

Cours 4

JDBC

HEI 2021 / 2022

Introduction

JDBC?

Qu'est ce que JDBC?

- Acronyme de "Java DataBase Connectivity".
- C'est une API du JDK qui permet d'accéder à n'importe quelle BDD SQL et de manipuler les données de cette base.

Une spécification

- JDBC décrit via des interfaces comment interagir avec une base de données.
- Les interfaces de l'API JDBC sont dans les packages suivants :
 - `java.sql`
 - `javax.sql`

Implémentations

- Toutes les bases de données sont différentes les unes des autres.
- JDBC ne fourni pas d'implémentation.
- Ce sont les constructeurs des bases de données qui fournissent les implémentations (Oracle, PostgreSQL ou Microsoft).

Driver

- Une implémentation de JDBC est appelé un driver.
- Le driver fourni les différentes classes qui implémentent l'API JDBC.
- Par exemple, la fondation MariaDB fourni un driver pour MariaDB appelé Connector/J.

Connector/J

- Comme les drivers ne sont pas fournis avec le JDK il est nécessaire d'importer ceux-ci dans notre projet.
- La dépendance maven pour Connector/J est la suivante :

```
<dependency>  
  <groupId>org.mariadb.jdbc</groupId>  
  <artifactId>mariadb-java-client</artifactId>  
  <version>2.5.0</version>  
</dependency>
```

```
graph TD; Code[Code] --> JDBC_API[JDBC API]; JDBC_API --> JDBC_Driver[JDBC Driver]; JDBC_Driver --> DB[(DB)];
```

Code

JDBC API

- API Générique pour les base de données relationnelles

JDBC Driver

- Implémentation pour MySQL par exemple

DB

Communication


- La communication avec la bdd est faite avec le langage SQL.
- JDBC met à disposition des méthodes permettant d'exécuter des requêtes SQL (SELECT, INSERT, UPDATE, ...)


Etapes d'une requête


Connexion
Ouverture
de la
connexion
avec la bdd


Statement
Création
d'un
statement
qui va
décrire la
requête
SQL


Requête
Exécution
de la
requête


Résultats
Traitement
du résultat
de la
requête au
besoin (rq
SELECT)


Fermeture
Fermeture
des objets
(résultats,
statement
et
connection)

Connexion

Connexion à une base de données avec JDBC

L'interface DataSource

- L'interface `javax.sql.DataSource` décrit la manière de se connecter à une base de données.
- Une fois configurée, les connexions peuvent être obtenues via la méthode `getConnection`.

```
Connection connection = dataSource.getConnection();
```

La classe MariaDbDataSource

- Connector/J possède une classe qui implémente l'interface DataSource : la classe **MariaDbDataSource**.
 - Cette classe est utilisée pour fournir les informations sur la manière de se connecter à MariaDB.
- Cette classe est la seule implémentation provenant Connector/J que nous utiliserons dans nos projets.

Paramètres de connexion

- Les paramètres suivants sont nécessaires pour se connecter à une base MariaDB :

| Informations | Description |
|---------------|--|
| Server name | Url du server qui héberge la bdd |
| Port | Port sur lequel tourne la base de données |
| Database name | Nom de la base (ou schéma) à utiliser |
| User | Login de l'utilisateur qui doit se connecter |
| Password | Mot de passe associé au login |

Paramètres de connexion

- La classe MariaDbDataSource a les setters nécessaire pour renseigner les informations :

```
MariaDbDataSource mariadbDataSource = new MariaDbDataSource();  
mariadbDataSource.setServerName("localhost");  
mariadbDataSource.setPort(3306);  
mariadbDataSource.setDatabaseName("hei_db");  
mariadbDataSource.setUser("hei");  
mariadbDataSource.setPassword("hei_password");
```

L'interface Connection

- L'interface `java.sql.Connection` représente une connexion à la base de donnée.
- Lorsque l'on appelle la méthode `getConnection()` de l'objet `DataSource` une connexion vers la base de données est établie.

