

Persistence en Java Objectifs

Objectifs

- Connaître les principales classes de JDBC
- Savoir configurer des connexions vers les sources de données
- Savoir interroger une base de données avec JDBC
- Connaître les concepts des ORM
- Savoir configurer une entité avec JPA
- Utiliser les entités avec JPA

Chapitres

0.Objectifs

1.Rappels sur les bases de données

2.Persistance en Java avec JDBC

3.Persistance en Java avec JPA

copyleft

Professionnalisati
on des métiers
informatiques

Support de formation créé par
Franck SIMON

<http://www.franck-simon.com>



Cette œuvre est mise à disposition sous licence
Attribution

Pas d'Utilisation Commerciale

Partage dans les Mêmes Conditions 3.0 France.

Pour voir une copie de cette licence, visitez

<http://creativecommons.org/licenses/by-nc-sa/3.0/fr/>

ou écrivez à

Creative Commons, 444 Castro Street, Suite 900,
Mountain View, California, 94041, USA.

Persistence avec Java

Rappels sur les bases de données

Verrouillages

- Plusieurs utilisateurs peuvent accéder en même temps à une même base de données
 - lectures, écritures, suppressions
- Le verrouillage consiste à bloquer l'accès à un niveau de données pour toutes les requêtes
 - pour un temps déterminé : la requête en cours
 - le niveau de verrouillage peut-être
 - la table, la vue, l'enregistrement
 - dépend de la base de données

Gestion de la concurrence

- Les traitements métiers nécessitent une série d'interactions avec l'utilisateur
- Ces interactions sont entrecoupées d'accès à la base de données
- Dans les applications distribuées, une transaction en base ne doit pas se dérouler sur plusieurs interactions avec l'utilisateur

Les transactions

- Une transaction représente une série de traitements élémentaires
 - un SELECT, une série de UPDATE, un batch de mise à jour de données, ...
 - exemple : virement de 100 € d'un compte courant vers un compte épargne
 - étape 1 : débit de 100 € du compte courant
 - étape 2 : crédit de 100 € vers le compte épargne
 - que ce passe-t-il si un problème survient entre les étapes 1 et 2 ?

Les transactions

- Une transaction doit respecter les propriétés ACID
 - **A**tomacité
 - **C**ohérence
 - **I**solation
 - **D**urabilité
- Exemple : processus de réservation d'un trajet de train
Paris → Carnac
 - pas de train direct => deux trajets
 - Paris → Rennes et Rennes → Carnac
 - paiement du billet

Les transactions

- **Atomicité** : la transaction doit être entièrement réalisée, sinon elle est annulée
 - c'est une unité de travail unique et indivisible
- Exemple du trajet Paris → Carnac
 - la réservation Paris → Rennes a été effectué
 - la réservation Rennes → Carnac **n'a pas pu être effectué**
 - la transaction est **abandonnée** et la réservation Paris → Rennes n'est pas enregistrée dans la base

Les transactions

- **Cohérence** : l'état de la base est cohérente si toutes les contraintes d'intégrité sur les données sont satisfaites
 - une transaction fait passer une base de données d'un état cohérent à un nouvel état cohérent
- Exemple du trajet Paris → Carnac
 - si une défaillance a pour conséquence que le trajet Rennes → Carnac ne soit pas enregistré
 - le client possède son billet mais la réservation est incomplète en base

Les transactions

- **Isolation** : une transaction en doit pas montrer son effet aux autres transactions avant sa validation
 - durant la transaction, les données ne peuvent pas être utilisées par une autre transaction
- Exemple du trajet Paris → Carnac
 - que ce passe-t-il si une réservation R1 prend la dernière place sur le trajet Paris → Rennes ?
 - une transaction concurrente R2 sur ce même trajet ne peut pas connaître l'état de la base avant que la réservation R1 ne soit validée
- Le standard SQL définit 4 niveaux d'isolation

Les transactions

- **Durabilité** : une fois la transaction validée, la base reste dans un état cohérent même en cas de défaillance du système
- Exemple du trajet Paris → Carnac
 - une fois le commit effectué, aucune défaillance ne peut changer l'état de la base
 - que se passerait-il si le client avait son billet mais qu'une défaillance d'un des trajets disparaissait ?

Niveaux d'isolation des transactions

- Les transactions concurrentes peuvent générer des phénomènes indésirables (violations)
 - lecture sale - dirty read
 - une transaction lit des données écrites par une transaction non validée concurrente
 - lecture non reproductible - non-repeatable read
 - une transaction relit des données qu'elle a lu précédemment et ces données ont été modifiées par une autre transaction (validée depuis la lecture initiale)
 - lecture fantôme - phantom read
 - une transaction ré-exécute une requête sur une condition de recherche, et trouve un nombre de ligne différent du fait de la validation d'une autre transaction

Dirty Read

Transaction A

UPDATE voyageurs SET
age=12 WHERE id=1;

rollback

Transaction B

SELECT * FROM voyageurs
WHERE id=1;

L'enregistrement de la
transaction B est "sale"

Non-Repeatable Read

Transaction A

SELECT * FROM voyageurs
WHERE id=1;

SELECT * FROM voyageurs
WHERE id=1;

Les deux lectures renvoient
des valeurs différentes pour
le même enregistrement

Transaction B

UPDATE voyageurs SET
age=12 WHERE id=1;

commit

Phantom Read

Transaction A

SELECT * FROM voyageurs
WHERE age>10 AND age<20;

SELECT * FROM voyageurs
WHERE age>10 AND age<20;

Les deux requêtes renvoient
un nombre d'enregistrements
différent

Transaction B

INSERT INTO voyageurs
(age) VALUES (18);

commit

Niveaux d'isolation des transactions

- SQL définit 4 niveaux d'isolation afin d'empêcher les violations précédentes

Niveau d'isolation	Lecture sale	Lecture non reproductible	Lecture fantôme
Read Uncommitted Lecture de données non validées	Possible	Possible	Possible
Read Committed Lecture de données validées	Impossible	Possible	Possible
Repeatable Read Lecture répétée	Impossible	Impossible	Possible
Serializable Sérialisable	Impossible	Impossible	Impossible

Niveaux d'isolation des transactions

- Consultez la documentation de votre base de données pour connaître le niveau d'isolation par défaut
 - ex. : PostgreSQL : Read Committed
- Plus le niveau d'isolation est élevé, moins la performance est présente
 - à confirmer par la documentation de votre base
 - Serializable émule les exécutions des transactions en série plutôt que parallèlement
 - les applications doivent prendre en compte les tentatives de ré-essai en cas d'échec de la transaction

