2ème Ingéniorat en Informatique

Programmation Orientée Objet II

Chapitre IV
Programmation Orientée Objet et Bases de Données

2023/2024

Dr. HANNOUSSE Abdelhakim – hannousse.abdelhakim@univ-guelma.dz

Contenu du chapitre

- Introduction
- Concepts de base
  - Clonage des objets
  - Persistance et sérialisation
- Bases de données orientées objets
  - Systèmes de gestion des bases de données orientées objets
  - Schéma de bases de données orientées objets
  - Bases de données relationnelles vs Bases de données orientées objets
  - Bases de données orientées objets avec ObjectDB
    - Les entités
    - Les relations entre entités
    - Connexion et transaction
    - Manipulation des entités avec EntityManager
    - Requêtes JPA avec JPQL
- Java et bases de données relationnelles

Introduction

Introduction (1/3)

- Les bases de données orientées objet (BDDOO) adoptent le paradigme orienté objet pour le stockage, la gestion et la récupération des données.
  - Les données sont représentées sous forme d'objets. Ces objets peuvent être des instances de classes, similaires à la programmation orientée objet.
  - Prennent en charge les concepts de l'héritage et du polymorphisme, ce qui signifie que les objets peuvent être organisés en hiérarchies de classes et que les objets de sous-classes peuvent être traités de manière polymorphique.
  - Peuvent gérer des relations entre les objets, similaires aux relations entre les objets dans les langages de programmation orientés objet. Ces relations peuvent être un-à-un, un-à-plusieurs ou plusieurs-à-plusieurs.
  - Utilisent souvent un langage de requête orienté objet pour interagir avec la base de données. Ce langage permet de formuler des requêtes en utilisant des concepts tels que les classes, les objets et les relations.

Introduction (2/3)

- Les systèmes de gestion des BDDOO garantissent souvent les propriétés ACID (*Atomicité, Cohérence, Isolation, Durabilité*) pour les transactions. Cela assure la fiabilité et la cohérence des données lors d'opérations complexes.
  - **Atomicité:** Les transactions sont généralement mises en œuvre de manière à ce qu'elles soient soit complètement accomplies, soit complètement annulées en cas d'échec.
  - **Cohérence:** Les règles métier et les contraintes d'intégrité définies dans le modèle de données sont respectées pour garantir que les données restent cohérentes.
  - **Isolation:** Chaque transaction voit un état cohérent et isolé de la base de données, même si d'autres transactions sont en cours d'exécution en même temps.
  - **Durabilité:** Les transactions confirmées sont généralement stockées de manière permanente sur un support de stockage stable pour garantir leur durabilité.

Introduction (3/3)

- Certains systèmes BDDOO cherchent à intégrer de manière transparente les données objet avec les bases de données relationnelles, en utilisant des ponts objets-relationnels.
  - Les données sont représentées sous forme d'objets avec des attributs et des méthodes.
  - Des ponts objets-relationnels sont utilisés pour traduire les concepts orientés objet en structures relationnelles, et inversement, de manière transparente pour les développeurs.
  - Les ponts objets-relationnels défini comment les objets et leurs relations sont représentés dans la base de données relationnelle.
  - Les données sont finalement stockées dans la base de données relationnelle sous forme de tables avec des lignes et des colonnes, conformément à la correspondance établie par le pont objet-relationnel.

Concepts de base

Clonage, persistance et sérialisation des objets

Clonage des objets

- Le clonage des objets se réfère à la création d'une copie d'un objet existant dans le système orienté objets.
- Le processus de clonage permet de reproduire un objet tout en conservant sa structure et ses propriétés, de sorte que la copie résultante soit indépendante de l'objet original.
- Il existe deux types de clonage d'objets:
  - **Clonage superficiel :** Il crée une copie de l'objet principal, mais les références aux objets imbriqués ne sont pas dupliquées. Ainsi, les modifications dans les objets imbriqués d'une copie peuvent affecter l'original et vice versa.
  - **Clonage profond :** Il crée une copie complètement indépendante, y compris des objets imbriqués. Les modifications dans les objets imbriqués de la copie n'affectent pas l'original et vice versa.

Utilité de clonage des objets (1/2)

- **Conservation d'états historiques :** Le clonage permet de créer des copies d'objets représentant des états historiques. Cela peut être utile pour suivre les modifications d'objets dans le temps, offrant ainsi une gestion historique des données.
- **Sauvegarde et restauration des données :** En clonant des objets, il est possible de créer des sauvegardes de l'état actuel de la base de données. Ces sauvegardes peuvent être utilisées pour restaurer la base de données à un état antérieur en cas de besoin.
- **Transactions atomiques :** Le clonage peut être utilisé pour effectuer des opérations de manière atomique. Avant d'apporter des modifications à un objet dans la base de données, une copie de l'objet peut être clonée. Si la transaction échoue, la copie peut être jetée, préservant ainsi l'intégrité des données.

Utilité de clonage des objets (2/2)

- **Isolation et réduction des conflits de concurrence :** Lorsque plusieurs utilisateurs tentent de modifier simultanément un même objet, le clonage peut aider à réduire les conflits en attribuant à chaque utilisateur une copie clonée de l'objet. Les modifications sont ensuite fusionnées de manière contrôlée.
- **Optimisation des performances :** Le clonage peut également être utilisé pour optimiser les performances. Plutôt que de travailler directement avec des objets partagés, chaque opération peut être effectuée sur des copies clonées, ce qui réduit les risques de contention et d'attente.

Le clonage des objets en Java (1/6)

- En Java, le clonage d'objets est implémenté à l'aide de l'interface ***Cloneable*** et de la méthode ***clone()***.
- L'interface ***Cloneable*** est une interface de marquage qui indique que la classe peut être clonée. Elle ne contient aucune méthode à implémenter. C'est simplement un moyen de signaler que la classe prend en charge le clonage.
- La méthode ***clone()*** est définie dans la classe ***Object*** et doit être redéfinie dans la classe que l'on souhaite rendre clonable. Elle renvoie une copie de l'objet sur lequel elle est appelée.
- La méthode ***clone()*** est généralement déclarée **protected** et redéfinie avec une exception ***CloneNotSupportedException***. Il est important de lancer cette exception pour signaler que la classe supporte le clonage.
- La méthode ***clone()*** renvoie une référence à un objet de type **Object**, donc il faut utiliser le casting des objets pour récupérer l'objet souhaité.

Le clonage des objets en Java (2/6)

- **Clonage superficiel:** La méthode **clone()** de la classe Object effectue un clonage superficiel.

```java
class Author implements Cloneable {
    private String name;

    public Author(String name) { this.name = name; }
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Book implements Cloneable {
    private String title;
    private Author author;

    public Book(String title, Author author) {
        this.title = title; this.author = author;
    }

    public String getTitle() { return this.title; }
    public void setTitle(String title) { this.title = title; }
    public Author getAuthor() { return this.author; }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Le clonage des objets en Java (3/6)

- **Clonage superficiel:** La méthode **clone()** de la classe **Object** effectue un clonage superficiel.

```
34 class Main {
35     public static void main(String[] args) {
36         Author originalAuthor = new Author("Abdelhakim Hannnousse");
37         Book originalBook = new Book("Object Oriented Programming Course II", originalAuthor);
38 
39         try {
40             Book clonedBook = (Book) originalBook.clone();
41             System.out.println(originalBook.getTitle() + " : " + originalBook.getAuthor().getName());
42             System.out.println(clonedBook.getTitle() + " : " + clonedBook.getAuthor().getName());
43             clonedBook.getAuthor().setName("Mohamed Cherif Nait-Hamoud");
44             System.out.println(originalBook.getTitle() + " : " + originalBook.getAuthor().getName());
45             System.out.println(clonedBook.getTitle() + " : " + clonedBook.getAuthor().getName());
46         } catch (CloneNotSupportedException e) {
47             e.printStackTrace();
48         }
49     }
50 }
```

Le clonage des objets en Java (4/6)

- **Clonage superficiel:** La méthode **clone()** de la classe **Object** effectue un clonage superficiel.

```
Object Oriented Programming Course II : Abdelhakim Hannnousse
Object Oriented Programming Course II : Abdelhakim Hannnousse
Object Oriented Programming Course II : Mohamed Cherif Nait-Hamoud
Object Oriented Programming Course II : Mohamed Cherif Nait-Hamoud
```

Le clonage des objets en Java (5/6)

- **Clonage profond:** nécessite souvent une implémentation personnalisée de la méthode ***clone()*** pour cloner les objets internes de manière récursive.

