachedRowSetImpl`

   ```
   CachedRowSetImpl cRowSet = new CachedRowSetImpl();
   ```

2. Il est nécessaire ensuite de spécifier les paramètres de connexion avec les setters (`setUsername`, `setPassword` et `setUrl`)

3. Suit la spécification de la requête avec `setCommand("select...")`

4. Si le `CachedRowSet` est utilisé pour faire des modifications sur les données, il est nécessaire spécifier les colonnes formant la clé primaire

   ```
   int [] keys = {1};
   cRowSet.setKeyColumns(keys);
   ```

5. Exécution avec la méthode `execute`

3. CachedRowSet - c. Modification

- Les données d'un `CachedRowSet` peuvent être modifiées par divers méthodes `updateXXX` de la même manière qu'un `ResultSet`
  (il est nécessaire d'ajouter un `updateRow`)

- `acceptChanges` : enregistre les modifications et effectue un commit

- `undoInsert`, `undoDelete` et `undoUpdate` : permettent d'annuler resp. la dernière insertion, suppression et mise-à-jour effectuée depuis `acceptChanges`

- Certains SGBD (Oracle en particulier) imposent d'indiquer la table sur laquelle les modifications sont appliquées avec la méthode `setTableName("nomTable")`

3. CachedRowSet - d. Conflits à la reconnexion

- Conflit :
  - Les données modifiées en hors-ligne par un `CachedRowSet` ont également été modifiées dans la base pendant sa déconnexion par un tiers.
  - Déclenché lors de l'exécution de la méthode `acceptChanges`.
- En cas de conflit
  - Annulation des modifications du `CachedRowSet` avec `release`, ou
  - Résolution par un autre traitement.
- Résolution des conflits :
  - Dépend de l'implantation.
  - `CachedRowSetImpl` : utilise une résolution "optimiste"
  - Résolution optimiste : pas de verrouillage lors de la récupération et supposition qu'il ne peut y avoir que peu de conflits

3. CachedRowSet - d. Conflits à la reconnexion

Principe de la résolution

1. Lors d'un `acceptChanges`
   - Les lignes du `CachedRowSet` sont comparées une à une avec les lignes correspondantes dans la base

   ⇒ d'où la nécessite de spécifier la clé primaire lors de la création du `CachedRowSet`

2. Une exception `SyncProviderException` est levée si `acceptChanges` détecte un conflit

3. Les valeurs des colonnes provoquant un conflit sont récupérées depuis la base dans un objet de type `SyncResolver`

4. `SyncResolver` a exactement la même structure (mêmes lignes et mêmes colonne) que le `CachedResultSet`

5. Les valeurs de la base ne provoquant pas de conflits sont remplacées par `null` dans l'objet `SyncResolver`

3. CachedRowSet - d. Conflits à la reconnexion

Principe de la résolution (suite)

6. Les colonnes de chaque ligne de l'objet `SyncResolver` sont parcourues une à une.
7. Lors du parcours, chaque fois qu'une valeur conflictuelle (`!=null`) est rencontrée, la valeur conflictuelle est récupérée avec la méthode `getConflictValue()` du `SyncResolver`
8. On accède à la valeur correspondante du `CachedRowSet` et on décide de la valeur à rendre persistante avec la méthode `setResolvedValue` du `SyncResolver`
9. Une fois tous les conflits résolus, il est nécessaire de refaire un `acceptChanges`

3. CachedRowSet - d. Conflits à la reconnexion

```
try {
    cRowSet.acceptChanges();
}
catch (SyncProviderException spe) {

  //Objet dans lequel sont récupérées les valeurs conflictuelles
  //même structure que cRowSet
  SyncResolver resolver = spe.getSyncResolver();

  // Valeur se trouvant dans cRowSet
  Object cRowSetValue;

  // Valeur se trouvant dans l'objet SyncResolver
  Object resolverValue;

  // Valeur qui sera stockée dans la base
  Object resolvedValue;
```

3. CachedRowSet - d. Conflits à la reconnexion

```
while (resolver.nextConflict()) {
    if (resolver.getStatus() == SyncResolver.UPDATE_ROW_CONFLICT) {
        // Positionnement sur la ligne ayant une colonne conflictuelle
        int row = resolver.getRow();
        cRowSet.absolute(row);
        int colCount = cRowSet.getMetaData().getColumnCount();
        for (int j = 1; j <= colCount; j++) {
            if (resolver.getConflictValue(j)!= null) {
                cRowSetValue = cRowSet.getObject(j);
                resolverValue = resolver.getConflictValue(j);
                // ...
                // On compare les 2 valeurs et on décide
                // de celle à enregistrer.
                resolvedValue = cRowSetValue;
                resolver.setResolvedValue(
                    j, resolvedValue);
            }
        }
    }
}
}
```

Chapitre 3 - Java DataBase Connectivity : aspects avancés

1. Pool de connexions

- Les connexions sont des ressources coûteuses et longues à obtenir
- On peut vouloir les réutiliser dans des threads différents. Malheureusement, ce n'est pas possible.
- À la place on peut utiliser un pool de connexions :
  - Les connexions ne sont pas fermées mais gardées en mémoire dans une sorte de tampon (le pool)
  - Chaque fois qu'un programme demande une connexion, on vérifie d'abord s'il y en a une de disponible dans le pool
  - C'est uniquement dans le cas contraire (connexion non disponible) qu'une nouvelle connexion est créée

1. Pool de connexions

- Objet `DataSource`
  1. À partir de JDBC 3.0, paquetage `javax.sql.DataSource`
  2. Permet d'obtenir une connexion (alternative à `DriverManager`)
  3. Crée et récupère les connexions à partir d'un pool
  4. Gère les transactions réparties
  5. Méthode préférée si l'application est destinée à un déploiement dans un conteneur : le pool peut alors être partagé entre les applications gérées par le conteneur

2. Les objets DataSource

- `DataSource` : représente le plus souvent une BD, mais peut également représenter un fichier texte ou XML.
- Tout pilote JDBC doit fournir une implantation de `DataSource`
  Oracle : `oracle.jdbc.pool.OracleDataSource`
- Une fois la connexion renvoyée, elle s'utilise de la même manière qu'une connexion obtenue par `DriverManager` $\Rightarrow$ la gestion du pool est transparente.
- Configuration d'une `DataSource` :
  - Soit avec des setters
  - Soit par un fichier XML (méthode la plus courante)

2.a. Configuration avec setters

```
OracleDataSource ds = new OracleDataSource();
ds.setURL("jdbc:oracle:thin:@localhost:1521:dbstcm");
Connection con = ds.getConnection("khaled", "isitc0m");
// Le reste est identique au code qui
// n'utilise pas de source de données
```

2.b. Configuration avec fichier XML

- Méthode utilisée lorsque l'application est exécutée par un serveur d'applications (ex. glassfish) ou un conteneur Web (ex. Tomcat)
- Dans ce cas les ressources utilisées par l'application (une source de données en l'occurrence) doivent être déclarées au niveau du serveur d'applications.
- La déclaration se fait dans des descripteurs de déploiement (Ex. `glassfish-resources.xml`)

2.b. Configuration avec fichier XML

- Le serveur d'applications enregistre les ressources récupérées dans les descripteurs de déploiement dans un registre JNDI (transparent à l'utilisateur)
- Registre JNDI : associe (*bind*) chaque ressource à sa description.

  Exemple : source de données aux informations permettant d'accéder à la source (URL, utilisateur, mot de passe, taille du pool, etc.)

N.B. Une fois la ressource enregistrée dans JNDI, le serveur n'utilise plus le fichier glassfish-resources.xml

2.c. Obtention d'un DataSource déclarée dans JNDI

- Le code ci-dessous permet d'obtenir une source de données déclarée auprès du registre JNDI du serveur d'applications

```
InitialContext context = new InitialContext();
DataSource ds = (DataSource) context.lookup("jdbc/dbstcm_jndi");
Connection con = ds.getConnection();
```

- Le registre est organisé sous forme d'arborescence (comme un système de fichiers)
- `InitialContext()` : permet de se positionner à la racine de l'arborescence du registre JNDI du serveur d'applications.
- `lookup("jdbc/dbstcm_jndi")` : recherche dans l'arborescence de la source de données `"jdbc/dbstcm_jndi"`
- Un registre JNDI peut contenir différents types de ressources. `lookup("jdbc/dbstcm_jndi")` retourne un objet que nous devons caster en un objet de type `DataSource`

2.c. Obtention d'un DataSource déclarée dans JNDI

En plus de la gestion des transactions réparties et pools, l'utilisation des DataSource a d'autres avantages :

- **Facilité d'utilisation** : une fois une source de données déclarée au niveau du serveur d'applications, le programmeur n'a plus besoin d'en spécifier les paramètres de connexions
- **Portabilité** : si les paramètres de connexions changent (migration de la BD, modification du password, etc.), la modification se fait au niveau du JNDI, mais le code des applications accédant à la source reste le même

Chapitre 4 - Mapping Objet-Relationnel

Introduction

- Bases de données relationnelles : position dominante sur le marché (théorie solide, concepts éprouvés et normes reconnues)
- Tendance actuelle : de plus en plus d'applications développées en langage de programmation orienté-objet dans les différents secteurs du développement logiciel.
- UML : s'est imposé comme langage de référence pour la modélisation des applications
- Situation courante : modélisation avec UML, traitement des données avec une application orientée objet et stockage dans une base de données relationnelles

Mais alors comment rendre persistantes dans une base de données relationnelles les données manipulées dans une application orientée objet?

Introduction

- Passage Objet/Relationnel : problème non-trivial à cause de la non correspondance des paradigmes Objet/Relationnel
- Modèle relationnel :
  - Données organisées en tables formées par des tuples (lignes) et des attributs (colonnes) mono-valués
  - Tuples identifiés par des clés primaires (attribut ou ensemble d'attirubuts permettant d'identifier de manière unique un tuple)
  - Relations entre tables : clés étrangères joignant les clés primaires
  - Interrogation par SQL, gestion de la concurrence et des transactions, reprise sur panne, indexation, etc.
- Toutes ces notions sont étrangères au modèle objet

Introduction

- Modèle Objet
  - Objets instances de classes identifiés par leurs adresses en mémoire
  - Héritage, classes abstraites/concrètes, polymorphisme, etc.
  - Il n'est pas simple de lui faire correspondre un modèle relationnel

  $\Rightarrow$ C'est là qu'interviennent les outils ORM (`Object/Relational Mapping`) pour
    - automatiser le passage et donner une vue orientée objet des données relationnelles,
    - abstraire autant que possible les appels SQL de bas niveau;
    - fournir une assistance à la gestion des aspects non-fonctionnels (transaction, accès concurrents, etc.)

- Ce chapitre étudie les principaux problèmes liés au passage Relationnel/Objet ainsi que le design pattern DAO

Chapitre 4 - Mapping Objet-Relationnel

Exemple simple de correspondance

- Généralement une application ne travaille pas directement sur la représentation tabulaire des entités [1] (via JDBC par exemple)
- A la place, elle possède son propre modèle. Dans le cas le plus simple une table est traduite en une classe
- Les tuples de la table sont récupérés dans des objets de la classe. Les objets entités jouent alors le rôle de tampon.
- Solution courante lorsque la BD est préexistante
- Classe **entité**
  - Classe dont les instances (les objets) sont stockées
  - Généralement, implémentation POJO (*Plain Old Java Object*) sérialisable (pour les envoies via le réseau ou d'un composant JEE à un autre) avec accesseurs

[1] Les objets sont des instances volatiles qui résident uniquement en mémoire vive. Les entités sont des objets qui résident brièvement en mémoire vive et qui sont stockées d'une manière permanente dans une base de données

Exemple simple de correspondance

- Table `CREATE TABLE dept (idDept NUMBER CONSTRAINT pk_dept PRIMARY KEY, nomDept varchar(15))`

```
public class Dept implements java.io.Serializable {
// Fields
    private int idDept;
    private String nomDept;
    /** default constructor */
    public Departement() {}
    /** full constructor */
    public Departement(int idDept, String nomDept) {
    this.idDept = idDept;
    this.nomDept = nomDept;
    }
    // Property accessors
...
}
```

- Traduction de la table Dept en une classe Dept (ou l'inverse)
- Chaque tuple est chargé dans un objet (ou chaque objet est conservé dans un tuple de la table)
- L'application manipule les tuples via les objets correspondants

Chapitre 4 - Mapping Objet-Relationnel

1. Identification en objet vs. en relationnel

- Problème de l'identification : savoir si 2 objets en mémoire correspondent ou non à un même tuple
- Monde objet : identification des objets par l'adresse de leur emplacement mémoire.
- 2 objets a et b peuvent être les mêmes selon deux modalités différentes : par identité ou par équivalence
- Par identité des objets : a==b
  - a et b sont identiques s'ils référencent un même emplacement mémoire.
  - 2 objets distincts peuvent avoir les mêmes valeurs pour leurs propriétés
- Par égalité ou équivalence :
  - méthode `equals()` à implanter par la classe.
  - a et b sont équivalents s'ils possèdent la même valeur (même état)

1. Identification en objet vs. en relationnel

- Modèle relationnel :
  1. Identification des tuples par la valeur de la clé primaire
  2. Impossible d'avoir 2 tuples avec la même valeur de la clé primaire
- == et `equals()` : n'équivalent ni l'une ni l'autre à une comparaison de la valeur de la clé primaire
- Faut-il ajouter aux objets une propriété pour représenter une clé primaire?
- Souvent **réponse positive** : identification des objets par la propriété correspondant à la clé primaire

1. Identification en objet vs. en relationnel

- Identification des objets par la propriété
  - Certaines précautions doivent être prises, aussi bien du côté de la BD que du code Java

1. Préférer les clés de substitution synthétiques (*Surrogate*) aux clés ayant une valeur significative (*i.e.* clés naturelles : NSS, CIN, etc.)
2. Interdire la modification de la valeur de la propriété
3. Redéfinir la méthode `equals`
4. Interdire la représentation d'un tuple par plus d'un objet (duplication)

2. Clé de substitution (Surrogate)

Clé de substitution

- Exemple : entiers générés automatiquement par le SGBD par `Auto_increment` ou séquence
- Intérêt
  - Éviter les clés composées de plusieurs attributs (comparaison, jointure, etc., plus lentes)
  - Faciliter l'indexation
  - Faciliter les mises-à-jour : ne pas avoir à faire des modifications en cascade en cas de mise-à-jour de la valeur de la clé
  - Éliminer le lien avec le domaine métier : l'identification est un problème lié à la persistance

3. Accès privé en écriture

Rendre la méthode `setId()` privée

- La valeur d'une propriété d'identification, de la même manière qu'une clé primaire ne doit/peut pas être modifiée

Redéfinir la méthode `equals`

- Exemple :

```
@Override
public boolean equals(Object object) {
    if (!(object instanceof Dept)) {
       return false;
    }
    Dept other = (Dept) object;
    if ((idDept == null && other.idDept != null) ||
       (idDept != null && !idDept.equals(other.idDept))) {
       return false;
    }
    return true;
}
```

4. Duplication en mémoire

Interdire la représentation d'un tuple par plus d'un objet

- Un même tuple ne doit pas être représenté par plusieurs objets en mémoire vive
- Si elle n'est pas gérée, cette situation peut conduire à des pertes de données ou à des incohérences
- Exemple :
  1. Création en mémoire d'un objet d_1 de la classe `Dept` à l'occasion d'une navigation à partir d'un objet `Emp`
  2. Puis création du même département dans un objet d_2 à l'occasion du parcours de la table `Dept`.

  $\Rightarrow$ Si l'un ou les des deux objets sont modifiés, comment savoir quelle valeur est valide et laquelle rendre persistante?
- **Solution** : utilisation d'un tampon (appelé également "unité de travail" ou **"Contexte de persistance"**)

4. Duplication en mémoire

Tampon

- Fonctionnement
  - Conserver une référence des objets liés à la base : valeur de la propriété d'identification, associée à l'adresse mémoire
  - Lors d'une navigation, si l'objet (la clé) recherché(e) est référencé(e) dans le tampon, celui-ci fournit sa référence
  - Sinon, création d'un nouvel objet et enregistrement de sa clé et son adresse mémoire dans le tampon
- Fonction de tampon : remplie par les sessions de Hibernate et les gestionnaires d'entités de JPA (nous y reviendrons)

Chapitre 4 - Mapping Objet-Relationnel

1. Granularité

- Granularité du modèle objet plus fine que le modèle relationnel

⇒ Pour un même modèle de domaine, on peut avoir "plus" de classes que de tables
Ex. association 1-1 traduite en relationnelle par la fusion des deux entités dans une même table et en objet par deux classes

⇒ Un tuple peut être décomposé en plusieurs objets et inversement plusieurs objets peuvent se recomposer pour se sauvegarder dans un seul tuple

- Ces objets sont appelées des objets intégrés (*embedded* qui signifie intégré, imbriqué)

- Les objets intégrés ne nécessitent pas de clé primaire

2. Objets intégrés

- Exemple :
  - Classe `Emp` en association avec une classe Adresse (association 1-1 par exemple)
  - La classe `Adresse` peut ne pas avoir de table correspondante dans la base
  - À la place, les attributs de la classe `Adresse` sont intégrées dans la table qui représente la classe Employé.
  - Les objets `Adresse` n'ont pas d'identifiants propres dans la base
- `Adresse` est dite classe intégrée (*embedded*)

Chapitre 4 - Mapping Objet-Relationnel

1. Les associations en objet

- Association : liens entre les objets/tuples
- Modèle relationnel : une association se matérialise par
  - Une clé étrangère (association 1-1 ou 1-N ou N-1)
  - Table dont la clé primaire est composé des clés étrangères (association M-N)
  - Les associations sont bidirectionelles par jointure, *i.e.* il est possible de parcourir une association dans les deux sens.
    Exemple : à partir d'un employé on peut trouver son département et inversement à partir d'un département on peut trouver ses employés

1. Les associations en objet

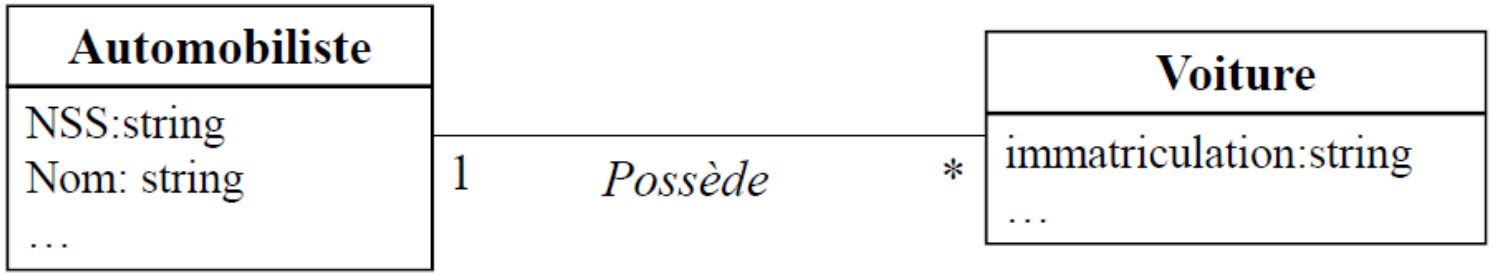
- Modèle objet : une association se matérialise par
  - Une variable d'instance référençant l'objet associé (association 1-1 ou N-1)
  - Une variable d'instance de type `Collection`, `List` ou `Map` contenant les références vers les objets associés (association 1-N ou M-N)
  - Une classe "association" (association M-N)
  - Une association peut être unidirectionnelle ou bidirectionnelle
- Contrairement au relationnel, il est difficile de parcourir tous les objets d'une classe à la recherche d'une valeur donnée [2]

  $\Rightarrow$ On ne peut pas utiliser l'opération de jointure

  $\Rightarrow$ D'où la nécessité dans certains cas de représenter l'association dans les deux classes (associations bidirectionnelles)

[2] Il serait nécessaire de passer par une collection statique référençant toutes les instances

2. Association 1-1/N-1

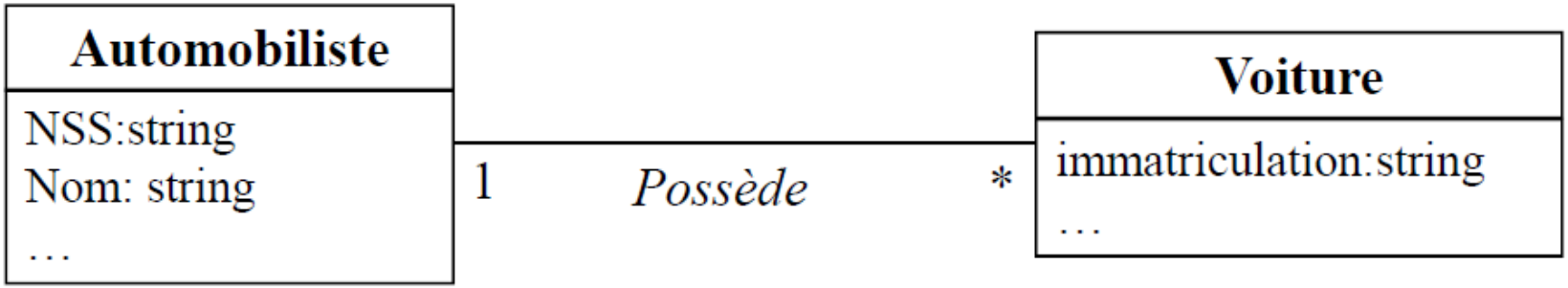



```
CREATE TABLE Automobilsite
( Automobiliste_ID  SERIAL,
  NSS varchar(10) NOT NULL,
  Nom varchar(20) NOT NULL,
  ...
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Automobiliste_ID),
) ;

CREATE TABLE Voiture
( Voiture_ID  SERIAL,
  Immatriculation varchar(10) NOT NULL,
  ...
  Proprietaire_ID int NOT NULL,
  CONSTRAINT PK_Automobiliste PRIMARY KEY (Voiture_ID),
  CONSTRAINT PK_Automobiliste_Voiture
   FOREIGN KEY (Proprietaire_ID) REFERENCES Automobiliste (Automobiliste_ID)
);
```

Figure: Exemple repris de ["Persistance des objets". Notes de cours. M. Manouvrier. 2010]

2. Association 1-1/N-1



De Automobiliste vers Voiture

```
public class Automobiliste {
  // Fields
  private String NSS;
  private String nom;
  private Collection<Voiture> parc_automobile;
  ...
}
```

De Voiture vers Automobiliste

```
public class Voiture {
  // Fields
  private String immatriculation;
  private Automobiliste proprietaire;
  ...
}
```

Figure: Exemple repris de ["Persistance des objets". Notes de cours. M. Manouvrier. 2010]

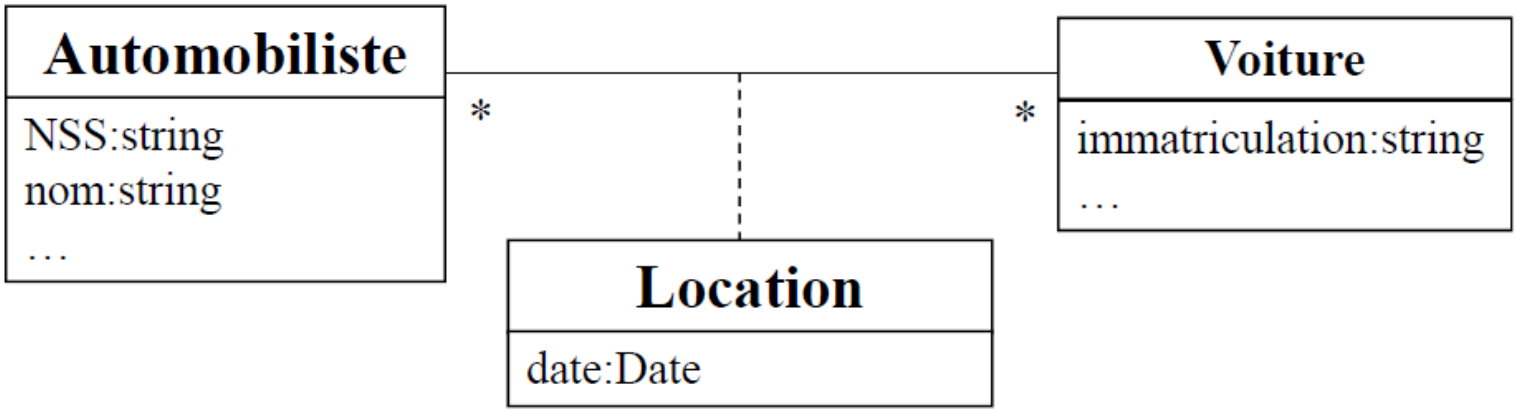
2. Association 1-1/N-1

- Il est possible d'utiliser des `Map` ou des `Collections`. Exemple :
  - `private Map<String, Voiture> parc_automobile;`

    au lieu de `private Collection<Voiture> parc_automobile;`
  - Variante avec `Map` : permet un accès rapide à une des voitures d'un automobiliste (si on connaît son immatriculation)
- En objet, une association peut être unidirectionnelle. Exemple :
  - la classe `Voiture` a une variable d'instance donnant son propriétaire, mais la classe `Automobiliste` n'a pas de collection `parc_automobile` (ou l'inverse).