Persistence en Java JDBC

Objectifs

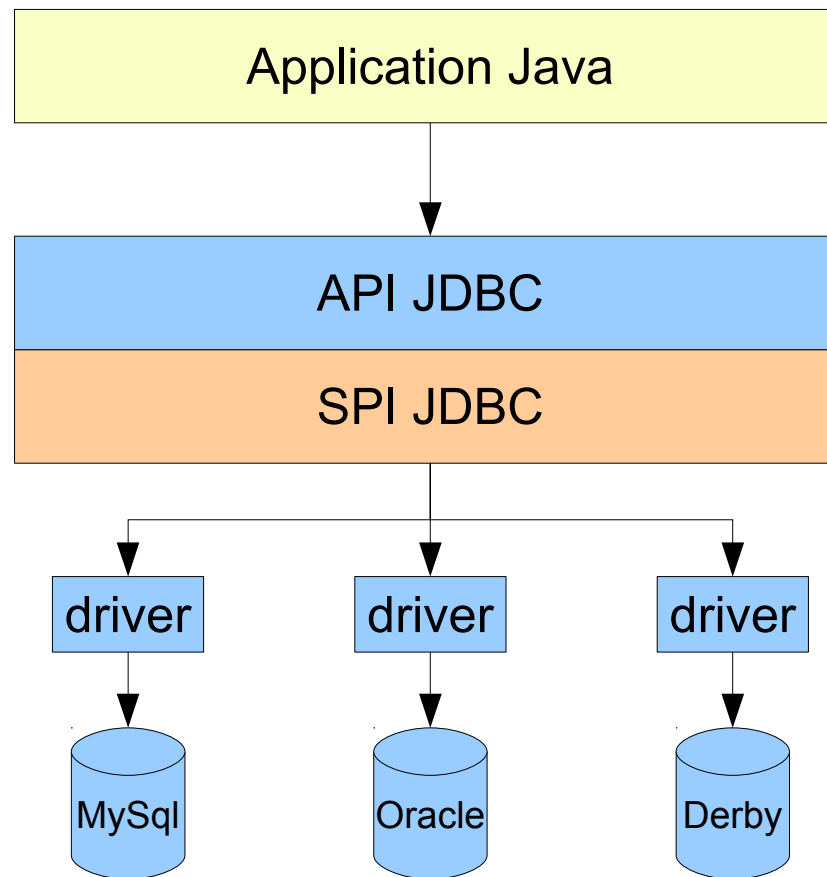
- Comprendre l'architecture JDBC
- Connaître les classes principales de JDBC
- Savoir interroger une base de données avec JDBC
- Savoir gérer les transactions avec JDBC

Introduction

- JDBC : Java DataBase Connectivity
 - API Java pour l'accès aux sources de données
 - actuellement version 4.2 avec Java SE 8
 - permet l'interrogation uniforme des bases de données
 - packages : `java.sql` et `javax.sql`
- Langage d'interrogation des bases de données : SQL
 - Structured Query Language

Introduction

- Architecture de base



API : Application Programming Interface
SPI : Service Provider Interface

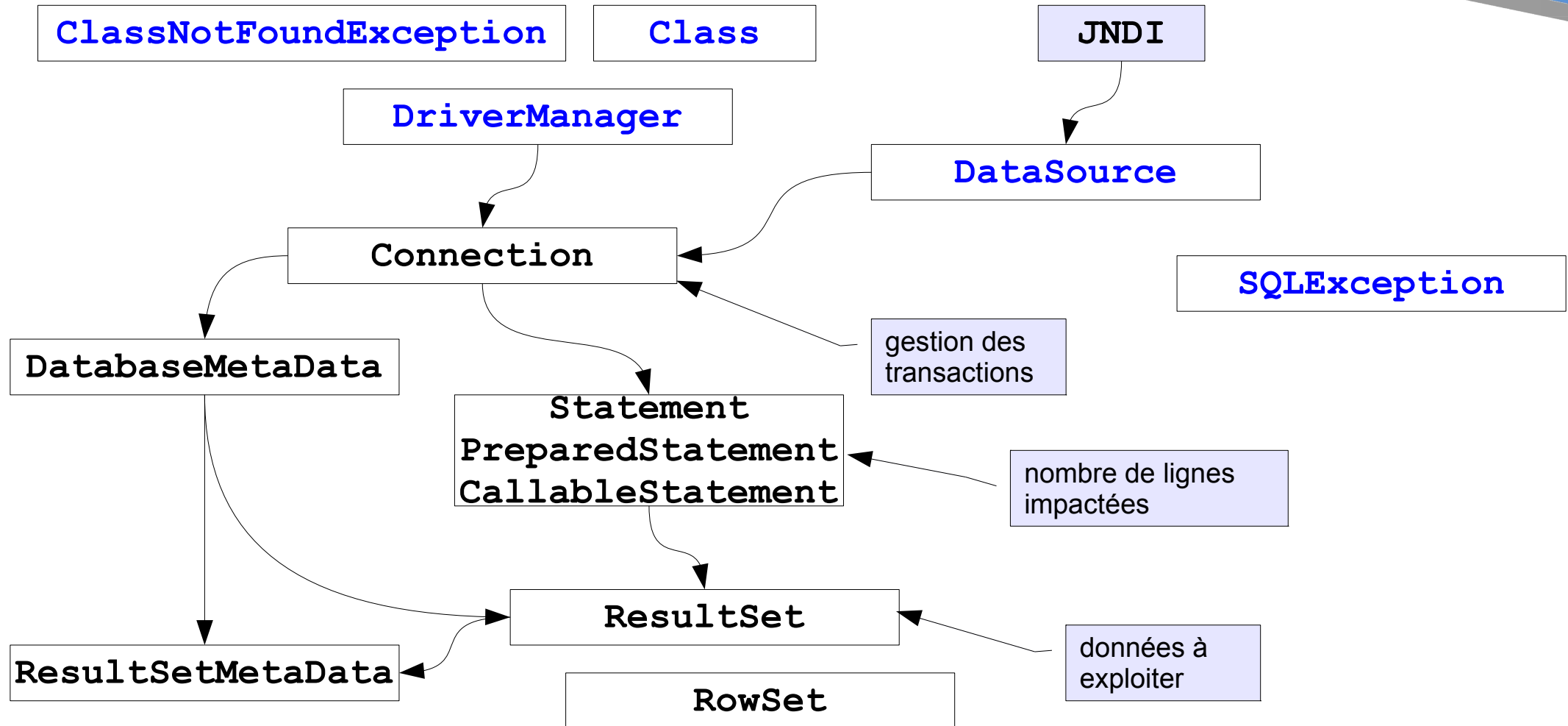
Les drivers

- Java ne fournit aucun driver
 - seulement un pont JDBC-ODBC
 - ODBC : Open DataBase Connectivity
 - ce sont les éditeurs de bases de données qui fournissent les drivers
- Types de driver
 - type 1 : pont entre JDBC et une autre API
 - type 2 : communication avec le SGBD en code natif
 - type 3 : pont réseau - communication avec le SGBD via une serveur intermédiaire
 - type 4 : 100% Java

Les classes à connaître

- DriverManager
 - gère les instances de driver
- Connection
 - gère la connexion vers la base
- Statement
 - gère l'environnement de requête
- ResultSet
 - résultat d'une requête

Les classes à connaître



Démarche à suivre

- Il n'y a aucune classe à instancier
 - les méthodes renvoient les instances
- Étapes
 1. charger le driver
 2. obtenir une `Connection`
 3. obtenir un `Statement` qui contient la requête SQL
 4. exécuter la requête et récupérer le `ResultSet`
 5. rendre la `Connection`

Étape 1 : charger le driver

- Il s'agit d'enregistrer le driver fourni par l'éditeur de la base de données auprès du `DriverManager`
 - le nom du driver est souvent externalisé dans un fichier de paramètres
 - nom complet : package + nom de la classe
 - utilisation de la classe `Class` pour charger le driver
 - peut générer une `ClassNotFoundException`
 - le code statique du driver permet l'enregistrement auprès du `DriverManager`

```
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";  
Class.forName(driverName);
```