```
Object Oriented Programming Course II : Abdelhakim Hannnousse
Object Oriented Programming Course II : Abdelhakim Hannnousse
Object Oriented Programming Course II : Abdelhakim Hannnousse
Object Oriented Programming Course II : Mohamed Cherif Nait-Hamoud
```

```java
class Author implements Cloneable {
    private String name;

    public Author(String name) { this.name = name; }
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Book implements Cloneable {
    private String title;
    private Author author;

    public Book(String title, Author author) {
        this.title = title; this.author = author;
    }

    public String getTitle() { return this.title; }
    public void setTitle(String title) { this.title = title; }
    public Author getAuthor() { return this.author; }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Book clone = (Book) super.clone();
        clone.author = (Author) author.clone();
        return clone;
    }
```

Le clonage des objets en Java (6/6)

- Les types **primitifs** ainsi que les **objets immtuables** ne sont pas clonés de la même manière que les objets mutables:
  - Les types primitifs en Java, tels que int, **float**, **double**, **boolean**, etc., représentent des valeurs de données de base et ne sont pas des objets. Lorsqu'une variable de type primitif est clonée, sa valeur est simplement copiée dans la nouvelle variable.
  - Les objets immutables, tels que **String**, **Integer**, **LocalTime**, etc., sont des objets dont l'état ne peut pas être modifié une fois qu'ils ont été créés. Lorsqu'un objet immtuable est cloné, en réalité, une nouvelle référence à cet objet est créée, donc, il n'y a pas de nécessité de créer une copie profonde de l'objet.
  - Les objets mutables, tels que les listes, les ensembles, les tableaux, et les objets des classes definies par les utilisateurs, peuvent être modifiés après leur création. Lorsqu'un objet mutable est cloné, une nouvelle instance de cet objet est créée, mais son état peut également nécessiter une copie profonde pour garantir que les modifications à l'objet original ne modifient pas l'objet cloné et vice versa.

Persistance des objets

- La persistance des objets fait référence à la capacité de stocker durablement des objets, généralement dans une base de données ou un autre support de stockage, de manière à ce qu'ils puissent être récupérés ultérieurement. Cela permet aux objets de maintenir leur état entre différentes exécutions d'un programme.
  - Les objets persistants sont sauvegardés dans un support de stockage durable, tel qu'une base de données ou un fichier. Cela permet de conserver l'état de l'objet même lorsque le programme se termine.
  - Les objets persistants peuvent être récupérés à partir du support de stockage pour être réutilisés ultérieurement. Cela permet de restaurer l'état précédent de l'objet.
- Pour rendre les objets persistants, ils sont souvent **sérialisés**, c'est-à-dire convertis en une séquence d'octets pouvant être stockée ou transmise. La **dé-sérialisation** est le processus inverse, où les octets sont utilisés pour recréer l'objet en mémoire.

Sérialisation des objets

- La sérialization transforme un objet en un flux de données, généralement une séquence d'octets. Ce flux peut être enregistré dans un fichier, transmis sur un réseau, ou stocké dans une base de données.
- Lors de la sérialization, l'ensemble de l'état de l'objet est inclus dans le flux de données. Cela comprend les valeurs de tous les attributs de l'objet ainsi que toute la hiérarchie des objets imbriqués.
- Il existe différents formats de sérialization, tels que le format binaire, le format JSON, XML, etc. Chaque format a ses avantages et ses inconvénients en fonction des besoins spécifiques de l'application.
- La désérialization est le processus inverse de la sérialization, où le flux de données est converti en un objet réel dans la mémoire. Il est essentiel que le processus de sérialization soit réversible pour permettre une utilisation cohérente des données.

Sérialisation des objets en Java (1/4)

- Pour rendre un objet sérialisable en Java, sa classe doit implémenter l'interface ***Serializable***. Cette interface est simplement une interface de marquage. Elle indique au mécanisme de sérialisation que la classe peut être sérialisée.
- Lors de la tentative de sérialisation d'un objet d'une classe qui n'implémente pas l'interface **Serializable**, Java lèvera une exception de type **java.io.NotSerializableException**.
- Java utilise les classes ***ObjectOutputStream*** et ***ObjectInputStream*** pour sérialiser et désérialiser des objets (format binaire) respectivement. Les méthodes ***writeObject()*** et ***readObject()*** des classes ***ObjectOutputStream*** et ***ObjectInputStream*** respectivement effectue la sérialisation et la désérialisation des objets passés en paramètres.
- Dans certains cas, certains champs d'une classe ne doivent pas etre sérialisés. On peut exclure certains champs de la sérialisation en les déclarant avec le mot-clé ***transient***.

Sérialisation des objets en Java (2/4)

- L'exclusion certains champs de la sérialisation est utile pour:
  - **Eviter la sérialisation inutile de certains champs:** Certains champs d'une classe peuvent contenir des données qui ne doivent pas être sérialisées, par exemple des références temporaires, des caches de données volatiles.
  - **Réduction de la taille des données et optimization de performance :** En excluant certains champs de la sérialisation, on peut réduire la taille des données sérialisées. Cela peut être important dans le cas de transfér des données sur le réseau ou leurs stockage sur un support de stockage de taille limitée.
  - **Protection des données sensibles :** Les champs contenant des informations sensibles telles que des mots de passe, des clés de cryptage, des jetons d'authentification, etc., peuvent être déclarés comme transient pour éviter qu'ils ne soient écrits dans un flux de sortie, ce qui pourrait compromettre leur sécurité.
  - **Gestion de la compatibilité :** L'utilisation de champs transient peut aider à gérer la compatibilité entre les différentes versions des applications en excluant les champs obsolètes lors de la désérialisation.

Sérialisation des objets en Java (3/4)

- Lors de la sérialisation, tous les champs, y compris les champs transient, sont inclus dans le flux de sérialisation. Les valeurs des champs transient ne sont pas écrites dans le flux de sérialisation.
- Lors de la désérialisation, les champs transient sont initialisés avec leurs valeurs par défaut. Il faut donc rétablir les valeurs des champs transient après la désérialisation en utilisant des méthodes spécifiques pour réinitialiser les valeurs des champs transient.
- En Java, la sérialisation est **superficielle** par défaut, ce qui signifie que seule l'instance de l'objet sérialisé est enregistrée dans le flux de sortie sans prendre en compte les objets référencés à l'intérieur de cet objet. Cela peut poser problème si l'objet a sérialiser contient des références à d'autres objets.
- Pour une sérialisation profonde, il est recommandé d'implémenter l'interface **Serializable** dans toutes les classes des objets référencés. Cela garantit que l'ensemble de la hiérarchie d'objets est sérialisable, permettant ainsi une sérialisation complète des objets.

Sérialisation des objets en Java (4/4)

- Lorsque des modifications sont apportées à une classe sérialisée, il est recommandé de spécifier un numéro de série (***serialVersionUID***). Cela permet de gérer les versions de classe et d'éviter des erreurs lors de la désérialisation si la classe a été modifiée.
- La non spécification d'un numéro de série (***serialVersionUID***) peut poser des problèmes lors de la désérialisation, car Java vérifie si le numéro de série de la classe sérialisée correspond à celui de la classe de désérialisation. Si les numéros de série ne correspondent pas, une ***InvalidClassException*** sera levée.
- Java utilise un format binaire pour la sérialisation par défaut. Cependant, il est également possible d'utiliser d'autres formats tels que JSON ou XML en utilisant des bibliothèques externes comme Jackson ou JAXB. Dans ce cas l'implémentation de l'interface ***Serializable*** peut être ignorée.

Sérialisation des objets en Java - Exemple (1/4)

```java
import java.io.*;
class Auteur implements Serializable {
    private String nom;
    public Auteur(String nom) { this.nom = nom;}
    public String getNom() { return this.nom; }
    public void setNom(String nom) { this.nom = nom; }
}

class Livre implements Serializable {
    private String titre;
    private Auteur auteur;
    private int annee;
    private String publisher;

    public Livre(String titre, Auteur auteur, int annee, String publisher) {
        this.titre = titre; this.auteur = auteur; this.annee = annee; this.publisher = publisher;
    }
    @Override
    public String toString() {
        return "Livre{" + "titre= '" + titre + ", auteur= '" + auteur.getNom() +
                ", annee= " + annee + ", publisher= '" + publisher + '}';
    }
}
```

Sérialisation des objets en Java - Exemple (2/4)

```java
public class SerializationExample {
    public static void main(String[] args) {
        // Création d'un objet Livre
        Livre livre = new Livre(
            "Programmtion Orienté Objet II", new Auteur("Abdelhakim Hannousse"), 2023,
            "Université de Guelma");

        // Sérialisation de l'objet Livre
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("livre.dat"))) {
            oos.writeObject(livre);
            System.out.println("L'objet Livre a été sérialisé avec succès.");
        } catch (IOException e) { e.printStackTrace(); }

        // Désérialisation de l'objet Livre
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("livre.dat"))) {
            Livre livreDeserialise = (Livre) ois.readObject();
            System.out.println("L'objet Livre a été désérialisé : " + livreDeserialise);
        } catch (IOException | ClassNotFoundException e) { e.printStackTrace(); }
    }
}
```

Résultat d'affichage:

```
L'objet Livre a été sérialisé avec succès.
L'objet Livre a été désérialisé : Livre{titre= 'Programmtion Orienté Objet II', auteur= 'Abdelhakim
    Hannousse', annee = 2023, publisher= 'Université de Guelma'}
```

Sérialisation des objets en Java - Exemple (3/4)

```java
import java.io.*;
class Auteur implements Serializable {
    private String nom;
    public Auteur(String nom) { this.nom = nom;}
    public String getNom() { return this.nom; }
    public void setNom(String nom) { this.nom = nom; }
}

class Livre implements Serializable {
    private String titre;
    private Auteur auteur;
    private int annee;
    private transient String publisher;

    public Livre(String titre, Auteur auteur, int annee, String publisher) {
        this.titre = titre; this.auteur = auteur; this.annee = annee; this.publisher = publisher;
    }
    @Override
    public String toString() {
        return "Livre{" + "titre= '" + titre + ", auteur= '" + auteur.getNom() +
                ", annee= " + annee + ", publisher= '" + publisher + '}';
    }
}
```

Résultat d'affichage:

```
L'objet Livre a été sérialisé avec succès.
L'objet Livre a été désérialisé : Livre{titre= 'Programmtion Orienté Objet II', auteur= 'Abdelhakim
    Hannousse', annee = 2023, publisher= null}
```

Sérialisation des objets en Java - Exemple (4/4)

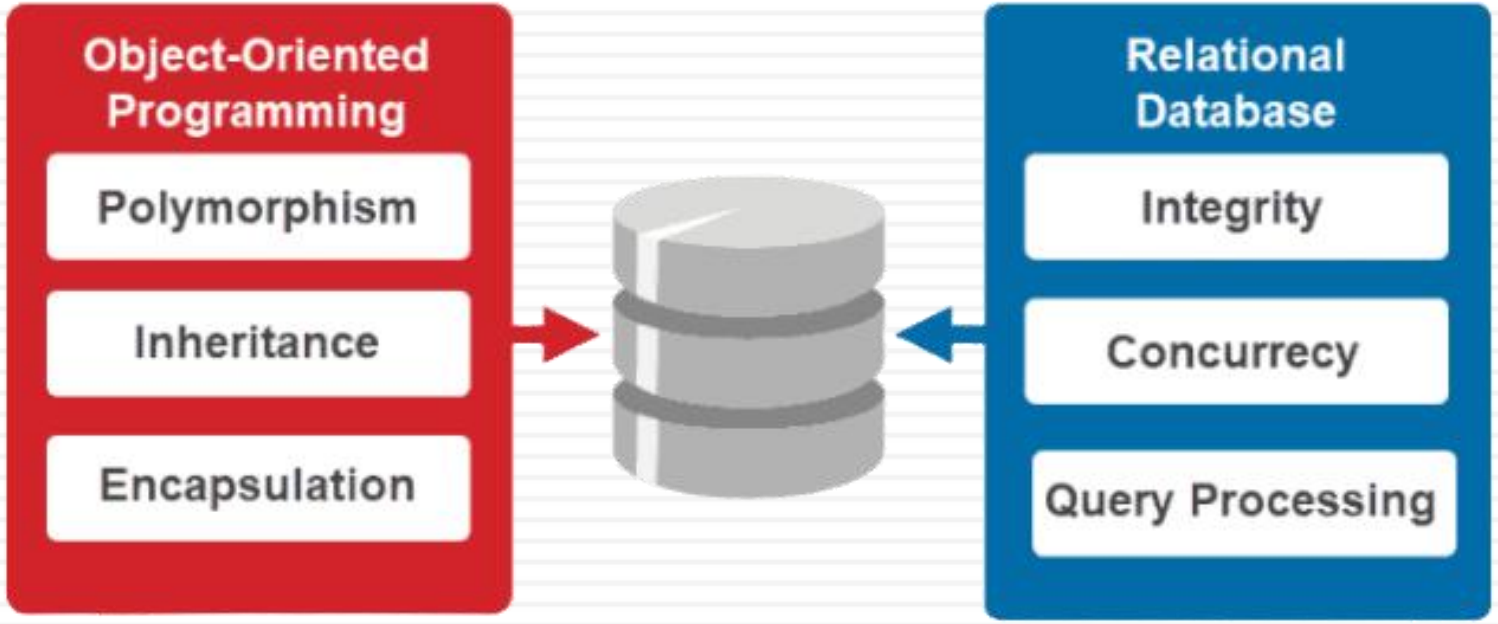
```java
import com.fasterxml.jackson.databind.*;
class Auteur {
    private String nom;
    public Auteur(String auteur) { this.nom = auteur;}
    // Setters et Getters
}
class Livre {
    private String titre;
    private Auteur auteur;
    private int annee;
    private String publisher;
    public Livre(String titre, Auteur auteur, int annee, String publisher) {
        this.titre = titre; this.auteur = auteur; this.annee = annee; this.publisher = publisher;
    }
    // Setters et Getters
}
public class Main {
public static void main(String[] args) throws Exception {
    Livre livre = new Livre("POO II",new Auteur("Abdehakim Hannousse"),2024,"Guelma University");

    // Utiliser ObjectMapper de Jackson pour sérialiser l'objet en JSON
    ObjectMapper mapper = new ObjectMapper();
    mapper.enable(SerializationFeature.INDENT_OUTPUT); // Pour une belle indentation du JSON
    String json = mapper.writeValueAsString(livre);

    // Afficher le JSON résultant
    System.out.println(json);
    }
}
```

Résultat d'affichage:

```
{
    "titre" : "POO II",
    "auteur" : { "nom" : "Abdelhakim Hannousse"},
    "annee" : 2024,
    "publisher" : "Guelma University"
}
```

En Jackson, le mot-clé transient n'est pas directement pris en charge comme c'est le cas dans la sérialisation Java. Il faut utiliser des annotations pour ignorer les champs lors de la sérialisation et de la désérialisation. L'annotation **@JsonIgnore** est utilisée à cette fin.

Bases de données orientées objets



Introduction (1/2)

- Les bases de données orientées objets sont conçues pour gérer la persistance des objets, c'est-à-dire la capacité à stocker les objets de manière durable afin qu'ils puissent être récupérés même après la fermeture de l'application.
- Elles utilisent une approche de modélisation basée sur des objets, ce qui signifie qu'elles représentent les données sous forme d'objets, similaires à la POO. Ces objets peuvent avoir des attributs (champs) et des méthodes associées.
- Au lieu des types de données traditionnels tels que les entiers et les chaînes de caractères, les bases de données orientées objets prennent en charge des types de données plus complexes tels que les objets, les tableaux et d'autres structures de données.
- Elles cherchent à combiner les avantages de la programmation orientée objet avec les fonctionnalités des bases de données relationnelles. Elles permettent de stocker des objets complexes tout en maintenant des relations entre eux.

Introduction (2/2)

- Les concepts d'héritage et de polymorphisme de la programmation orientée objet sont souvent intégrés dans la modélisation des données. Cela permet de créer des hiérarchies d'objets avec des propriétés communes.
- Certains systèmes de gestion de bases de données orientées objets utilisent un langage de requête basé sur des objets. Cela permet aux utilisateurs de formuler des requêtes en utilisant des concepts orientés objets, ce qui peut rendre les opérations plus intuitives.
- Certains exemples de systèmes de gestions de bases de données orientées objets comprennent:
  - db4o (https://sourceforge.net/projects/db4o/)
  - **ObjectDB** (https://www.objectdb.com/)
  - ObjectStore (https://ignitetech.com/softwarelibrary/objectstore)

Systèmes de gestion des Bases de données orientées objets

- Similaire à un système de gestion de bases de données relationnelles (SGBDR), un système de gestion de bases de données (SGBDO) doit assurer:
  - **Persistance des objets.** Tout objet doit pouvoir persister sur disque au-delà du programme qui le crée.
  - **Concurrence d'accès.** La base d'objets doit pouvoir être partagée simultanément par les transactions qui la consultent et la modifient ; les blocages doivent être minimaux afin d'assurer la cohérence de la base.
  - **Fiabilité des objets.** Les objets doivent être restaurés en cas de panne d'un programme dans l'état où ils étaient avant la panne. Les transactions doivent être atomiques, c'est-à-dire totalement exécutées ou pas du tout.
  - **Facilité d'interrogation.** Il doit être possible de retrouver un objet à partir de valeurs de ses propriétés, qu'il s'agisse de valeurs d'attributs, de résultats de méthodes appliquées à l'objet ou de liens avec les objets référencés ou référençants.

Schéma de bases de données orientées objets (1/2)

- Le schéma de base de données à objets repose sur la représentation et la structuration des données sous forme d'objets, en accord avec les principes de la programmation orientée objet (POO).
- Le diagramme de classe joue un rôle essentiel dans le schéma de base de données orientées objets.
  - **Classes d'objets :** sont l'équivalent des tables dans les bases de données relationnelles. Chaque classe représente un type d'objet avec des attributs spécifiques.
  - **Attributs :** sont les propriétés ou les caractéristiques d'une classe d'objet. Chaque attribut est associé à un type de données.
  - **Relations :** sont représentées de manière similaire aux clés étrangères dans les bases de données relationnelles. Une classe peut avoir des références à d'autres classes, créant ainsi des relations.

Schéma de bases de données orientées objets (2/2)

- **Héritage :** l'héritage peut être utilisé pour modéliser des relations entre différentes entités en partageant des caractéristiques communes.
- **Méthodes :** les méthodes dans un schéma de base de données orienté objet apportent une dimension comportementale aux entités, permettant ainsi une modélisation plus riche et expressive des interactions entre les données.
- **Identifiants :** chaque objet doit avoir un identifiant unique. Cet identifiant est défini comme une clé primaire, mais il peut aussi être un attribut unique.

- Dans ce modèle, les données peuvent être représentées sous forme de types de données complexes, tels que des images et du multimédia, ce qui les rend adaptées aux applications modernes.
- Le principal avantage du modèle orienté objet est sa compatibilité avec les langages de programmation orientés objet, permettant aux développeurs de travailler de manière plus transparente avec les bases de données.

Bases de données relationnelles vs Bases de données orientées objets

BDDR	BDDO
Utilise des tables avec des lignes et des colonnes pour représenter les données. Les relations entre les tables sont établies à l'aide de clés étrangères.	Utilise des classes et des objets pour représenter les données. Les relations entre les objets sont établies par des associations et des références entre les instances.
Moins flexible en termes de types de données. Le schéma est souvent rigide et les modifications peuvent être complexes.	Plus flexible en termes de modélisation. Les objets peuvent contenir des structures de données complexes, et la modification du schéma est souvent plus naturelle.
Mettent souvent en œuvre des contraintes d'intégrité pour assurer la cohérence des données.	L'intégrité des données dépend souvent de la logique métier intégrée dans les méthodes des objets.
Utilise le SQL pour les requêtes.	Souvent utilise des langages de requêtes orientés objets.

Bases de données orientées objets avec ObjectDB

- ObjectDB est un sysème de gestion de base de données orientée objets qui prend en charge le stockage et la gestion de données basées sur le modèle objet.
- ObjectDB utilise un langage de requête objet appelé JPQL (Java Persistence Query Language). Ce langage permet d'exprimer des requêtes de manière similaire à SQL, mais en utilisant des concepts orientés objets. Cela rend les requêtes plus naturelles pour les développeurs Java.
- ObjectDB prend en charge les transactions, ce qui signifie qu'il offre des mécanismes pour garantir la cohérence des données lors des opérations de lecture et d'écriture. Les transactions permettent d'assurer l'intégrité des données, même en cas de situations exceptionnelles.
- ObjectDB est conçu pour offrir des performances élevées en manipulant directement des objets en mémoire. Cela peut être particulièrement avantageux dans les scénarios où la représentation objet naturelle de données est essentielle.

Bases de données orientées objets avec ObjectDB

- ObjectDB gère automatiquement les index pour améliorer les performances des requêtes. Les développeurs peuvent également définir des index personnalisés pour répondre à des besoins spécifiques.
- En utilisant ObjectDB, les développeurs peuvent souvent concevoir et développer plus rapidement, car ils travaillent avec un modèle de données plus proche de leur modèle conceptuel.

Bases de données orientées objets avec ObjectDB - Entité

- En ObjectDB, une entité représente un objet stocké dans la base de données.
- Souvent associée à une classe Java, chaque instance de cette classe devient une entité lorsqu'elle est persistée.
- Pour rendre une classe persistante, utiliser l'annotation ***@Entity*** de JPA (Java Persistent API).
- Par défaut, tous les champs non ***final*** ou ***static*** sont persistants. Pour exclure un champ de la sauvegarde, spécifier l'annotation ***@Transient***
- L'annotation ***@Id*** est utilisée pour spécifier l'attribut qui servira d'identifiant primaire (ID) pour une entité persistante.
- La génération automatique des clés en ObjectDB, souvent utilisée en conjonction avec l'annotation ***@GeneratedValue***, permet à la base de données de générer automatiquement des valeurs pour les clés primaires lors de l'insertion d'une nouvelle entité.

Exemple