  $\Rightarrow$ En partant d'une Voiture il est aisée de trouver son propriétaire, mais partant d'un Automobiliste on n'a pas alors un moyen simple pour trouver ses voitures

3. Association M-N




```
public class Location {
  private Automobiliste loueur;
  private Voiture vehicule;
  private Date date;
  …
}
```

```
CREATE TABLE Location
( …
 CONSTRAINT PK_Location PRIMARY KEY (Automobiliste_ID,Voiture_ID, Date),
 CONSTRAINT FK_Location_Voiture
   FOREIGN KEY (Voiture_ID) REFERENCES Voiture (Voiture_ID),
 CONSTRAINT FK_Location_Automobiliste
   FOREIGN KEY (Automobiliste_ID) REFERENCES Automobiliste (Automobiliste_ID),
);
```

3. Association M-N

- 2 façons différentes pour représenter une association M-N
  1. une collection ou une map dans chacune (ou dans une, selon la directionnalité) des classes associées
  2. Une classe association qui représente une instance de l'association
- Seule la deuxième méthode convient si l'association est porteuse d'informations

4. Gestion des associations

- Gestion des associations plus complexe en objet qu'en relationnel
- Exemple : modification du propriétaire d'une voiture
  - Relationnel : modification de la valeur de la clé étrangère dans la table `Voiture`
  - Objet :
    suppression du véhicule correspondant dans la collection `parc_automobiliste` de l'objet `Automobiliste`

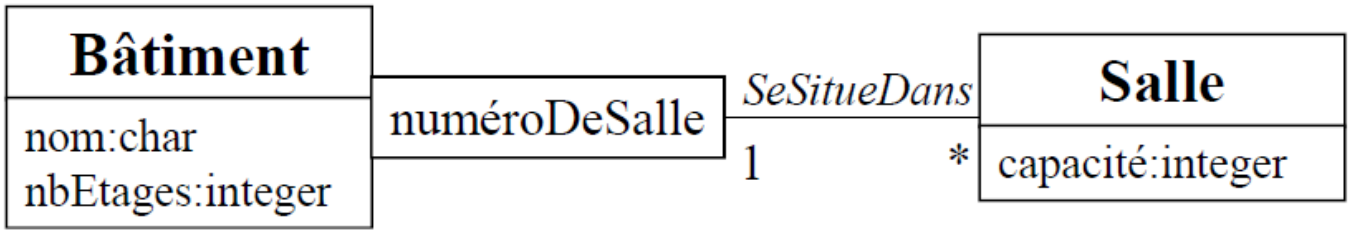
    +

    mise-à-jour de l'instance propriétaire de l'objet `Voiture`

Chapitre 4 - Mapping Objet-Relationnel

Objets dépendants

- Objet dépendant : objet dont le cycle de vie dépend d'un autre objet auquel il est associé
- Suppression de l'objet propriétaire implique celle de ses objets dépendants (suppression en cascade)
- Tous les outils de mapping (comme JPA) offrent la possibilité d'automatiser les suppressions en cascade des objets dépendants

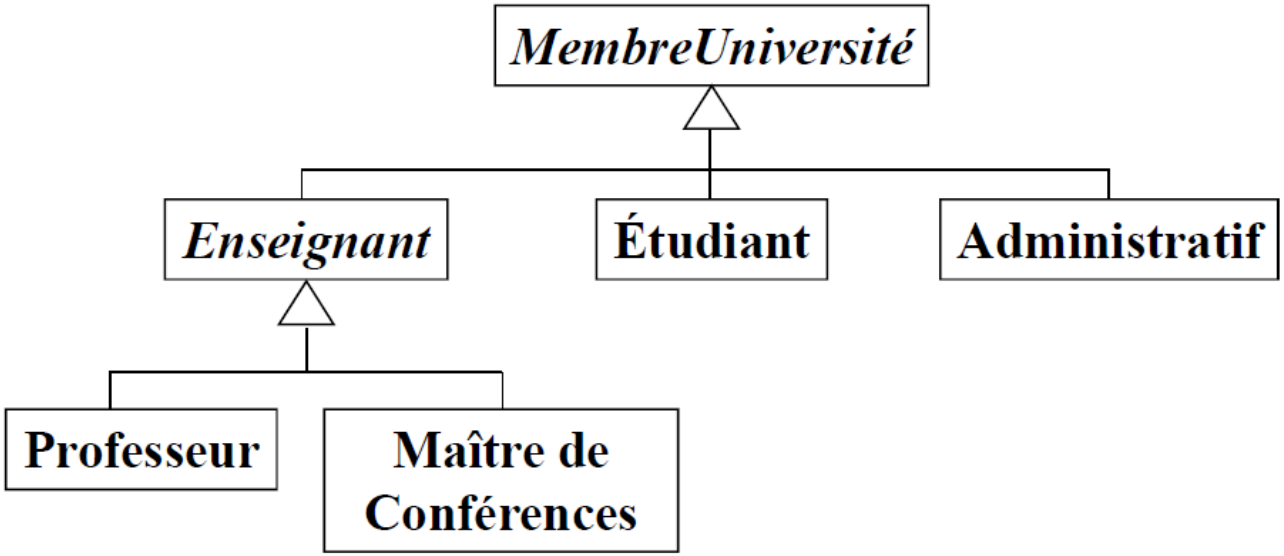



```
Batiment(IDBatiment, nom, nbEtages)
Salle(#IDBatiment, numeroSalle, capacité)
```

Chapitre 4 - Mapping Objet-Relationnel

1. Stratégies de traduction

- **Héritage** en Objet : permet des associations et des requêtes polymorphes



- **Association polymorphe**
  - Association représentée par une référence vers une instance d'une classe ou des sous-classes de cette classe
  - Exemple : un département garde une référence vers tout membre de l'université qui travaille, enseigne ou étudie en son sein.
  - En objet le département référence uniquement la classe racine de la hiérarchie (MembreUniversité)

1. Stratégies de traduction

- **Requête polymorphe**
  - Requête renvoyant une information conservée dans les instances d'une classe et des sous-classes de cette classe
  - Exemple : requête renvoyant les noms de tous les membres de l'université.
  - En Objet on interroge uniquement la classe racine de la hiérarchie (MembreUniversité)

1. Stratégies de traduction

- Héritage : non supporté dans le modèle relationnel
- Possibilités de traduction de l'héritage
  - Faire correspondre toutes les classes de la hiérarchie à une seule table
  - Représenter chaque classe, abstraite ou concrète, par une table
  - Représenter chaque classe concrète par une table (perte du polymorphisme)

2. Une seule table pour toute la hiérarchie

- Table avec tous les attributs de chacune des classes en plus d'un attribut pour distinguer les différents types (**attribut discriminant**)

- Avantages et inconvénients

  + Facilité de mise en oeuvre. Solution la plus courante

  + Possibilité des requêtes et associations polymorphes

  + Bonnes performances (pas de jointures, on y reviendra)

  - Valeurs `NULL` pour les attributs non communs

  - Pas de possibilité de déclarer des contraintes NOT NULL, même si celles-ci doivent être vérifiées (Exemple, date de recrutement d'un enseignant)

3. Table pour chaque classe

- Table pour chaque classe (abstraite ou concrète). Chaque table ne contient que les attributs non hérités

  ⇒ Répartition des attributs d'un objet dans plusieurs tuples

  Ex. Nom, Adresse, DateNaiss dans un tuple de la table MembreUniversité et DateRecrut dans un tuple de la table Enseignant

- Pour pouvoir reconstituer un objet : préservation de l'identité en donnant la même valeur de clé primaire à tous les niveaux de la hiérarchie

  ⇒ Les clés primaires des tables correspondant aux sous-classes, sont également des clés étrangères référençant la classe mère

  Ex. la clé primaire de la table Etudiant est aussi une clé étrangère qui fait référence à la clé primaire de la table MembreUniversité

3. Table pour chaque classe

- Avantages et inconvénients
  - \+ Simple bijection entre les classes et les relations
  - \+ Possibilité de faire des des requêtes et associations polymorphes
  - \+ Vérification possible des contraintes d'intégrité
  - \- Nombreuses jointures pour reconstituer un objet
    $\Rightarrow$ mauvaises performances si hiérarchie profonde

4. Table pour chaque classe concrète

- Table pour chaque classe concrète. Chaque table contient tous les attributs (hérités ou non)
- Préservation de l'identité en cas de classe concrète avec des classes filles : clé primaire des tables correspondant aux classes filles = clés étrangères faisant référence à la clé primaire de la table correspondant à la classe mère
- Avantages et inconvénients
  - \+ Pas de jointure pour retrouver les informations
  - \- Problème pour les requêtes et les associations polymorphes. Exemple "trouver le nom des membres", solution : union de plusieurs select
  - \- Redondance source d'incohérences
- Solution à éviter (optionnelle dans JPA 2.1)

Chapitre 4 - Mapping Objet-Relationnel

1. Problématique

- **Problème** : lorsqu'un objet créé à partir des données récupérées dans la base référence d'autres objets non encore créés, faut il créer aussi les objets référencés?
- Exemple :
  - Objet facture référençant plusieurs lignes et chaque ligne référençant un produit.
  - Recherche dans la base d'une `Facture` vérifiant un certain critère
  - Est-ce que les objets `LigneFacture` associés à cette facture doivent aussi être créés?
  - Et si la réponse est positive, faut-il créer aussi les objets `Produit` référencés par les lignes extraites
  - Et si la réponse est positive et que chaque produit référence un `Fournisseur` ...

2. Chargement immédiat

- Deux stratégies possibles pour la récupération des objets référencés :
  - *Eager loading* : récupération immédiate
  - *Lazy loading* : récupération à la demande (différée)

- **Eager loading**
  - Récupération immédiate des objets référencés
  - Risque de créer un très grand nombre d'objets inutilement
  - Exemple : nous souhaitons uniquement avoir la date de la facture.
  - Les objets associés sont inutiles et leur chargement dégrade les performances et pollue la mémoire vive

3. Récupération à la demande

- **Lazy loading**
  - Récupération à la demande des objets référencés
  - Les objets référencés ne sont pas créés mais remplacés par des objets "proxy"
  - "Proxy" ne contient que l'information nécessaire (clé primaire) pour pouvoir créer le vrai objet (encore une raison pour utiliser des Surrogates)
  - Les vrais objets ne sont créés que s'il contiennent une information requise par l'application
  - Exemple : récupération uniquement des données de la facture (Ex. date), puis si jamais l'application accède à une ligne, celle-ci est chargée dans un objet.
  - De même le produit référencé dans la ligne n'est chargé dans un objet que si l'application y accède
- Performances améliorées, mais problème des "N+1" sélections

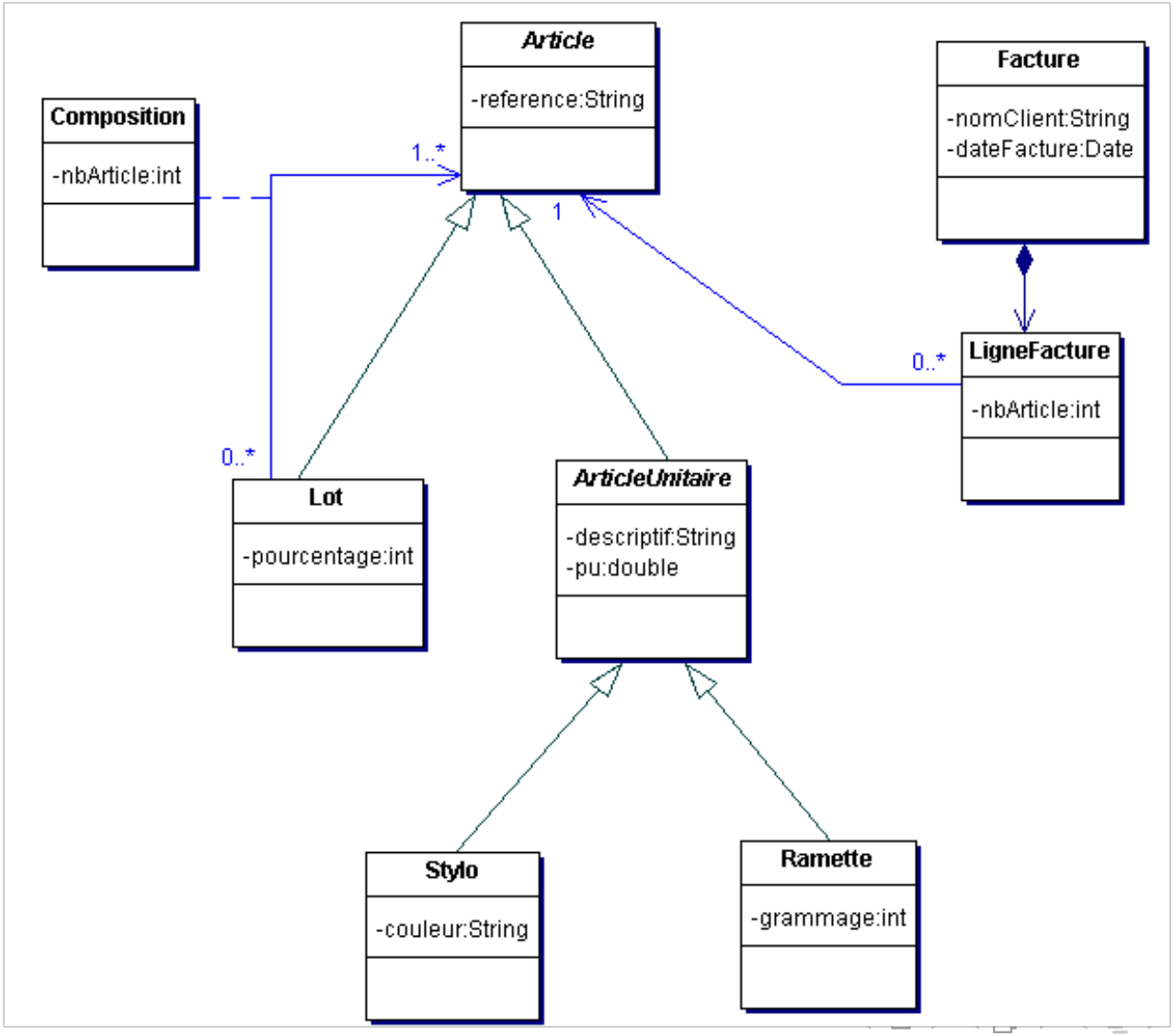
3. Récupération à la demande

- Exemple de "N+1" sélections
  - Afficher les lignes des 100 dernières factures, avec une récupération à la demande des lignes
  - Les 100 objets factures sont tout d'abord récupérés ⇒ 1 `Select`
  - Dans un deuxième temps, on récupère pour chaque facture ses lignes : 100 `Select` (un `Select` par (numéro) facture)
  - D'où un total de 101 `Select`, alors que le résultat peut s'obtenir avec un seul

- **Solutions possibles**
  - Lancer un ordre SQL *ad hoc* qui ne renvoie que ce qui est recherché.
  - Choisir le mode adéquat pour chaque requête. Tous les outils de mapping le permettent

Exercice

Donnez le schéma relationnel correspondant au modèle du domaine ci-dessous



Chapitre 4 - Mapping Objet-Relationnel

1. Présentation - a. Qu'est ce que DAO?

- *Design Pattern* (patron ou modèle de conception) :
  - Bonne pratique ou réponse standard à un problème de conception
  - Issue de l'expérience des concepteurs de logiciels.

- **DAO** (*Data Access Object*) : pattern consistant à
  - isoler le code de persistance (Ex. Ordres SQL `Insert into`, `Delete from`, ...) en dehors des classes entités et des classes métiers
  - abstraire les appels de bas niveau (Ex. `persist(Objet)` au lieu de `INSERT INTO...VALUES(Objet.getId(),,,))`

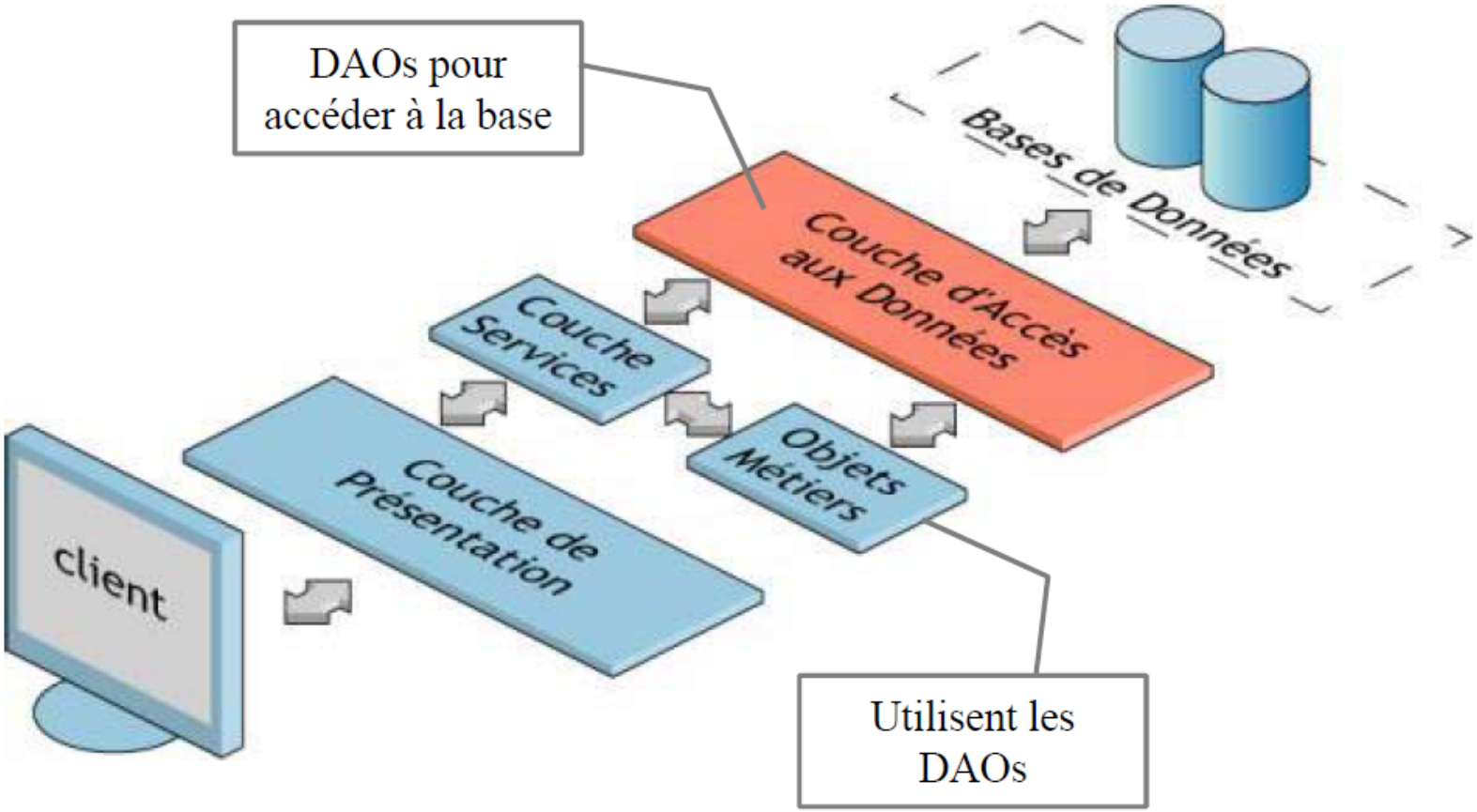
1. Présentation - a. Qu'est ce que DAO?

- Principe :
  - Chaque classe entité a une classe correspondante qui se charge d'exécuter ses opérations CRUD (*Create, Retrieve, Update, Delete*). Ex. `EmpDAO`, `DeptDAO`
  - Les ordres SQL de gestion des données (Insert, Delete, Update, Select) ne sont utilisés que dans les méthodes des classes DAO
  - Les classes métiers appellent des méthodes d'autres classes, *i.e.* les classes DAO, qui se chargent d'exécuter le code SQL.
  - Les classes métier n'interagissent pas directement avec la BD, mais uniquement avec les classes DAO

1. Présentation - b. Utilité des DAO

- Utilité des DAO
  - En cas de changement du mode de stockage des objets entités, le code des classes métier et entités n'est pas affecté. Seules les classes DAO sont modifiées
  - Changement : changement de SGBD, évolution d'un SGBD, etc.
  - Il est plus facile d'effectuer le changement si tout le code SQL est regroupé dans les classes DAO (il n'est pas nécessaire de parcourir tout le code de l'application pour examiner les ordres SQL)
  - Factoriser le code d'accès aux données (si plusieurs classes métier ont besoin de manipuler les mêmes entités.)
- Sans doute le modèle de conception le plus utilisé dans le monde de la persistance objet

1. Présentation - c. Emplacement des DAOs



1. Présentation - c. Emplacement des DAOs

- Objets DAO :
  - Placés dans la couche "d'accès aux données"
  - Couche pouvant être sur une machine différente de la couche où sont instanciées les classes métier
  - Les échanges de messages entre DAOs et les objets métiers engendrent souvent des appels
  - Pour réduire les appels distants, on n'envoie pas les résultats des requêtes ou les attributs des entités interrogés un à un mais, regroupés soit dans un objet entité soit dans un DTO
  - DTO (*Data Transfer Objects*) : s'utilisent pour améliorer la vitesse des échanges

1. Présentation - c. Emplacement des DAOs

- Objet de transfert DTO :
  - Exemple d'utilisation : application souhaitant récupérer les noms des employés chefs de projets
  - L'appel distant un à un des accesseurs (`getNom`) est inefficace
  - Solution : construire un objet (le DTO) contenant toutes les valeurs
  - À utiliser si les objets entités ne sont pas sérialisables ou si on souhaite transférer plusieurs objets en un seul appel
  - A éviter si l'application est locale (objets entités et programme appelant sur une même machine)

2. CRUD

- Une DAO implante (au moins) 4 opérations de persistance de base :
  - Create : création d'une nouvelle entité dans la base
  - Retrieve : recherche d'une ou de plusieurs entités de la base
  - Update : modification d'une entité dans la base
  - Remove : suppression d'une entité de la base
- Plusieurs variantes pour les signatures de ces méthodes dans un même DAO

2. Opérations CRUD - a. Méthode create

- Prend en paramètres l'état de la nouvelle entité
- Cet état peut être donné par
  - une série de paramètres : `create(int idE, String nomE,..)`
  - un DTO : `create(DTOEmp dtoE)`
  - L'objet entité que l'on veut persister : `create(Emp emp)`
- Type de retour
  - `void` (variante la plus utilisée)
  - `boolean` pour indiquer si la création a pu avoir lieu
  - identificateur de l'entité ajoutée (utile si la clé primaire est générée automatiquement)
  - objet entité ou un DTO correspondant à l'entité ajoutée

2. Opérations CRUD - a. Méthode create

```
//variable d'instance
private String insert =
"insert into Emp (idE, nomE, sal, refDept)
values(?, ?, ?, ?)";
...
//Méthode create
Emp create (Long idE, String nomE, Float sal, Long refDept){
...
PreparedStatement ps = connexion.prepareStatement(insert);
ps.setLong(1, idEmp);
ps.setString(2, nomE);
ps.setDouble(3, sal);
ps.setLong(4, refDept);
ps.executeUpdate();
...
}
```

2. Opérations CRUD - b. Retrieve

- 3 types de méthodes, selon qu'elle retourne
  - un seul objet
  - une collection d'objets
  - une valeur calculée à partir de plusieurs entités (agrégation)
- Une méthode (ou un objet) qui retourne des données de la base est souvent appelé *finder*

2. Opérations CRUD - b. Retrieve

- *finders*
  - `find(id)` : retrouve une entité en donnant son identificateur à la BD
  - `findAll()` : retrouve toutes les entités du type géré par le DAO
  - D'autres *finders* cherchant les entités selon des critères donnés sont également rencontrés (dépendent des traitements métier)
- Types de retour
  - Objet ou null
  - ResultSet, RowSet, Collection (`Collection`, `List`, `Set`, ...)

