Programmation orientée objet avec Java (application console Restaurant)

Guide adapté au projet : POO en Java, Maven, JDK 25. Modèle Restaurant / Table (OneToMany), Hibernate avec MySQL, DAO, interface console CRUD et Faker pour données de test.

Développement Web Programmation 26 min de lecture Niveau Intermédiaire

Document généré le 25/05/2026 à 20h08 · nouvy.fr/wiki/java-poo-restaurant-console

Sommaire

30 section(s) · 26 min de lecture

Résumé des concepts utilisés

Aller plus loin

Repository de travail et d'exemple : java-di25

Ce guide couvre la **programmation orientée objet (POO)** en Java à travers une **application console** de gestion de restaurants. Vous y trouverez les concepts POO, l'utilisation de **Maven**, **Hibernate** avec **MySQL**, et une interface en **ligne de commande** (CRUD restaurants, données de test avec Faker).

Prérequis : **JDK 25** (ou 21 LTS), **Maven**, **IntelliJ IDEA** et une base **MySQL** (schéma `test_di25`).

Pourquoi Java pour ce projet ?

- **Maven** : gestion des dépendances et structure de projet standardisée.
- **Multiplateforme** : une même base de code tourne sur Windows, macOS et Linux.
- **POO native** : Java est conçu autour des classes et objets.
- **IntelliJ IDEA** : IDE recommandé pour le développement Java (refactoring, Maven intégré).
- **Hibernate / MySQL** : persistance des données avec un ORM mature et une base répandue.

Les piliers de la POO en Java

La programmation orientée objet en Java repose sur quatre piliers fondamentaux. Chacun apporte une façon de structurer le code pour le rendre plus sûr, réutilisable et maintenable.

---

1. L'encapsulation

Idée : regrouper les **données** (attributs) et les **opérations** (méthodes) dans une classe, et **cacher les détails internes** au reste du programme. L'accès aux données se fait uniquement via des méthodes, jamais en exposant directement les champs.

Pourquoi c'est important :

- **Contrôle** : on peut valider les valeurs (ex. refuser un montant négatif) avant de modifier l'état.
- **Évolution** : on peut changer la façon dont les données sont stockées en interne sans casser le code qui utilise la classe.
- **Cohérence** : les invariants du métier (ex. « le solde ne peut pas être négatif ») sont garantis à un seul endroit.

Modificateurs de visibilité en Java :

Modificateur	Visible dans la classe	Dans les sous-classes	Partout
`private`	oui	non	non
`protected`	oui	oui	non
`public`	oui	oui	oui

En pratique, on met les **attributs en `private`** et on expose des **getters** (lecture) et **setters** (écriture) si besoin. Pour une donnée en lecture seule, on ne fournit qu'un getter.

Exemple :

```
public class CompteBancaire {
    private double solde;  // privé : invisible de l'extérieur

    public void deposer(double montant) {
        if (montant > 0) {
            solde += montant;
        }
    }

    public boolean retirer(double montant) {
        if (montant > 0 && montant <= solde) {
            solde -= montant;
            return true;
        }
        return false;
    }

    public double getSolde() {
        return solde;
    }
    // Pas de setSolde(double) : on évite de modifier le solde sans passer par
deposer/retirer
}
```

Copier

L'utilisateur du `CompteBancaire` ne peut pas faire `compte.solde = -100` ; il est obligé de passer par `deposer` et `retirer`, où les règles métier sont appliquées.

---

2. L'héritage

Idée : une classe peut **étendre** une autre classe (la **classe parente** ou **superclasse**) pour en réutiliser les attributs et méthodes, tout en ajoutant ou en modifiant du comportement. On évite de dupliquer du code et on crée une hiérarchie de types (ex. `Chien` et `Chat` sont des `Animal`).

Mécanismes clés :

- **`extends`** : la classe fille hérite des membres `public` et `protected` de la parente.
- **`super`** : pour appeler le constructeur ou une méthode de la superclasse (ex. `super(nom)` dans un constructeur).
- **`@Override`** : indique qu'on redéfinit une méthode déjà présente dans la parente ; le comportement de l'objet sera celui de la classe **réelle** (polymorphisme).

Exemple :

```
public class Animal {
    protected String nom;

    public Animal(String nom) {
        this.nom = nom;
    }

    public void parler() {
        System.out.println("...");
    }

    public String getNom() {
        return nom;
    }
}

public class Chien extends Animal {
    public Chien(String nom) {
        super(nom);  // obligatoire : appeler le constructeur du parent
    }

    @Override
    public void parler() {
        System.out.println("Wouf ! Je m'appelle " + nom);
    }
}

public class Chat extends Animal {
    public Chat(String nom) {
        super(nom);
    }

    @Override
    public void parler() {
        System.out.println("Miaou !");
    }
}
```

Copier

`Chien` et `Chat` ont tout ce qu'un `Animal` a (nom, getNom, parler), mais chacun donne sa propre implémentation de `parler()`.

