Compte Rendu : Programmation Orientée Objet en Java

Ce document présente un ensemble d'exercices pratiques en Java qui illustrent les concepts clés de la programmation orientée objet (POO). Ces exercices visent à approfondir la compréhension et l'application de l'héritage, la redéfinition des méthodes, le polymorphisme, les classes abstraites, les interfaces, ainsi que les associations de classes. En abordant chaque concept de manière pratique, ces exercices offrent un aperçu des bonnes pratiques et des techniques fondamentales de la POO, utilisées couramment dans le développement d'applications.

Objectifs du Compte Rendu

L'objectif de ce compte rendu est de :

1. Décrire et expliquer le code développé pour chaque exercice.
2. Mettre en avant les concepts de la POO appliqués dans chaque cas, notamment l'importance de l'héritage, du polymorphisme, et de la modularité du code.
3. Justifier les choix techniques effectués, en expliquant comment chaque élément du code répond aux exigences de l'exercice.
4. Identifier les acquis en termes de programmation orientée objet, ainsi que les éléments techniques et conceptuels qui ont nécessité une attention particulière.

Exercice 1 : Gestion d'une Bibliothèque avec Héritage et Redéfinition

Introduction

Cet exercice consiste en la création d'une application Java pour gérer les informations des adhérents, auteurs, et livres d'une bibliothèque. À travers cet exercice, plusieurs concepts de la programmation orientée objet en Java, tels que l'héritage, la redéfinition de méthodes, l'encapsulation et l'association, sont mis en application. Ce document offre une analyse détaillée de chaque classe et de son rôle dans l'application, ainsi qu'une explication des concepts de POO utilisés et de leur utilité.

Détails du Code par Classe

1. Classe Personne

- **Rôle** : La classe Personne représente la structure de base pour toute personne liée à la bibliothèque, contenant des informations communes comme le nom, le prénom, l'email, le téléphone, et l'âge.
- **Attributs** :
    - Tous les attributs (nom, prenom, email, telephone, age) sont définis comme privés pour assurer l'encapsulation.
    - La gestion des informations se fait uniquement via des méthodes publiques pour protéger et contrôler l'accès aux données.
- **Méthodes** :
    - **Constructeur avec paramètres** : Initialise les attributs d'une personne à partir des valeurs fournies en paramètre.
    - **Constructeur sans paramètres** : Permet de créer une instance de Personne sans informations initiales.
    - **Getters et Setters** : Chaque attribut possède une méthode d'accès (getter) et de modification (setter) pour préserver l'encapsulation.
    - **Méthode afficher()** : Affiche les informations personnelles de la personne. Cette méthode est destinée à être redéfinie dans les classes dérivées pour enrichir l'affichage avec des informations spécifiques.
- **Concepts appliqués** :
    - **Encapsulation** : En masquant les attributs, on limite l'accès direct aux données. Cela rend le code plus sûr et réduit les risques d'erreurs.
    - **Constructeurs** : Le constructeur permet de contrôler l'initialisation des objets, évitant la création d'objets incomplets.

2. Classe Adherent

- **Rôle** : La classe Adherent représente un adhérent de la bibliothèque et hérite de la classe Personne. Elle introduit un attribut supplémentaire, numAdherent, pour différencier chaque adhérent par un numéro unique.

- **Attributs** :
    - numAdherent est défini comme un entier privé, permettant de stocker un numéro unique pour chaque adhérent.
- **Méthodes** :
    - **Constructeur** : Utilise le constructeur de Personne via **super()** pour initialiser les attributs hérités et ajoute numAdherent comme un nouvel attribut spécifique à Adherent.
    - **Getters et Setters** : Fournissent un moyen sécurisé d'accéder et de modifier le numéro de l'adhérent.
    - **Méthode afficher() (redéfinition)** : Redéfinit la méthode afficher() pour afficher les informations d'un adhérent, incluant son numéro unique. Cette redéfinition permet de conserver l'affichage des informations générales d'une personne et d'y ajouter des détails spécifiques à Adherent.
- **Concepts appliqués** :
    - **Héritage** : Permet de réutiliser les attributs et méthodes de Personne sans avoir à les redéfinir, ce qui rend le code plus concis et facile à maintenir.
    - **Redéfinition de méthode** : En redéfinissant afficher(), on ajoute des informations spécifiques à chaque type de personne.