```java
import javax.persistence.*;

@Entity
public class Livre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private String titre;
    private String auteur;
    private int anneePublication;
    private String publisher;

    @Transient
    private String commentaire;  // Exclu de la sauvegarde

    public Livre(String titre, String auteur, int anneePublication, String publisher) {
        this.titre = titre;
        this.auteur = auteur;
        this.anneePublication = anneePublication;
        this.publisher = publisher;
    }

    @Override
    public String toString() {
        return "Livre{" + "titre= '" + titre + '\'' + ", auteur= '" + auteur + '\'' +
                ", anneePublication= " + anneePublication + ", publisher= '" + publisher + '}';
    }
}
```

Bases de données orientées objets avec ObjectDB – Relations entres les entités

- JPA utilise une liste des annotations pour définir les relations entre entités dans une base de données:
  - **@OneToOne:** définir une relation un-à-un entre deux entités:
    - *targetEntity*: la classe de l'entité cible.
    - *cascade*: définit les opérations qui doivent être propagées de l'entité parente à l'entité enfant.
  - **@OneToMany:** définir une relation un-à-plusieurs entre deux entités.
    - *targetEntity*: la classe de l'entité cible.
    - *mappedBy*: le nom du champ dans l'entité cible qui maintient la relation.
    - *cascade*: définit les opérations qui doivent être propagées de l'entité parente à l'entité enfant.

Bases de données orientées objets avec ObjectDB – Relations entres les entités

- **@ManyToOne:** définir une relation plusieurs-à-un entre deux entités.
  - *targetEntity*: la classe de l'entité cible.
  - *fetch*: spécifie la stratégie de récupération des données associées.
  - *cascade*: définit les opérations qui doivent être propagées de l'entité parente à l'entité enfant.
- **@ManyToMany:** définir une relation plusieurs-à-plusieurs entre deux entités.
  - *targetEntity*: la classe de l'entité cible.
  - *mappedBy*: le nom du champ dans l'entité cible qui maintient la relation.
  - *cascade*: Indique les opérations à propager à la relation.
- **@JoinColumn:** spécifier une colonne de jointure dans une relation.
  - *name*: le nom de la colonne de jointure.
  - *referencedColumnName*: le nom de la colonne référencée.

Bases de données orientées objets avec ObjectDB – Relations entres les entités

- ***CascadeType*** est une énumération qui définit les différentes opérations qui peuvent être propagées de l'entité parente à l'entité enfant:
  - **CascadeType.PERSIST:** propage l'opération de persistance (persist). Utile lors de l'ajout d'une nouvelle entité parente pour enregistrer automatiquement les entités enfants associées.
  - **CascadeType.REMOVE:** propage l'opération de suppression (remove). Utile pour supprimer automatiquement les entités enfants lorsqu'une entité parente est supprimée.
  - **CascadeType.REFRESH:** propage l'opération de rafraîchissement (refresh). Utile lorsque l'entité parente doit automatiquement rafraîchir les entités enfants avec les données les plus récentes de la base de données.
  - **CascadeType.DETACH:** propage l'opération de détachement (detach). Utile lorsqu'une entité parente doit automatiquement détacher les entités enfants.

Bases de données orientées objets avec ObjectDB – Relations entres les entités

- **CascadeType.MERGE:** propage l'opération de fusion (merge). Utile lorsqu'une entité parente modifiée doit fusionner automatiquement les modifications dans les entités enfants associées.
- **CascadeType.ALL:** propage toutes les opérations.

- ***FetchType*** est une énumération qui spécifie comment les données associées à une relation doivent être chargées.
  - **FetchType.LAZY :** les données associées ne sont chargées que lorsque l'attribut est accédé pour la première fois. Utile pour optimiser les performances lorsque toutes les données ne sont pas toujours nécessaires.
  - **FetchType.EAGER :** les données associées sont chargées immédiatement avec l'entité principale. Toutes les données, même celles qui ne sont pas immédiatement utilisées, sont chargées.

Bases de données orientées objets avec ObjectDB – Relations entres les entités - Exemples

- Livre à Auteur (un-à-un) - un livre ait un seul auteur.

```
@OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
@JoinColumn(name = "auteur_id")
private Auteur auteur;
```

- Auteur à Livres (un-à-plusieurs) - Un auteur peut avoir écrit plusieurs livres.

```
@OneToMany(mappedBy = "auteur", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Livre> livres;
```

- Livre à Auteur (plusieurs-à-un) - Plusieurs livres peuvent avoir le même auteur.

```
@ManyToOne(fetch = FetchType.EAGER)
private Auteur auteur;
```

- Auteurs et Livres (plusieurs-à-plusieurs) - Plusieurs auteurs peuvent avoir écrit plusieurs livres.