---

3. Le polymorphisme

Idée : une **même référence** (variable ou paramètre) peut désigner des objets de **types concrets différents**. Le code qui utilise cette référence n'a pas besoin de savoir quel type exact est utilisé ; c'est la **méthode de l'objet réel** qui est appelée à l'exécution.

Exemple :

```
Animal a1 = new Chien("Rex");
Animal a2 = new Chat("Minou");

a1.parler();  // Affiche "Wouf ! Je m'appelle Rex"  → méthode de Chien
a2.parler();  // Affiche "Miaou !"                   → méthode de Chat
```

Copier

On peut écrire une méthode qui travaille avec *n'importe quel* `Animal` :

```
public static void faireParler(Animal a) {
    a.parler();  // selon le type réel (Chien, Chat, etc.)
}

faireParler(new Chien("Rex"));   // Wouf !
faireParler(new Chat("Minou"));  // Miaou !
```

Copier

Intérêt : écrire du code **générique** sans `if (a instanceof Chien) ... else if (a instanceof Chat) ...`. C'est un des piliers du design orienté objet.

---

4. L'abstraction

Idée : définir des **contrats** (ce que doit faire un type) sans donner toute l'implémentation. En Java : **interfaces** et **classes abstraites**. Ils permettent de manipuler des concepts (ex. « quelque chose qui peut être dessiné ») sans se soucier du détail concret (cercle, rectangle, etc.).

- **Interface** : liste de méthodes que toute classe « implémentant » l'interface doit fournir. Une classe peut implémenter **plusieurs** interfaces.
- **Classe abstraite** : classe qu'on ne peut pas **instancier** directement ; elle sert de modèle pour des sous-classes. Une classe ne peut **étendre** qu'une seule classe abstraite.

Exemple :

```
public interface Drawable {
    void draw();
}

public abstract class Shape implements Drawable {
    protected String color;

    public Shape(String color) {
        this.color = color;
    }

    public abstract double area();

    public String getColor() {
        return color;
    }
}

public class Rectangle extends Shape {
    private double largeur, hauteur;

    public Rectangle(String color, double largeur, double hauteur) {
        super(color);
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    @Override
    public double area() {
        return largeur * hauteur;
    }

    @Override
    public void draw() {
        System.out.println("Dessin d'un rectangle " + color + " (" + largeur + "x" + hauteur
+ ")");
    }
}
```

Copier

L'abstraction permet de **découpler** le code qui utilise ces types de leurs implémentations concrètes.

Prérequis : installer le JDK et Maven

Vérifier le JDK

```
java -version
javac -version
```

Copier

Recommandé : **JDK 25** (ou **JDK 21** LTS). Les versions LTS (17, 21, 25) sont privilégiées pour la stabilité et le support à long terme.

Installation rapide :

- **Windows** : Télécharger le JDK depuis Adoptium ou Oracle.
- **macOS** : `brew install openjdk@25` (ou `openjdk@21`)
- **Linux** : `sudo apt install openjdk-25-jdk` (Ubuntu/Debian) ou `openjdk-21-jdk`

Installer Maven

- **Windows** : Télécharger depuis maven.apache.org, extraire, ajouter `bin` au PATH.
- **macOS** : `brew install maven`
- **Linux** : `sudo apt install maven`

Vérification :

```
mvn -version
```

Copier

Installer IntelliJ IDEA

- **Téléchargement** : jetbrains.com/idea/download — choisir **IntelliJ IDEA Community** (gratuit) ou Ultimate.
- **Premier lancement** : indiquer le JDK 25 (ou 21) si demandé ; IntelliJ peut aussi télécharger un JDK pour vous.
- **Maven** : IntelliJ embarque Maven ; pas besoin de l'installer séparément pour travailler dans l'IDE (optionnel en ligne de commande).

Base de données MySQL

- **MySQL** (ou MariaDB) doit être installé et démarré.
- Créer un schéma pour le projet, par exemple : `CREATE DATABASE test_di25 CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;`
- Adapter dans `hibernate.cfg.xml` l'URL, le nom d'utilisateur et le mot de passe selon votre installation.