2. Opérations CRUD - c. Update

- Paramètres, soit
  - identificateur + valeurs
  - objet entité dont on veut sauvegarder les modifications
- Type de retour
  - `void`
  - `boolean` : pour indiquer si la modification a pu avoir lieu

2. Opérations CRUD - d. Remove

- Paramètres, soit
  - identificateur
  - objet entité à supprimer (de la base)
- Type de retour
  - `void`
  - `boolean` : pour indiquer si la suppression a pu avoir lieu

3. Stratégies d'utilisation des DAOs

- Utilisation des DAO : deux stratégies possibles
- Première stratégie
  - Chaque objet entité a une référence à son DAO et l'utilise pour sa propre persistance.
  - Les classes métier ne connaissent pas les DAOs.
  - La référence peut être obtenue par une méthode `static` de la classe DAO
  - Ceci permet de partager un DAO entre les entités d'une même classe
- Deuxième stratégie
  - Les classes métier utilisent directement les DAOs
  - Les objets entité n'ont pas de référence à un DAO
  - Stratégie la plus fréquemment utilisée (y compris dans JEE)
  - Plus de souplesse (Ex. objets intégrés), mais perte de la pureté de la programmation OO

3. Stratégies d'utilisation des DAOs - a. Première stratégie

Exemple de la première stratégie

```
class Emp {
    private EmpDAO dao;
    ...
    public void sauvegardeToi() {
        dao = getDAO();
        dao.insertOrUpdate(this);
    }
    private EmpDAO getDAO() {
        if (dao == null)
        dao = EmpDAO.getDAO();
    return dao;
    }
...
}

//Utilisation en dehors de la classe
Emp emp;
emp.devientPersistant();
```

En dehors de la classe entité, la DAO n'est pas visible.

3. Stratégies d'utilisation des DAOs - b. deuxième stratégie

Exemple de la deuxième stratégie

```
EmpDAO empDAO = new EmpDAO();
int idEmp =...
empDAO.create(idEmp, "Foulen ", 1000,...);
. . .
Emp emp = empDAO.findById(idEmp);
emp.setSalaire(1500);
empDAO.update(emp);
List<Emp> lEmp = empDAO.findAll();
```

4. DAO et connexions

- Une connexion peut être ouverte au début des méthodes du DAO et fermée à la fin

  $\Rightarrow$ Stratégie coûteuse si un pool de connexions n'est pas utilisé

- Alternativement, le client du DAO (Ex. Objet métier) ouvre la connexion
  - Une fois ouverte, la connexion est passée au DAO
  - Le DAO doit alors avoir une méthode `setConnection(Connection conn)`
  - Ceci permet de partager une même connexion entre plusieurs objets et méthodes DAO
  - Avec JPA, c'est un objet `EntityManager` qui est passé en paramètre (contrairement à utilisation avec JDBC, `EntityManager` de JPA exécute plusieurs opérations de manière automatique en se basant sur les méta-données)

4. Utilisation des DAO - b. DAO et transactions

- Il n'est pas rare de vouloir inclure un ou plusieurs appels de méthodes d'un ou de plusieurs DAO dans une seule transaction
- $\Rightarrow$ C'est donc au client, et non au DAO, de démarrer et valider ou annuler une transaction
- Le DAO utilise la transaction en cours si elle existe

4. Utilisation des DAO - c. Exemple schématique JDBC - DAO

Exemple schématique JDBC - DAO

```
public class EmpDao {
private Connection connect;
public void setConnection(Connection conn) {
    this.connect = conn;
}
public long create(...) {
    PreparedStatement pstmt =
    conn.prepareStatement(...);
    pstmt.setString(...);
    ...
    pstmt.executeUpdate();
}
```

4. Utilisation des DAO - c. Exemple schématique JDBC - client

Exemple schématique JDBC - client

```
Connection conn = ... ;
daoEmp.setConnection(conn);
daoDept.setConnection(conn);
...
daoEmp.create(...);
daoDept.update(...);
conn.commit();
```

4. Utilisation des DAO - d. DAO et EJB

- Dans les applications construites selon la spécification EJB, les DAOs sont implantés le plus souvent par des beans de session sans état (`@stateless`) avec des contextes de persistances

Chapitre 4 - Mapping Objet-Relationnel

Persistance avec JDBC

- Persistance avec JDBC : le programmeur a la charge
  - de traduire le modèle objet en modèle relationnel (ou inversement)
  - d'implanter les méthodes permettant la sauvegarde d'un objet dans la base et l'instanciation d'un objet à partir de données récupérées dans la base
  - d'implanter la récupération à la demande et la récupération immédiate
  - d'implanter un cache pour éviter la duplication
  - de gérer les suppressions en cascade
  - de gérer les transactions
  - etc.
- Tâche difficile, risquée et consommatrice de temps (on estime que 30% du temps de développement se perd dans la gestion de la persistance)

Outils de mapping

- Des outils ORM (Object/Relational Mapping) ont été développés pour automatiser la plupart des tâches et réduire grandement la difficulté
- Offrent une vue orientée objet des données relationnelles et vice versa
- Si la base préexiste, ils permettent de créer des squelettes de classes Java à partir du schéma de la base et inversement
- S'appuient sur les méta-données
- Principaux outils : Hibernate, TopLinks, Java Data Objects (JDO), JPA (Java Persistence API), etc.
- Il est préférable d'utiliser JPA (Java Persistence API) car intégrée à Java EE

Chapitre 5 - Java Persistence API - Partie 1 : ORM

1. Présentation

- JPA (*Java Persistence API*) : API facilitant la gestion de la persistance des objets et l'accès aux données relationnelles à partir d'une application Java
- Créée avec JEE 5 (JPA 1.0) et améliorée dans JEE 6 (JPA 2.0).
- Interfaces, classes et annotations réunies dans le paquetage `javax.persistence`.
- EclipseLink 1.1 (intégré à GlassFish) est l'implantation open-source de référence de JPA 2.0. Il en existe d'autres (Ex. Hibernate Entity Manager, Apache OpenJPA, etc.)
- Peut s'appliquer à toutes les applications Java, même celles s'exécutant en dehors d'un serveur d'applications (Java SE, application Web, etc.).
- Couche d'abstraction au-dessus de JDBC. Appels simples de haut niveau.
  Exemple : `persist(objet)` au lieu de `executeUpdate("update...")`

1. Présentation

- Principaux composants :
  - ORM : gestion et génération automatisée d'une BD à partir des classes entités et inversement.
  - Une API gestionnaire d'entités `EntityManager` permettant d'effectuer des opérations CRUD sans utiliser directement JDBC
  - JPQL (*Java Persistence Query Language*), permettant de récupérer les données à l'aide d'un langage de requête orienté objet
  - Mécanismes de transactions et de verrouillage fournis par JTA (*Java Transaction API*)
  - Des méthodes de rappel et des écouteurs permettant d'ajouter la logique métier au cycle de vie d'une entité

2. Mapping Object-Relational avec JPA

- Mapping Objet-relationnel réversible à l'aide de méta-données (annotations ou XML)
- Méta-données : indiquent comment se fait la correspondance les objets et les données relationnelles
- Peuvent s'exprimer sous forme
  - Annotations : le code de l'entité est directement annotés
  - Descripteurs XML : fichier XML externe déployé avec les entités
- Pour réduire le nombre de méta-données, JPA utilise le principe de "convention plutôt que configuration" (ou "configuration par exception").
  Exemple : par défaut si une classe et la table correspondante portent le même nom, on n'ajoute pas d'annotation sur le nom

2. Mapping Object-Relational avec JPA

- Exemple :

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  @Column(nullable = false)
  private String title;
  private Float price;
  @Column(length = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;

  // Constructeurs, getters, setters
}
```

Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

3. Le gestionnaire d'entités

- `EntityManager` (gestionnaire d'entités) est une interface implantée par le fournisseur de persistance (Ex. EclipseLink)
- Il est le principal interlocuteur du programmeur
- Un objet `EntityManager` fournit les méthodes pour rendre persistant un objet, le supprimer de la base, lire son état à partir de la base, etc.
- Exemple : sauvegarde dans la base d'un objet dept

```
Dept dept = new dept(1, "INFO");
EntityManagerFactory emf =
 Persistence.createEntityManagerFactory("rhPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.persist(dept);
```

- `EntityManager` utilise les méta-données pour un mapping automatisé

3. Le gestionnaire d'entités

- **Unité de persistance** (dans l'exemple `"rhPersistenceUnit"`) :
  - Pont reliant le contexte de persistance à la base
  - Indique les paramètres de connexion à la base et le fournisseur de persistance implantant `EntityManager`.
  - Indique également la liste des classes entités dont les objets sont à gérer par l'instance `EntityManager`[3].
  - Nom de l'unité de persistance et informations déclarés dans le fichier XML `persistence.xml`
  - On en a besoin pour pouvoir créer des objets `EntityManager`

- **Contexte de persistance** :
  - Ensemble d'entités gérées par un gestionnaire d'entités à un instant donné.
  - Le gestionnaire veille à ce qu'il n'existe pas 2 entités avec le même identifiant dans son contexte

[3] Optionnel parce qu'elles peuvent être déterminées dynamiquement à l'exécution grâce aux annotations `@Entity`

3. Le gestionnaire d'entités

- **Fabrique de gestionnaires d'entités** (`EntityManagerFactory`) :
  - La configuration d'un gestionnaire d'entités est liée à la fabrique qui l'a créé
  - La fabrique prend en paramètre le nom d'une unité de persistance
  - Elle consulte le fichier `persistence.xml` pour récupérer les informations de connexion, la liste des entités à gérer et le fournisseur de persistance associés au nom de l'unité de persistance
  - Ces informations lui permettent la création des gestionnaires d'entités pour cette unité, quelque soit le fournisseur et la base d'une manière transparente à l'utilisateur (*i.e.* suit en cela les principes du design pattern *Factory*)
  - La création d'un gestionnaire d'entités se fait par la méthode statique `createEntityManager()`
  - Généralement Il n'existe qu'une seule fabrique par unité de persistance (mais il peut exister plusieurs `EntityManager` par unité de persistance).

3. Le gestionnaire d'entités

- **Fabrique de gestionnaires d'entités** `EntityManagerFactory` : (suite)
  - Dans un environnement Java EE, comme par exemple dans Glassfish, l'instanciation d'un objet de type `EntityManagerFactory` se fait par le conteneur d'une manière transparente au programmeur

⇒ le programmeur se contente de fournir un fichier persistence.xml et ne créé pas d'objet `EntityManagerFactory`

- Exemple :

```
... @Stateless
public class EmpFacade {
    @PersistenceContext(unitName = "rhPersistenceUnit")
    private EntityManager em;
...
}
```

  - **Rq.** `EmpFacade` dans l'exemple : joue le rôle de DAO pour les entités Emp

4. Méthodes de rappel et écouteurs

- Cycle de vie d'une entité
  - Gérée (par le gestionnaire d'entités) : a une identité de persistance et son état est synchronisé avec la base
  - Détachée : l'entité existe en mémoire mais n'appartient pas au contexte de persistance. Son état n'est pas synchronisé avec la base.
    Exemple : création d'une entité sans appel à `persist`
  - Lorsque le contexte de persistance est fermé (`EntityManager.close()`) toutes les entités deviennent détachées.

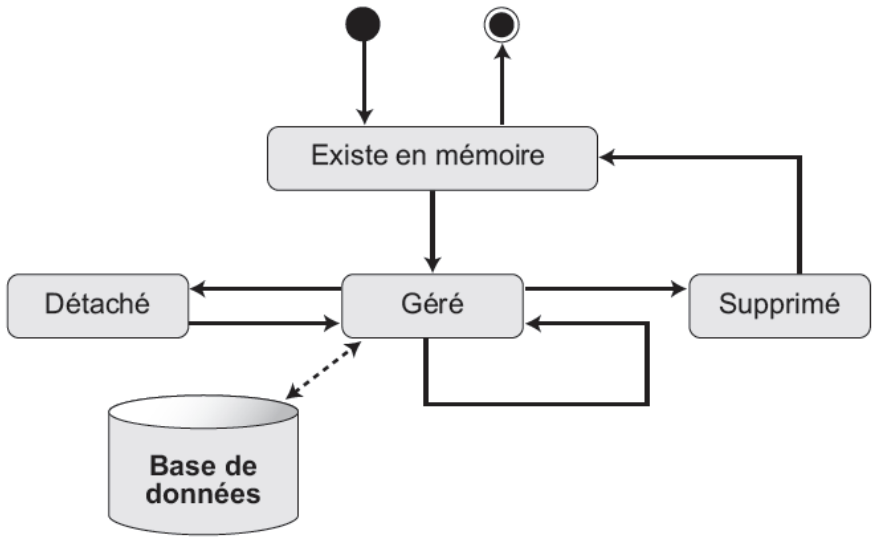


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

4. Méthodes de rappel et écouteurs

- Chaque opération exécutée par le gestionnaire sur une entité donnée a un événement "Pre" et un événement "Post"
- Ces événements peuvent être interceptés par le gestionnaire d'entité pour invoquer une méthode contenant un code métier
- On peut ainsi annoter une certaine méthode `@PrePersist`. La méthode sera exécutée avant l'appel à la méthode `persist`

Chapitre 5 - Java Persistence API - Partie 1 : ORM

1. Entités

- Entité : classe dont les instances peuvent être persistantes
- Il s'agit d'une classe POJO simple à laquelle le développeur associe l'annotation `@Entity`
  Rq. une classe entité s'instancie et s'utilise comme n'importe quelle autre classe Java
- Exemple :

```
@Entity
public class Dept {
      @id
      private Long idD
      ...
}
```

- Dans la suite, quand il n'y a pas d'ambiguïté, nous utilisons le terme entité pour désigner soit une classe entité soit l'une de ses instances (objet persistant)

1. Entités

- Pour être une entité, une classe doit en plus respecter quelques règles
  - L'annotation `@javax.Persistance.Id` doit être utilisée pour désigner une clé primaire simple (non composée)
  - La classe doit disposer d'un constructeur sans paramètre, `public` ou `protected`
  - Aucune méthode ou champ persistant ne doit être `final`
  - La classe doit implanter `Serializable` : utile pour le transfert d'une instance d'une couche à une autre
  - Peut être une classe abstraite mais pas une interface

2. Configuration par exception

- JPA adopte un ensemble de conventions pour réduire autant que possible l'effort de configuration du mapping (le moins d'annotations possible)
- Exemples de conventions
  - Nom d'une classe entité le même que la table correspondante
    sinon ajouter une annotation `@table(name=...)`
  - Noms des attributs les mêmes que ceux des colonnes correspondantes
    sinon ajouter une annotation `@Column(name=...)`
  - Les règles de correspondance des types Java/SQL sont le mêmes que pour JDBC

2. Configuration par exception

- Les outils de mapping (EclipseLink en particulier) permettent de produire la BD à partir des classes entité et inversement.
- Schéma relationnel généré à partir des classes entités
  - Seules les annotations `@Entity` et `@id` sont nécessaires
  - Les valeurs par défaut sont généralement suffisantes pour le reste
- Classes entités générées à partir des tables de la base
  - Les valeurs par défaut ne conviennent souvent pas, surtout en ce qui concerne les associations

3. Clés primaires - a. Clés générées automatiquement

- Valeurs d'une clé : produites par l'application ou générées automatiquement au niveau de la base
- Annotation `@GeneratedValue` : indique que la valeur de la clé est générée automatiquement
- `@GeneratedValue` : peut prendre une des quatre valeurs suivantes :
  - `SEQUENCE` ou `IDENTITY` : valeur de clé générée par une séquence SQL (Oracle, PostGreSQL) ou une colonne `IDENTITY` (SQLServer)
  - `TABLE` (Apache Derby) : le fournisseur de persistance créé une table de deux colonnes, une pour le nom de la séquence et l'autre pour la valeur actuelle de la séquence. La valeur est incrémentée à chaque insertion d'une nouvelle entité
  - `AUTO` : la valeur est générée automatiquement par le SGBD sous-jacent (MySQL). Valeur par défaut.

3. Clés primaires - a. Clés générées automatiquement

Exemple de clé gérée par une séquence Oracle SEQ_DEPT

```
...
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator="idSequence")
@SequenceGenerator(name="idSequence",
    sequenceName="SEQ_DEPT", allocationSize=1, initialValue =1)
private Long idD;
...
```

- `idSequence` : identificateur de la séquence dans le programme Java
- `SEQ_DEPT` : nom de la séquence dans la base
- `allocationSize=1` : pas de la séquence
- `initialValue=1` : valeur initiale de la séquence

3. Clés primaires - b. Clés composées

- Exemple : `Personne(nom, prenom, dateNaiss, ...)`
- Annotations : soit `@EmbeddedId` (à préférer) soit `@IdClass` (déprécié, existe pour une compatibilité avec EJB 2.0.)
- Exemple

```
@Embeddable
public class IdPersonne {
   private String nom;
   private String prenom;
   ...
}
```

```
@Entity
public class Personne {
   @EmbeddedId
   private IdPersonne idP;
   ...
}
```

4. Attributs - a. Propriétés des colonnes

- Annotation `@Column` : définit les propriétés d'une colonne (nom, taille, null autorisé, etc.)

- Exemple

```
@Column(name = "NOME", nullable = false, length = 30)
private String nome;
```

4. Attributs - a. Propriétés des colonnes

- Annotation `@Transient` :
  - Indique qu'un attribut ne doit pas être sauvegardé dans la base (pas de colonne associée)
  - Ex. L'âge peut être calculé à partir d'une date de naissance. On peut l'avoir comme attribut dans la classe, mais pas comme colonne dans la table.

- Annotation `@Basic` :
  - Indique (essentiellement) si la récupération de la valeur d'un attribut est `EAGER` (valeur par défaut) ou `LAZY`
  - Exemple : colonne dont la valeur est volumineuse

```
@Basic(fetch = FetchType.Lazy)
@Lob
private byte [] cvPDF;
```

  - Annotation `@Lob` : indique que la valeur doit être stockée dans une colonne de type LOB (*Large Object*). Ce type nécessite des appels JDBC spéciaux et il faut en informer le fournisseur de persistance avec l'annotation `@Lob`.

4. Attributs - b. Types d'accès

- Annotation des attributs : s'applique alternativement
  - soit à l'attribut (**accès par champ**)
  - soit au `getter` correspondant (**accès par propriété**)

- Exemple accès par propriété

```
...
private Long idD;
...
@iD @GeneratedValue
getIdD(){...}
```

- Si accès par champ, le fournisseur de persistance accède à l'état persistant via les champs

- Si accès par propriété, le fournisseur accède à l'état par les `getters`

- Dans un accès par propriété, il est possible d'inclure dans les getters des traitements supplémentaires que le fournisseur exécute à chaque appel

4. Attributs - b. Types d'accès

- Possibilité d'indiquer un type d'accès par défaut à tous les attributs :
- Il suffit alors d'annoter la classe par l'une des annotations suivantes : `@Access(AccessType.FIELD)` ou `@Access(AccessType.PROPERTY)`
- **N.B.** tous les attributs d'une hiérarchie de classes doivent avoir le même type d'accès, le mélange peut conduire à des erreurs
- Dans le cas général, préférer l'accès par champ pour distinguer les accès par JPA aux accès par l'application (via les accesseurs)
- Pour l'exécution du code métier, préférer les méthodes de rappel plutôt que son inclusion dans un getter

5. Objets intégrés - a. Principe

- Les instances d'une classe annotée `Embeddable` sont persistants mais n'ont pas d'identité propre dans la BD

- Exemple

```
@Embeddable
public class Adresse{
    private int numero;
    private String rue;
    private String ville;
    ...
    public String getVille()
    ...
}
```

```
@Entity
public class Emp{
...
    @Embedded
    private Adresse adresse;
...
}
```

- Les attributs d'une instance `Embeddable` sont stockées dans la table associée à l'objet contenant

- Éviter de mettre un type d'accès différent de celui de la classe intégrante

5. Objets intégrés - b. Référencement multiple

- Une même classe peut référencer plusieurs instances d'une même classe `Embeddable`
  Exemple : Adresse domicile et Adresse vacances

- Dans un tel cas il est nécessaire d'utiliser l'annotation `@AttributeOverrides` pour associer les attributs d'une adresse aux colonnes correspondantes de la table.

Chapitre 5 - Java Persistence API - Partie 1 : ORM

1. Généralités

- Une association est indiquée par une annotation sur les attributs permettant de faire le lien
- Elle peut être de type 1-1, 1-N, N-1, M-N ou M-N porteuse d'informations
- Peut être uni-directionnelle (représentée dans un seul bout) ou bi-directionnelle (représentée dans les 2 bouts)
- Si elle est bi-directionnelle, elle a un **bout propriétaire** et un **bout cible** (nous y reviendrons)

2. Association N-1 unidirectionnelle

Association N-1 unidirectionnelle

- N-1 : Ex. plusieurs employés peuvent être associés à un seul département
- Unidirectionelle : Association représentée uniquement dans un des bouts de l'association, l'entité `Emp` en occurrence.
  Ex. A partir d'un employé on peut atteindre son département, mais à partir d'un département on ne pas atteindre ses employés
- Se traduit en Objet par attribut dans `Emp` annoté par `@ManyToOne` et référençant un département
- Exemple

```
@Entity
public class Emp{
    @Id
    private Long idE;
     ...
    @ManyToOne
    private Dept dept;
     ...
}
```

2. Association N-1 unidirectionnelle

Association N-1 : mode de récupération de l'objet référencé

- Par défaut le chargement de l'objets référencé est `EAGER` (immédiat).
- Exemple : le chargement d'un employé, implique celui immédiat de l'objet département correspondant
- Modification du mode de récupération par défaut : `@ManyToOne(fetch = FetchType.LAZY)`
- Exemple : dans la Classe Emp

```
@ManytoOne(fetch = FetchType.LAZY)
private Dept dept;
```

2. Association N-1 unidirectionnelle

Association N-1 : schéma relationnel

- Si le schéma de la base est créé à partir des classes entités, l'association se traduit par la création d'une clé étrangère référençant une clé primaire
- Nom de l'attribut clé étrangère : par défaut nom de l'entité référencée (Ex. Dept), concaténé à "_", concaténé au nom de l'attribut d'identification dans l'entité référencée (Ex. IdD).
- Dans l'exemple : `Dept_IdD`
- Annotation `@JoinColumn` : permet de modifier le nom par défaut
- Exemple : dans la Classe Emp