```
@ManyToMany
private List<Auteur> auteurs;
```

Bases de données orientées objets avec ObjectDB – Connexion et transaction (1/3)

- La **connexion** en ObjectDB fait référence à l'établissement d'une liaison entre l'application Java et la base de données orientée objets.
- Toute activité sur la base de données doit être effectuée dans une **transaction**. Une transaction est une séquence d'une ou plusieurs opérations de base de données qui doit être traitée comme une unité atomique
- Pour établir une connexion avec ObjectDB dans une application Java :
  - **Chargement du driver/ObjectDB :** Assurer que le JAR du driver ObjectDB est inclus dans le projet Java.
  - **Création de l'EntityManagerFactory :** C'est l'objet responsable de créer des instances d'***EntityManager***. Il est généralement créé une seule fois pendant le cycle de vie de l'application. Il est configuré avec des informations sur la source de données, telles que l'emplacement de la base de données, le nom d'utilisateur et mot de passe.

Bases de données orientées objets avec ObjectDB – Connexion et transaction (2/3)

- **Début de la transaction :** Le début de la transaction (méthode ***begin()***) signifie que toutes les opérations suivantes seront regroupées dans une transaction cohérente.
- **Création et persistance des objets :** Les objets Java à persister sont créés et associés à l'***EntityManager***. La méthode ***persist()*** est utilisée pour les ajouter à la transaction courante. Si un objet contient des références vers d'autres entités, il faut explicitement les rendre persistants s'ils ne sont pas de type **Embeddable**.
  - Un type **Embeddable** fait référence à une classe dont les instances sont intégrées directement dans l'objet parent et n'ont pas d'existence autonome en tant qu'entité persistante distincte.
  - Les instances d'un type **Embeddable** sont intégrées directement dans la table de l'objet parent plutôt que d'avoir leur propre table dédiée.
  - Contrairement aux entités, les types **Embeddable** n'ont généralement pas d'identifiant propre (clé primaire). Leur existence dépend entièrement de leur relation avec l'objet parent.

Bases de données orientées objets avec ObjectDB – Connexion et transaction (3/3)

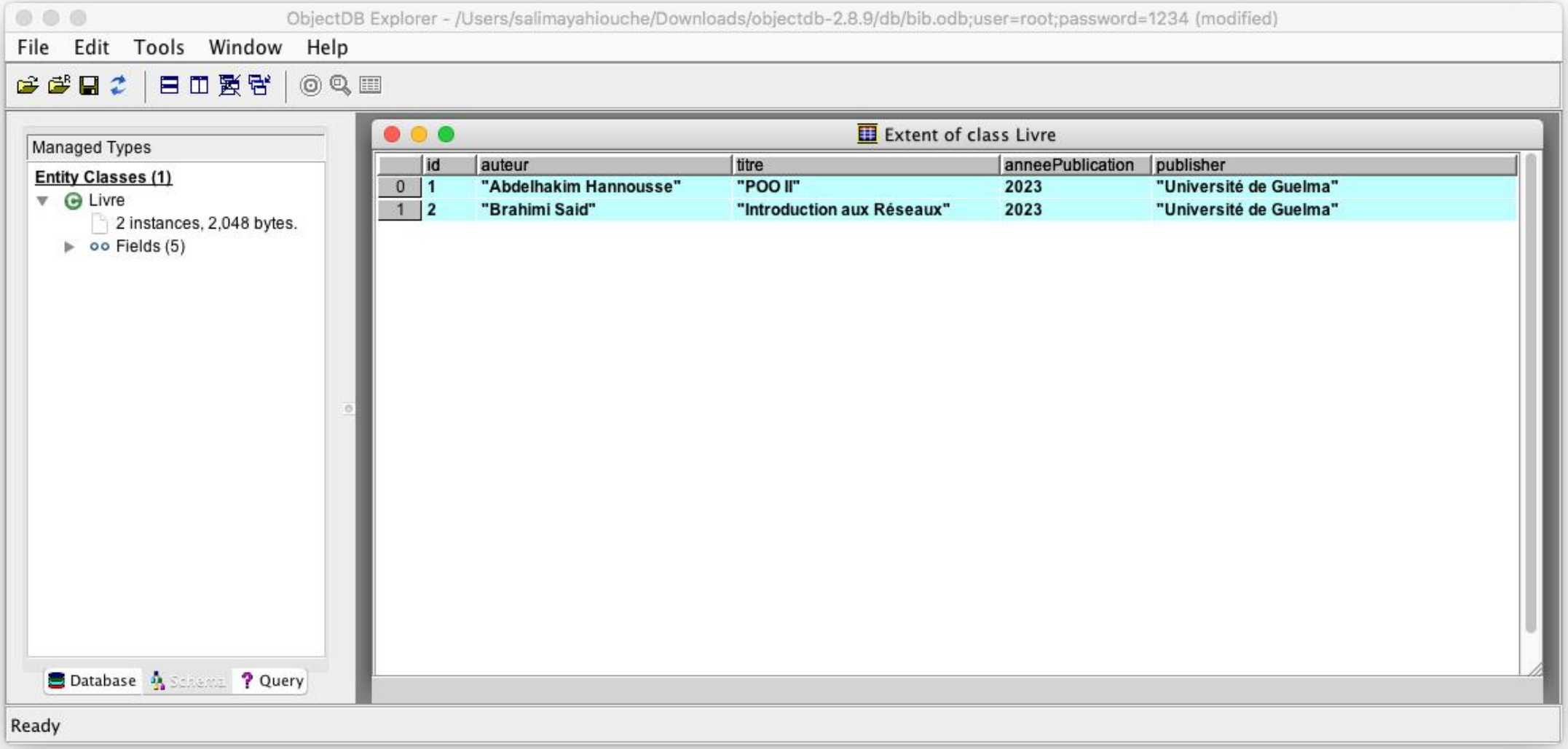
- La spécification des types Embeddable se fait à l'aide de l'annotation **@Embeddable** et **@Embeded.**
- Il est aussi possible d'utiliser ***@OneToOne(cascade=CascadeType.PERSIST)*** pour render la persistence entre deux entités transparente, dans ce cas uniquement l'objet parent doit être persister.

- **Valider la transaction :** Lorsque toutes les opérations de persistance sont prêtes et que la cohérence des données est assurée, la transaction est validée en utilisant la méthode ***commit()***. Toutes les modifications apportées à la base de données au cours de la transaction deviennent permanentes.
- **Fermeture de la connexion :** Une fois la transaction terminée, les ressources associées à l'***EntityManager*** et à l'***EntityManagerFactory*** doivent être libérées. La fermeture de l'***EntityManager*** libère les ressources temporaires liées à la gestion de la transaction. La fermeture de l'***EntityManagerFactory*** peut également libérer des ressources globales et terminer la connexion à la base de données.

Exemple

```java
import java.util.*;
import javax.persistence.*;
public class Main {
    public static void main(String[] args) {
        Map<String, String> ps= new HashMap<String, String>();
        ps.put("javax.persistence.jdbc.user", "root");
        ps.put("javax.persistence.jdbc.password", "1234");

        // Création de l'EntityManagerFactory
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("$objectdb/db/bib.odb", ps);

        // Création de l'EntityManager
        EntityManager em = emf.createEntityManager();

        // Début de la transaction
        em.getTransaction().begin();

        // Création et persistance des objets Livre
        Livre livre = new Livre("POO II", "Abdelhakim Hannousse", 2023, "Université de Guelma");
        em.persist(livre);
        livre = new Livre("Introduction aux Réseaux", "Said Brahimi", 2023, "Université de Guelma");
        em.persist(livre);

        // Valider la transaction
        em.getTransaction().commit();