Créer et mettre en place le projet

Structure Maven

1. **Créer un projet Maven** dans IntelliJ : **File → New → Project** → **Maven**.
2. **Project SDK** : JDK 25 (ou 21).
3. Renseigner **GroupId** (ex. `com.test.di25`), **ArtifactId** (ex. `TestDI25Java`), **Location**.
4. Structure obtenue : `src/main/java`, `src/main/resources`, `pom.xml`.

Dépendances (pom.xml)

Dans `<dependencies>`, ajouter au minimum : **hibernate-core**, **mysql-connector-j**, **jakarta.persistence-api**. Optionnel : **datafaker** pour les données de test, **testng** pour les tests.

Propriétés recommandées : `maven.compiler.source` et `target` à 25, `project.build.sourceEncoding` à UTF-8.

Configuration Hibernate

1. Créer `src/main/resources/hibernate.cfg.xml` avec le driver MySQL, l'URL (`jdbc:mysql://localhost:3306/test_di25?serverTimezone=UTC`), utilisateur, mot de passe, dialecte `MySQLDialect`.
2. Activer `hibernate.hbm2ddl.auto=update` pour créer ou mettre à jour les tables.
3. Déclarer les entités : `<mapping class="com.test.di25.Restaurant"/>`, `<mapping class="com.test.di25.Table"/>`.

Lancer l'application

1. Créer la base `test_di25` si ce n'est pas déjà fait.
2. Dans IntelliJ : ouvrir `Main.java`, cliquer sur la flèche verte à côté de `public static void main` → **Run 'Main.main()'**.
3. Le menu principal s'affiche : option 1 pour générer les données Faker, option 2 pour la gestion des restaurants, 0 pour quitter.

Contexte du projet (Restaurant / Table)

- **Package** : `com.test.di25`
- **Build** : Maven (JDK 25, encodage UTF-8)
- **Persistance** : Hibernate (ORM) + MySQL
- **Interface** : console (menus texte)
- **Données de test** : DataFaker (locale française) pour générer restaurants et tables

Dans ce projet, l'**encapsulation** est utilisée sur les entités (attributs privés, getters/setters). La relation **OneToMany / ManyToOne** entre `Restaurant` et `Table` est maintenue de façon cohérente via une méthode dédiée :

```
// Restaurant : côté « un » de la relation OneToMany
public void addTable(Table table) {
    this.tables.add(table);
    table.setRestaurant(this);
}
```

Copier

- **Restaurant** : `@OneToMany(mappedBy = "restaurant", cascade = CascadeType.ALL)` sur `List<Table> tables`
- **Table** : `@ManyToOne` + `@JoinColumn(name = "restaurant_id")` sur `Restaurant restaurant`

La clé étrangère est en base côté **Table** (`restaurant_id`). Le DAO et la console travaillent sur le **modèle** (entités) sans écrire de SQL à la main.

Structure du projet

```
TestDI25Java/
├── pom.xml
└── src/main/
    ├── java/com/test/di25/
    │   ├── Main.java              # Point d'entrée, menu principal
    │   ├── Restaurant.java        # Entité JPA
    │   ├── Table.java             # Entité JPA (ManyToOne vers Restaurant)
    │   ├── RestaurantDao.java     # CRUD Hibernate (SessionFactory)
    │   ├── RestaurantConsole.java # Menu console CRUD restaurants
    │   └── InitDatabase.java      # Génération données Faker
    └── resources/
        └── hibernate.cfg.xml      # Connexion MySQL, dialecte, entités
```

Copier

Modèle : entités JPA (Restaurant, Table)

Restaurant

- **Table** : `restaurants`
- **Champs** : `id` (PK, identité), `nom` (longueur 30), `adresse`, `codePostal`, `ville`
- **Relation** : une liste de `Table` (`@OneToMany(mappedBy = "restaurant", cascade = CascadeType.ALL)`)

Table

- **Table** : `tables` (réservé SQL : nom de table explicite)
- **Champs** : `id` (PK), `numero`, `places`
- **Relation** : `@ManyToOne` + `@JoinColumn(name = "restaurant_id")` vers `Restaurant`

Résumé : un restaurant **a plusieurs** tables ; une table **appartient à un** restaurant. Encapsulation et cohérence bidirectionnelle via `addTable`.

Configuration Hibernate et MySQL

- **Fichier** : `src/main/resources/hibernate.cfg.xml`
- **Driver** : `com.mysql.cj.jdbc.Driver`
- **URL** : `jdbc:mysql://localhost:3306/test_di25?serverTimezone=UTC`
- **Dialecte** : `org.hibernate.dialect.MySQLDialect`
- **Schéma** : `hibernate.hbm2ddl.auto=update` (création/mise à jour des tables)
- **Debug** : `show_sql`, `format_sql` pour afficher le SQL généré
- **Entités** : `<mapping class="com.test.di25.Restaurant"/>`, `<mapping class="com.test.di25.Table"/>`

Le **RestaurantDao** construit la `SessionFactory` avec

`Configuration().configure("hibernate.cfg.xml")` et `addAnnotatedClass(Restaurant.class)`, `addAnnotatedClass(Table.class)` (sans dépendre aux mappings XML).