```
@ManytoOne(fetch = FetchType.LAZY)
@JoinColumn(name="RefDept", nullable = false)
private Dept dept;
```

3. Association 1-1 unidirectionnelle

Association 1-1 unidirectionnelle

- Exemple : une place de parking appartient à un seul employé et un employé ne peut avoir au plus qu'une seule place.
- Unidirectionnelle : à partir d'une place de parking on peut trouver l'employé la possédant, mais nous n'avons pas besoin de chercher une place à partir d'un employé.
- Se traduit en Objet par un attribut dans Parc annoté par `@OneToOne` et référençant un employé
- Se traduit en relationnel (dans la base) par une clé étrangère dans la table Parking
- Mode de récupération et nom de l'attribut clé étrangère : mêmes règles que pour une association `@ManyToOne`

3. Association 1-1 unidirectionnelle

Association 1-1 sur les clés

- Association 1-1 sur les clés : deux entités de types différents identifiées par la même valeur de clé
- Exemple : entité `Park` identifiée par l'identifiant de l'employé possédant cette place de parking
- Depuis JPA 2.0 représentée par les **identités dérivées**
- Exemple (dans `Parking`) :

```
...
@Id @OneToOne
private Emp emp;
...
```

4. Association 1-N unidirectionnelle

Association 1-N unidirectionnelle

- 1-N : se traduit par attribut de type Collection, annoté par `@OneToMany` et référençant une collection d'entités cibles
  Ex. un département référençant les employés qui y travaillent.

- Unidirectionnelle : représentée dans un seul bout (Ex. uniquement dans `Dept`).

- Type de la collection choisi dans l'un des types (`java.lang.util`) suivants
  - `Collection` : ensemble non-trié acceptant les doublons (Type le plus utilisé).
  - `Set` : ensemble non-trié n'acceptant pas les doublons.
  - `List` : ensemble trié acceptant les doublons. Certaines précautions doivent être prises (nous y reviendrons)
  - `Map` : ensemble de clé-valeur permettant un accès rapide aux entités selon la valeur de clé

4. Association 1-N unidirectionnelle

Association 1-N unidirectionnelle

- Le type déclaré ne doit pas être concret (`ArrayList`/`LinkedList` pour l'interface `List`, `HashSet`/`TreeSet` pour `Set`, `HashMap`/`TreeMap` pour `Map`)
- Ceci laisse la liberté au fournisseur de persistance pour utiliser son propre type concret
- Exemple

```
@Entity
public class Dept {
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator="idSequence")
    @SequenceGenerator(name="idSequence",
        sequenceName="SEQ_DEPT", allocationSize=1, initialValue =1)
    private Long idD;
    private Sting nomD;
    @OneToMany
    private List<Emp> Employes;
    ...
}
```

4. Association 1-N unidirectionnelle

Association 1-N unidirectionnelle : schéma relationnel

- Mode de récupération par défaut (uni ou bi-directionnelle) : `LAZY` (à la demande). Peut être modifié.
- Schéma relationnel
  - Table de jointure, associant chaque département l'ensemble de ses employés
  - Nom par défaut de la table : Nom de l'entité, concaténé à "_", concaténé au nom de la classe des entités référencées
    Exemple : `DEPT_EMP`
  - La clé primaire de la table est composée par l'identificateur des deux classes entités aux bouts de l'association
    Exemple : `DEPT_EMP(DEPT_IDD, EMP_IDE)`
  - Annotation `@JoinTable` : à utiliser si les valeurs par défaut ne conviennent pas

4. Association 1-N unidirectionnelle

Association 1-N unidirectionnelle : schéma relationnel

- Schéma relationnel
  - Exemple :

    Classe Dept

```
...
@OneToMany
@JoinTable(
    name = "DEPT_EMP"
    joinColumns=@JoinColumn(name="refDept")
    inverseJoinColumns=@JoinColumn(name="refEmp")
)
private List<Emp> Employes;
...
```

4. Association 1-N unidirectionnelle

Association 1-N unidirectionnelle : schéma relationnel

- Schéma relationnel
  - Il est possible d'obliger JPA à traduire l'association par une clé étrangère plutôt que par table
  - Il suffit pour cela d'utiliser l'annotation `@JoinColumn`

```
...
@OneToMany
@JoinColumn(name = "refDept")
private List<Emp> Employes;
...
```

5. Association M-N unidirectionnelle

Association M-N unidirectionnelle

- Exemple : à un projet correspond plusieurs employés qui y participent
- Se traduit par attribut dans Projet de type Collection, annoté par `@ManyToMany` et référençant une collection d'employés
- Mode de récupération et traduction en relationnel : mêmes règles que pour les associations `@OneToMany`
- Exemple (classe projet) :

```
...
@ManyToMany
private Collection <Emp> employes;
...
```

6. Annotation des associations unidirectionnelles

- Annotations `@OneToOne`, `@ManyToOne`, `@OneToMany` et `@ManyToMany` :
  - Optionnelles si l'association est unidirectionnelle
  - Une référence à une entité est considérée par défaut comme une association 1
  - Les références à une collection d'entités sont considérées par défaut comme une association N
  - Annotations : s'utilisent si les conventions par défaut ne sont pas satisfaisantes

7. Association bi-directionnelle

- Association bi-directionnelle : représentée dans les deux classes (les 2 bouts) participant à l'association
- Lorsqu'une association est bi-directionnelle, la même information est représentée deux fois (dans les 2 classes). Il s'agit en quelque sorte de deux associations unidirectionnelles
- Exemple : l'employé x travaille au département y. Cette information est représentée par l'attribut `dept` de l'entité x et dans la collection `employes` de l'entité y.
- Le développeur est responsable de la gestion des 2 bouts de l'association
- Ex. si une employé change de département, les collections des employés des 2 départements doivent être modifiées en conséquence

7. Association bi-directionnelle

- JPA distingue un **bout propriétaire** (*owner side*) et un **bout cible** (*inverse side*)
- Les modifications concernant l'association faites dans le **bout propriétaire** sont celles **enregistrées dans la base**

  Exemple : le changement de l'attribut dept d'un employé est répercuté dans la base
- Les modifications concernant l'association faites dans le **bout non-propriétaire** (bout cible) sont **ignorées** lors de l'enregistrement dans la base

  Exemple : l'ajout d'un employé à la collection d'un département n'est pas répercuté dans la base

7. Association bi-directionnelle

Associations bi-directionnelles autres que les associations N-M

- Traduites dans la base par le mécanisme de clé primaire/clé étrangère
- **Bout propriétaire** : entité dont la table correspondante accueille la clé étrangère. Exemple : association Emp/Dept le bout propriétaire est Emp
- Le bout cible (non-propriétaire) devrait être qualifié par l'annotation `@OneToXXX (mappedBy = "nomAttribut")`
- `nomAttribut` : nom de l'attribut dans le bout propriétaire qui représente la même association (attribut propriétaire de l'association)
- La modification du nom par défaut donné à l'attribut faisant la jointure, se fait obligatoirement au niveau de la classe propriétaire

7. Association bi-directionnelle

Associations bi-directionnelles autres que les associations N-M

- Exemple :

  Dans la classe `Emp`

```
@ManyToOne
private Dept dept;
```

  Dans la classe `Dept`

```
@OneToMany(mappedBy="dept")
private Collection <Emp> employes;
```

- Quel bout est propriétaire?

7. Association bi-directionnelle

Association N-M bi-directionnelle non porteuse d'informations

- Le choix du bout propriétaire est laissé au programmeur (il doit **obligatoirement** en choisir un)
- Exemple :

```
@ManyToMany
private Collection <Projet> projets;
```

```
@ManyToMany (mappedBy = "projets")
private Collection <Emp> employes;
```

- Utilisation de l'annotation `@joinTable` dans le bout propriétaire si les valeurs par défaut (noms de la table faisant la jointure et des clés étrangères) ne conviennent pas

7. Association bi-directionnelle

Association N-M bi-directionnelle non porteuse d'informations

- Exemple :

  Classe Emp (propriétaire)

```
@ManyToMany
@JoinTable(
   name = "EMP_PROJET"
   joinColumns=@JoinColumn(name="refEmp")
   inverseJoinColumns=@JoinColumn(name="refProj")
)
private Collection<Projet> projets;
...
```

  Classe Projet (Cible)

```
@ManyToMany(mappedBy="projets")
private Collection<Emp> employes;
...
```

8. Association M-N bi-directionnelle porteuse d'informations

Association N-M bi-directionnelle porteuse d'informations

- Deux cas sont distingués :
  - L'association dispose de son propre identificateur (cas le plus simple, à privilégier)
  - L'association ne dispose pas de son propre identificateur : l'identificateur est la concaténation des identificateurs des classes participant l'association.

8. Association N-M porteuse d'informations

Association avec identificateur

- Exemple : une classe association `LigneFacture` avec identificateur

```
@Entity
public class LigneFacture {
   @Id
   private int id;
   @ManyToOne
   private Facture facture;
   @ManyToOne
   private Produit produit;
   private int qtte;
...
}
```

- Bout(s) propriétaire(s)?

8. Association N-M porteuse d'informations

Association avec identificateur

- Si le schéma de la base est généré à partir des classes entités, il est possible d'indiquer une contrainte d'unicité sur (`PRODUIT, FACTURE`)

- Contrainte d'unicité : indique dans l'exemple que la paire (IdP, Id) ne peut apparaître qu'une seule fois dans la table (un même produit ne peut pas apparaître deux fois dans une même facture)

```
@Entity
@Table(uniqueConstraints=@UniqueConstraint(
                columnNames{"PRODUIT","FACTURE"})
private class LigneFacture {...}
```

8. Association N-M porteuse d'informations

Association sans identificateur

- Solution à éviter, non couverte par les spécifications (chaque fournisseur implante sa propre stratégie)
- Ne doit être choisie que si la base est préexistante et que la table association a pour clé primaire la concaténation de deux clés étrangères (Ex. `PRODUIT_ID`, `FACTURE_ID`)
- On peut utiliser une classe `Embedded` pour représenter la clé composée (vu précédemment dans la section *II.2.a. Clés composées* - diapositive 183) en faisant attention à ce que la clé soit composée de types primitifs et non d'entités.
- On peut alternativement traiter l'association comme étant deux associations N-1, l'une vers produit et l'autre vers facture.

9. Gestion des bouts d'une association bi-directionnelle

- Propagation des modifications faites dans un bout sur l'autre bout
  - à la charge du programmeur
  - Il est commode d'écrire une méthode qui automatise la répercussion
  - Exemple : dans la classe `Dept` (le bout cible)

```
public void ajouterEmploye(Emp e) {
    Dept d = e.getDept();
    if (d != null){d.employes.remove(e);}
    this.employes.add(e);
    e.setDept(this);
}
```

10. Récupération des entités associées

- JPA permet une récupération immédiate (**EAGER**) ou à la demande (**LAZY**) des entités référencées par une entité gérée. Le choix est laissé au programmeur
- Comportement par défaut

Association	Stratégie	Exemple
`@OneToOne`	**EAGER**	Le chargement d'une place de parking implique celui immédiat de l'employé
`@ManyToOne`	**EAGER**	Le chargement d'un employé implique celui immédiat de son département
`@OneToMany`	**LAZY**	Le chargement d'un département n'implique pas le chargement de ses employés
`@ManyToMany`	**LAZY**	Le chargement d'un projet n'implique pas celui des employés qui y participent

- Il est possible de modifier le comportement par défaut

```
@OneToMany(mappedBy="dept",fetch=FetchType.EAGER)
private Collection<Emp> employes;
```

Chapitre 5 - Java Persistence API - Partie 1 : ORM

1. Stratégies

- Stratégies de traduction couverte par JPA
  - Une seule table pour toute la hiérarchie (`SINGLE_TABLE`)
  - Une table par classe avec jointure pour reconstituer les entités (`JOINED`)
- La stratégie "une table par classe concrète" est optionnelle (`TABLE_PER_CLASSE`) dans la spécification

2. Une table par hiérarchie

- Stratégie par défaut et également la plus utilisée
- Exemple :

  Classe à la racine de la hiérarchie

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Personne {...}
```

  Classe héritant de la classe racine

```
@Entity
@DiscriminatorValue("E")
public class Emp extends Personne {...}
```

- Le nom de la table est celui de la classe racine

2. Une table par hiérarchie

- Colonne discriminante
  - Permet de différencier les lignes des différentes classes de la hiérarchie
  - Indispensable au chargement des tuples dans les bons objets et au bon fonctionnement des requêtes
  - Par défaut la colonne s'appelle `DTYPE` et elle est de type `Discriminator.STRING`
  - L'annotation `@DiscriminatorValue` mise dans chacune des classes permet de spécifier la valuer prise par cette colonne pour ce type d'objets
  - Exemple : table personne

IdPersonne	**DTYPE**	**Nom**	...
1	E	Foulen	...
2	C	Foulena	...
...	...	...	...

3. Une table par classe

Rappel :

- Toutes les classes, concrètes et abstraites, sont représentées par une table
- Une même entité garde la même valeur de la clé primaire à tous les niveaux de la hiérarchie
- Nécessite des jointures pour retrouver les propriétés d'une entité
- Il est possible d'ajouter une colonne discriminatrice dans la table correspondant à la classe racine de la hiérarchie

3. Une table par classe

- Exemple

Classe à la racine de la hiérarchie

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Personne {...}
```

Classe héritant de la classe racine

```
@Entity
public class Emp extends Personne {...}
```

Chapitre 5 - Java Persistence API - Partie 1 : ORM

Configuration du mapping avec XML

- Il est possible de configurer la mapping Objet/Relationnel par un fichier XML
- Toutes les annotations vues dans ce chapitre ont un marqueur XML équivalent et vice versa
- Si utilisés ensemble, le fichier XML a une priorité plus haute que les annotations
- Dans le cas général, les annotations sont à préférer aux fichiers XML, car plus simple d'utilisation pour un même résultat

Chapitre 6 - Java Persistence API - Partie 2 : Gestion des objets persistants

Rappel

- **JPA (Java persistence API)** : couvre essentiellement 2 aspects
  - ORM : persistance automatisée d'objets entité dans une BD relationnelle
  - Gestion des objets entités : manipulation, interrogation, accès concurrents, etc.

- **Gestionnaire d'entités** `EntityManager`
  - Gestion des objets entités de manière simple : manipulation des entités sans se soucier du modèle relationnel sous-jacent et sans appels (SQL) de bas niveau

    Ex. persist(e), remove(e), findById(), etc...
  - Peut être considéré comme une classe DAO générique
  - Possibilité de déléguer les services techniques ($e.g.$ gestion des transactions) au conteneur

- **Contexte de persistance**
  - Objets entités gérés par un `EntityManager` à un instant donné
  - Il ne peut exister 2 objets entités avec une même valeur de l'attribut identificateur dans le contexte

Rappel

- **Objet entité**
  - **Géré** : associé à un contexte de persistance (référencé dans le contexte de persistance)
  - **Détaché** : existe en mémoire (comme simple objet POJO Java) mais non référencé dans un contexte de persistance
- **Entité gérée**
  - état automatiquement sauvegardé dans la base au moment du commit de la transaction
  - Commit : par le programme ou par le conteneur.

Rappel

- **Fournisseur de persistance**
  - Implante les les classes et les interfaces de JPA et notamment l'interface `EntityManager`
  - Traduit les appels de haut niveau en ordres SQL de bas niveau
- **Unité de persistance** : configuration des objets `EntityManager`
  - Informations sur le fournisseur de persistance, le pilote JBDC, les entités gérées, paramètres de connexion à la base, etc.
  - Informations stockées dans le descripteur de déploiement `persistence.xml`
- **Obtention d'un gestionnaire d'entités**
  - À partir d'une fabrique prenant en paramètre une unité de persistance
  - Application JSE : la fabrique est créée par le programmeur
  - Application JEE : par injection de dépendance. Le programmeur fournit une unité de persistance et le conteneur crée et instancie la fabrique de manière transparente. Le conteneur maintient alors un **pool de gestionnaires d'entités**. Généralement une seule fabrique est créée pour une unité de persistance.

Rappel

Exemple (application Java SE)

```
   // Création d'une entité Emp.  emp est à ce stade détachée,
                //il s'agit d'un simple POJO
Emp emp = new Emp(123L,"Foulen",...);
   // Création d'un gestionnaire d'entités et d'une transaction.
EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("rhPersistenceUnit");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
   // Stockage de l'employé dans la base de données.
tx.begin();
em.persist(emp);
   // emp est gérée.  Son état est sauvegardé au prochain commit.
tx.commit();
   // Récupération de l'employé par son identifiant.
emp = em.find(Emp.class, 123L);
System.out.println(emp.toString());
em.close();
emf.close();
```

Rappel

Exemple équivalent en utilisant un EJB sans état (déployé côté serveur)[4]

```
@Stateless
public class EmpFacade {
@PersistenceContext(unitName = "rhPersistenceUnit")
private EntityManager em;
...
public void createEmp(Long idE, String nomE, ...)  {
     //Création d'une instance de Emp.
  Emp emp = new Emp(idE, nomE, ...);
    //Stockage de l'instance dans la base de données.
  em.persist(emp);
    // Récupération d'une instance par son identifiant.
  emp = em.find(Emp.class, 123L);
  ...}
}
```

[4] la création d'une fabrique de gestionnaires d'entités ainsi que la gestion des transactions sont délégués au conteneur JEE. Ici nous obtenons par injection de dépendance une référence vers un gestionnaire d'entités. Le conteneur gère un pool de gestionnaires d'entités.

Chapitre 6 - Java Persistence API - Partie 2 : Gestion des objets persistants

1. Méthodes de l'interface `EntityManager`

Principales méthodes de l'interface `EntityManager` liées à la gestion des entités

Méthode	Description
`void persist(Object entity)`	Crée une instance gérée et persistante
`<T> T find(Class<T> class, Object primaryKey)`	Recherche par clé primaire d'une entité. L'objet retourné est rattaché au C.P.
`<T> T getReference(Class<T> class, Object primaryKey)`	Obtient une référence à une entité dont l'état peut être récupéré à la demande. L'objet retourné est rattaché au C.P.
`void remove(Object entity)`	Supprime l'entité du C.P. et de la BD. L'entité reste en mémoire et se manipule comme un objet ordinaire.
`<T> T merge(Object entity)`	Rattache l'entité au C.P.
`void refresh(Object entity)`	Rafraîchit l'état de l'entité à partir de la BD. Écrase les éventuelles modifs apportées à l'entité
`void flush()`	Enregistre les entités référencées dans le C.P. dans la BD
`void clear()`	Vide le C.P. Toutes les entités deviennent détachées
`void clear(Object entity)`	Supprime l'entité du C.P. (mais pas de la BD)
`boolean contains(Object entity)`	Teste si l'entity est actuellement gérée (référencée) dans le C.P.

Tableau repris de [*Java EE 6 et Glassfish*. A. Goncalves. Pearson Education. 2010]

2. Rendre une entité persistante

- Exemple :

```
Emp emp = new emp (123L, "Foulen", ...);
Dept dept = new dept ("Info");

emp.setDept(dept);

tx.begin();
em.persist(emp);
em.persist(dept);
tx.commit();

assertNotNull(dept.getId());
em.refresh(dept);
```

- em et tx sont resp. de type EntiyManager et EntityTransaction
- Identificateur de dept : généré automatiquement grâce aux annotations (soit par une séquence soit par le fournisseur de persistance)

2. Rendre une entité persistante

- `persist(E)`
  - rend une entité gérée, *i.e.* référencée dans le contexte de persistance
  - mais l'entité n'est **pas encore écrite dans la BD**
- Les entités ne sont écrites dans la base que lorsque la transaction est validée (`tx.commit()`)
- Le gestionnaire d'entités (`em` dans l'exemple)
  - réordonne les ordres d'écriture pour ne pas violer l'intégrité référentielle.

  $\Rightarrow$ `dept` est écrite avant `emp`.
  - En coulisse, le fournisseur de persistance traduit les insertions dans la base par des ordres SQL.
- `assertNotNull(dept.getId())` (optionnel) : teste si l'entité a bien été écrite (a bien reçu un identifiant)
- `refresh(dept)` : re-chargement de l'entité `dept` à partir de la base (remplit entre autres la liste de ses employés)

3. Suppression d'une entité

- `remove(E)`
  - Supprime une entité du contexte de persistance
  - La suppression de la base se fait au moment de la validation de la transaction ou suite à l'exécution de `flush()`
  - Ne peut être utilisée que dans le contexte d'une transaction
  - L'entité supprimée devient un simple POJO et peut être utilisée en tant que telle
  - Exemple

```
tx.begin();
em.remove(emp);
tx.commit();
System.out.println("Employé supprimé " + emp.getNome());
```

3. Suppression des orphelins

- Orphelin : tuple qui n'est référencé par aucun autre
- Exemple :
  - Un fournisseur f est référencé par un seul produit p.
  - Si p est supprimé, f n'est plus atteignable
  - f est dit **orphelin**
- JPA permet d'automatiser la suppression des orphelins avec la propriété `orphanRemoval` d'une association
- La suppression de la base se fait au moment de la validation de la transaction ou suite à l'exécution de `flush()`

3. Suppression des orphelins

- Exemple :

```
@Entity
public class produit {
   @Id
   private Long id;
   ...
   @OneToOne (fetch = FetchType.LAZY, orphanRemoval = true)
   private Fournisseur fournisseur;
   ...
}
```

4. Recherche

- `find(E.Class, identificateur de E)`
  - Prend deux paramètres : la classe de l'entité recherchée et son identifiant
  - L'appel renvoie `null` ou l'entité recherchée qui **devient automatiquement gérée**
  - Exemple :

```
Emp emp = em.find(Emp.class, 1234L);
if (emp!= null) {
   ...
}
```

4. Recherche

- `getReference(E.Class, identificateur de E)`
  - Prend les mêmes paramètres que `find()`
  - Permet de récupérer une référence d'entité à partir de sa clé primaire, sans que cette entité ne soit nécessairement initialisée
  - Les **attributs** de l'entité, autres que la clé primaire, sont **récupérés à la demande**.
  - `getReference` peut être utilisée pour améliorer les performances quand une entité non initialisée peut être suffisante (sans que l'entité entière soit retrouvée dans la base de données)
  - Dans le cas général, privilégier `find`

4. Recherche

- Toutes les entités retrouvées par `find`, `getReference` ou un `query` sont automatiquement gérées par le gestionnaire d'entités
- Les modifications apportées à l'entité sont donc enregistrées au prochain `commit`
- Cependant les objets référencés par l'entité ne sont pas automatiquement persistées [5]

[5] même s'il y a une cascade sur persist ; ça ne marche que pour un appel explicite de la méthode `persist`

5. Synchronisation avec la base de données

- `flush()` :
  - Permet d'enregistrer les entités référencées dans un contexte de persistance sans attendre la validation de la transaction.
  - En coulisse, un `commit()` exécute un `flush()`
  - En cas de `rollback()` les modifications faites par `flush()` sont annulées.
  - Exemple :

```
tx.begin();
em.persist(emp);
em.flush();
em.persist(dept);
tx.commit();
```

⇒ le code précédent génère une erreur, pourquoi?

- `refresh(E)` :
  - Recharge l'état d'une entité référencée dans le contexte de persistance à partir de la base

6. Contenu du contexte de persistance

- `contains(E)` :
  - Prend en paramètre une (référence d'une) entité
  - Renvoie un booléen indiquant si l'entité passée en paramètre est actuellement gérée dans le contexte de persistance
- `clear()` :
  - vide le contexte de persistance
  - toutes les entités gérées, sont détachées (**sans enregistrement dans la base**)
- `detach(E)` :
  - Supprime l'entité du contexte de persistance
  - Les modifications sur l'objet ne sont pas enregistrées dans la base aux prochains commit

7. Fusion d'une entité

- Une entité détachée (ne faisant plus ou pas partie du contexte de persistance) peut être modifiée

- `merge(E)` : permet de **rattacher une entité au contexte de persistance**. Les modifications apportées en dehors du contexte, sont enregistrées au prochain commit.

- `merge(E)` :
  - Si E est une entité nouvelle ou détachée, son état est copié dans une entité gérée E' qui a la même identité que E. merge renvoie E'
  - Si E est déjà gérée, `merge` renvoie E

8. Propagation d'opérations

- Par défaut, chaque opération du gestionnaire d'entités ne s'applique qu'à l'entité passée en paramètre à l'opération
- Des fois il peut s'avérer utile de propager automatiquement l'action faite par l'opération à d'autres entités
- Exemple : rendre persistant une entité Employé, rend automatiquement persistante l'entité place parking référencée par l'employé
- Option `cascade` d'une association : permet d'indiquer s'il est souhaité ou non de propager l'action d'une opération

8. Propagation d'opérations

- Exemple :