        // Fermeture des ressources
        em.close(); emf.close();
    }
}
```

Bases de données orientées objets avec ObjectDB – Connexion et transaction – Exemple (ObjectDB Explorer)

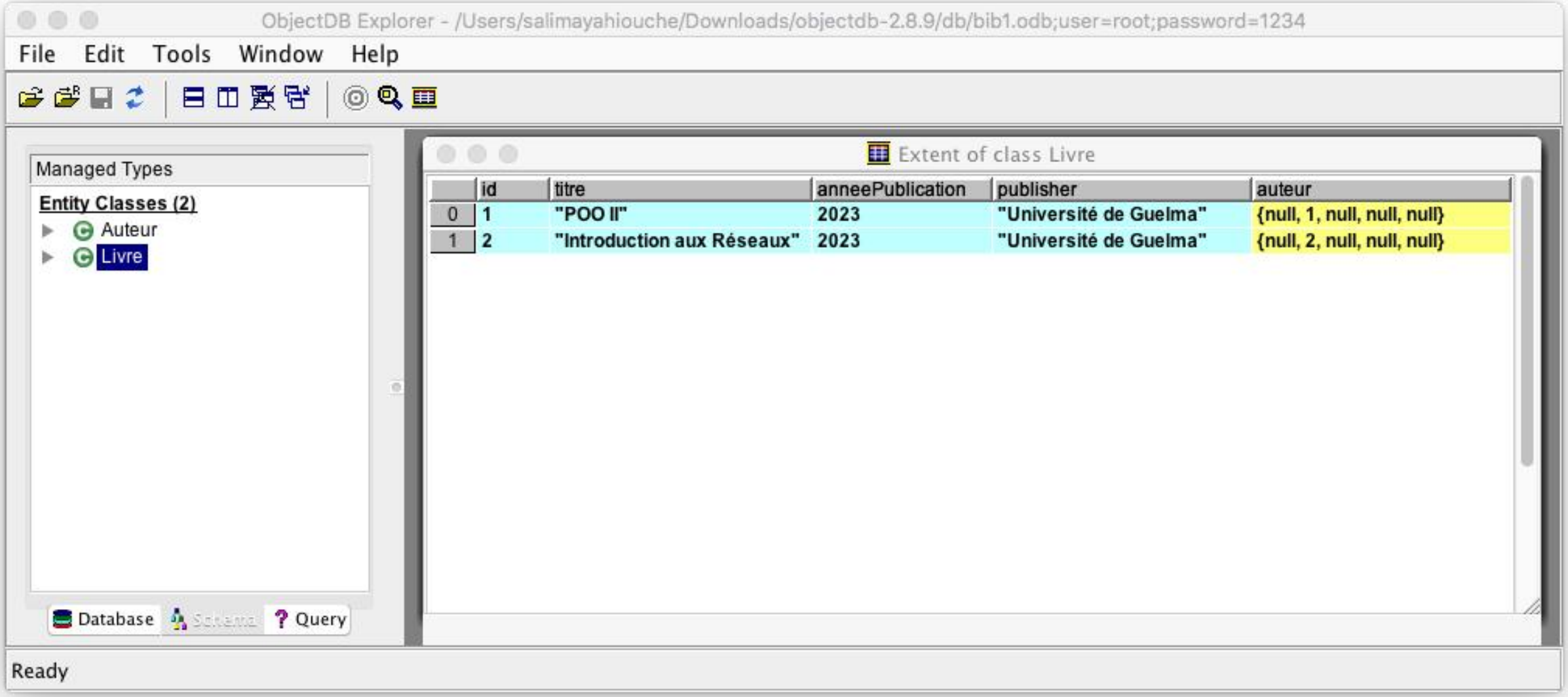


Bases de données orientées objets avec ObjectDB – Connexion et transaction - Exemple

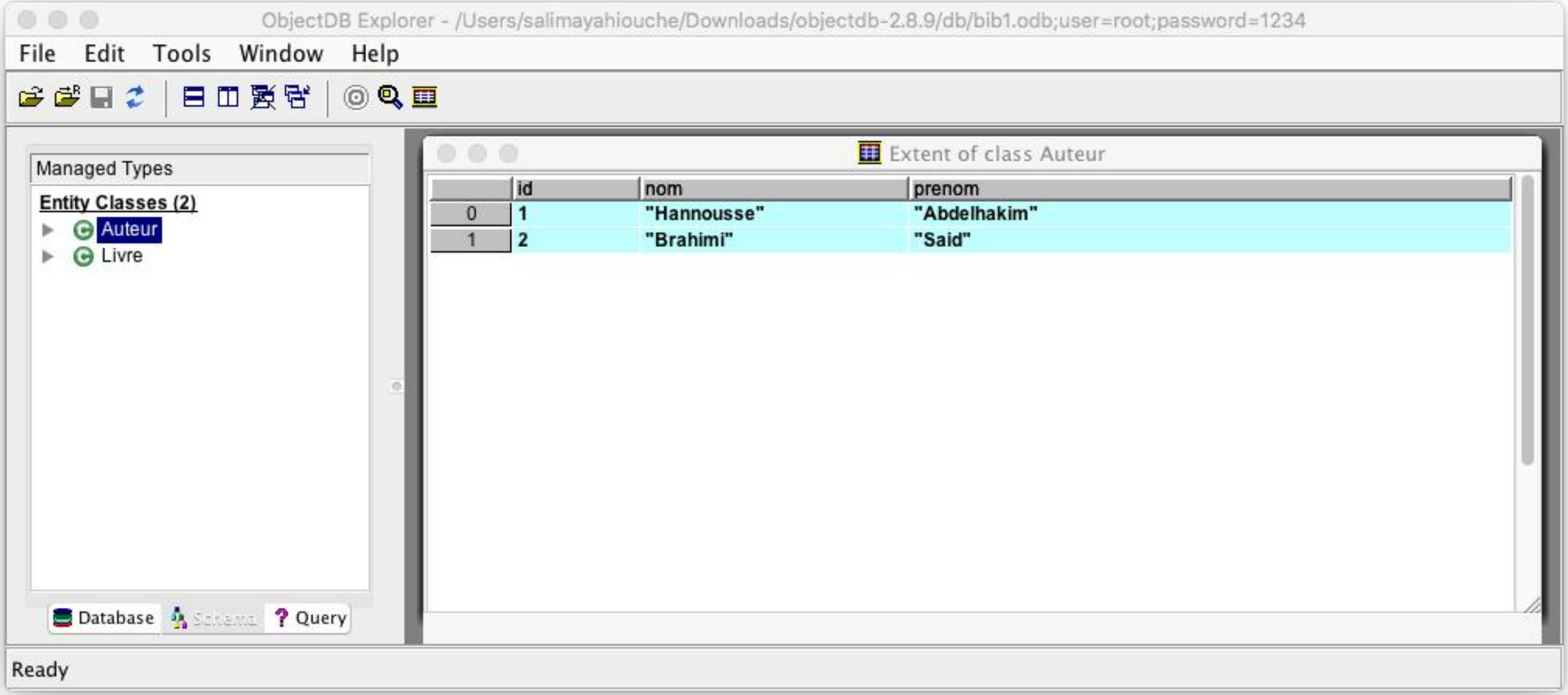
```java
import javax.persistence.*;
@Entity
class Livre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String titre, publisher;
    private Auteur auteur; private int anneePublication;

    public Livre(String titre, Auteur auteur, int anneePublication, String publisher) {
        this.titre = titre; this.auteur = auteur;
        this.anneePublication = anneePublication; this.publisher = publisher;
    }
    @Override
    public String toString() {
        return "Livre{ titre= '" + titre + ", auteur= '" + auteur + ", anneePublication= " +
            anneePublication + ", publisher= '" + publisher + '}';
    }
}
@Entity
class Auteur {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nom, prenom;

    public Auteur(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }
    @Override
    public String toString() {
        return "Auteur{ nom= '" + nom + ", prénom= '" + prenom + '}';
    }
}
```

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<String, String> ps= new HashMap<String, String>();
        ps.put("javax.persistence.jdbc.user", "root");
        ps.put("javax.persistence.jdbc.password", "1234");
        // Création de l'EntityManagerFactory
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("$objectdb/db/bib1.odb", ps);

        // Création de l'EntityManager
        EntityManager em = emf.createEntityManager();

        // Début de la transaction
        em.getTransaction().begin();

        Auteur auteur; Livre livre;

        // Création et persistance des objets Livre
        auteur = new Auteur("Hannousse", "Abdelhakim");
        livre = new Livre("POO II", auteur, 2023, "Université de Guelma");
        em.persist(auteur);
        em.persist(livre);
        auteur = new Auteur("Brahimi", "Said");
        livre = new Livre("Introduction aux Réseaux", auteur, 2023, "Université de Guelma");
        em.persist(auteur);
        em.persist(livre);

        // Valider la transaction
        em.getTransaction().commit();

        // Fermeture des ressources
        em.close(); emf.close();
    }
}
```

Bases de données orientées objets avec ObjectDB – Connexion et transaction – Exemple (ObjectDB Explorer)



Bases de données orientées objets avec ObjectDB – Connexion et transaction – Exemple (ObjectDB Explorer)

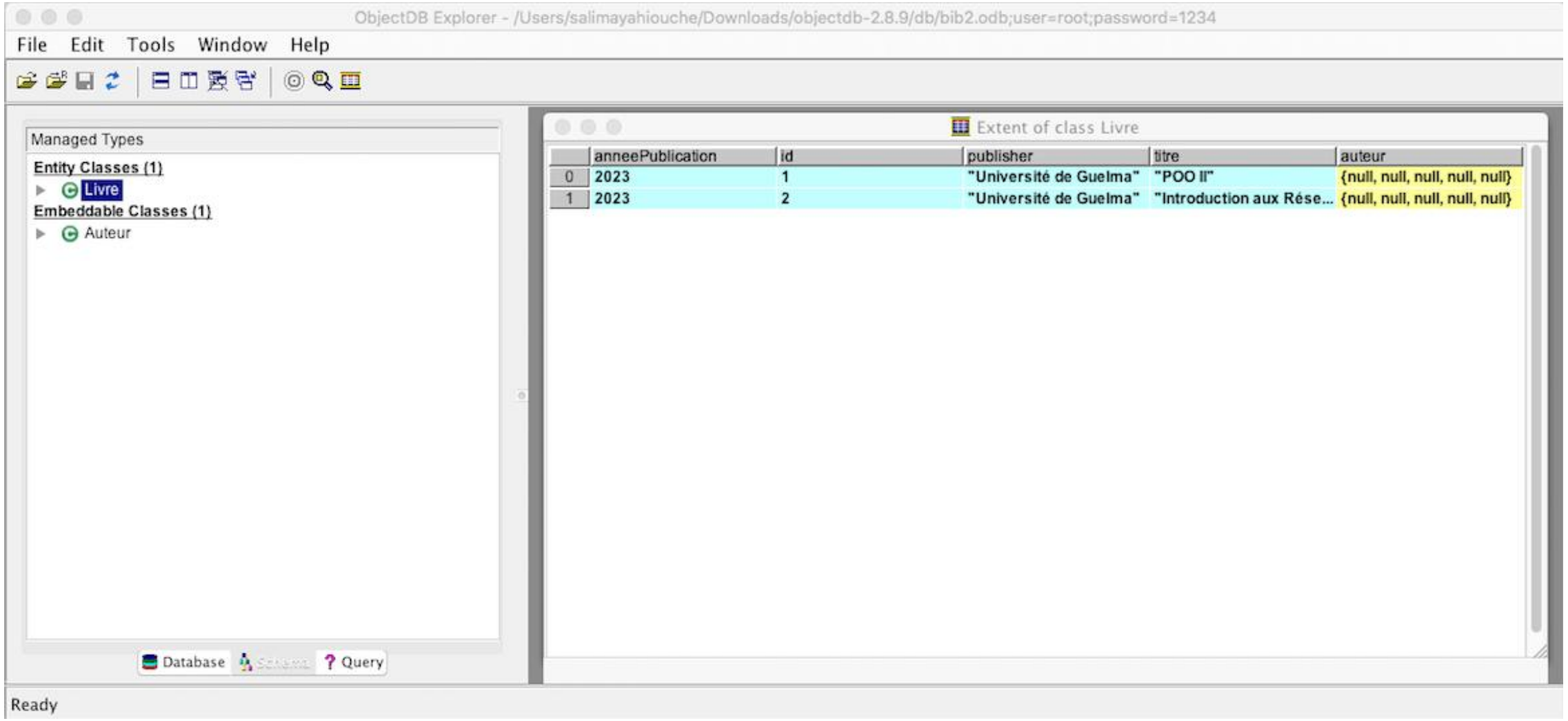


Bases de données orientées objets avec ObjectDB – Connexion et transaction - Exemple

```java
import javax.persistence.*;
@Entity
public class Livre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String titre;
    @Embedded
    private Auteur auteur;
    private int anneePublication;
    private String publisher;

    public Livre(String titre, Auteur auteur, int anneePublication, String publisher) {
        this.titre = titre; this.auteur = auteur;
        this.anneePublication = anneePublication; this.publisher = publisher;
    }
    @Override
    public String toString() {
        return "Livre{ titre= '" + titre + ", auteur= '" + auteur + ", anneePublication= " +
            anneePublication + ", publisher= '" + publisher + '}';
    }
}

@Embeddable
class Auteur {
    private String nom, prenom;
    public Auteur(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }
    @Override
    public String toString() {
        return "Auteur{ nom= '" + nom + ", prénom= '" + prenom + '}';
    }
}
```