Fermer une connexion

- Une connexion à une base de donnée doit être fermée lorsqu'elle n'est plus utilisée.
 - La méthode `close()` de l'interface `Connection` ferme la connexion :

```
// Creation d'une nouvelle connexion
Connection connection = dataSource.getConnection();
// Utilisation de la connexion
...
// fermeture
connection.close();
```

Fermer une connexion

- Même en cas d'erreur il faut penser à fermer la connexion :

```
try {  
    connection = dataSource.getConnection();  
    // utilisation de la connexion  
} catch (SQLException e) {  
    // Gestion erreur  
} finally {  
    try {  
        connection.close();  
    } catch (SQLException e) {  
        // Gestion erreur  
    }  
}
```

Try with resource

- L'interface `Connection` étend `AutoCloseable` et peut donc être utilisée dans un try with resource.

```
try(Connection connection = dataSource.getConnection()) {  
    // Utilisation de la connexion  
} catch (SQLException e) {  
    // Gestion exception  
}
```

Statements

L'interface Statement

- Toutes les requêtes SQL sont faites via l'utilisation d'un objet statement.
 - L'interface `java.sql.Statement` représente ces objets.
- Une instance de statement est récupérée depuis l'objet connexion via la méthode `createStatement()`.

```
Statement statement = connection.createStatement();
```

Fermeture d'un statement

- Comme pour les connexions, il est nécessaire de fermer les statement lorsqu'ils ne sont plus utilisés.
 - L'interface `Statement` étend également `AutoCloseable`.

```
try (Statement statement = connection.createStatement()) {  
    // Utilisation du statement  
} catch (SQLException e) {  
    // Gestion des exceptions  
}
```

Requêtes SELECT

- Pour faire une requête SELECT, l'interface `Statement` possède la méthode `executeQuery()`.
 - Cette méthode prend une requête SQL en paramètre et retourne un `ResultSet`.

```
String sqlQuery = "SELECT title, author FROM book";  
ResultSet resultSet = statement.executeQuery(sqlQuery);
```

Autres requêtes

- Pour les autres requêtes SQL comme INSERT INTO, UPDATE or DELETE, la méthode `executeUpdate()` doit être utilisée.
 - Elle retourne le nombre de lignes modifiées dans la base.

```
String sqlQuery = "DELETE FROM book WHERE author='Some author'";  
int nbLines = statement.executeUpdate(sqlQuery);
```


Interface PreparedStatement

- L'interface `java.sql.PreparedStatement` est un type particulier de statements.
- Une instance est créée via l'usage de la méthode `prepareStatement()` de la classe `Connection`.
 - Une requête SQL est directement passée à la création du statement.

```
PreparedStatement statement = connection.prepareStatement(sqlQuery);
```

Exécution de requêtes

- Un `PreparedStatement` possède les même méthode pour lancer les requêtes :
 - `executeQuery()` pour les requêtes `SELECT`;
 - `executeUpdate()` pour les autres requêtes.
- Ces 2 méthodes ne prennent pas de paramètre car la requête a été passée lors de la création du statement.

Exécution de requêtes

```
String sqlQuery = "SELECT title, author FROM book";  
PreparedStatement statement = connection.prepareStatement(sqlQuery);  
ResultSet resultSet = statement.executeQuery();
```

```
String sqlQuery = "DELETE FROM book WHERE author='Some author'";  
PreparedStatement statement = connection.prepareStatement(sqlQuery);  
int nbLines = statement.executeUpdate();
```

Requêtes avec paramètres

- Le but principal des PreparedStatement est d'ajouter des paramètres à nos requêtes.

Récupération des livres d'un auteur :

`SELECT title, author FROM book WHERE author='Racine'`



`SELECT title, author FROM book WHERE author= ?`

Passage de l'auteur en paramètre

Requêtes avec paramètres

- Si la requête est paramétrée, il est nécessaire de passer les valeurs des paramètres avant d'exécuter celle-ci.
 - Si la paramètres ne sont pas défini un Exception est levée.
- L'interface **PreparedStatement** fournit plusieurs méthodes pour passer les valeurs de paramètres.

Passage de paramètres

- Chaque méthode permettant de setter un paramètre ont cet aspect :

setType(int parameterIndex, T value)

The diagram consists of a green box labeled 'Type du paramètre' centered below the text. Two green lines extend upwards from the box, one to the left and one to the right. Each line ends in a horizontal bar that underlines the 'Type' in 'setType' and the 'T' in 'T value' respectively.

- Chaque méthode formatera la valeur en fonction de son type, et l'ajoutera à la requête (plus de problèmes de guillemets).