```
@Entity
public class Emp {
   ...
   @OneToOne (fetch = FetchType.LAZY,
              cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
   private Parking parking;
   ...
}
```

```
...
Emp emp = new Emp(4561,"Foulen",...);
Parking placePkg = new Parking("Couverte");
emp.setParking(placePkg);
tx.begin();
em.persist(emp);
tx.commit();
//persist est propagée à l'entité placePkg
...
}
```

8. Propagation d'opérations

- Évènements pouvant être propagés

Type	Description
`PERSIST`	Propage les opérations `persist` à la cible de l'association
`REMOVE`	Propage les opérations `remove` à la cible de l'association
`MERGE`	Propage les opérations `merge` à la cible de l'association
`REFRESH`	Propage les opérations `refresh` à la cible de l'association
`CLEAR`	Propage les opérations `clear` à la cible de l'association
`ALL`	Propage toutes les opérations précédantes

Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

Chapitre 6 - Java Persistence API - Partie 2 : Gestion des objets persistants

1. Présentation

- `find` et `getReference` : recherche par identifiant.
- JPA permet de formuler des recherches plus complexes avec deux langages JPQL (*Java Persistence Query Language*) et l'API Criteria
- Requêtes JPQL et Criteria : **manipulent le modèle objet et non le modèle Relationnel**
- JPQL/API Criteria vs. requêtes SQL natives :
  - Code portable
  - Le résultat d'une requête est converti automatiquement en entités gérées.
- JPA utilise 2 interfaces pour définir les requêtes :
  - `Query` (depuis JPA 1.0)
  - `TypedQuery` (à partir de JPA 2.0)

1. Présentation

- Query
  - Type de retour de la requête non connu ou non spécifié
  - Exemple (JPQL) :

```
String s = "SELECT e FROM Emp e WHERE e.nome = 'Foulen'";
Query q1 = em.createQuery(s);
Emp e = (Emp) q1.getSingleResult();
```

- TypedQuery
  - Type de retour de la requête connu
  - Exemple (JPQL) :

```
String s = "SELECT e FROM Emp e WHERE e.nome = 'Foulen'";
TypedQuery<Emp> q2 = em.createQuery(s, Emp.class);
Emp emp = q2.getSingleResult();
```

2. JPQL - a. Syntaxe

- Syntaxe assez proche de SQL, avec comme principales différences
  1. l'utilisation obligatoire des alias pour les entités interrogées

     Exemple : `SELECT e FROM Emp e`
  2. possibilité d'utiliser la notation pointée propre au modèle objet (navigation d'un objet à un autre)

     Exemple : `SELECT e FROM Emp e WHERE e.dept.nomd = "INFO"`
  3. Une requête JPQL peut retourner une entité ou une collection d'entités

- En coulisse,
  - Une requête JPQL est traduite en une requête SQL et en des appels JDBC; puis
  - Les instances des objets sont initialisées et sont renvoyées à l'application

2. JPQL - a. Syntaxe

- Syntaxe de l'ordre SELECT

```
SELECT <expression select>
 FROM <clause from>
 [WHERE <expression conditionnelle>]
 [ORDER BY <clause order by>]
 [GROUP BY <clause group by>]
 [HAVING <clause having>]
```

- On peut également formuler un `UPDATE` ou un `DELETE`

2. JPQL - a. Syntaxe

- Exemples de requêtes JPQL
  - `select e.nome, e.salaire from Emp e`
  - ```
    select e.nome, d.nomd
    from Emp e, Dept d
    where d = e.dept AND d.nomd='Info'
    ```
  - ```
    select e.nome, count(p)
    from Emp e join e.projets p
    group by e
    having count(p) > 2
    ```
  - ```
    select e
    from Emp e
    where e.salaire
      >= ALL (select e2.salaire from Emp e2 where e2.dept = e.dept)
    ```
    ```

2. JPQL - b. Requêtes paramétrées

- **Requêtes paramétrées**
  - Paramètre : identifié soit par sa position ($?i$) ou par un nom ($:nomParam$)
  - `setParameter()` : donne une valeur à un paramètre
  - Exemple :

```
String jpql = "SELECT e FROM Emp e WHERE e.dept.nomd= ?1";
q = em.createQuery(jpql);
query.setParameter(1, "INFO");
List<Emp> emps = (List<Emp>) q.getResultList();
```

  - Alternative :

```
String jpql = "SELECT e FROM Emp e WHERE e.dept.nomd=:nomDept";
q = em.createQuery(jpql);
q.setParameter("nomDept", "INFO");
List<Emp> emps = (List<Emp>) q.getResultList();
```

2. JPQL - c. Obtention d'une requête

- `EntityManager` fournit différentes méthodes pour déclarer une requête `Query`

Méthode	Description
`createQuery(String jpql)`	requête JPQL
`createQuery(CriteriaQuery criteria)`	requête Criteria
`createNativeQuery(String sql)`	requête native SQL
`createNamedQuery(String nom)`	requête nommée

- `Query` dispose de plusieurs méthodes pour exécuter une requête (liste non exhaustive)

Méthode	Description
`Object getSingleResult()`	renvoie un seul objet ou une seule valeur
`List getResultList()`	renvoie plusieurs lignes (objets ou valeurs)
`setMaxResult(int nb)`	renvoie des nb premiers résultats
`int executeUpdate()`	exécute un update/delete et renvoie le nombre d'entités modifiées/supprimées

- Exemple :

```
String s = "select e from Emp e";
Query q = em.createQuery(s);
List<Emp> listeEmployes = (List<Emp>) q.getResultList();
```

2. JPQL - d. Requêtes nommées

- **Requêtes nommées**
  - Statiques et non modifiables
  - Traduites en SQL au démarrage de l'application : **consomment moins de temps que les autres types de requêtes**
  - Déclarées à l'aide d'annotations `@NamedQuery=(name="...", query="...")` placées dans les classes entités

```
@Entity
@NamedQueries({
  @NamedQuery(name="Emp.findAll", query="SELECT e FROM Emp e"),
  @NamedQuery(name="Emp.findWithParam", query="SELECT e FROM Emp e
           WHERE e.dept.nomd=:nd")
})
public class Emp implements Serializable {...}
```

```
Query q = em.createNamedQuery("Emp.findWithParam");
q.setParameter("nd", "INFO");
q.setMaxResults(3);
List<Emp> emps = (List<Emp>) q.getResultList();
```

2. JPQL - e. Requêtes natives

- Requêtes natives :
  - requêtes écrites dans le langage SQL de la base
  - permet de bénéficier des fonctionnalités spécifiques d'un SGBD, mais le code n'est pas portable (à éviter)
  - Avantage du passage par JPA au lieu des appels directs JDBC : résultat de la requête automatiquement converti en entité(s)

- Exemple :

```
TypedQuery<Emp> q =
          em.createNativeQuery("SELECT * FROM Emp",Emp.class);
List<Emp> emps = q.getResultList();
```

3. API Criteria

- API introduite par JPA 2.0 (`javax.persistence.CriteriaQuery`)
- Tout ce qui peut être fait avec JPQL peut l'être avec l'API "critères"
- Les requêtes JPQL sont des `String` qui peuvent contenir des erreurs (par exemple le nom d'une classe qui n'existe pas)
- L'avantage de l'API criteria est que les requêtes peuvent être vérifiées à la compilation
- L'inconvénient est que le code est un peu plus complexe à écrire, et sans doute moins lisible
- Requête critère :
  - Les différentes clauses de la requête (`SELECT`, `FROM`, `WHERE`, etc.) sont définies dynamiquement.
  - Peut nécessiter le recours au **méta-modèle** des entités.

3. API Criteria

Construction d'une requête criteria

- Deux principaux objets à manipuler : `CriteriaBuilder` et `CriteriaQuery`.
- Objet `CriteriaQuery` :
  - Permet de construire dynamiquement les différents éléments d'une requête (select, from, where, group by, etc.)
- `CriteriaBuilder` :
  - Fabrique de `CriteriaQuery` pour une unité de persistance donnée (et donc un méta-modèle donné).
  - S'obtient auprès de d'un `EntityManager` ou d'un `EntityManagerFactory`.
- Exemple :

```
CriteriaBuilder cqFabrique = em.getCriteriaBuilder();
CriteriaQuery cqDefinition = cqFabrique.createQuery();
```

3. API Criteria

Construction d'une requête criteria

- Éléments à définir d'une requête `CriteriaQuery`

Élément	Description	Création
`Select`	Les objets ou les valeurs à retourner	`CriteriaQuery.select`
`From`	Les entités interrogées et éventuellement les jointures les liants	`CriteriaQuery.from`
`Where`	Conditions ou **prédicats** sur les entités interrogées. Il est possible de créer des objets prédicats et de les ajouter à une requête	`CriteriaQuery.where`, `CriteriaBuilder.equals`, `CriteriaBuilder.lessThanOEquals`, etc.
`Order By`	Tri des résulats	`CriteriaQuery.orderBy`
`Group By`	Calcul d'agrégats groupé selon un attribut	`CriteriaQuery.groupBy`

- La création des différents éléments d'une requête doit suivre un certain ordre

3. API Criteria

Construction d'une requête criteria

- Étapes de construction d'une requête criteria
  1. **Étape 1 :** création des objets `CriteriaBuilder` et `CriteriaQuery`
  2. **Étape 2 :** spécification de la clause **from**
  3. **Étape 3** : spécification de la clause **select**
  4. **Étape 4** : spécification des **prédicats**
  5. **Étape 5** : spécification de la clause **where** en utilisant les prédicats
  6. **Étape 6** : **exécution** de la requête

3. API Criteria

Construction d'une requête criteria

- Exemple simple :

```
/*étape 1*/ CriteriaBuilder cqFabrique = em.getCriteriaBuilder();
/*étape 1*/ CriteriaQuery cqDefinition = cqFabrique.createQuery();

/*étape 2*/ Root<Emp> cqRacine = cqDefinition.from(Emp.class);

/*étape 3*/ cqDefinition.select(cqRacine);

/*étape 6*/ Query requete = em.CreateQuery(cqDefinition);
/*étape 6*/ List<Emp> resultats = requete.getResultList();
```

- Étapes 2 à 5 : construction des éléments de la requête
- La requête obtenue retourne la liste de tous les employés
- `Root<T>` : indique les classes où le résultat est à chercher. Une même requête peut avoir plusieurs root. On aurait pu écrire

```
cqDefinition.select(cqDefinition.from(Emp.class))
```

3. API Criteria

Construction d'une requête criteria

- **Exemple avec un prédicat** : employés ayant un salaire > 1000

```
/*étape 2*/ Root<Emp> cqRacine = cqDefinition.from(Emp.class);

/*étape 3*/ cqDefinition.select(cqRacine);

/*étape 4*/ Predicate pGtSal = cqFabrique.gt(cqRacine.get("salaire"),1000);

/*étape 5*/ cqDefinition.where(pGtSal);
```

- Prédicat : indique quel attribut est concerné par la comparaison
- `cqRacine.get("salaire")` : permet d'accéder à la valeur de l'attribut salaire de l'objet Emp en-cours de traitement
- Un prédicat peut se ré-utiliser dans la définition de plusieurs requêtes criteria. Ça explique pourquoi il s'obtient à partir d'un objet `CriteriaBuilder` plutôt que d'un `CriteriaQuery`.
- Une requête criteria peut comporter plusieurs prédicats.

3. API Criteria

Construction d'une requête criteria

- **Exemple avec plusieurs prédicats** : employés ayant un salaire > 1000 ET une date de naissance $\leq$ "$31/12/1985$"

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
Date date = sdf.parse("31/12/1985");
...
/*étape 2*/ Root<Emp> cqRacine = cqDefinition.from(Emp.class);

/*étape 3*/ cqDefinition.select(cqRacine);

/*étape 4*/ Predicate pGtSal = cqFabrique.gt(cqRacine.get("salaire"),1000);
/*étape 4*/ Predicate pLeDN = cqFabrique.le(cqRacine.get("dNaiss"),date);
/*étape 4*/ Predicate pAND = cqFabrique.and(pLeDN,pGtSal);

/*étape 5*/ cqDefinition.where(pAND);
```

3. API Criteria

Construction d'une requête en utilisant des classes du méta-modèle

- Exemples précédents :
  - Utilisation de chaînes de caractères pour construire certains éléments de la requête(Ex. `get("salaire")`)
  - La vérification de l'exactitude des chaînes (existe t-il un attribut salaire?) est faîte à l'exécution et non à la compilation.

  ⇒ **même problème qu'avec JPQL**

- **Solution** :
  - Utilisation des classes du méta-modèle pour se référer aux attributs.
  - Exemple : `Emp_` désigne une classe du méta-modèle et `Emp_.salaire` un attribut.

3. API Criteria

- Méta-modèle d'une classe entité :
  - Décrit les attributs et les méthodes de la classe entité (chaque classe entité a une classe du méta-modèle équivalente)
  - Chaque classe du méta-modèle traduit l'ensemble des annotations et des informations sur le mapping issues du fichier de configuration XML relatives à une classe entité
  - Une classe du méta-modèle porte le nom de la classe entité correspondante suivi de " _"
    Exemple : Emp_ est la classe du méta-modèle correspondant à la classe entité Emp
  - Les classes du méta-modèle sont générées au déploiement et peuvent s'utiliser dans les requêtes criteria
  - L'ensemble des classes du méta-modèle, joue en quelque sorte le rôle de dictionnaire de données

3. API Criteria

Construction d'une requête en utilisant des classes du méta-modèle

- Exemple

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
Date date = sdf.parse("31/12/1985");
...
/*étape 2*/ Root<Emp> cqRacine = cqDefinition.from(Emp.class);

/*étape 3*/ cqDefinition.select(cqRacine);

/*étape 4*/ Predicate pGtSal = cqFabrique.gt(Emp_.salaire,1000);
/*étape 4*/ Predicate pLeDN = cqFabrique.le(Emp_.dNaiss,date);
/*étape 4*/ Predicate pAND = cqFabrique.and(pLeDN,pGtSal);

/*étape 5*/ cqDefinition.where(pAND);
```

3. API Criteria

Construction d'une requête criteria avec jointure

- Exemple : les employés du département 'INFO' ayant un salaire > 1000 ET une date de naissance $\leq$ "$31/12/1985$"

```
/*étape 2*/ Root<Emp> cqRacine = cqDefinition.from(Emp.class);
/*étape 2*/ Join<Emp,Dept> cqJointure = cqRacine.join(Emp_.dept);

/*étape 3*/ cqDefinition.select(cqRacine);

/*étape 4*/ Predicate pEqInfo =
              cqFabrique.equals(cqJointure.get(Dept_.nomd),"INFO");
/*étape 4*/ Predicate pGtSal = cqFabrique.gt(Emp_.salaire,1000);
/*étape 4*/ Predicate pLeDN = cqFabrique.le(Emp_.dNaiss,date);
/*étape 4*/ Predicate pAND = cqFabrique.and(pEqInfo,pLeDN,pGtSal);

/*étape 5*/ cqDefinition.where(pAND);
```

3. API Criteria

Construction d'une requête criteria calculant un agrégat

- Exemple : Nombre d'employés ayant un salaire > 1000 ET une date de naissance $\leq$ "$31/12/1985$"

```
/*étape 2*/ Root<Emp> cqRacine = cqDefinition.from(Emp.class);
/*étape 2*/ Join<Emp,Dept> cqJointure = cqRacine.join(Emp_.dept);
/*étape 2*/ Expression compter = cqFabrique.count(cqRacine.get(Emp_idE));

/*étape 3*/ cqDefinition.select(compter);

/*étape 4*/ Predicate pGtSal = cqFabrique.gt(Emp_.salaire,1000);
/*étape 4*/ Predicate pLeDN = cqFabrique.le(Emp_.dNaiss,date);
/*étape 4*/ Predicate pAND = cqFabrique.and(pLeDN,pGtSal);

/*étape 5*/ cqDefinition.where(pAND);

/*étape 6*/ Query requete = em.CreateQuery(cqDefinition);
/*étape 6*/ Long nbEmp = (Long) requete.getSingleResult().intValue();
```

3. API Criteria

Construction d'une requête criteria compacte

- Exemple : Nombre total des employés avec imbrication de la construction des différents éléments

```
CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
cq.select(em.getCriteriaBuilder().count(cq.from(Emp.class)));
Long nbEmps = (Long) em.createQuery(cq).getSingleResult().intValue();
```

- Pour plus d'exemples : *The Java EE 6 Tutorial* http://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html.

Chapitre 6 - Java Persistence API - Partie 2 : Gestion des objets persistants

1. Verrouillage - a. Types de verrouillage

- **Concurrence d'accès** : accès simultané à une même entité par deux transactions (chacune exécutée par exemple dans un thread différent)

- Si la concurrence d'accès n'est pas gérée, des modifications peuvent se perdre (seules les modifications faîtes par la transaction validée en dernier sont prises en compte)

- JPA :
  - Permet de **gérer la concurrence au niveau du programme Java** (et non au niveau de la base).
  - Dispose de deux types de verrouillage : **optimiste** et **pessimiste**

1. Verrouillage - a. Types de verrouillage

- **Verrouillage optimiste**
  - Mode par défaut.
  - Supposition qu'il ne peut y avoir que peu de conflits.
  - Toutes les transactions ont accès en lecture à l'entité.
  - Avant d'enregistrer une modification une transaction vérifie que l'entité n'a pas changé ($i.e.$ n'a pas reçu un nouveau numéro de version)
  - Si tel n'est pas le cas, une exception `OptimisticLockException` est levée. Le programmeur a la charge de gérer la situation
  - Performances accrues : pas de blocage et déchargement de la base de la gestion du verrouillage

- **Verrouillage pessimiste**
  - Verrouillage strict nécessitant un verrouillage des tuples au niveau de la base.
  - Peut dégrader les performances.
  - Impose d'obtenir un verrou sur une entité avant de la manipuler.

1. Verrouillage - b. Modification du mode par défaut

- Principaux modes de verrouillage (`LockModeType`)
  - `OPTIMISTIC` : verrouillage optimiste sans détection de conflits.
  - `OPTIMISTIC_FORCE_INCREMENT` : verrouillage optimiste et incrémentation du numéro de version
  - `PESSIMISTIC_READ` : verrouillage pessimiste. Equivalent au niveau d'isolation `REPEATABLE_READ`
  - `PESSIMISTIC_WRITE` : verrouillage pessimiste. Equivalent au niveau d'isolation `TRANSACTION_SERIALIZABLE`
  - `PESSIMISTIC_FORCE_INCREMENT` : Verrouillage pessimiste et incrémentation du numéro de version de l'entité
- Verrouillage avec les méthodes de l'`EntityManager`
  - `find(Class c, Object primaryKey, LockModeType LockMode)`
  - `lock(Object entity, LockModeType lockMode)`
  - `refresh(Object entity, LockModeType lockMode)`
- Avec la méthode de `setLockMode(LockModeType lockMode)` de `Query`

2. Versionnement

- Une classe entité peut disposer d'un attribut indiquant la version de l'entité.
- Exemple :

```
@Version
private long versionID = 1L;
```

- En mode `OPTIMISTIC_FORCE_INCREMENT`, chaque fois qu'une entité est modifiée, son numéro de version est incrémenté
- Détection des conflits
  - Comparaison du numéro de version trouvé lors d'un `commit()` ou d'un `flush()` à celui trouvé lors du démarrage de la transaction
  - Si les numéros sont différents une exception `OptimisticLockException` est levée.
  - La résolution est à la charge du programmeur

Chapitre 6 - Java Persistence API - Partie 2 : Gestion des objets persistants

1. Méthodes de rappel

- Entité gérée :
  - Référencée dans le contexte de persistance
  - État sauvegardé suite à un commit ou un flush

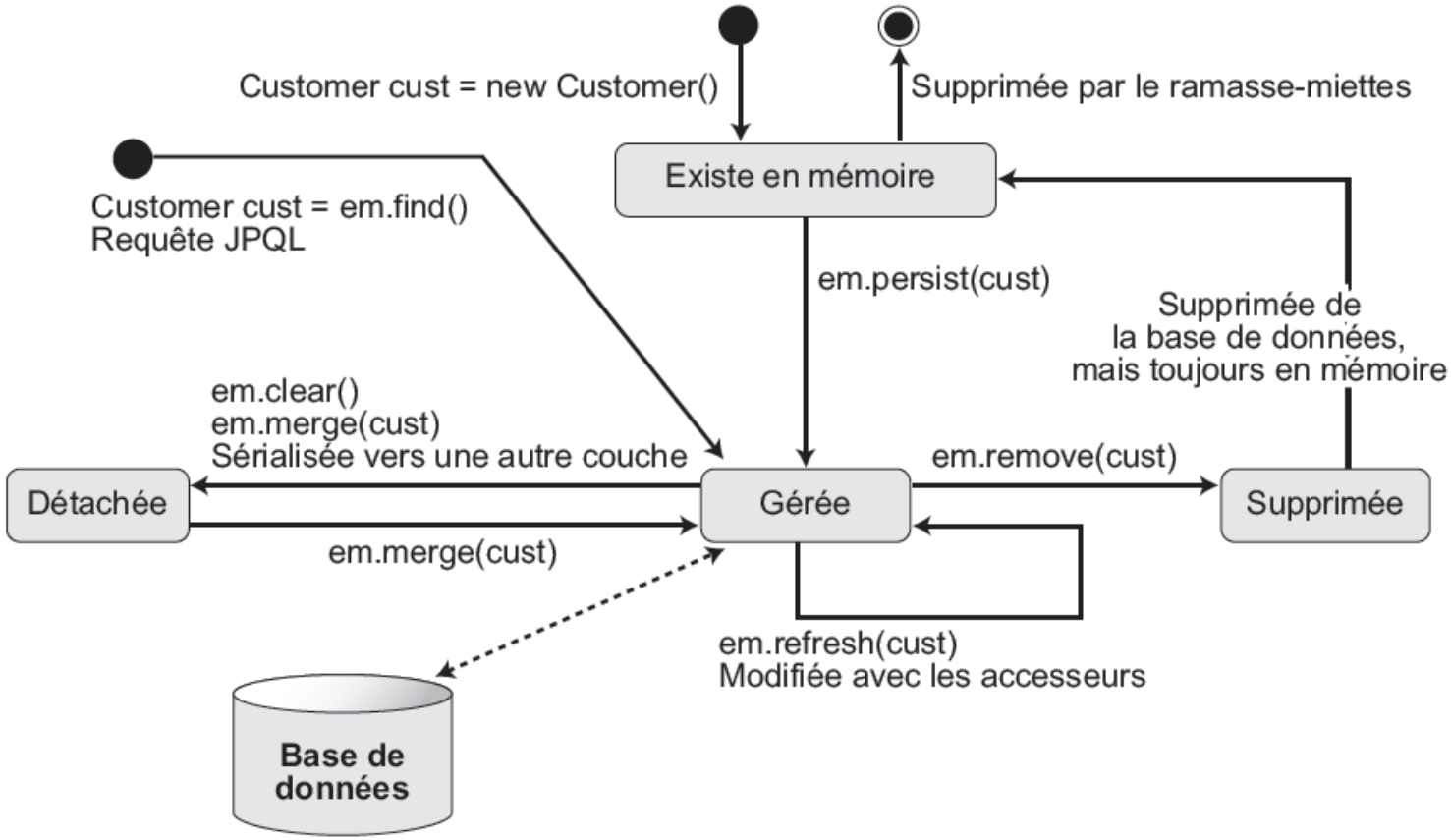


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

2. Méthodes de rappel

- JPA permet d'associer des méthodes de rappel pour chaque passage d'une entité d'un état à un autre
- Ces méthodes agissent comme des triggers (déclencheurs)
- Ils sont appelées automatiquement par le gestionnaire d'entités chaque fois qu'un événement particulier se produit
- Méthodes de la classe entité annotées par `@PreXXX` ou `@PostXXX`