Étape 1 : chargement du driver

- Lorsque l'application est exécutée dans un serveur
 - la méthode précédente n'est pas utilisée
 - une `DataSource` est configurée dans le serveur
 - configuration propre à chaque serveur
 - le développeur retrouve la source de données par une interrogation JNDI ou par CDI
 - JNDI : Java Naming and Directory Interface
 - CDI : Context and Dependency Injection

Étape 2 : obtention d'une Connection

- L'instance de Connection est obtenu
 - auprès du DriverManager
 - ou de la DataSource si l'application est exécutée dans un serveur
- Dans les deux cas il est nécessaire d'avoir
 - l'URL de connexion à la base de données
 - les identifiants et mots de passe de connexion à la base
 - dans le cas de la DataSource, ces éléments sont dans le fichier de configuration de la source de données

Étape 2 : obtention d'une Connection

- L'URL de connexion est constituée de trois parties
 - **protocole:sous-protocole:infos-complémentaires**
 - protocole : toujours jdbc
 - sous-protocole : dépend de la base, correspond au driver utilisé
 - infos-complémentaires : chaîne de connexion à la base, dépend de la base utilisée
 - exemples

```
jdbc:odbc:cds_et_dvds
```

```
jdbc:derby://localhost:1527/D:/SERVEURS/db-derby-10.10.1.1-bin/databases/france
```

```
jdbc:mysql:///france?  
user=root&password=password
```

```
jdbc:mysql://localhost:3306/  
bovoyage
```


Étape 2 : obtention d'une Connection

- Obtention de la connexion
 - par le DriverManager

```
Connection connection = DriverManager.getConnection(url,user,password);
```

- par la DataSource
 - l'instance `maDataSource` a été récupérée via JNDI ou CDI

```
Connection connection = maDataSource.getConnection(url,user,password);
```

Étape 3 : obtenir un environnement de requête

- Type de déclaration de requête
 - `Statement` : requête SQL simple, sans paramètres à passer au moment de la requête
 - ou avec les paramètres construits

```
SELECT * FROM  
destinations
```

- `PreparedStatement` : requêtes SQL avec paramètres passés au moment de la requête

```
SELECT * FROM destinations WHERE kp_destination=?
```

- `CallableStatement` : pour les procédures stockées

Étape 4 : exécuter la requête

- Avec un Statement

```
String sql = "SELECT * FROM destinations";  
Statement statement = connection.createStatement();  
ResultSet rs = statement.executeQuery(sql);
```

- Avec un PreparedStatement

```
String sql = "SELECT * FROM dates_voyages WHERE ke_destination=?";  
PreparedStatement st = con.prepareStatement(sql);  
st.setInt(1, id);  
ResultSet rs = st.executeQuery();
```

paramètre

position et
valeur du
paramètre

Étape 4 : exécuter la requête

- Le `ResultSet` correspond à la table résultat de la requête
 - le curseur du `ResultSet` est positionné avant la première ligne de la table
 - la méthode `next()` renvoie `true` si il y a une ligne suivante, et passe à cette ligne automatiquement
 - une simple boucle `while` permet d'exploiter le `ResultSet`

```
ResultSet rs = st.executeQuery();
while(rs.next())
{
    DatesVoyage s = this.construireSejour(rs);
    sejours.add(s);
}
```

Étape 4 : exécuter la requête

- Les différents champs sont récupérés par leur type et le nom de la colonne

```
private DatesVoyage construireSejour(ResultSet rs) throws SQLException
{
    DatesVoyage s = new DatesVoyage();
    s.setDepart(rs.getDate("date_depart"));
    s.setRetour(rs.getDate("date_retour"));
    s.setPrix(rs.getDouble("prixHT"));
    s.setId(rs.getInt("kp_date_voyage"));
    return s;
}
```

Étape 5 : rendre la connexion

- Une fois le traitement exécuter, il faut rendre la connexion.
 - méthode close sur l'instance de `Connection`

```
connection.close();
```
 - si la connexion a été reçu du `DriverManager`, elle est physiquement fermée
 - si la connexion provient d'une source de donnée, elle retourne dans le pool de connexion

Autre méthodes

- Insertion, modification ou suppression d'enregistrement
 - `statement.executeUpdate()`
 - retourne le nombre de lignes impactées
 - peut dépendre de la base de données
 - pour récupérer l'identifiant généré en base
 - ajouter `Statement.RETURN_GENERATED_KEYS` au `prepareStatement`
 - interroger le `Statement` par `getGeneratedKeys()`

Insertion et récupération de la clé primaire

- Extrait

```
public void create(User user) throws SQLException {
    try (
        Connection connection = dataSource.getConnection();
        PreparedStatement statement = connection.prepareStatement(SQL_INSERT,
            Statement.RETURN_GENERATED_KEYS);
    ) {
        statement.setString(1, user.getName());
        statement.setString(2, user.getPassword());
        statement.setString(3, user.getEmail());
        // ...

        int affectedRows = statement.executeUpdate();

        if (affectedRows == 0) {
            throw new SQLException("Creating user failed, no rows affected.");
        }

        try (ResultSet generatedKeys = statement.getGeneratedKeys()) {
            if (generatedKeys.next()) {
                user.setId(generatedKeys.getLong(1));
            }
            else {
                throw new SQLException("Creating user failed, no ID obtained.");
            }
        }
    }
}
```


Transactions

- Les transactions permettent de :
 - confirmer un changement (commit)
 - de revenir en arrière si une erreur apparaît (rollback)
 - de positionner des points de restauration (savepoint)

Transactions

- La stratégie de verrouillage dépend de la base de données utilisée
 - ne dépend pas de JDBC
- Le niveau d'isolation des transactions peut être spécifié à JDBC
- La connexion est créée en mode auto-commit
 - mode par défaut
 - chaque requête SQL exécutée comme une transaction qui est automatiquement confirmée

Transactions

- Pour permettre l'exécution de plusieurs requêtes dans une seule transaction il faut annuler le mode auto-commit

```
connection.setAutoCommit(false);
```

- Puis lancer l'ensemble des requêtes composant la transaction

- si succès :

```
connection.commit();
```

- si échec :

```
connection.rollback();
```

Transactions

- Pour permettre l'exécution de plusieurs requêtes dans une seule transaction il faut annuler le mode auto-commit

```
connection.setAutoCommit(false);
```

- Puis lancer l'ensemble des requêtes composant la transaction

- si succès :

```
connection.commit();
```

- si échec :

```
connection.rollback();
```

Transaction

```

...
try {
    con.setAutoCommit(false);
    updateSales = con.prepareStatement(updateString);
    updateTotal = con.prepareStatement(updateStatement);

    //initialisation des paramètres de requêtes
    ...
    updateSales.executeUpdate();
    updateTotal.executeUpdate();
    con.commit();
} catch (SQLException e ) {
    if (con != null) {
        try {
            System.err.print("Transaction is being rolled back");
            con.rollback();
        } catch (SQLException excep) {
            System.err.print(excep);
        }
    }
} finally {
    if (updateSales != null)
        updateSales.close();
    if (updateTotal != null)
        updateTotal.close();
    con.setAutoCommit(true);
}
...