Passage de paramètres

| Type | Method | Format |
|---------|-----------|--|
| Integer | setInt | None |
| Float | setFloat | None |
| String | setString | Ajoute des quotes autour de la chaîne et échappe celle à l'intérieur |
| Date | setDate | Transforme la date pour être compatible avec la bdd |
| ... | | |

Paramètre index (1-based)

- Le premier paramètre de setter définit la position du paramètre dans la requête.

UPDATE book set title=?, author=? WHERE id_book=?

Paramètre 1

Paramètre 2

Paramètre 3

```
statement.setString(1, "The Lord of the ring");  
statement.setString(2, "JRR Tolkien");  
statement.setInt(3, 42);
```


Passage de paramètres

- Exemple complet :

```
String sqlQuery = "UPDATE book set title=?, author=? WHERE id_book=?";
try (PreparedStatement statement = connection.prepareStatement(sqlQuery)) {
    statement.setString(1, "The Lord of the ring");
    statement.setString(2, "JRR Tolkien");
    statement.setInt(3, 42);
    int nbRows = statement.executeUpdate();
    System.out.println(String.format("%d rows have been modified.", nbRows));
} catch (SQLException e) {
    // Gestion des exceptions
}
```

Avantages des PreparedStatement

- On pourrait d'utiliser la concaténation au lieu d'utiliser les PreparedStatement :

```
String query = "SELECT * FROM book WHERE author='" + author + "'";
```

- Mais les PreparedStatement on certains avantages sur la concaténation.
- La concaténation est **très fortement** déconseillée pour écrire des requêtes SQL.

Lisibilité

- L'utilisation de la concaténation n'est pas aussi simple qu'il n'y paraît. Tout le formatage doit être géré par le développeur :

```
String queryWithConcatenation = "UPDATE book set  
    title='"+title.replace("'", "\\')+ "',  
    author='"+author.replace("'", "\\')+ "',  
    publish_date='"+dateFormat.format(publishDate)+"'  
WHERE id_book="+bookId;
```

Sécurité (SQL Injection)

- Si un développeur utilise la concaténation et oublie certains aspect sécurité cela peut mener à de grave problèmes de sécurité :

```
"DELETE FROM book WHERE author='" + author + "'";  
author = "xxx' OR 1=1; -- ";
```

- ➔ "DELETE FROM book WHERE author='xxx' OR 1=1; -- "
- ➔ "DELETE FROM book WHERE 1=1"
- ➔ "DELETE FROM book"

- Si les paramètres en entrée ne sont pas vérifiés, on perd toutes les données de la base.

Performances

- L'utilisation des PreparedStatement permet aussi à la base de donnée d'utiliser du cache.
- Les requêtes SQL sont envoyées à la base et compilées à la création du Statement (requête avec les "?").
 - La requête sera considérée comme étant la même, même si la valeurs des paramètres sont différents.

ResultSet

Lecture des résultats d'un SELECT

L'interface ResultSet

- L'appel de la méthode `executeQuery()` sur `Statement` ou un `PreparedStatement`, retourne un `java.sql.ResultSet`.

```
ResultSet resultSet = statement.executeQuery(query);
```

- Un `ResultSet` contient toutes les lignes que le SELECT a retourné.

Fermer un ResultSet

- Comme pour la connection ou un statement, un resultSet doit être fermé après utilisation.
 - L'interface `ResultSet` étend également `AutoCloseable`.

```
try (ResultSet resultSet = statement.executeQuery(query)) {  
    // Exploitation des résultats  
} catch (SQLException e) {  
    // Gestion des erreurs  
}
```

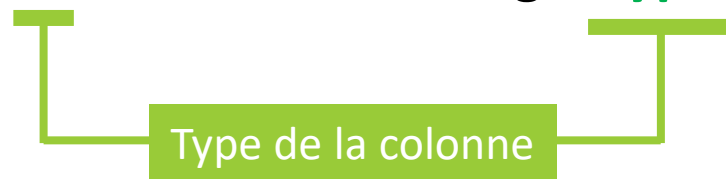

Parcours d'un ResultSet

- Un **ResultSet** contient 0, 1 ou plusieurs lignes en fonction du résultat de la requête. Ces lignes seront lues une par une.
- Pour chaque ligne, les différentes colonnes de celle-ci sont accessibles.