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<String, String> ps= new HashMap<String, String>();
        ps.put("javax.persistence.jdbc.user", "root");
        ps.put("javax.persistence.jdbc.password", "1234");
        // Création de l'EntityManagerFactory
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("$objectdb/db/bib1.odb", ps);

        // Création de l'EntityManager
        EntityManager em = emf.createEntityManager();

        // Début de la transaction
        em.getTransaction().begin();

        Auteur auteur; Livre livre;

        // Création et persistance des objets Livre
        auteur = new Auteur("Hannousse", "Abdelhakim");
        livre = new Livre("POO II", auteur, 2023, "Université de Guelma");
        em.persist(livre);
        auteur = new Auteur("Brahimi", "Said");
        livre = new Livre("Introduction aux Réseaux", auteur, 2023, "Université de Guelma");
        em.persist(livre);

        // Valider la transaction
        em.getTransaction().commit();

        // Fermeture des ressources
        em.close(); emf.close();
    }
}
```

Bases de données orientées objets avec ObjectDB – Connexion et transaction – Exemple (ObjectDB Explorer)



Bases de données orientées objets avec ObjectDB – Connexion et transaction - Exemple

```java
import javax.persistence.*;
@Entity
public class Livre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String titre;
    @OneToOne(cascade=CascadeType.PERSIST)
    private Auteur auteur;
    private int anneePublication;
    private String publisher;

    public Livre(String titre, Auteur auteur, int anneePublication, String publisher) {
        this.titre = titre; this.auteur = auteur;
        this.anneePublication = anneePublication; this.publisher = publisher;
    }
    @Override
    public String toString() {
        return "Livre{ titre= '" + titre + ", auteur= '" + auteur + ", anneePublication= " +
            anneePublication + ", publisher= '" + publisher + '}';
    }
}
@Entity
class Auteur {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String nom, prenom;
    public Auteur(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }
    @Override
    public String toString() { return "Auteur{ nom= '" + nom + ", prénom= '" + prenom + '}'; }
}
```

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<String, String> ps= new HashMap<String, String>();
        ps.put("javax.persistence.jdbc.user", "root");
        ps.put("javax.persistence.jdbc.password", "1234");
        // Création de l'EntityManagerFactory
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("$objectdb/db/bib1.odb", ps);

        // Création de l'EntityManager
        EntityManager em = emf.createEntityManager();

        // Début de la transaction
        em.getTransaction().begin();

        Auteur auteur; Livre livre;

        // Création et persistance des objets Livre
        auteur = new Auteur("Hannousse", "Abdelhakim");
        livre = new Livre("POO II", auteur, 2023, "Université de Guelma");
        em.persist(livre);
        auteur = new Auteur("Brahimi", "Said");
        livre = new Livre("Introduction aux Réseaux", auteur, 2023, "Université de Guelma");
        em.persist(livre);

        // Valider la transaction
        em.getTransaction().commit();

        // Fermeture des ressources
        em.close(); emf.close();
    }
}
```

Bases de données orientées objets avec ObjectDB – Manipulation des entités avec EntityManager

- L' ***EntityManager*** offre plusieurs méthodes pour manipuler directement les entités :
  - Méthode ***find(Class, ID)***: récupère une entité spécifique de la BDD en fonction de sa clé primaire. Une fois l'objet récupéré, il peut être modifié et sauvegardé.
  - Méthode ***refrech(Object)***: rafraîchit l'état d'une entité pour s'assurer d'avoir la dernière version de l'objet. Utile si un autre ***EntityManager*** a modifié l'objet.
  - Méthode ***remove(Object)***: supprime une entité persistante de la base de données. Attention à la propagation de la suppressions des objets référencés.

```java
import javax.persistence.*;
public class ManipulationEntites {
    public static void main(String[] args) {
        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("$objectdb/db/bib2.odb");
        EntityManager em = emf.createEntityManager();

        // Exemple d'utilisation de find
        Livre livre = em.find(Livre.class, 1);
        System.out.println("Livre trouvé : " + livre);

        // Exemple d'utilisation de refresh
        em.getTransaction().begin();
        em.refresh(livre);
        System.out.println("Livre après rafraîchissement : " + livre);
        em.getTransaction().commit();

        // Exemple d'utilisation de remove
        em.getTransaction().begin();
        em.remove(livre);
        System.out.println("Livre supprimé");
        em.getTransaction().commit();

        // Fermeture des ressources
        em.close();
        emf.close();
    }
}
```

Bases de données orientées objets avec ObjectDB – Requêtes JPA avec JPQL

- Les requêtes JPA (Java Persistence API) avec JPQL (Java Persistence Query Language) sont utilisées pour interagir avec des entités persistantes dans une base de données à objets.
- JPQL est un langage de requête indépendant du fournisseur qui permet d'effectuer des requêtes sur des objets entité, plutôt que sur des tables de base de données.
- JPQL ressemble à SQL, mais il utilise des noms d'entités Java et de champs plutôt que des noms de tables et de colonnes.
- Similaire aux requêtes SQL, les requêtes JPQL comprennent des clauses telles que **SELECT**, **FROM**, **WHERE**, **ORDER BY**, **GROUP BY**, etc. Elles sont utilisées pour décrire les données que vous souhaitez récupérer.
- Contrairement aux requêtes SQL, les requêtes JPQL peuvent inclure des paramètres qui sont remplacés par des valeurs spécifiques lors de l'exécution de la requête. Les paramètres sont souvent utilisés pour rendre les requêtes dynamiques.

Bases de données orientées objets avec ObjectDB – Requêtes JPA avec JPQL

- Les requêtes JPQL s'exécutent via l'***EntityManager*** en créant un objet ***Query*** ou ***TypedQuery*** avec ***createQuery(..)***.
  - ***Query:*** est une interface générale pour toutes les requêtes JPA. Elle permet d'exprimer des requêtes de manière flexible sans spécifier le type des résultats. Utile si le type de retour n'est pas connu.
  - ***TypedQuery***: sous-interface de ***Query*** qui est paramétrée avec le type des résultats attendus. Elle offre une sécurité de type, car le type des résultats est spécifié lors de la création de la requête. Utilie si le type de retour est connu.
- En JPA, ***NativeQuery*** permet d'exécuter des requêtes SQL natives directement dans une application Java. Une ***NativeQuery*** est créée par l'***EntityManager*** en appelant **createNativeQuery(..)**.
- Une ***NativeQuery*** est utile dans les cas suivants:
  - Utilise des fonctionnalités spécifiques qui ne sont pas prises en charge par JPQL.

Bases de données orientées objets avec ObjectDB – Requêtes JPA avec JPQL

- Migration des bases de données.
- Assurer une certaine performance d'exécution des opérations complexes.

```
String sqlQuery = "SELECT * FROM Livre";
Query nativeQuery = em.createNativeQuery(sqlQuery, Livre.class);
```

- ***NamedQuery*** en JPA est une fonctionnalité qui permet de définir des requêtes SQL dans les entités Java de manière statique. Ces requêtes sont précompilées et associées à un nom dans la classe entité, ce qui facilite leur réutilisation à plusieurs endroits dans le code.

```
@Entity
@NamedQuery(
        name = "chercheLivreParTitle",
        query = // requête ici
)
class Livre { ... }
// Utilisation
TypedQuery<Livre> query = em.createNamedQuery("chercheLivreParTitle", Livre.class);
```

Bases de données orientées objets avec ObjectDB – Requêtes JPA avec JPQL

- Il existe deux méthodes pour récupérer les résultats d'une requête JPQL:
  - ***getResultList()*** **:** pour une liste de résultats.
  - ***getSingleResult()*** pour un résultat unique. peut retourner une erreur si plus d'une valeur est retournée par la requête, ou si aucune valeur n'est présente.