```

Transactions

- Utilisation des isolations de la base de données

```
connection.setTransactionIsolation(level);
```

- où `level` est un `static int` de `Connection` :
 - `TRANSACTION_READ_UNCOMMITTED`
 - `TRANSACTION_READ_COMMITTED`
 - `TRANSACTION_REPEATABLE_READ`
 - `TRANSACTION_SERIALIZABLE`
- il est possible de récupérer auprès de la base son niveau d'isolation

```
DatabaseMetaData.supportsTransactionIsolationLevel(level).getTransactionIsolation();
```

Transactions

- Mise en place de points de restauration
 - méthode `Savepoint Connection.setSavePoint()`
- Annulation jusqu'au point de restauration
 - méthode `Connection.rollback(savepoint)`

```
...  
Savepoint save1 = connection.setSavepoint();  
...  
connection.rollback(save1);  
...
```

Persistance en Java JPA

Introduction

- JPA - Java Persistence API
 - spécification ORM (Object Relational Mapping) Java
 - JPA - JSR 220
 - JPA 2.0 - JSR 317
 - JPA 2.1 - JSR 338
 - package : `javax.persistence`
 - implémentations
 - Hibernate
 - EclipseLink
 - DataNucleus
 - OpenJPA
 - ...

Introduction

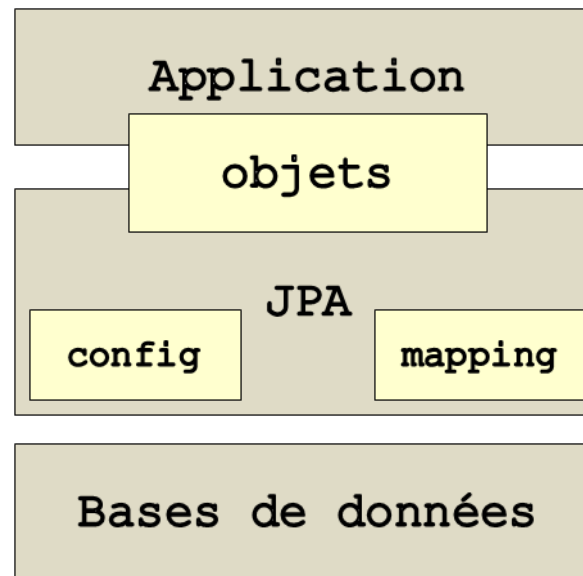
- Contrairement aux EJB 2 Entity, JPA ne fait pas partie de la spécification JAVA EE (EJB 3)
- JPA peut-être utilisé dans une application Java SE ou Java EE
 - le serveur Java EE fournit l'implémentation JPA
 - Java SE ne fournit pas d'implémentation JPA

Objectifs de JPA

- Fournir une vue orienté Java de la persistance
 - approche standardisée
 - utilisation des meilleurs concepts de Hibernate, JDO
 - JDO : Java Data Objects
- Travailler avec des POJO
- Supporter les concepts objets
 - polymorphisme, héritage
- Éliminer le codage des requêtes SQL
 - requêtes en JPQL, Criteria
 - JPQL : Java Persistence Query Language

Architecture JPA

- JPA implémente la couche d'accès aux données
 - retourne des objets du domaine comme réponse aux requêtes
 - persiste les objets du domaine



Architecture JPA

- Le mapping est effectué à l'aide d'annotations sur le POJO
 - peut-être effectué via un fichier XML

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String civilite;
    private String nom;
    private String prenom;
    ...
}
```

Architecture JPA

- La configuration est effectuée par le fichier *META-INF/persistence.xml*

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="jpa">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/persistence"/>
      <property name="hibernate.connection.username" value="toto"/>
      <property name="hibernate.connection.password" value="totopw"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

Entité JPA

- Écrire la classe Java qui suit la spécification JavaBean
 - fournir un constructeur sans argument
 - chaque propriété est associée à des méthodes get/set
- Fournir une propriété représentant la clé primaire
 - c'est avec cette propriété que l'ORM maintient le lien entre l'instance et l'enregistrement en base

Entité JPA

- Annoter la classe avec `@Entity`
- Annoter les propriétés
 - l'identifiant avec `@Id`
 - par défaut toutes les propriétés sont persistées
 - annoter avec `@Transient` si une propriété ne doit pas être persistée
 - les annotations peuvent être mise sur les propriétés ou les méthodes get/set
 - attention il peut y avoir des différences de comportement en fonction des implémentations

Entité JPA

- Au minimum deux annotations sont nécessaires
 - `@Entity` pour marquer la classe
 - `@Id` pour l'identifiant
- Par défaut le nom de la classe correspond au nom de la table
 - sinon utiliser l'annotation `@Table`
- Par défaut les noms des colonnes correspondent aux nom des propriétés
 - sinon utiliser l'annotation `@Column`

EntityManager

- L'utilisation de JPA nécessite une interaction avec un `EntityManager`
 - fournit les opération de création, recherche, mise à jour et de suppression
 - une instance d'`EntityManager` est fournit par le serveur
 - par injection avec l'annotation `@PersistenceContext`
 - si JPA est utilisé dans une application autonome il faut utiliser un `EntityManagerFactory`

EntityManager

- Le conteneur Java EE propose le support des transactions et gère le cycle de vie de l'EntityManager
- Dans une application non Java EE
 - c'est à dire une application Java SE, ou un application web utilisant un conteneur Servlet/JSP
 - les transactions doivent être gérées
 - le cycle de vie l'EntityManager doit géré
 - il doit être fermé en fin d'utilisation

Fichier *persistence.xml*

- Le fichier *persistence.xml* définit :
 - le nom de l'unité de persistance
 - élément `<persistence-unit>`
 - la classe implémentant JPA
 - élément `<provider>`
 - la connexion à la source de donnée
 - via JNDI sur le serveur, avec les éléments `<jta-data-source>` ou `<non-jta-data-source>`
 - ou par les propriétés fournies à l'implémentation de JPA, avec les éléments `<properties>` et `<property>`

Récupération de l'EntityManager

- Hors conteneur Java EE
 - récupération d'un `EntityManager` via un `EntityManagerFactory`
 - on précise le nom de l'unité de persistance définie dans le fichier *persistence.xml*

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

- L'`EntityManagerFactory` doit être fermé à la fin de l'application
 - il est exécuté dans un thread indépendant
 - utiliser un `ApplicationListener` pour les conteneurs Servlet/JSP

Récupération de l'EntityManager

- Exemple avec Java SE

```
public static void main(String[] args) {
    Contact c1 = new Contact("M", "LAGAFFE", "Gaston");
    EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("foo");
    ContactDAO dao = new ContactDAO(emf);
    dao.save(c1);
    emf.close();
}
```

```
public class ContactDAO {
    private EntityManagerFactory emf;

    public ContactDAO(EntityManagerFactory emf){
        this.emf = emf;
    }

    public void save(Contact contact){
        EntityManager em= emf.createEntityManager();
        ...
        em.close();
    }
    ...
}
```

Récupération de l'EntityManager

- Exemple sous Tomcat
 - utilisation d'un ServletContextListener

```
@WebListener
public class ApplicationListener implements ServletContextListener {
    public static final String EMF = "emf";