Récupérer la valeur d'une colonne

- L'interface `ResultSet` possède des méthodes pour récupérer les valeurs des colonnes.
- La méthode dépend du type de la valeur récupérée.
 - Comme pour les `PreparedStatement` avec les setters :

`T value = resultSet.getType(String columnName)`



Récupérer la valeur d'une colonne

- Comme pour les setters de `PreparedStatement`, les getter s'occupe de tout formatage :

```
String title = resultSet.getString("title");  
Date publishingDate = resultSet.getDate("publish_date");  
Integer bookId = resultSet.getInt("id");
```

Ligne courante

- Tout au long du parcours d'un `ResultSet`, il y a une ligne courante.
 - Elle est déterminée par un curseur interne qui pointe vers une des ligne du Set.
- L'appel d'un getter s'applique à la colonne spécifiée de la **ligne courante**.

Méthode next()

- La méthode `next()` déplace le curseur sur la ligne suivante du Set.
 - Lors de la création du `ResultSet`, le curseur est positionné **avant** la première ligne.
- Cette méthode retourne un booléen, true s'il y a une ligne, false sinon.

```
resultSet = statement.executeQuery(query); // pas de ligne courante  
resultSet.next(); // ligne 1  
resultSet.next(); // ligne 2
```

Lecture d'un ResultSet

- Exemple complet :

```
String query = "SELECT title, author FROM book";
try (ResultSet resultSet = statement.executeQuery(query)){
    while(resultSet.next()) {
        String title = resultSet.getString("title");
        Date publishingDate = resultSet.getDate("publish_date");
        Integer bookId = resultSet.getInt("id");
        System.out.println(String.format("Book n°%d: %s (Published %s)",
                                         bookId, title, publishingDate));
    }
} catch (SQLException e) {
    // Gestion des erreurs
}
```

Gestion des dates

Type Date

- En java, l'objet `Date` classique est issu du package `java.util`.

```
Date date = new Date();
```

- L' API JDBC utilise un autre type de date contenu dans le package `java.sql` package:
 - `java.sql.Date` pour une date;
 - `java.sql.Timestamp` pour une date et un temps.

Récupérer Date

- Comme `java.sql.Date` et `java.sql.Timestamp` étendent la classe `java.util.Date`, rien ne doit être fait pour récupérer une date :

```
Date dateValue = resultSet.getDate("date_column");  
Date sqlDateAndTime = resultSet.getTimestamp("datetime_column");
```

`java.util.Date`

Type JDBC

Positionner une Date

- Lorsque l'on passe une date à un statement, il est nécessaire de convertir l'objet `java.util.Date` via la méthode `getTime()` :

```
preparedStatement.setDate(1, new java.sql.Date(classicDateObject.getTime()));  
preparedStatement.setTimestamp(1, new Timestamp(classicDateObject.getTime()));
```

API Java Time

- Depuis Java 8 une nouvelle API est disponible pour gérer les dates.
 - Ces classes et interface sont définies dans le package `java.time`.
- Les classes principales sont :
 - `java.time.LocalDate` pour une date;
 - `java.time.LocalDateTime` pour une date et un temps.

Compatibilité JDBC

- L'API JDBC n'a pas de getter/setter pour gérer ces types de dates.

```
Date date = resultSet.getDate("date_column");
```

Classic `java.util.Date` type

```
LocalDate dateJava8 = resultSet.getLocalDate("date_column");
```

New Java Time API `java.time.LocalDate`

Conversion de types

- Une première solution pour utiliser des `LocalDate` est de les convertir en objets `Date` avant d'appeler les méthodes JDBC.
- Une fois converties, les méthodes classiques peuvent être utilisées.

java.sql.Date en LocalDate

- La classe `java.sql.Date` possède la méthode `toLocalDate()` :

```
java.sql.Date classicDate = resultSet.getDate("date_column");  
LocalDate dateJava8 = classicDate.toLocalDate();
```

LocalDate en java.sql.Date

- La classe `java.sql.Date` possède la méthode statique `valueOf()` qui transforme une `LocalDate` en `java.sql.Date`:

```
LocalDate dateJava8 = LocalDate.now();  
statement.setDate(1, java.sql.Date.valueOf(dateJava8));
```

Utilisation de méthode générique

- L'API JDBC API possède des méthodes qui gèrent tout type d'objet.
 - La transformation est faite via l'utilisation du driver de bdd.
 - Si le driver ne sait pas gérer le type cela déclenche des erreurs

```
LocalDate dateJava8 = resultSet.getObject("date_column", LocalDate.class);  
statement.setObject(1, dateJava8, Types.DATE);
```

SQL Column type

Identifiants générés

Auto increment en MariaDB

- En MariaDB, les colonnes peuvent être déclarées en "auto increment".
- Lors de l'ajout d'une ligne dans une table la colonne avec l'auto increment, sera incrémentée.