2. Méthodes de rappel

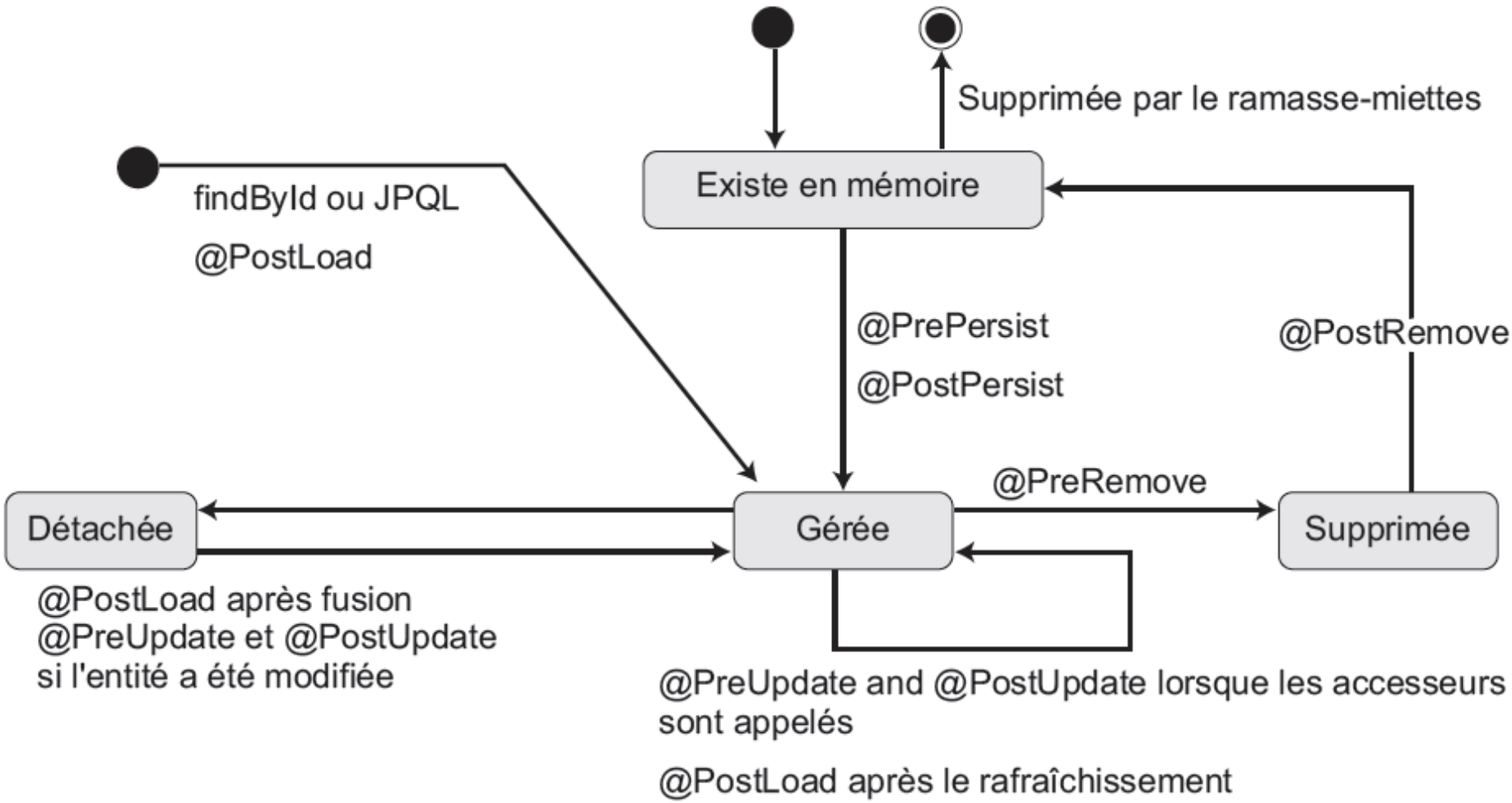


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

2. Méthodes de rappel

- Annotations

Annotation	Description
`@PrePersist`	Appel avant l'exécution d'`EntityManager.persist()`
`@PostPersist`	Appel après l'enregistrement dans la BD de l'entité. Si la génération de la clé primaire est automatisée, la valeur est accessible dans la méthode
`@PreUpdate`	Appel avant la modification de l'entité dans la base (suite à l'appel à un setter ou à EntityManager.merge())
`@PostUpdate`	Appel suite à l'enregistrement dans la base
`@PreRemove`	Appel avant l'exécution de `EntityManager.remove()`
`@PostRemove`	Appel après la suppression de l'entité
`@PostLoad`	Appel suite au chargement d'un tuple dans un objet entité (par une requête ou par un `EntityManager.find()`)

R.Q. Les méthodes de rappel peuvent invoquer JNDI, JDBC, JMS et les EJB, mais aucune opération d'`EntityManager` ou de `Query`.

2. Méthodes de rappel

- Exemple

```
@PostLoad
@PostPersist
@PostUpdate
public void calculateAge() {
  if (dateOfBirth == null) {
    age = null;
    return;
  }
  Calendar birth = new GregorianCalendar();
  birth.setTime(dateOfBirth);
  Calendar now = new GregorianCalendar();
  now.setTime(new Date());
  int adjust = 0;
  if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0)
    adjust = -1;
  }
  age = now.get(YEAR) - birth.get(YEAR) + adjust;
}
```

Chapitre 7 - Enterprise Java Beans

1. Tier métier

- EJB (*Enterprise Java Beans*) : composants côté serveur encapsulant la logique métier d'une application d'entreprise

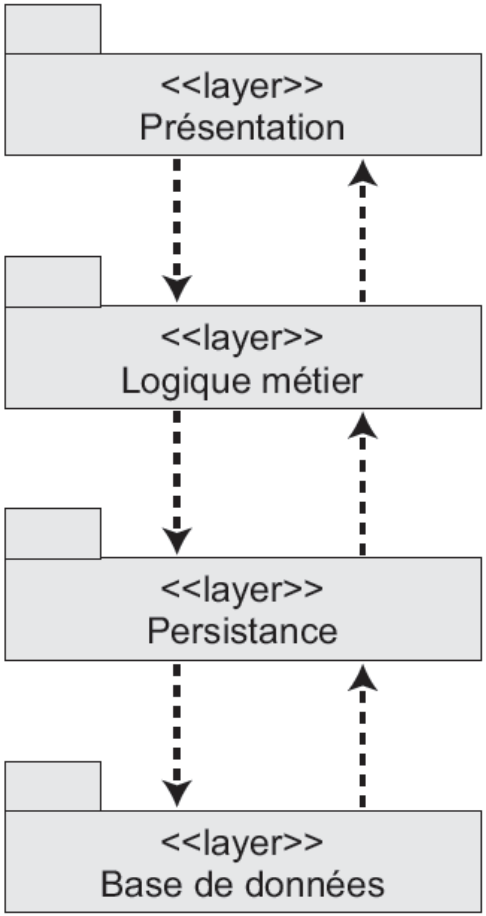


Figure: Architecture n-tiers. ["Objet-relationnel". Notes de cours. R. Grin. 2007]

1. Exemple simple

- EJB : simples classes POJO annotées, dont les **instances** sont **gérées dans un pool** et les **méthodes** sont **transactionnelles**

```
@Stateless
public class EmpDAO {

  //Injection de dépendance
  @PersistenceContext(unitName = "rhPersistenceUnit")
  private EntityManager em;

  public Emp findEmpById(Long id) {
    return em.find(Emp.class, id);
  }

  public Emp createEmp(Emp emp) {
    em.persist(emp);
    return emp;
  }
}
```

Définition d'une classe métier

1. Exemple simple

- Le conteneur peut partager une instance d'un bean entre plusieurs clients
- Le client ne créé donc pas d'instance du bean, mais obtient une référence d'un bean existant (c'est pour cette raison qu'on n'utilise pas `new()`)
- Injection de dépendance : permet d'obtenir une référence à une instance du bean

```
public class Main {

  //Injection de dépendance
  @EJB
  private static EmpDAO empDAO;

  public static void main(String[] args) {
     Emp emp = new Emp();
     ...
     empDAO.createEmp(emp);
  }
}
```

Code client

2. Conteneur d'EJB

- EJB : ne peuvent être exécutés que par un conteneur
- Conteneur EJB : environnement d'exécution des EJB fournissant certains services
- Principaux services offerts par un conteneur EJB
  - **Pooling** : le conteneur crée un pool pour les instances de certains EJB. Ces instances peuvent être partagées par plusieurs clients EJB. Il en fait de même pour d'autres ressources (Ex. DataSource). Le pooling permet de réduire le nombre d'objets créés en mémoire.
  - **Annuaire** : le conteneur dispose d'un annuaire JNDI, qui enregistre toutes les ressources qu'il gère et expose. Les ressources gérées sont accessibles simplement par leur nom.
  - **Gestion du cycle de vie** : le conteneur prend en charge la création, la mise dans un pool, le partage et la suppression des instances EJB

2. Conteneur d'EJB

- Principaux services offerts par un conteneur EJB (suite)
  - **Gestion de la concurrence** : les EJB sont *thread-safe* (excepté les EJB singleton). La synchronisation est gérée par le conteneur.
  - **Injection de dépendance** : le conteneur permet le référencement ou l'injection de dépendances des ressources qu'il gère (DataSource, autres EJB)
  - **Gestion des transactions** : Un EJB informe le conteneur par annotations de sa politique de gestion des transactions. La délimitation, la validation et l'annulation sont gérés en arrière plan par le conteneur que les transactions soient réparties ou non.
  - **Sécurité** : les EJB peuvent préciser un contrôle d'accès au niveau de la classe afin d'imposer une authentification de l'utilisateur
  - **Journalisation** : le conteneur gère la journalisation de tous ces éléments, de même que le suivi des performances, et différentes fonctions de monitoring.

Chapitre 7 - Enterprise Java Beans

Types d'EJB

- Deux grandes familles d'EJB :
  - *Session Beans*
  - *Message Driven Beans* [6]
- **Beans de session** : trois types
  - **Bean sans état** (*stateless*) : une même instance du bean peut être partagée par plusieurs clients (mode partagé).
  - **Bean avec état** (*stateful*) : chaque client utilise sa propre instance du bean (mode dédié).
  - **Bean Singleton** (*singleton*) : une instance unique existe et est partagée entre tous les clients

[6] Dans la suite nous nous intéressons aux beans de session. Pour les MDB, le lecteur est invité à consulter d'autres sources

1. Beans sans état

- Classes annotées par `@javax.ejb.Stateless`. Beans les plus utilisés.
- Beans les plus efficaces et les plus souples :
  - Placés dans un pool et partagés par plusieurs clients
  - Lorsqu'un client appelle une méthode d'un bean sans état, le conteneur choisit une instance du pool et l'affecte au client.
  - Quand le client en a fini, l'instance retourne au pool

  $\Rightarrow$ Il suffit d'un petit nombre de beans pour gérer plusieurs clients
  - Peuvent s'utiliser. entre autres. comme **DAO**

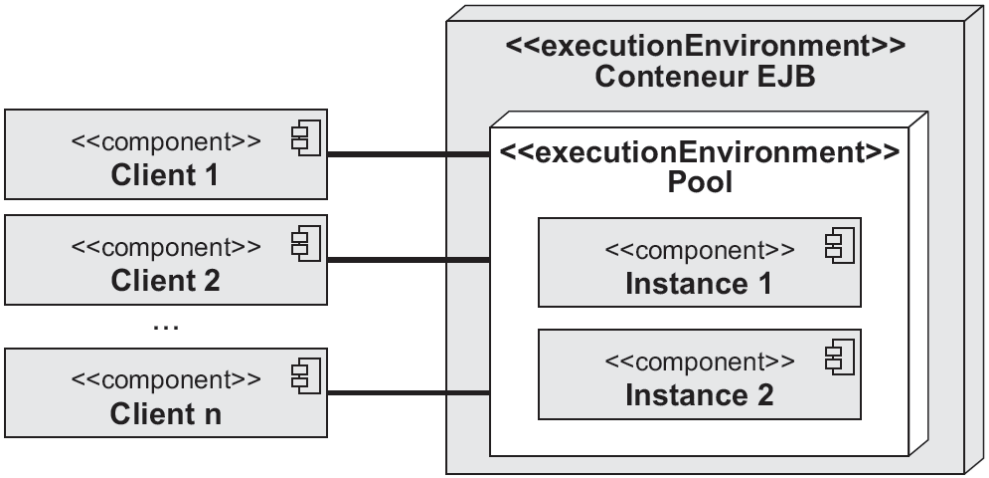


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

2. Beans avec état

- Classes annotées par `@javax.ejb.Stateful`.
- S'utilisent lorsqu'une même tâche métier s'exécute en plusieurs étapes (*i.e.* avec plusieurs appels de méthodes du bean) et que chaque étape nécessite la prise en compte du résultat des étapes précédentes

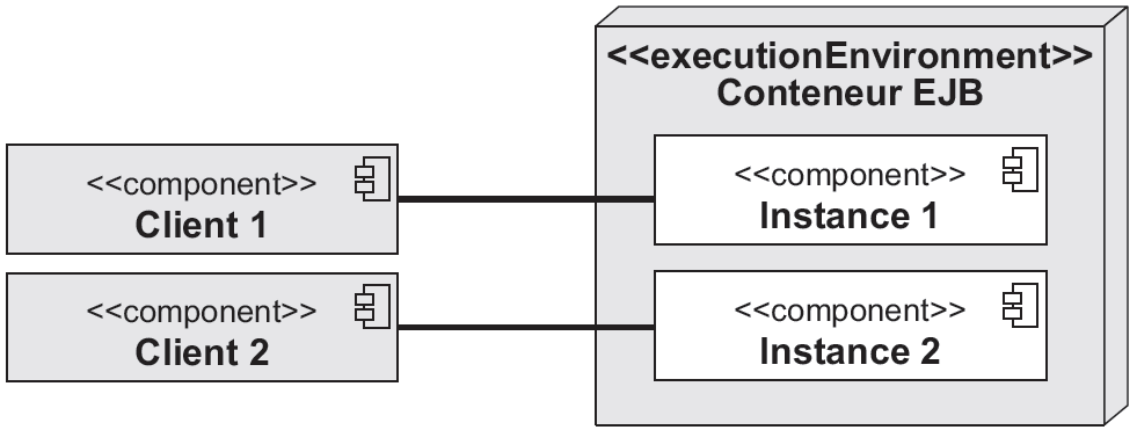


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

2. Beans avec état

- Exemple : site de vente en ligne
  - Ajout de produits au panier virtuel : plusieurs appels à la méthode `addItem()`
  - Le bean doit mémoriser les différents produits ajoutés par les différents appels `addItem()`

⇒ l'application cliente doit utiliser le même bean, qui doit mémoriser l'état de la conversation

2. Beans avec état

```
@Stateful
statefullTimeout(20000)
public class purchase {

  private List<Item> cartItems = new ArrayList<Item>();

  public void addItem(item item){
    if(!cartItems.contains(item)){
      cartItems.add(item);
    }
  }
 ...
}
```

```
Item item = statelessBItem.findById(123L);
statefullBPurchase.addItem(item);
item = statelessBItem.findById(456);
statefullBPurchase.addItem(item);
statefullBPurchase.Validate();
```

2. Beans avec état

- Mode dédié : pas de réutilisation, peut engendrer la création de beaucoup d'objets en mémoire
- Utilisation du mécanisme de passification/activation :
  - Stockage temporaire d'une instance sur disque dur
  - Chargement de l'instance au besoin
  - Problèmes potentiels de lenteur

3. Beans Singleton

- Classes annotées par `@javax.ejb.Singleton`.
- Bean de session qui n'est instancié qu'une seule fois (inspiré du design pattern *Singleton* vu en TP)
- Une fois instancié, le conteneur garantit qu'il n'y aura qu'une seule instance de singleton pour toute l'application.

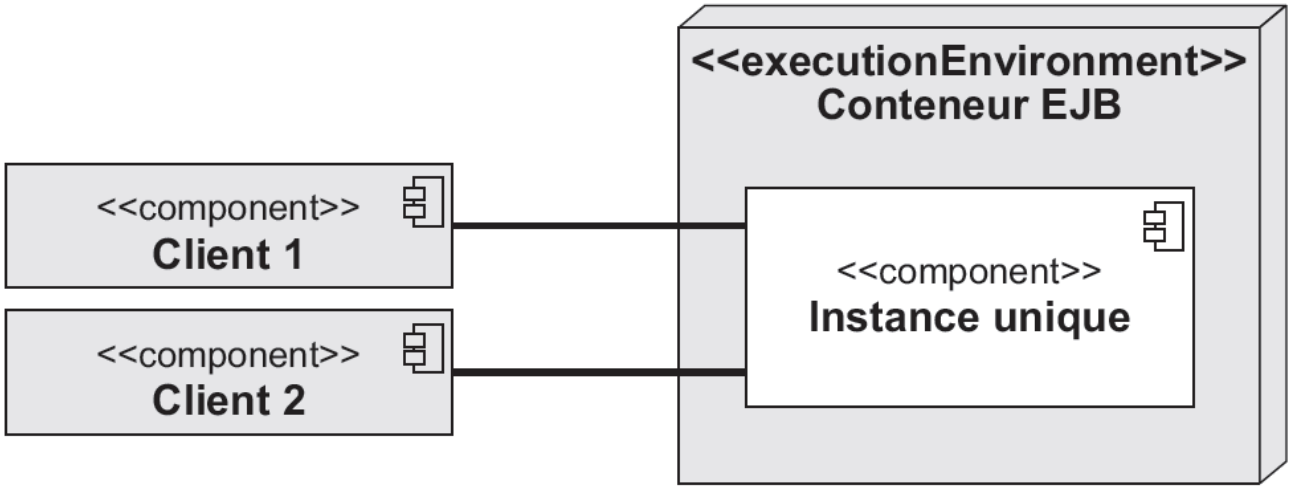
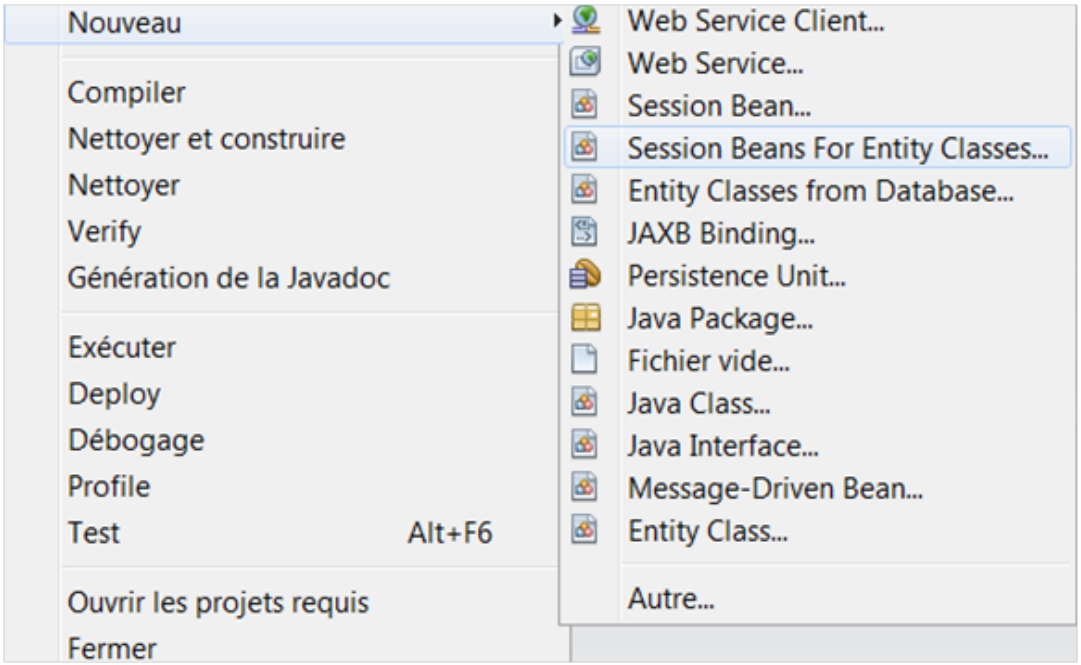


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

Chapitre 7 - Enterprise Java Beans

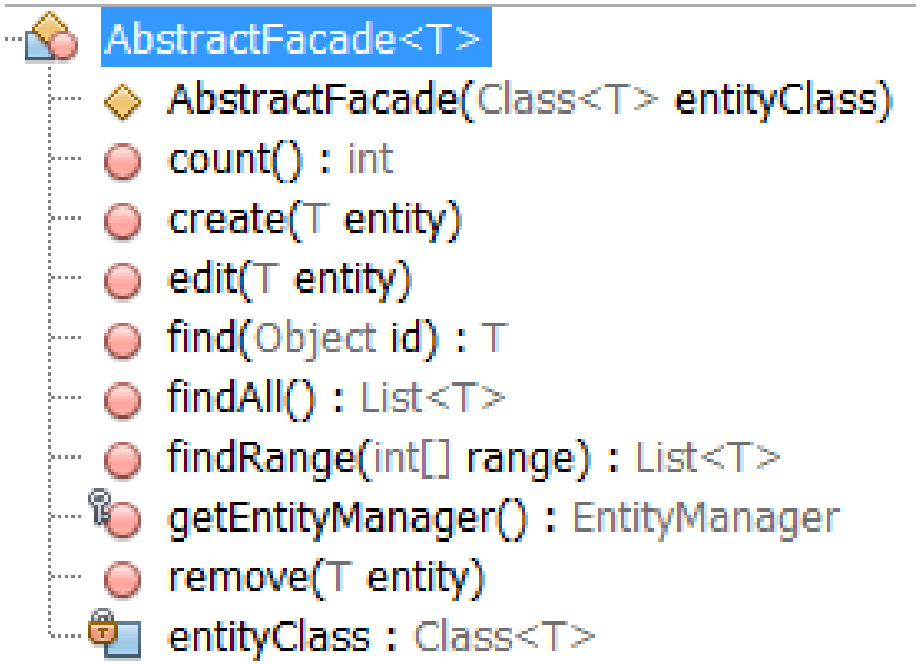
1. Génération des beans de session sans état

- Il est possible d'utiliser un EJB sans état pour implanter les opérations CRUD sur une entité JPA : l'EJB joue ainsi le rôle de DAO
- Les environnements de développement permettent la génération automatique d'EJB sans état à partir d'entités JPA
- Cette fonctionnalité combinée avec la génération automatique d'objets entité permet une génération automatique de toute la couche accès aux données



2. Génération des beans de session sans état

- Les classes DAO générées utilisent un objet `EntityManager` pour réaliser les différentes opérations de manipulation des objets entité
- Le code de gestion est mis en facteur dans une classe abstraite générique `AbstractFacade`
- Les autres classes (une classe par entité) sont des extensions de la classe abstraite



2. Génération des beans de session sans état

- Avantage d'une classe DAO par rapport à une utilisation directe d'`EntityManager` :
  - On peut définir dans ces classes des requêtes plus élaborées que les simples `find()`, `merge()`, etc.
  - Ces requêtes sont définies une seule fois dans la DAO et peuvent se réutiliser sans être redéfinies dans le code métier.