    @Override
    public void contextInitialized(ServletContextEvent event) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("foo");
        event.getServletContext().setAttribute(EMF, emf);
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        EntityManagerFactory emf = (EntityManagerFactory)
event.getServletContext().getAttribute(EMF);
        emf.close();
    }
}
```

Récupération de l'EntityManager

- Exemple sous Tomcat
 - récupération du factory

```
public class AddContactServlet extends HttpServlet{
    private static final long serialVersionUID = 1L;
    private ContactDAO dao;

    @Override
    public void init() throws ServletException {
        EntityManagerFactory emf = null;
        emf =
        (EntityManagerFactory)this.getServletContext().getAttribute(ApplicationListener.EMF);
        dao = new ContactDAO(emf);
    }
    ...
}
```


Récupération de l'EntityManager

- Dans un serveur Java EE
 - la récupération d'un `EntityManager` est effectuée par injection dans un EJB

```
@PersistenceContext(unitName="jpa") private EntityManager em;
```

- il est automatiquement fermé à la sortie de la méthode

Développer avec JPA

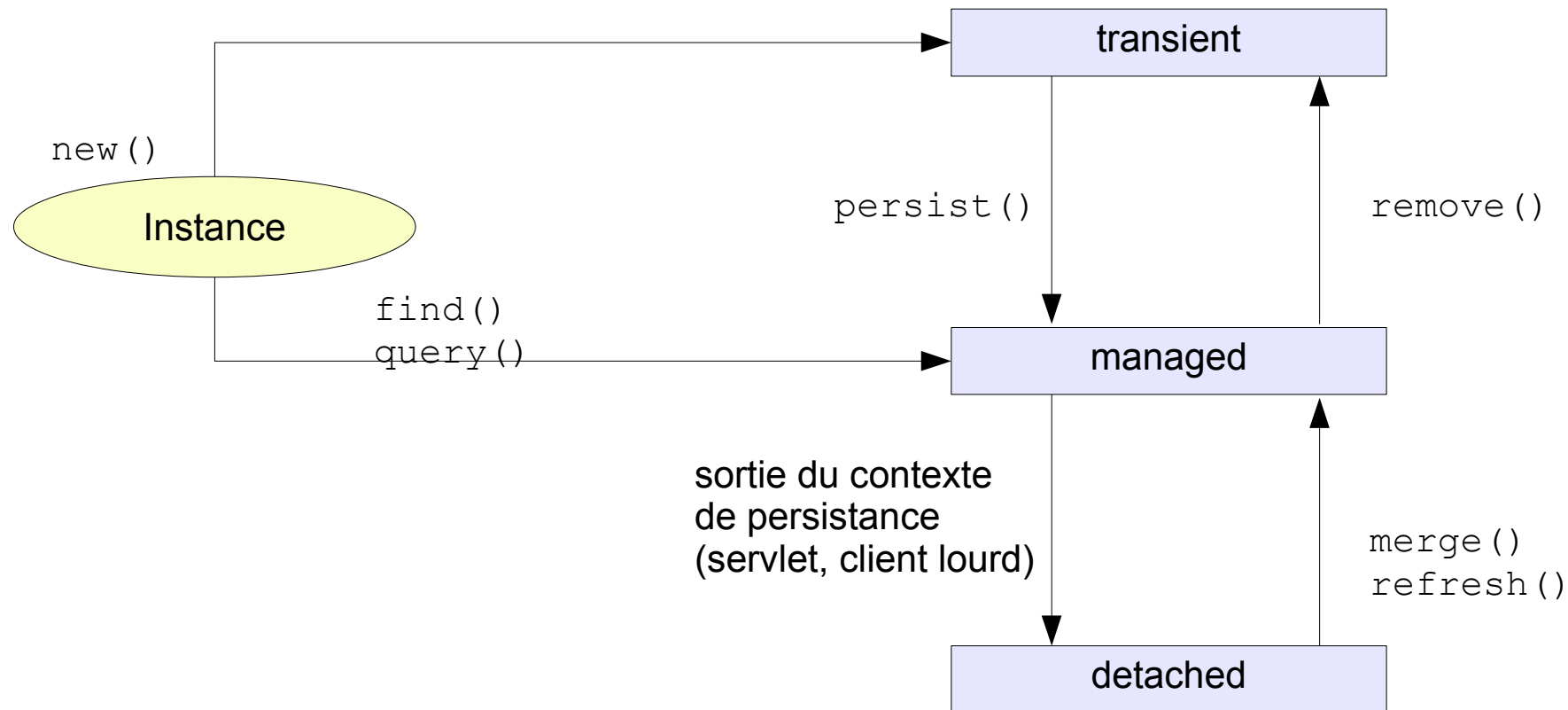
- Toutes les opérations effectuées sur l'`EntityManager` doivent être effectuées dans une transaction

```
...
public void save(Contact contact)
{
    EntityManager em= emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();
    em.persist(contact);
    transaction.commit();
    em.close();
}
...
```

EntityManager

- Opérations de base :
 - sauvegarde des entités avec `persist(...)`
 - récupération d'un objet par son identifiant : `find(...)`
 - suppression d'une entité en base : `remove(...)`
 - mise à jour de la base avec l'objet : `merge()`
 - mise à jour de l'objet avec la base : `refresh()`
 - les modifications sur les objets sont propagées vers la base lors du commit sur la transaction

Cycle de vie des entités



Objets embarqués

- Dans le modèle JPA fait la différence entre les entités et les objets internes
- Les entités ont leur propres identifiant
 - la classe est annotée avec `@Entity`
- Un objet interne à une entité peut ne pas être lui-même une entité
 - c'est un objet embarqué
 - la classe est annotée par `@Embeddable`

Objets embarqués

- L'entité contient un identifiant
 - la classe embarquée n'en contient pas

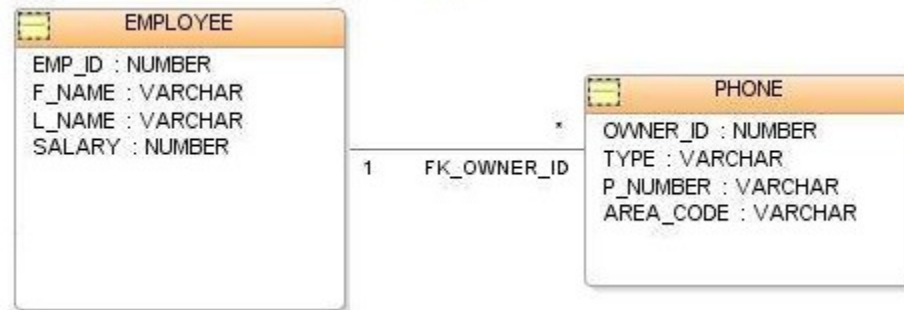


- Une seule table en base contient tous les champs des deux classes

Column Name	Datatype	NOT NULL
id	INT(11)	✓
codePostal	VARCHAR(255)	
rue	VARCHAR(255)	
ville	VARCHAR(255)	
civilite	VARCHAR(255)	
nom	VARCHAR(255)	
prenom	VARCHAR(255)	

Collection de types

- De nombreuses tables comportent des listes de types
 - String, int, double, ...

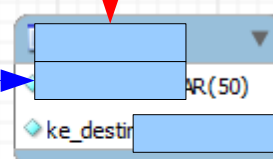
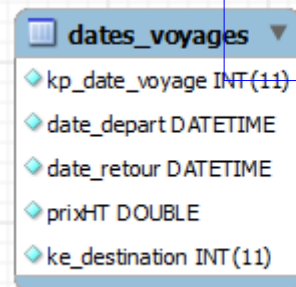


- Le type `String` ne peut pas être marqué comme `@Embeddable`
- Il faut marquer la collection comme `@ElementCollection`

Collection de types

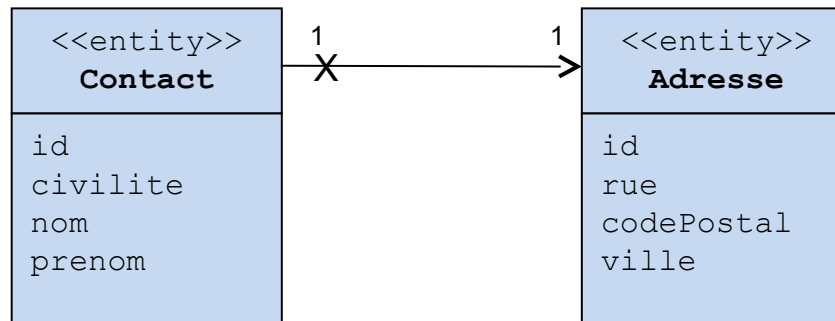
```
@Entity
@Table(name="destinations")
public class Destination {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="kp_destination")
    private long id;
    private String region;