Generated Keys

- Avec JDBC, les auto increment sont appelés des generated keys.
- Ils peuvent être retournés par la base de données lors de l'exécution de requêtes.
 - Par défaut, ce n'est pas le cas

Récupération de generated keys

- Pour récupérer les generated keys, un paramètre supplémentaire doit être passé lors de la spécification de la requête SQL :

```
PreparedStatement preparedStatement = connection.prepareStatement(  
    sqlQuery, Statement.RETURN_GENERATED_KEYS);  
  
Statement statement = connection.createStatement();  
statement.executeUpdate(sqlQuery, Statement.RETURN_GENERATED_KEYS);
```

Getting a generated key

- Après l'exécution de la requête, les generated keys peuvent être récupérées dans le `ResultSet` par la méthode `getGeneratedKeys()`.

```
ResultSet ids = stmt.getGeneratedKeys();
if (ids.next()) {
    generatedId = ids.getLong(1);
}
```

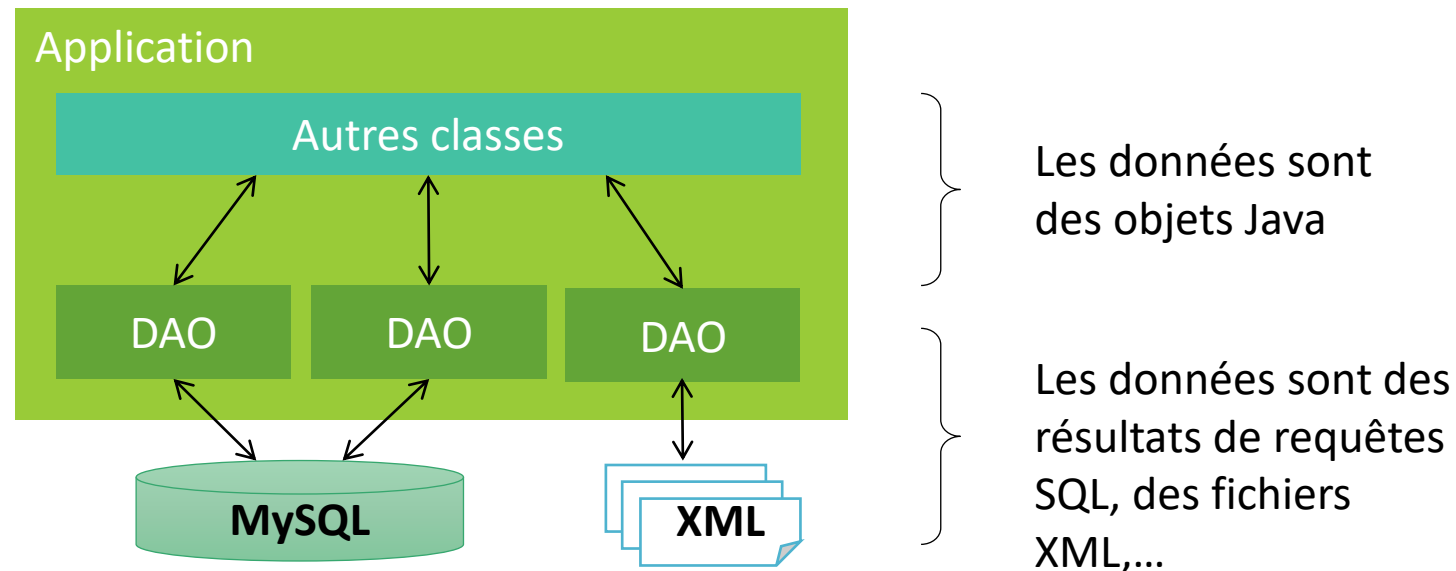
Data Access Object

Organisation

- Lors du design d'un projet, la plupart du temps on organise notre code dans différentes classe en fonction de leur usage.
- Les classes qui gèrent l'accès aux données sont appelées Data Access Objects ou DAO.

Abstraction

- Le but d'un DAO est d'abstraire la partie technique de l'accès aux données du reste de l'appli.