Bases de données orientées objets avec ObjectDB – Requêtes JPA avec JPQL – Exemples des requêtes

- Sélection:

```
TypedQuery<Livre> query = em.createQuery("SELECT l FROM Livre l", Livre.class);
List<Livre> livres = query.getResultList();
```

- Sélection conditionnée:

```
TypedQuery<Livre> query =
   em.createQuery("SELECT l FROM Livre l WHERE l.anneePublication > 2020", Livre.class);
List<Livre> livres = query.getResultList();
```

- Jointure et sélection des entités distincts:

```
TypedQuery<Auteur> query =
    em.createQuery("SELECT DISTINCT a FROM Livre l JOIN l.auteur a", Auteur.class);
List<Auteur> auteurs = query.getResultList();
```

```
TypedQuery<String> query =
  em.createQuery("SELECT l.titre FROM Livre l JOIN l.auteur a WHERE a.nom = :nomAuteur", String.class);
query.setParameter("nomAuteur", "Hannousse");
List<String> titres = query.getResultList();
```

Bases de données orientées objets avec ObjectDB – Requêtes JPA avec JPQL – Exemple des requêtes

- Utilisation des fonctions:

```
TypedQuery<Long> query = em.createQuery("SELECT COUNT(l) FROM Livre l", Long.class);
Long nombreLivres = query.getSingleResult();
```

- Mise à jour des entités:

```
em.getTransaction().begin();
Query updateQuery = em.createQuery("UPDATE Livre l SET l.anneePublication = :nouvelleAnnee");
updateQuery.setParameter("nouvelleAnnee", 2022);
updateQuery.executeUpdate();
em.getTransaction().commit();
```

```
em.getTransaction().begin();
Query deleteQuery = em.createQuery("DELETE FROM Livre l WHERE l.anneePublication <= :anneeLimite");
deleteQuery.setParameter("anneeLimite", 2022);
deleteQuery.executeUpdate();
em.getTransaction().commit();
```

Java et bases de données relationnelles

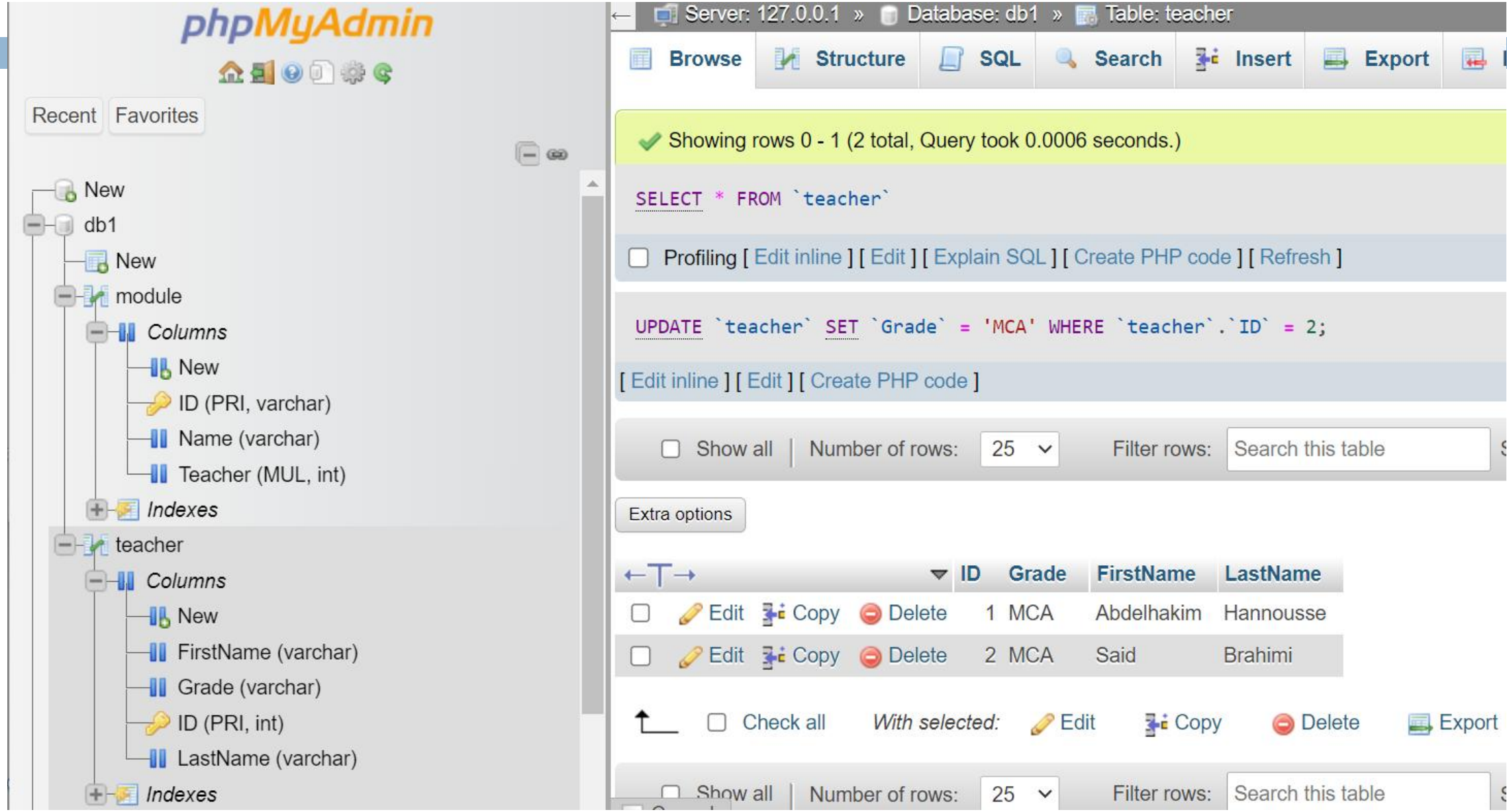
Java et bases de données relationnelles (1/3)

- Il est possible de se connecter à une base de données relationnelle en Java.
- La manière standard de se connecter à une base de données relationnelle en Java est d'utiliser JDBC (Java Database Connectivity), qui est une API standard pour l'accès aux bases de données relationnelles.
  - **Chargement du pilote JDBC :** Charger le pilote JDBC spécifique à la base de données souhaitee.
    - MySQL JDBC pour une base de données MySQL depuis: https://www.mysql.com/products/connector/
    - Oracle JDBC pour une base de données Oracle depuis: https://www.oracle.com/database/technologies/appdev/jdbc-downloads.html
  - **Création de l'URL de connexion :** Créer une URL de connexion qui spécifie l'emplacement du base de données, ainsi que d'autres paramètres de connexion tels que le nom d'utilisateur et le mot de passe.

Java et bases de données relationnelles (2/3)

- **Établissement de la connexion :** Utiliser la classe **DriverManager** pour obtenir une connexion à la base de données en fournissant l'URL de connexion, le nom d'utilisateur et le mot de passe.
- **Exécution de requêtes SQL :** Créer des objets **Statement** ou **PreparedStatement** pour exécuter des requêtes SQL telles que **SELECT, INSERT, UPDATE, DELETE,** etc.
  - **Statement** est utilisée pour exécuter des requêtes SQL simples sans paramètres.
  - **PreparedStatement** est utilisée pour exécuter des requêtes SQL paramétrées, ce qui permet de passer des paramètres dynamiques à la requête.
- **Gestion des exceptions :** Gérer les exceptions qui peuvent survenir lors de la connexion à la base de données en utilisant des blocs **try-catch** pour assurer une gestion appropriée des erreurs.

Exemple (3/3)


phpMyAdmin
Recent
Favorites
New
db1
New
module
Columns
New
ID (PRI, varchar)
Name (varchar)
Teacher (MUL, int)
Indexes
teacher
Columns
New
FirstName (varchar)
Grade (varchar)
ID (PRI, int)
LastName (varchar)
Indexes
Server: 127.0.0.1 » Database: db1 » Table: teacher
Browse
Structure
SQL
Search
Insert
Export
Showing rows 0 - 1 (2 total, Query took 0.0006 seconds.)
SELECT * FROM `teacher`
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]
UPDATE `teacher` SET `Grade` = 'MCA' WHERE `teacher`.`ID` = 2;
[Edit inline] [Edit] [Create PHP code]
Show all
Number of rows: 25
Filter rows: Search this table
Extra options
ID Grade FirstName LastName
Edit Copy Delete 1 MCA Abdelhakim Hannousse
Edit Copy Delete 2 MCA Said Brahimi
Check all
With selected:
Edit
Copy
Delete
Export
Show all
Number of rows: 25
Filter rows: Search this table

Exemple (3/3)

```java
import java.sql.*;
public class MySQLConnectionExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/db1";
        String username = "hannousse";
        String password = "pass123";

        try {
            Connection connection = DriverManager.getConnection(url, username, password);
            Class.forName("com.mysql.cj.jdbc.Driver");
            Statement statement = connection.createStatement();
            String sql = "SELECT * FROM Teacher";
            ResultSet resultSet = statement.executeQuery(sql);
            while (resultSet.next())
                System.out.println(resultSet.getString("FirstName") + " " + resultSet.getString
                    ("LastName"));

            PreparedStatement preparedStatement =
            connection.prepareStatement("SELECT * FROM teacher WHERE Grade = ?");
            preparedStatement.setString(1, "MCA");
            resultSet = preparedStatement.executeQuery();
            while (resultSet.next())
                System.out.println(resultSet.getString("FirstName") + " " + resultSet.getString
                    ("LastName"));
        } catch (ClassNotFoundException e) {
            System.err.println("Error: MySQL JDBC driver not found!");
            e.printStackTrace();
        } catch (SQLException e) {
            System.err.println("Error: Failed to establish database connection!");
            e.printStackTrace();
        }
    }
}
```