    @ElementCollection
    @CollectionTable(name="images", joinColumns=@JoinColumn(name="ke_dest"))
    @Column(name="image")
    private List<String> images;
    ...
}
```



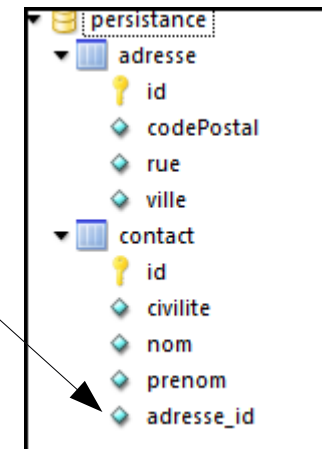
Mapping one-to-one unidirectionnel

- Les deux classes sont des entités
 - chacune possède un identifiant



- Il y a deux tables en base

Clé étrangère sur la
table adresse



Mapping one-to-one unidirectionnel

- La propriété adresse de `Contact` est annotée avec `@OneToOne`
 - si l'attribut cascade n'est pas mis en place il faudra sauver les entités `Adresse` et `Contact`
- L'annotation optionnelle `@JoinColumn` permet de préciser un nom de colonne pour la clé étrangère

Mapping one-to-one unidirectionnel

- La propriété adresse de `Contact` est annotée avec `@OneToOne`
 - si l'attribut cascade n'est pas mis en place il faudra sauver les entités `Adresse` et `Contact`
- L'annotation optionnelle `@JoinColumn` permet de préciser un nom de colonne pour la clé étrangère

Mapping one-to-one unidirectionnel

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String civilite;
    private String nom;
    private String prenom;
    @OneToOne(cascade=CascadeType.ALL)
    private Adresse adresse;
    ...
}
```

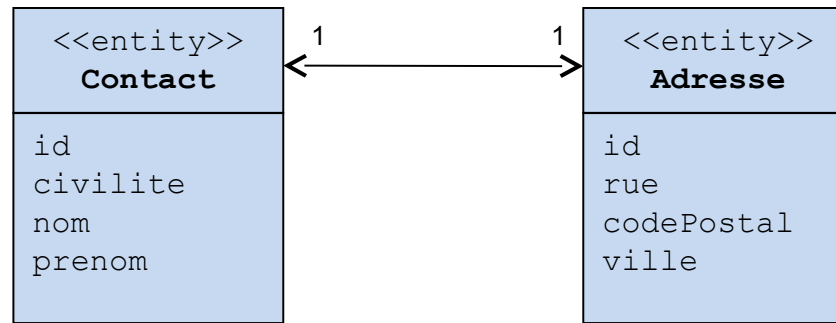
Mapping one-to-one unidirectionnel

- La classe `Adresse` ne contient pas de référence vers `Contact`

```
@Entity
public class Adresse
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String rue;
    private String codePostal;
    private String ville;
    ...
}
```

Mapping one-to-one bidirectionnel

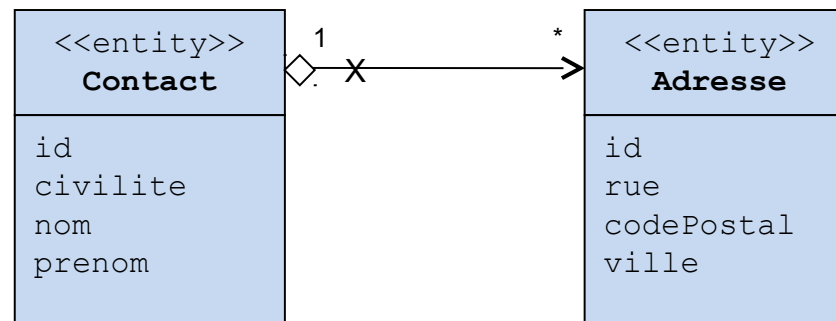
- Chaque entité possède une référence vers l'autre



- Une des extrémité, et seulement une, est responsable de la mise à jour des colonnes de l'association
- l'extrémité non responsable utilise l'attribut `mappedBy` sur l'annotation `@OneToOne`

Mapping one-to-many unidirectionnel

- Au niveau de l'implémentation, les associations un-
vers-plusieurs utilisent des collections
- JPA fournit des mappings pour les interfaces Java
List, Set, Map, ...



Mapping one-to-many unidirectionnel

- Utiliser `@OneToMany` pour mapper la relation
 - l'attribut cascade assure la sauvegarde des entités liées en même temps que l'entité principale
 - l'annotation `@JoinColumn` impose une clé étrangère dans la table des adresses
 - si omis, une table de liaison est créée

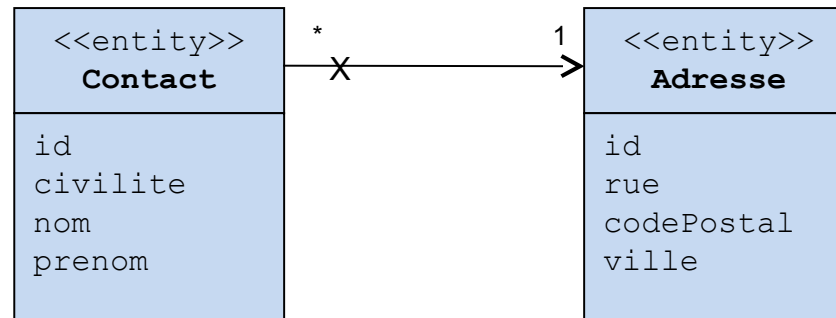
```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String civilite;
    private String nom;
    private String prenom;
    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="ke_contact")
    private List<Adresse> adresses = new ArrayList<Adresse>();
    ...
}
```


Mapping one-to-many unidirectionnel

- Il n'y a pas de description de mapping dans la classe Adresse
- Par défaut la récupération des collection est en mode *eager*.
 - les collections sont récupérées en base lors de l'appel de la méthode `getXxxx()`