Exemples

Nos données

- Imaginons que nous avons un application qui gère des livres.
- En base de données on a un table **book** avec les colonnes suivantes :
 - book_id
 - title
 - author
 - publication_date

Entité

- Dans le code java, un livre est un objet de type **Book**.

```
public class Book {  
    private Integer id;  
    private String title;  
    private String author;  
    private LocalDate publicationDate;  
  
    public Book() {}  
  
    public Book(Integer id, String title, String author, LocalDate publicationDate) {  
        this.id = id;  
        this.title = title;  
        this.author = author;  
        this.publicationDate = publicationDate;  
    }  
}
```

DAO

- Notre DAO aura les méthodes suivantes :
 - `private DataSource getDataSource();`
 - `public List<Book> listAll ();`
 - `public List<Book> listAllByAuthor(String author);`
 - `public Book getByld(Integer id);`
 - `public void delete(Integer id);`
 - `public Book add(String title, String author, LocalDate publicationDate);`

Lister tous les livres

```
public List<Book> listAll() {
    List<Book> listOfBooks = new ArrayList<>();
    try (Connection connection = getDataSource().getConnection()) {
        try (Statement statement = connection.createStatement()) {
            try (ResultSet results = statement.executeQuery("select * from books")) {
                while (results.next()) {
                    Book book = new Book(
                        results.getInt("book_id"),
                        results.getString("title"),
                        results.getString("author"),
                        results.getDate("publication_date").toLocalDate());
                    listOfBooks.add(book);
                }
            }
        }
    } catch (SQLException e) {
        // Gestion des erreurs
        e.printStackTrace();
    }
    return listOfBooks;
}
```

Liste par auteur

```
public List<Book> listAllByAuthor(String author) {
    List<Book> listOfBooks = new ArrayList<>();
    try (Connection connection = getDataSource().getConnection()) {
        try (PreparedStatement statement = connection.prepareStatement("select * from books where author=?")) {
            statement.setString(1, author);
            try (ResultSet results = statement.executeQuery()) {
                while (results.next()) {
                    Book book = new Book(
                        results.getInt("book_id"),
                        results.getString("title"),
                        results.getString("author"),
                        results.getDate("publication_date").toLocalDate());
                    listOfBooks.add(book);
                }
            }
        }
    } catch (SQLException e) {
        // Gestion des erreurs
        e.printStackTrace();
    }
    return listOfBooks;
}
```

Récupération d'un livre

```
public Book getById(Integer bookId) {
    try (Connection connection = getDataSource().getConnection()) {
        try (PreparedStatement statement = connection.prepareStatement(
            "select * from books where book_id=?")) {
            statement.setInt(1, bookId);
            try (ResultSet results = statement.executeQuery()) {
                if (results.next()) {
                    return new Book(
                        results.getInt("book_id"),
                        results.getString("title"),
                        results.getString("author"),
                        results.getDate("publication_date").toLocalDate());
                }
            }
        }
    } catch (SQLException e) {
        // Gestion des erreurs
        e.printStackTrace();
    }
    return null;
}
```

Suppression d'un livre

```
public void delete(Integer bookId) {  
    try (Connection connection = getDataSource().getConnection()) {  
        try (PreparedStatement statement = connection.prepareStatement(  
            "delete from book where book_id=?")) {  
            statement.setInt(1, bookId);  
            statement.executeUpdate();  
        }  
    } catch (SQLException e) {  
        // Gestion des erreurs  
        e.printStackTrace();  
    }  
}
```


Ajout d'un livre

```
public Book add(String title, String author, LocalDate publicationDate) {
    try (Connection connection = getDataSource().getConnection()) {
        String sqlQuery = "insert into book(title, author, publication_date) VALUES(?,?,?)";
        try (PreparedStatement statement = connection.prepareStatement(
            sqlQuery, Statement.RETURN_GENERATED_KEYS)) {
            statement.setString(1, title);
            statement.setString(2, author);
            statement.setDate(3, Date.valueOf(publicationDate));
            statement.executeUpdate();

            ResultSet ids = statement.getGeneratedKeys();
            if (ids.next()) {
                return new Book(ids.getInt(1), title, author, publicationDate);
            }
        }
    } catch (SQLException e) {
        // Gestion des erreurs
        e.printStackTrace();
    }
    return null;
}
```

Sources

- <http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/>