Couche d'accès aux données : RestaurantDao

Le **DAO** (Data Access Object) centralise toutes les opérations Hibernate sur les restaurants :

Méthode	Rôle
`create(Restaurant)`	Persiste un nouveau restaurant
`findById(Long)`	Retourne `Optional<Restaurant>`
`findAll()`	Liste tous les restaurants (requête HQL `from Restaurant r order by r.id`)
`update(Restaurant)`	Merge en base
`deleteById(Long)`	Supprime par id (retourne false si non trouvé)

Utilisation de **Session** en try-with-resources, **Transaction** pour create/update/delete. Une méthode `close()` ferme la `SessionFactory` (appelée dans `Main` en finally).

Interface console

Main

- Crée un `RestaurantDao` et un `RestaurantConsole(dao)`.
- Boucle de menu principal :
  - **1** : `InitDatabase.init()` (génération des données Faker)
  - **2** : `restaurantConsole.run()` (sous-menu CRUD restaurants)
  - **3** : Gestion des tables (à venir)
  - **0** : Quitter
- En sortie : `dao.close()`.

RestaurantConsole

- Reçoit le `RestaurantDao` par constructeur (injection de dépendance simple).
- Menu : 1 Créer, 2 Lister, 3 Afficher par id, 4 Modifier, 5 Supprimer, 0 Retour.
- Saisie via une classe utilitaire **IO** (`IO.print`, `IO.println`, `IO.readln`) pour alléger le code.
- Création : instanciation d'un `Restaurant`, setters, puis `restaurantDao.create(r)`.
- Liste / détail : utilisation de `findAll()` et `findById()` ; affichage du nombre de tables via `r.getTables().size()` pour le détail.

Séparation claire : **modèle** (entités), **DAO** (persistance), **console** (saisie/affichage).

Données de test : InitDatabase et Faker

- **InitDatabase.init()** : construit une `SessionFactory` (même config que le DAO), ouvre une session, démarre une transaction.
- **DataFaker** : `Faker faker = new Faker(new Locale("fr"))` pour des noms de restaurants, adresses, code postal, ville.
- Boucle (ex. 1000 restaurants) : pour chaque restaurant, création d'entité `Restaurant`, affectation des champs via Faker, puis création de `Table` (numero, places) et liaison avec `restaurant.addTable(table)` avant `session.persist(restaurant)`.
- Commit et fermeture session/factory.

Cela permet de tester l'application sans saisie manuelle et de valider les relations et le schéma Hibernate.

Dépendances Maven (extrait)

- **Hibernate** : `hibernate-core` (6.3.x)
- **MySQL** : `mysql-connector-j`
- **Jakarta Persistence** : `jakarta.persistence-api`
- **DataFaker** : `datafaker` (génération de données réalistes)
- **TestNG** : en scope compile si besoin de tests

Pas de JavaFX : application 100 % console.

Résumé des concepts utilisés

Élément	Rôle dans le projet
Modèle (Restaurant, Table)	Entités JPA, encapsulation, relation OneToMany / ManyToOne
Hibernate	ORM, configuration MySQL dans `hibernate.cfg.xml`, `SessionFactory` via `Configuration` + `addAnnotatedClass`
RestaurantDao	CRUD et fermeture `SessionFactory` ; abstraction de l'accès aux données
RestaurantConsole	Menu CRUD et saisie utilisateur (IO), utilise uniquement le DAO
Main	Point d'entrée, menu principal, initialisation Faker et fermeture DAO
InitDatabase	Génération de données de test avec Faker (restaurants + tables)
Maven	Build, JDK 25, gestion des dépendances
IntelliJ IDEA	Édition, exécution, configuration Maven

Aller plus loin

- **Gestion des tables** : menu dédié (création/suppression de tables liées à un restaurant) en réutilisant le même modèle et éventuellement un `TableDao`.
- **Classe IO** : centraliser `Scanner` et affichage (IO.print / IO.println / IO.readln) si pas déjà présente.
- **Tests** : tests unitaires sur le DAO (base H2 en test) ou tests d'intégration avec MySQL de test.
- **Validation** : contraintes Bean Validation sur les entités (ex. `@NotNull`, `@Size`) et vérifications dans la console.