Chapitre 7 - Enterprise Java Beans

1. Injection de dépendance

- Une même instance du bean peut être partagée par plusieurs applications clientes.
- Le client manipule un proxy et non la véritable instance du bean. La véritable instance est créée du coté du conteneur
- Obtention d'une référence :
  - Par une recherche dans le JNDI du conteneur
  - Par injection de dépendance avec l'annotation @EJB
  - Via un web service

1. Injection de dépendance

- Annotation `@EJB` :
  - Permet d'injecter une référence à un EJB dans l'application cliente (**injection de dépendances**)
  - Exemple :

```
@EJB
MySessionBean mySessionBean;
```

- Alternativement, la référence à un EJB peut s'obtenir par une recherche dans l'**annuaire JNDI** du conteneur de l'EJB
  - Lors du déploiement d'un bean de session, son nom est automatiquement lié à un nom du JNDI
  - Nom JNDI : `java:global/nom de l'EJB/nom de l'interface`
  - Recherche:

```
MySessionBean mySessionBean = (MySessionBeanRemote)
        InitialContext.lookup("java:global/EntAppEJB/MySessionBeanRemote")
```

2. Interfaces locales et distantes

- Un EJB peut être appelé à partir d'une application cliente locale (*i.e.* tournant sous la même JVM), distante (*i.e.* tournant sur une JVM différente) ou via un service web
- **Appel local** : les paramètres des méthodes du bean sont passés par référence.
- **Appel distant** : invocation par des appels RMI (*Remote Method Invocation*). Les paramètres sont passés par valeur.
- **Service web** (appel distant) : paramètres et résultats sérialisés en XML

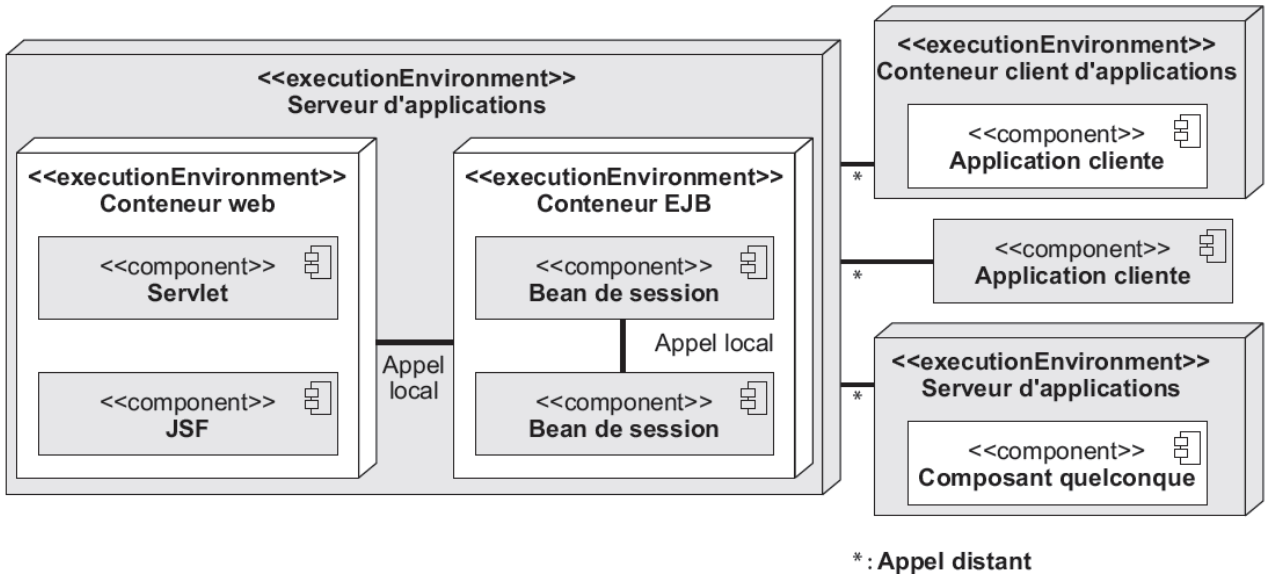


Figure: ["JEE 6 et GlassFish" .A. Goncalves. Pearson Education. 2010.]

2. Interfaces locales et distantes

- Appel distant
  - Le client d'un bean ne le référence pas directement, mais référence un proxy qui transforme l'appel local du client (vers le proxy) en un appel distant vers le bean
  - Le client invoque les méthodes du proxy et non directement les méthodes de la classe bean
  - Le proxy doit donc avoir la même interface que le bean (même signature des méthodes, etc.).
  - Pour pouvoir créer les objets proxy, le client doit connaître l'**interface** du bean
  - Souvent une bibliothèque contenant l'interface est créée. Cette bibliothèque est ensuite distribuée aux applications clientes distantes.

2. Interfaces locales et distantes

- **Interface métier** d'un bean
  - Contient les déclarations des méthodes visibles par les clients et implantées par la classe bean.
  - Contrat entre le bean et le client
  - Interface locale (annotée par *@Local*) : optionnelle, les méthodes publiques forment l'interface
  - Interface distante (annotée par *@Remote*) : obligatoire

Chapitre 7 - Enterprise Java Beans

1. Modes de gestion

- Les traitements effectués par un EJB sont encapsulés dans des transactions
- Ces transactions peuvent être gérées par le conteneur. On parle alors de *Container Managed Transaction* (CMT)
- Alternativement les transactions peuvent être gérées par l'EJB. On parle alors de *Bean Managed Transaction* (BMT)
- Dans les deux cas, les appels de bas niveau (verrouillage en 2 phases, etc.) sont réalisés par le conteneur

2. Container Managed Transaction (CMT)

- Annotation `@javax.ejb.TransactionAttribute(TransactionAttributeType.`$type$`)` d'un EJB : définit la stratégie de gestion des transactions d'un bean ou d'une méthode du bean

- Types possibles

Valeur	Description
`REQUIRED`	Valeur par défaut. La méthode doit toujours être invoquée dans une transaction. Le conteneur ne crée une nouvelle transaction que si l'appelant ne dispose pas de contexte de transaction. S'utilise si les données sont modifiées et que l'on ne sait pas si le client a lancé une transaction
`REQUIRES_NEW`	Le client créé toujours une nouvelle transaction. Si le client a lancé une transaction, celle-ci est suspendue jusqu'à la validation ou l'annulation de celle créée par le conteneur. S'utilise lorsque nous ne souhaitons pas que l'annulation de la méthode ait un effet sur le client.
`SUPPORTS`	La méthode de l'EJB hérite du contexte de transaction du client. Si celui-ci n'en possède pas, la méthode est exécutée sans contexte de transaction. S'utilise si on accède à une table en lecture seule.

2. Container Managed Transaction (CMT)

- Types possibles (suite)

`MANDATORY`	Le conteneur exige une transaction du client avant d'appeler la méthode, mais n'en créera pas de nouvelle. S'il n'en existe pas une exception est levée.
`NOT_SUPPORTED`	La méthode ne peut pas être appelée dans un contexte de transaction. Si le client en a un, le conteneur le suspend, appelle la méthode, puis relance la transaction
`NEVER`	La méthode ne peut pas être appelée dans un contexte de transaction. Si le client en a un, le conteneur lève une exception

2. Container Managed Transaction (CMT)

- Marquage d'un CMT pour annulation
  - Un bean CMT n'est pas autorisé à annuler explicitement une transaction, mais il peut demander au conteneur de le faire
  - Il doit pour cela obtenir une référence au contexte de sa session `SessionContext` et appeler la méthode `setRollbackOnly()` de cette interface.
  - Cet appel n'annule pas immédiatement la transaction. mais positionne un indicateur dont tiendra compte conteneur.
  - Exemple :

```
@Stateless public class EmpEJB {
   @Resource
   private SessionContext ctx;
   ...
   ctx.setRollbackOnly();
   ...
}
```

  - Un bean peut appeler la méthode `getRollbackOnly()` pour savoir si la transaction courante a été marquée pour annulation

3. Bean Managed Transaction (BMT)

- Le mode CMT ne permet pas dans certains cas d'avoir la finesse voulue de démarcation des transactions
- Pour ces cas, le conteneur permet au bean de gérer les transactions par lui-même (*Bean Managed Transaction*)
- Annotation `@TransactionManagement(TransactionManagementType.BEAN)` d'un EJB : indique que la gestion est de type BMT.
- L'interface `javax.transaction.UserTransaction` s'utilise par le bean pour la démarcation des transactions
- Un objet `javax.transaction.UserTransaction` s'obtient via une recherche dans le JNDI ou par `SessionContext.getUserTransaction()` (plus simple)

3. Bean Managed Transaction (BMT)

- Méthodes de l'interface `javax.transaction.UserTransaction`

Méthode	Description
`begin`	Débute une nouvelle transaction et l'associe au thread courant
`commit`	Valide la transaction attachée au thread courant
`rollback`	Annule la transaction attachée au thread courant
`setRollbackOnly`	Marque la transaction courante pour annulation
`getStatus`	Récupère l'état de la transaction courante
`setTransactionTimeout`	Modifie le délai d'expiration de la transaction courante

Chapitre 8 - Notes sur SOAP et REST

Introduction

- **Intérêt des services web** : permettre l'appel à distance (via le web) à des traitements métier (SOAP) ou à des ressources (REST)

- **En quoi est-ce différent d'autres technologies?**

  1. Interopérabilité : les messages échangés sont formatés en XML (ou JSON) et donc l'application cliente et le service web peuvent être écrits dans des langages différents (contrairement à RMI et à DCOM)
  2. Simplicité : utilisation de protocoles simples et universels (contrairement au très complexe CORBA)
  3. La plupart des pare-feux laissent passer le protocole HTTP (contrairement à d'autres protocoles).

- Deux grandes approches :
  - **SOAP** (*Simple Object Access Protocol*) : un protocole standardisé
  - **REST** (*Representational State Transfer*) : une architecture

Chapitre 8 - Notes sur SOAP et REST

1. Exemple rudimentaire avec JAX-WS

- Définition d'un service SOAP (à l'aide de JAX-WS), côté application serveur.

```
@WebService(serviceName = "serviceHello")
public class serviceHello {

  @WebMethod(operationName = "bonjour")
  public String bonjour(@WebParam(name = "aQui") String txt) {
     return "Bonjour " + txt + " !";
  }
  ...
}
```

- Le service web de l'exemple appelé "`serviceHello`", dispose d'une seule méthode web ("`bonjour`"). Cette méthode prend en entrée un paramètre appelé `aQui`
- Le service peut exposer plusieurs autres méthodes. Ces méthodes peuvent être invoquées à distance par les applications clientes.

1. Exemple rudimentaire avec JAX-WS

- Invocation d'une méthode du service côté application cliente

```
  //Instantiation d'un proxy du service
ServiceHello_Service service = new ServiceHello_Service();

  //Connexion à un EndPoint du service
ServiceHello port = service.getServiceHelloPort();

  //Appel à la méthode bonjour du service.
    //L'appel local est transformé à ce niveau en un appel distant
String reponse = port.bonjour("isitcom");
```

- L'exécution de la méthode se fait par l'application serveur.
- La définition et l'invocation de services SOAP, sont grandement simplifiés par JAX-WS. Les choses sont un peu plus complexes que l'exemple présenté!

2. Fournisseur, annuaire et client

Services SOAP : font intervenir trois types d'acteurs

- Le **fournisseur** du service
  - Définit le service et implémente ses méthodes.
  - Publie sa description dans l'annuaire sous la forme d'un document WSDL (*Web Services Description Language*)
  - Exécute les méthodes et renvoie les réponses formatées en XML
- L'**annuaire UDDI** (*Universal Description Discovery and Integration*)
  - Répertorie les descriptions des services publiés par les fournisseurs (les documents WSDL)
- Le **client**
  - Obtient une description du service grâce à l'annuaire.
  - Cette description (le WSDL) lui permet d'invoquer à distance les méthodes du service (URI du service, noms des méthodes, format des messages, etc.)
  - Formate les requêtes en XML (méthode invoquée, ses paramètres, etc.)

2. Fournisseur, annuaire et client

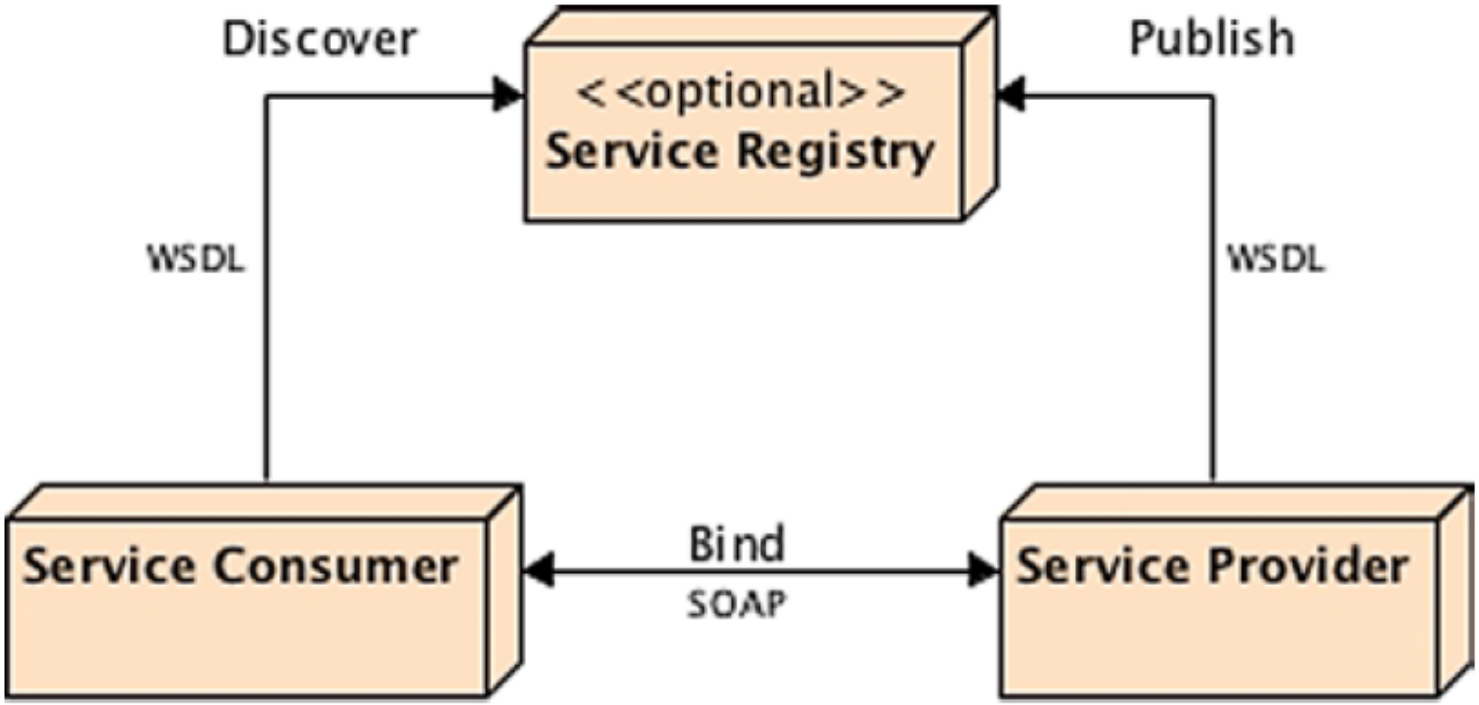
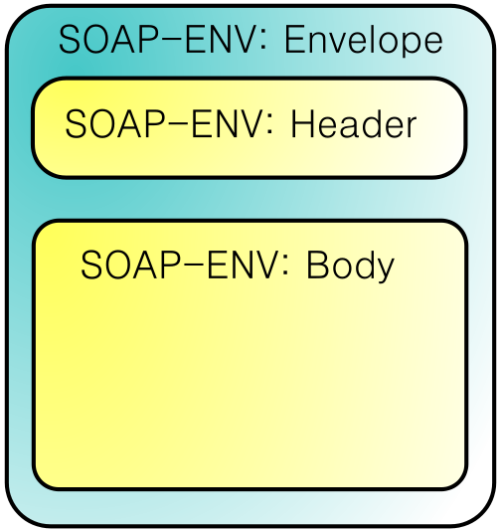


Figure: ["Beginning Java EE 7". A. Goncalves. éditions Apress. 2013]

3. Structure d'un message SOAP

- **SOAP** est un **protocole** : Les **messages XML** échangés, ainsi que les documents WSDL doivent suivre un certain **format**



- Structure des messages SOAP :
  - **Enveloppe** : élément racine
  - **En-tête** : contient des informations optionnelles sur le l'acheminement et la sécurisation des échanges
  - **Corps** : contient les informations échangées. S'il s'agit d'une requête SOAP, on y trouve le nom de la méthode invoquée et ses paramètres. Si c'est une réponse, les données résultant du traitement.

3. Structure d'un message SOAP

Exemple de messages SOAP échangés entre le client et le serveur

SOAP Request

```
<soapenv:Envelope xmlns:soapenv="http://schemas
   <soapenv:Header/>
   <soapenv:Body>
      <ser:bonjour>
         <aQui>isitcom</aQui>
      </ser:bonjour>
   </soapenv:Body>
</soapenv:Envelope>
```

SOAP Response

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/s
   <S:Body>
      <ns2:bonjourResponse xmlns:ns2="http://serv
         <return>Bonjour isitcom !</return>
      </ns2:bonjourResponse>
   </S:Body>
</S:Envelope>
```

4. WSDL

- Document **WSDL** : document XML respectant un certain format et indiquant
  - le schéma des données échangées (le format des messages acceptés et retournés par les méthodes web);
  - comment utiliser le service web (l'emplacement du service); et
  - comment interagir avec le service (méthodes, paramètres et valeurs de retour).

`Élément`	Descritpion
`definitions`	Racine du WSDL. Contient le déclaration des namespaces
`types`	Définit les types de données échangées dans les messages. On y trouve notamment un lien vers le **.xsd** (*XML Schema Definition*) définissant la structure des paramètres passés aux méthodes du service et les données renvoyées en réponse
`message`	Définit les types de messages échangés. Typiquement, pour chaque méthode du web service, on a deux balises `message`, la première définit le contenu du message permettant d'invoquer la méthode, et la deuxième le message renvoyé en réponse par la méthode.

4. WSDL

élément	Descritpion
portType	Décrit les méthodes du service. A chaque méthode, on associe un message requête et un message réponse. Il s'agit en quelques sortes de l'interface du service.
binding	Un binding propose une réalisation concrète d'un type de port (méthode du service). Par exemple, un type de port peut être réalisé par SOAP+HTTP et/ou par SOAP+STMP.
service	Contient une liste de ports
port	Spécifie pour chaque *Binding*, un *end point*. Par exemple, si le protocole de transport utilisé est HTTP, port indique l'URL permettant d'atteindre le service et ses méthodes)

4. WSDL

Exemple de WSDL (généré auotmatiquement grâce à JAW-WS).

4. WSDL

Exemple de WSDL (suite)

```
<binding name="serviceHelloPortBinding" type="tns:serviceHello">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
   <operation name="bonjour">
      <soap:operation soapAction=""/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>
<service name="serviceHello">
   <port name="serviceHelloPort" binding="tns:serviceHelloPortBinding">
      <soap:address location="http://localhost:8080/Bonjour/serviceHello"/>
   </port>
</service>
</definitions>
```

4. WSDL

XSD généré


```
<xs:schema version="1.0" targetNamespace="http://service/" xmlns:t
  <xs:element name="bonjour" type="tns:bonjour"/>
  <xs:element name="bonjourResponse" type="tns:bonjourResponse"/>
  <xs:complexType name="bonjour">
    <xs:sequence>
      <xs:element name="aQui" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="bonjourResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

5. JAX-WS

- **JAX-WS** (*Java API for XML-Based Web Services*) : API Java pour la création et la consommation de services web SOAP

- Masque grandement la complexité de la création des services web :
  - Service Web : POJO annoté par `@javax.jws.WebService` et déployé dans un conteneur web
  - Méthode web : méthode de la classe annotée par `@javax.jws.WebMethod`. Les paramètres que le client passe à la méthode Web sont annotés par `@javax.jws.WebWebParam`
  - Formatage des réponses en des messages SOAP (d'une manière transparente au développeur)
  - Fichier WSDL et schémas XSD : générés automatiquement à partir des annotations

5. JAX-WS

- Du côté de l'application cliente
  - Localisation (URI) du service web à partir du WSDL
  - Paramètres et appels aux méthodes web formatés en un *SOAP request* (d'une manière transparente au développeur)
  - Proxy (Stubs) générés automatiquement à partir du WSDL (les appels locaux aux proxy, sont transformés en appels distants)
  - Classes définissant les types de retour des différentes opérations générées automatiquement à partir du WSDL

5. JAX-WS

- Metro :
  - Implémentation de référence de `JAX-WS` (il en existe d'autres, Apache Axis2, JBoss JBossWS, etc.)
  - Permet de créer et de déployer des services web et des consommateurs sécurisés, fiables et transactionnels.
  - Produit par la communauté GlassFish, mais peut être utilisé en dehors de celui-ci, dans un environnement Java EE ou Java SE
  - Fournit également une implémentation de l'API JAXB (*Java Architecture for XML Binding*)

6. JAXB

- **JAXB** (*Java Architecture for XML Binding*) :
  - API Java pour la **sérialisation** et la **désérialisation** d'objets Java en XML
  - Joue en quelque sorte le rôle de JPA pour XML.
  - Comme JPA, elle permet la création de classes Java à partir de schémas XML (.XSD) et inversement, grâce à des annotations (Ex. `@XmlRootElement`).
  - Comme JPA, elle permet le stockage (sérialisation) d'objets Java dans un document XML et l'instantiation d'objets Java à partir d'un document XML (désérialisation)
  - Une même classe entité peut être annotée selon JAXB et JPA
- Utilité de JAXB pour les services Web
  - Côté service Web : sérialisation des objets entités dans les réponses SOAP
  - Côté client : création des classes entités et désérialisation des objets entités

7. EJB sous forme d'un service Web

- Les méthodes d'un EJB[7] peuvent être exposées sous forme d'un service web SOAP (alternativement REST)
- Il suffit de l'annoter par `@WebService` et certaines de ses méthodes par `@WebMethod`
- Avantage par rapport à RMI : le client peut être écrit dans n'importe quel langage.

[7] à condition qu'il ne soit pas stateful

Chapitre 8 - Notes sur SOAP et REST

1. Introduction

- **REST** (*Representational State Transfer*) web services : connaissent un grand essor (Google, Amazon, eBay, Yahoo, etc.)

- REST n'est pas un protocole, mais une **manière particulière d'utilisation du protocole HTTP** :
  - L'application cliente n'utilise que les méthodes HTTP standards (`GET`, `POST`, `PUT`, etc.) pour demander des services au serveur REST
  - Du côté du serveur des méthodes métiers sont mis en correspondance avec les méthodes HTTP. Exemple : si une requête `GET` est reçue par le serveur, ceci déclenche la méthode métier correspondante
  - Les messages échangés entre le client et le serveur, n'ont pas de formats propres à REST. Ils suivent le protocole HTTP (`HTTP Request`, `HTTP Response`)

  $\Rightarrow$ **Il s'agit d'un design pattern ou d'un style architectural** et non d'un protocole

2. Ressources web

- Serveur HTTP (web) : produit et stocke des ressources (web) et permet de leur accéder et de les manipuler par des requêtes HTTP

- **Ressource** (au sens web) :
  - Toute information, image, texte, donnée, etc. accessible via un serveur web.
  - Peut être une page web, un fragment d'une page web, ou une information que le serveur web met à disposition de ses clients (*e.g.* données recherchées depuis une BD, résultat d'un calcul, etc.)
  - Une ressource web a un **type MIME** et une **URI** (*Unique Ressource Identifier*)

- **Type MIME** (*Internet Media Type*)
  - Format général : `type/sous-type`
  - Exemples de types usuels : `text/plain`, `text/html`, `application/xml`, `application/xml`, `image/jpg`, `video/mp4`, etc.