Mapping many-to-one unidirectionnel

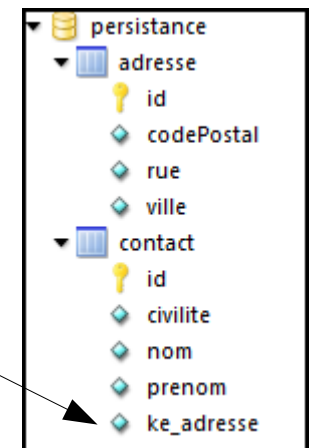
- Les associations plusieurs-vers-un sont déclarées au niveau de la propriété avec l'annotation `@ManyToOne`
 - `@JoinColumn` est optionnel



Mapping many-to-one unidirectionnel

- La liaison est décrite dans Contact

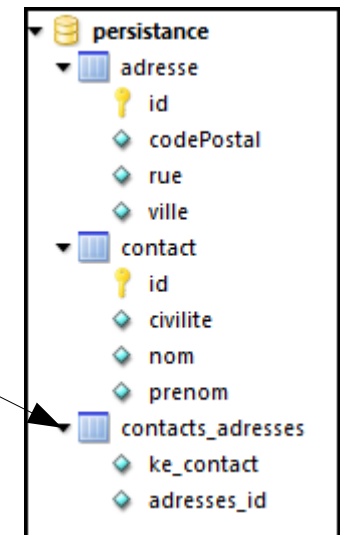
```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String civilite;
    private String nom;
    private String prenom;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "ke_adresse")
    private Adresse adresse;
    ...
}
```



Mapping many-to-many unidirectionnel

- Utilise l'annotation `@ManyToMany`
- L'annotation `@JoinTable` décrit la table de jointure

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String civilite;
    private String nom;
    private String prenom;
    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="contacts_adresses",
        joinColumns={@JoinColumn(name="ke_contact")})
    private List<Adresse> adresses = new ArrayList<Adresse>();
    ...
}
```

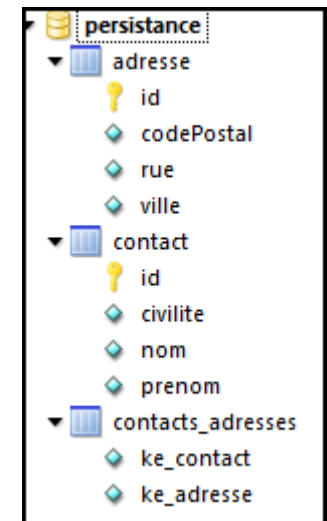


Mapping many-to-many bidirectionnel

- Si l'association est bidirectionnelle
 - une extrémité est considérée comme propriétaire, l'autre est marquée comme inverse

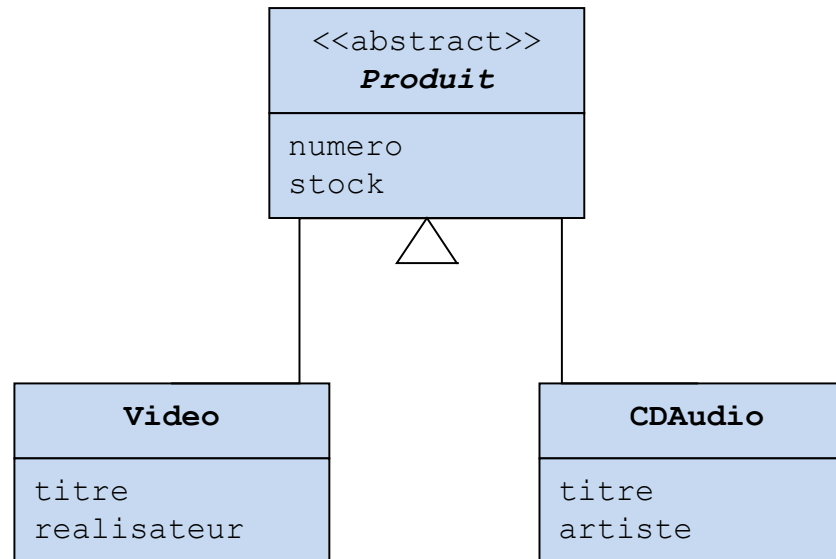
```
@Entity
public class Contact {
...
    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="contacts_adresses",
        joinColumns={@JoinColumn(name="ke_contact")},
        inverseJoinColumns={@JoinColumn(name="ke_adresse")})
    private List<Adresse> adresses = new ArrayList<Adresse>();
...
}
```

```
@Entity
public class Adresse {
...
    @ManyToMany(cascade=CascadeType.ALL,
        mappedBy="adresses")
    private List<Contact> contacts = new ArrayList<Contact>();
...
}
```



Mapper les héritages

- JPA propose trois stratégies pour mapper l'héritage
 - une table par classe concrète
 - une seule table pour la hiérarchie de classe
 - une table par classe fille



Une table par classe concrète

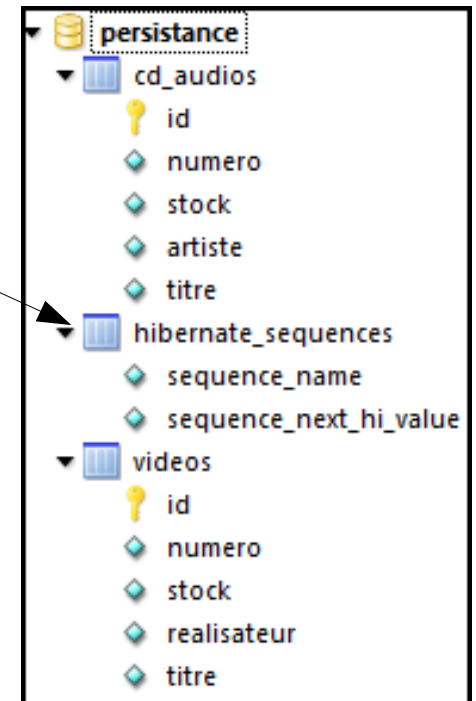
- La stratégie `TABLE_PER_CLASS` est mise en place dans l'annotation `@Inheritance`
- La plupart des ORM implémente cette stratégie en utilisant des UNION SQL pour le polymorphisme
- Le type de stratégie de création de la clé primaire est liée à l'implémentation

Une table par classe concrète

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Produit
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private int id;
    ...
}
```

```
@Entity
@Table(name="cd_audios")
public class CDAudio extends Produit
{
    ...
}
```

```
@Entity
@Table(name="videos")
public class Video extends Produit
{
    ...
}
```



Une table pour la hiérarchie de classes

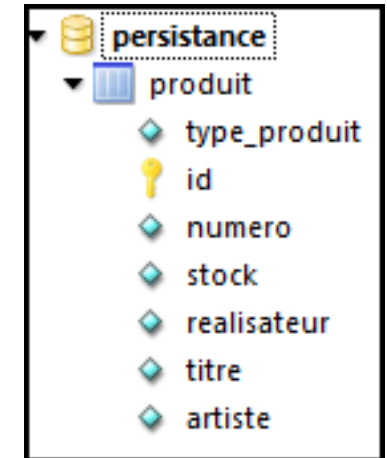
- La stratégie `SINGLE_TABLE` est mise en place dans l'annotation `@Inheritance`
- Une seule table est utilisée
 - requêtes polymorphiques plus rapides
 - colonnes "nullable"