2. Ressources web

- **URI** (*Unique Ressource Identifier*)
  - Identifie de manière unique une ressource.
  - Les ressources sont accessibles par leur URI
  - Format général d'une URI : $http://host:port/path?queryString\#fragment$
  - Exemple :
    $http://maps.google.com/maps/api/staticmap?center=34.50592,9.48310\&zoom=7\&size=400x800\&sensor=false$

3. HTTP - a. Requête

- Requête/Réponse HTTP :

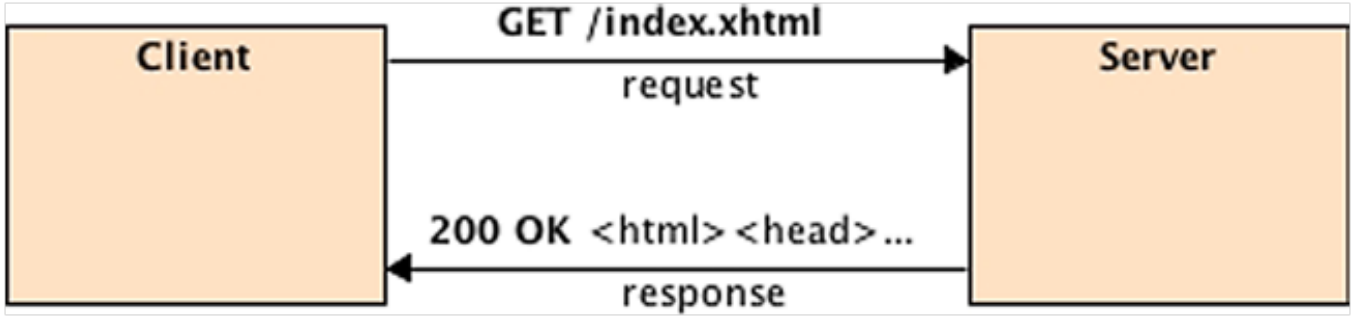


Figure: ["Beginning Java EE 7". A. Goncalves. éditions Apress. 2013]

- Une **requête** indique une **méthode http** (`GET`, `PUT`, `POST`, `DELETE` etc.) et éventuellement l'**URI** de la ressource sur laquelle la méthode s'applique
- En-tête de la requête : indique entre-autres, le **type MIME** de la ressource attendue en réponse.
- Corps de ma requête : informations envoyées au serveur (pour `PUT` et `POST`)

3. HTTP - a. Requête

- Méthodes HTTP usuels :
  - `GET` : lecture d'une ressource, sans modifier son état
  - `POST`: création d'une nouvelle ressource. Le corps de la requête contient les informations nécessaires à la création (Ex. *idDept=1&nomDept="Info"*)
  - `PUT`: mise-à-jour d'une ressource. Le corps de la requête contient les informations nécessaires à la mise-à-jour.
  - `DELETE` : suppression d'une ressource
- Il existe d'autres méthodes HTTP, mais elles sont peu utilisées (`HEAD`, `CONNECT`, etc.).

3. HTTP - b. Réponse

- Requête/Réponse HTTP :

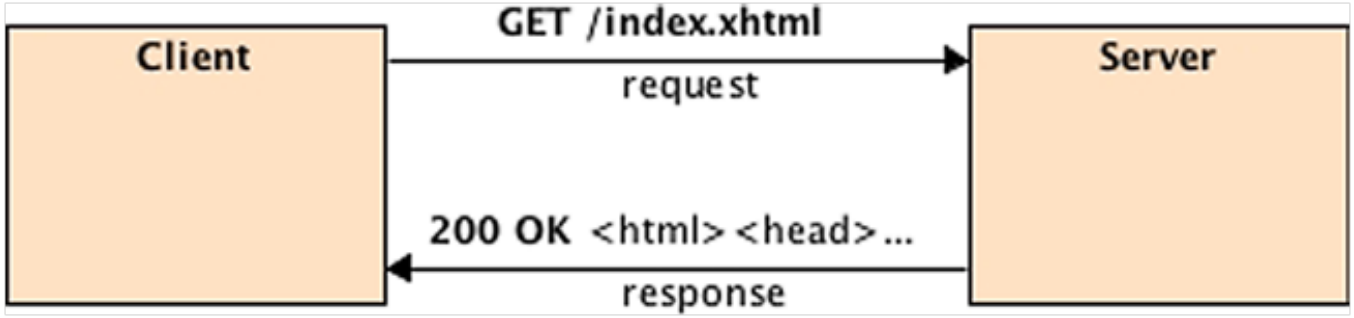


Figure: ["Beginning Java EE 7". A. Goncalves. éditions Apress. 2013]

- Une réponse est composée de :
  - **Code associé à la réponse (response code)**, par exemple 200-Ok si tout se passe bien, 404-not found, etc.
  - En-tête de la réponse : contient le **type MIME** de la ressource renvoyée en réponse (`text/plain`, `text/html`, etc.)
  - Corps de la réponse : la **ressource** renvoyée par le serveur (page html, document xml, JSON, image, etc.)

3. HTTP - c. REST et HTTP

- Dans une architecture REST, tout est ressource : on ne manipule et on n'obtient que des ressources

- **Serveur REST**
  - Produit les ressources sous différents formats (text/plain, text/xml, etc.)
  - Réceptionne les requêtes HTTP et renvoie des réponses HTTP
  - En fonction de la méthode HTTP reçue et du type MIME attendu par le client, il exécute la méthode métier correspondante et renvoie le résultat dans une réponse HTTP standard

- **Client REST**
  - Utilise les méthodes HTTP standards pour accéder aux ressources (`GET`) et demander au serveur de les modifier (`POST`, `PUT`, `DELETE`, etc.)
  - Envoie des requêtes HTTP et reçoit en retour des réponses HTTP

3. HTTP - c. REST et HTTP

Exemple rudimentaire avec JAX-RS :

```
@Path("bonjour")
public class BonjourFromRest {
  @GET
  @Path("{txt}")
  @Produces("MediaType.TEXT_PLAIN")
  public String direBonjour(@PathParam("txt") String txt) {
    return "Bonjour " + txt + "!";
  }
  @GET
  @Path("{txt}")
  @Produces(MediaType.TEXT_XML)
  public String direBonjourEnXML(@PathParam("txt") String txt) {
    return "<?xml version=Ï.0?>" + "<bonjour> Bonjour"
          + txt + "!  </bonjour>";
   }
}
```

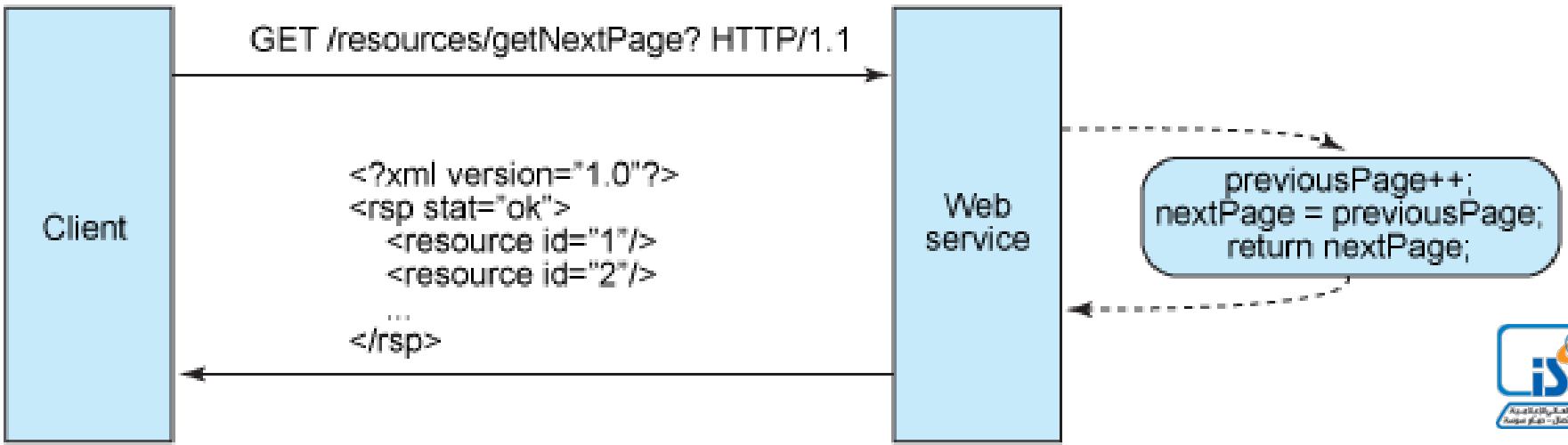
- Accès à la ressource à partir d'un navigateur (se traduit par une requête GET)

`http://www.monserveur.com/.../bonjour/isitcom`

- Réponse (affiché par le navigateur) : Bonjour isitcom!

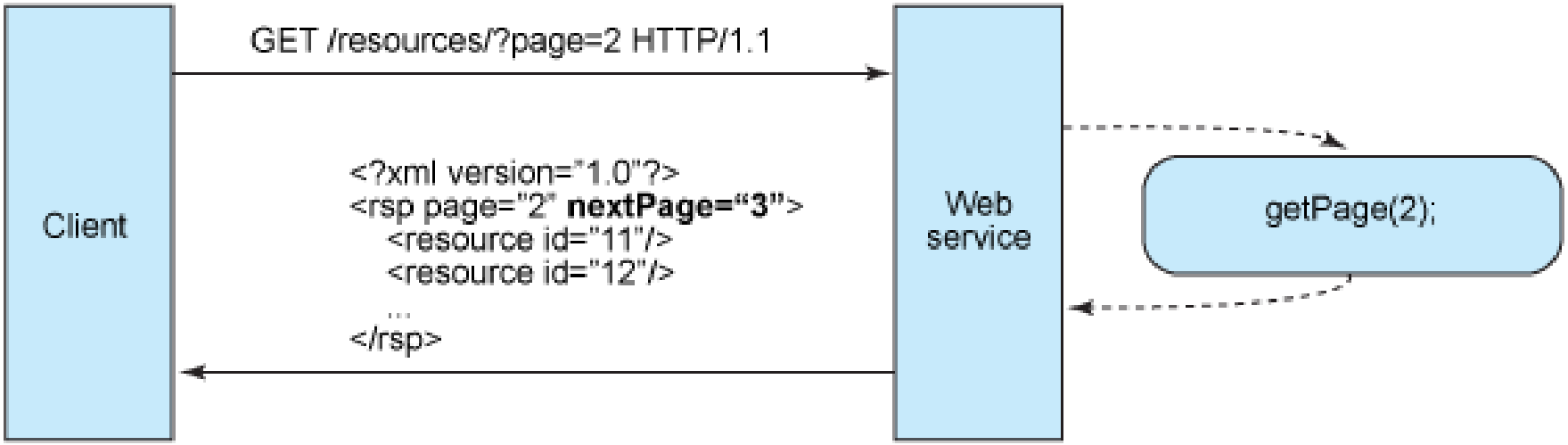
4. Non conservation d'état

- Un service Web respectant l'architecture RESTful doit suivre une conception $Stateless$ (pas de conservation d'état)
- Conception $Stateless$:
  - le serveur ne garde pas d'informations entre 2 requêtes successives d'un client.
  - Toute requête d'un client, doit contenir toutes les informations nécessaires à son exécution
- Exemple de (mauvaise) conception avec conservation d'état : le serveur doit mémoriser l'id de l'ancienne page pour pouvoir renvoyer celle qui suit



4. Non conservation d'état

- Exemple de (bonne) conception sans conservation d'état : le client indique la ressource à laquelle il voudrait accéder



5. JAX-RS, côté serveur - a. Présentation

- Java API for RESTful Web Services (**JAX-RS**) : API pour le développement de Web Services RESTful.
- **Jersey** est l'implémentation de référence (Il y en a d'autres : Apache CXF, JBoss RESTEasy et Restlet)
- JAX-RS, **côté serveur** :
  - Création de Web services REST
  - Faire la correspondance entre (transformer) les appels aux méthodes HTTP en des appels à des méthodes métiers
- JAX-RS, **côté client** :
  - Consommation des ressources produites par le serveur
  - Faire la correspondance (transformer) les appels aux méthodes métiers en des appels à des méthodes HTTP

5. JAX-RS, côté serveur - a. Présentation

- **Application JAX-RS**
  - Application Web JEE (archivée dans un war)
  - Contient une ou plusieurs classes implantant chacune un Web service REST (ressources racines)

- Création d'une application JAX-RS :
  1. Création d'une **application web JEE**
  2. Création d'une ou de plusieurs classes représentant les **ressources racines**
  3. **Configuration de l'URI** de base et des URIs associées à chacun des web services

5. JAX-RS, côté serveur - b. Création d'un service RESTful

Règles pour l'écriture d'une **classe représentant une ressource REST racine** :

- La classe doit être **annotée par** `@javax.ws.rs.Path`. `Path` indique le chemin relatif (bout de l'URL) du web service
- La classe doit être publique et disposer d'un constructeur public. La classe ne doit être ni `final` ni `abstract`
- La classe ne doit pas définir une méthode `finalize()`
- Le service doit être un **objet sans état** et ne doit pas garder des informations sur ses clients entre les appels
- Pour ajouter des **capacités EJB** au Web Services, la classe doit être **annotée soit par** `@javax.ejb.Stateless` **ou** `@javax.ejb.Singleton`

5. JAX-RS, côté serveur - b. Création d'un service RESTful

Principales annotations

Annotation	Descritpion
`@PATH(`$chemin$`)`	Placée au niveau de la classe ($e.g.$ dept), indique le chemin relatif du web service (http://.../dept). Placée au niveau d'une méthode ($e.g.$ create()), indique le sous-chemin vers la méthode (http://.../dept/create).
`@GET`	Indique que la méthode est invoquée en réponse à une requête HTTP `GET`
`@POST`	Indique que la méthode est invoquée en réponse à une requête HTTP `POST`
`@PUT`	Indique que la méthode est invoquée en réponse à une requête HTTP `PUT`
`@DELETE`	Indique que la méthode est invoquée en réponse à une requête HTTP DELETE
`@Produces(`MT`)`	Indique le type MIME ($i.e.$ MT) de la ressource renvoyée par la méthode
`@Consumes(`MT`)`	Indique le type MIME ($i.e.$ MT) consommé par la méthode
`@PathParam(`p`)`	Permet d'extraire un paramètre p à partir d'une URI

5. JAX-RS, côté serveur - d. Configuration d'une application JAX-RS

- Application JAX-RS : peut contenir plusieurs ressources racines (plusieurs services REST), chacune représentée par une classe différente
- **Configuration d'une application JAX-RS** :
  - Configuration de l'**URI de base** de l'application
  - **Enregistrement des ressources racines**, pour que le conteneur sache vers quelle ressource racine orienter les requêtes HTTP
- Deux méthodes :
  - Modification à la main du fichier `web.xml`
  - Alternativement, en créant et en annotant une classe fille de `javax.ws.rs.core.Application`

5. JAX-RS, côté serveur - d. Configuration d'une application JAX-RS

- Exemple :

```
@ApplicationPath("AppConf")
public class MaConfig extends Application {
  @Override
  public Set< Class<?> > getClasses() {
     final Set< Class<?> > classes = new HashSet<>();
     //Enregistrement des ressources racines
     classes.add(BonjourFromRest.class);
     classes.add(UnAutresService.class);
   return classes;
  }
}
```

- L'URI de base est *http://monserveur/nomAppliWebJEE/AppConf/*
- Le service BonjourFromRest de l'application est accessible via l'URI *http://monserveur/nomAppliWebJEE/AppConf/bonjour*

6. JAX-RS, côté client - a. Classes et interfaces

Principales classes et interfaces utilisées côté client

Classe/Interface	Descritpion
`Client`	Point d'entrée du service. Permet de construire et d'exécuter les requêtes HTTP
`ClientBuilder`	Permet d'obtenir et d'initialiser un objet `Client`
`WebTarget`	Permet de lier le client à un web service REST ou à l'une de ses ressources (identifiés par des URIs)
`Invocation`	Un objet Invocation est une requête HTTP préparée et prête à l'envoi

6. JAX-RS, côté client - b. Création d'un client

Création d'un client et connexion à un web service REST

```
...
String aQui = "isitcom";
String BASE_URI = "http://localhost:8080/BonjourFromRest/AppConf";

//Création d'un objet Client du service
Client client = ClientBuilder.newClient();

//Construction de l'URI de la ressource à manipuler
WebTarget webTarget = client.target(BASE_URI).path("bonjour");
WebTarget ressource = webTarget.path(aQui);

//Préparation de la requête en indiquant la méthode HTTP à appliquer
//et le type MIME attendu en retour
Response resp = ressource.request("text/plain").get();

System.out.println(resp.readEntity(String.class));
...
```

6. JAX-RS, côté client - c. Sérialisation

- Contrairement à JAX-WS, JAX-RS ne créé pas automatiquement de classes pour sérialiser les réponses dans les types appropriés.
- JAX-RS fournit uniquement une seule classe générique de type `Response` pour réceptionner les résultats.
- La création d'objets spécifiques (Ex. `Emp, Dept`) est à la charge du programmeur
- JAX-RS fournit néanmoins le xsd des données échangées, accessible via un lien spécifié au niveau du WADL du service.
- En général, l'URI prend la forme : `URI_Serveur/application.wadl/xsd0.xsd`
- JAXB pourrait être utilisé pour la création des classes permettant de sérialiser les objets résultats à partir du schéma XML.

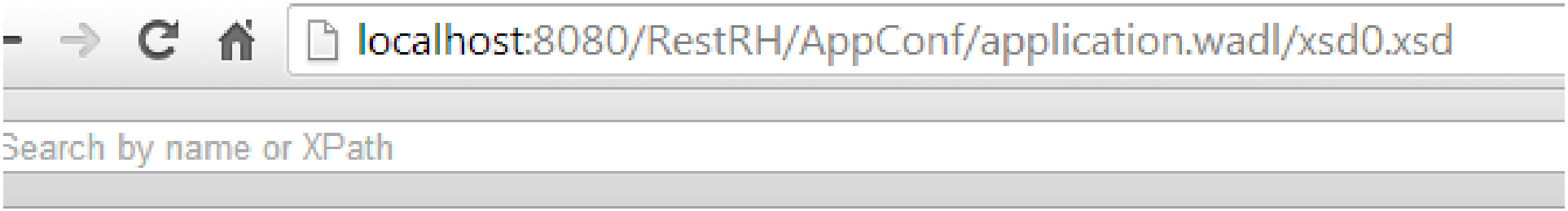
6. JAX-RS, côté client - c. Sérialisation

WADL


```
<application xmlns="http://wadl.dev.java.net/2009/02">
   <doc jersey:generatedBy="Jersey: 2.0 2013-05-14 20:07:34" xmlns:jersey
   <grammars>
      <include href="application.wadl/xsd0.xsd">
         <doc title="Generated" xml:lang="en"/>
      </include>
   </grammars>
   <resources base="http://localhost:8080/RestRH/AppConf/">
      <resource path="dept">
         <method id="create" name="POST">
            <request>
               <ns2:representation element="dept" mediaType="application/x
               <ns2:representation element="dept" mediaType="application/j
            </request>
```

6. JAX-RS, côté client - c. Sérialisation

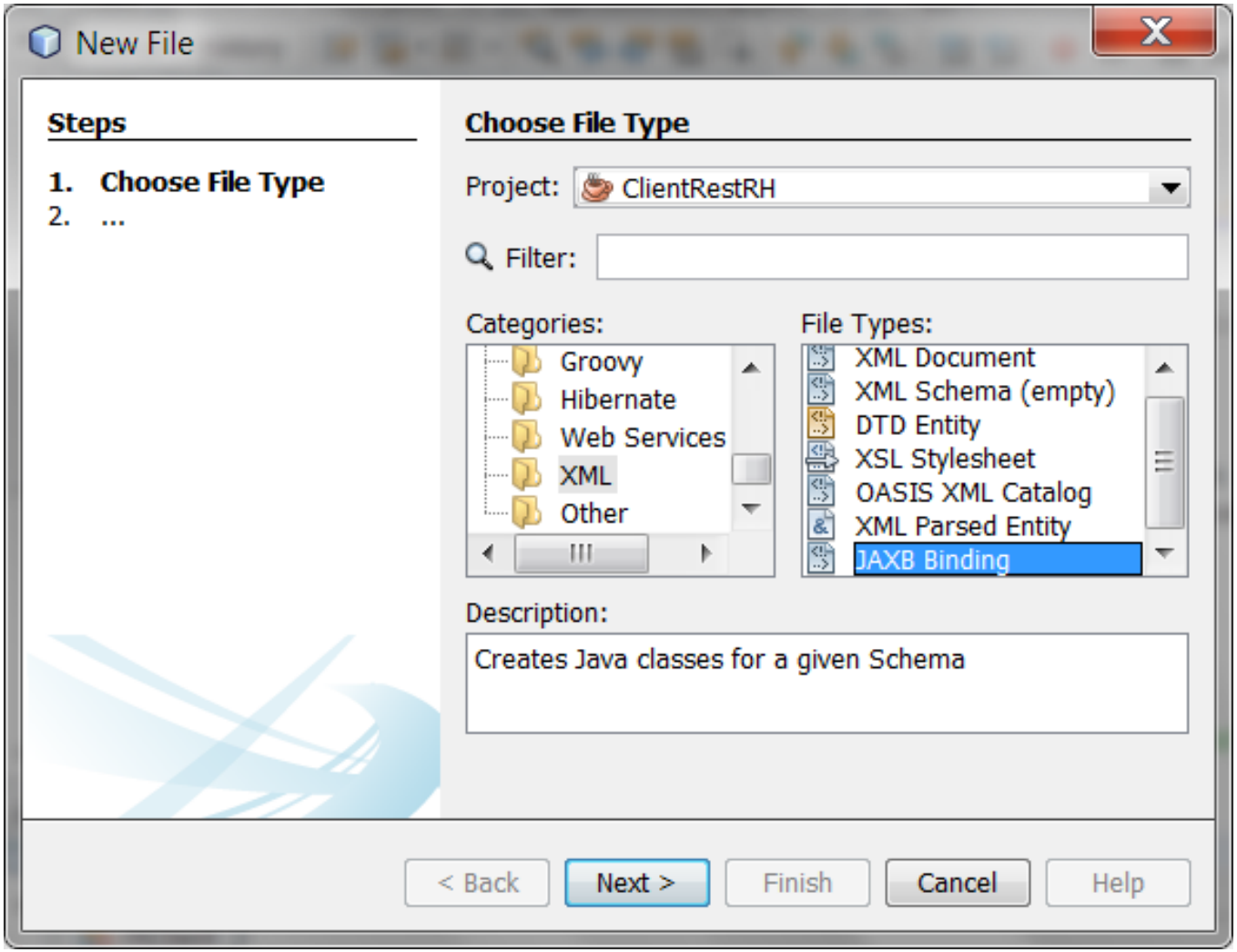
XSD




```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
   <xs:element name="dept" type="dept" />
   <xs:element name="emp" type="emp" />
   <xs:complexType name="dept">
      <xs:sequence>
         <xs:element name="iddept" type="xs:int" minOccurs="0" />
         <xs:element name="nomdept" type="xs:string" minOccurs="0" />
      </xs:sequence>
```

6. JAX-RS, côté client - c. Sérialisation

Génération des classes à partir du schéma xsd avec JAXB



6. JAX-RS, côté client - d. Exemple Client

Les classes créées avec JAXB peuvent être utilisées dans le client REST comme suit :

```
public class RHClient {
    private final WebTarget webTarget;
    private final Client client;
    private static final String BASE_URI = "http://localhost:8080/RestRH/AppConf";

    public RHClient() {
        client = javax.ws.rs.client.ClientBuilder.newClient();
        webTarget = client.target(BASE_URI).path("dept");
    }

   public Response create(Dept d) throws ClientErrorException {
       return webTarget.request(MediaType.APPLICATION_XML)
               .post(Entity.entity(d, MediaType.APPLICATION_XML), Response.class);
    }

   public List<Dept> retrieveAlldepts() {
    WebTarget resource = webTarget;
        List<Dept> depts = resource.request(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Dept>>(){});
    return depts;
   }
```

Chapitre 8 - Notes sur SOAP et REST

4. Exercice

En vous aidant des assistants de NetBeans :

1. Créez une BD Apache Derby (Java DB), appelée RH. Dans cette base créez une table Dept et insérez-y quelques tuples.
2. Créez une application Web
3. Dans la nouvelle application créez un paquetage entity contenant la classe entité Dept et un paquetage dao contenant l'EJB permettant de réaliser les opérations CRUD sur la classe entité
4. Dans un paquetage service, créez un web service REST et un web service SOAP permettant de réaliser les opérations CRUD sur l'entité Dept
5. Créez une application JSE cliente du web service SOAP
6. Créez une autre application JSE cliente du web service REST. Créez dans cette application une classe Dept à partir du schéma XSD fournit par JAX-RS