Une table pour la hiérarchie de classes

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type_produit")
public abstract class Produit
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    ...
}
```

```
@Entity
@Table(name="cd_audios")
public class CDAudio extends Produit
{ ...
}
```

```
@Entity
@Table(name="videos")
@DiscriminatorValue(value="video")
public class Video extends Produit
{ ...
}
```



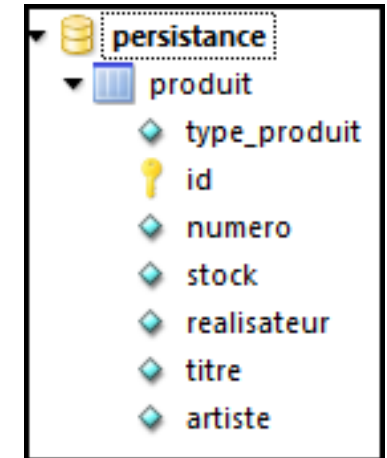
type_produit	id	numero	stock	realisateur	titre	artiste
	1	v1	3	Stanley KUBRICK	Orange mécanique	NULL
CDAudio	2	cd1	10	NULL	The Wall	Pink Floyd

Une table pour la hiérarchie de classes

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type_produit")
public abstract class Produit
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    ...
}
```

```
@Entity
@Table(name="cd_audios")
public class CDAudio extends Produit
{ ...
}
```

```
@Entity
@Table(name="videos")
@DiscriminatorValue(value="video")
public class Video extends Produit
{ ...
}
```



type_produit	id	numero	stock	realisateur	titre	artiste
	1	v1	3	Stanley KUBRICK	Orange mécanique	NULL
CDAudio	2	cd1	10	NULL	The Wall	Pink Floyd

Une table par classe fille

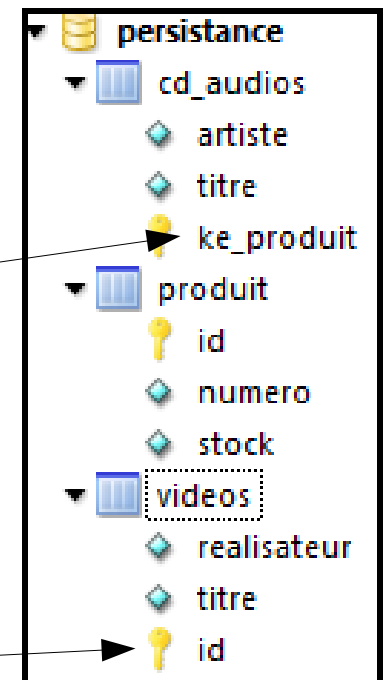
- La stratégie `JOINED` est mise en place dans l'annotation `@Inheritance`
- Une table regroupe les propriétés communes
- Les identifiants des tables de classes filles correspondent aux clés primaires de la classe mère

Une table par classe fille

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Produit
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    ...
}
```

```
@Entity
@Table(name="cd_audios")
@PrimaryKeyJoinColumn(name="ke_produit")
public class CDAudio extends Produit
{ ...
}
```

```
@Entity
@Table(name="videos")
public class Video extends Produit
{ ...
}
```



Langage JPQL

- JPQL : Java Persistence Query Language
- Langage similaire à SQL
 - mais JPQL permet une approche objet du requêtage
 - SQL est basé sur la connaissance des structures de la base
 - JPQL est basé sur la connaissance du modèle objet
- La recherche des adresses est effectuée par
 - `SELECT * FROM table_adresses` en SQL
 - `from Adresse` en JPQL
 - attention certaines implémentations imposent un `SELECT`

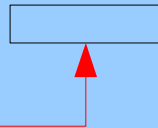
Langage JPQL

- Les requêtes sont créées via l'`EntityManager`
 - `createQuery()` pour les requêtes JPQL
 - `createNamedQuery()` pour les requêtes nommées définies par les annotations
 - `@NamedQueries({...})`
 - `@NamedQuery()`
 - `createNativeQuery()` pour les requêtes SQL
- Les requêtes peuvent comporter des paramètres

Langage JPQL

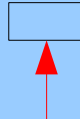
- Exemple de requête avec paramètres

```
public List<Video> getVideoParTitre(String titre)
{
    Query query = em.createQuery("from Produit p where p.titre like :titre");
    query.setParameter("t" + "%");
    return query.getResultList();
}
```



- Les paramètres peuvent aussi être indicés

```
public List<Video> getVideoParTitre(String titre)
{
    Query query = em.createQuery("from Produit p where p.titre like ?1");
    query.setParameter("e" + "%");
    return query.getResultList();
}
```



Langage JPQL

- Utilisation de requêtes nommées
- mise en place par annotation

```
@Entity
@Table(name="cd_audios")
@PrimaryKeyJoinColumn(name="ke_produit")
@NamedQueries({
    @NamedQuery( name="CDAudio.findByArtiste",
                  query="from CDAudio cd where cd.artiste = :artiste")
})
public class CDAudio extends Produit
{...
```

- utilisation

```
public List<CDAudio> getCDAudiosParArtiste(String artiste)
{
    Query query = em.createNamedQuery("CDAudio.findByArtiste");
    query.setParameter("artiste", artiste);
    return query.getResultList();
}
```

Criteria API

- Autre manière de créer des requêtes
 - les requêtes JPQL sont créées avec des chaînes de caractères
 - ce qui peut être complexe lorsqu'il faut créer des requêtes dynamiquement
 - concaténation de String
 - les requêtes basées sur Criteria sont créées par des appels à des méthodes
 - la requête ainsi créée est ensuite exécutée classiquement

Criteria API

- Exemple de requête simple par Criteria
 - équivalent à `SELECT d FROM Destination d`

```
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("bovoyage");
EntityManager entityManager = entityManagerFactory.createEntityManager();

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

CriteriaQuery<Destination> criteriaQuery = criteriaBuilder.createQuery(Destination.class);
Root<Destination> d = criteriaQuery.from(Destination.class);
criteriaQuery.select(d);

TypedQuery<Destination> query = entityManager.createQuery(criteriaQuery);

List<Destination> destinations = query.getResultList();
```

Criteria API

- `CriteriaBuilder` : fabrique pour les requêtes et éléments de requêtes basés sur Criteria API
 - peut être obtenu auprès de `EntityManagerFactory` ou `EntityManager`
- `CriteriaQuery` : la requête qui devra être créée
 - obtenu via le `CriteriaBuilder`
- `Root` : représente la clause FROM de la requête

Criteria API

- Une fois la requête Criteria créée

```
criteriaQuery.select(d);
```

- Elle peut-être utilisée via l'EntityManager

```
TypedQuery<Destination> query = entityManager.createQuery(criteriaQuery);  
List<Destination> destinations = query.getResultList();
```

Criteria API

- Si pour des requêtes simples Criteria semble plus bavard, cette API prend toute sa puissance avec des requêtes créées dynamiquement
 - méthodes de `CriteriaQuery`
 - `select()`, `where()`, `orderBy()`, `groupBy()`, ...
 - utilisation des paramètres
 - `ParameterExpression`
 - cf. documentation