3. Classe Auteur

- **Rôle** : La classe Auteur hérite également de Personne et ajoute un attribut numAuteur, représentant un numéro unique d'identification pour chaque auteur.
- **Attributs** :
    - numAuteur est un entier privé permettant d'identifier chaque auteur de façon unique.
- **Méthodes** :
    - **Constructeur** : Fait appel au constructeur de Personne avec super() pour initialiser les attributs hérités, puis initialise le numéro d'auteur.
    - **Getters et Setters** : Fournissent un accès sécurisé à numAuteur.

- **Méthode afficher() (redéfinition)** : Redéfinit afficher() pour inclure numAuteur, en plus des informations générales de Personne. Cela permet d'afficher des informations détaillées sur l'auteur lorsqu'un auteur est lié à un livre.

- **Concepts appliqués** :
  - **Héritage** et **Redéfinition de méthode** : La méthode afficher() redéfinie permet une personnalisation de l'affichage pour cette classe dérivée.

4. Classe Livre

- **Rôle** : La classe Livre représente un livre de la bibliothèque, associant un auteur (de type Auteur) à chaque instance de livre et incluant des informations spécifiques comme le titre et l'ISPN (numéro d'identification).
- **Attributs** :
  - titre et ispn (numéro de série) sont des attributs de type String et int pour décrire le livre.
  - auteur est une association de type Auteur, permettant de lier chaque livre à un auteur.
- **Méthodes** :
  - **Constructeur avec paramètres** : Permet d'initialiser un livre avec un titre, un ISPN, et un auteur.
  - **Constructeur sans paramètres** : Permet de créer un livre sans informations initiales, laissant la possibilité de les définir plus tard.
  - **Getters et Setters** : Fournissent des méthodes pour accéder et modifier les attributs du livre.
  - **Méthode afficher()** : Affiche les informations d'un livre en incluant les informations de l'auteur grâce à l'appel de auteur.afficher(). Cette méthode utilise la composition pour intégrer les informations d'une autre classe.
- **Concepts appliqués** :
  - **Association** : Livre est associé à Auteur, illustrant la relation entre deux objets distincts, où un livre est lié à un auteur spécifique.

- **Encapsulation** : Les informations sont protégées, et leur modification est contrôlée par des méthodes d'accès.

5. Classe Main

- **Rôle** : La classe Main sert de point d'entrée pour tester les fonctionnalités de l'application en instanciant des objets Adhèrent, Auteur, et Livre puis en affichant leurs informations.
- **Fonctionnalités** :
    - Crée un Adhèrent avec un nom, prénom, email, téléphone, âge et numéro d'adhérent.
    - Crée un Auteur avec les mêmes informations de base et un numéro d'auteur unique.
    - Crée un Livre qui est associé à un Auteur, permettant de tester la composition et les associations entre classes.
    - Appelle la méthode afficher() pour chaque objet, démontrant la redéfinition de méthode et l'héritage.
- **Concepts Appliqués**
    - **Polymorphisme** (indirectement) : La méthode afficher() dans Adhèrent et Auteur est redéfinie, permettant un comportement différent selon le type d'objet.

Conclusion

Cet exercice met en évidence plusieurs concepts fondamentaux de la programmation orientée objet. L'héritage et la redéfinition de méthodes permettent une structuration hiérarchique des classes, tandis que l'encapsulation protège les données des modifications non contrôlées. La composition et l'association entre Livre et Auteur démontrent comment structurer les relations entre objets, rendant le code modulaire et évolutif. En appliquant ces concepts, l'application offre une gestion organisée et flexible des informations d'une bibliothèque.

Exercice 2 : Gestion des Salaires des Employés avec Classes Abstraites et Polymorphisme

Introduction

Cet exercice a pour but de concevoir une application Java permettant de gérer les employés d'une entreprise, en distinguant les ingénieurs et les managers. À travers cette application, on met en pratique les concepts des classes abstraites, de l'héritage, de la redéfinition de méthodes et du polymorphisme. Le code est structuré de manière à assurer une flexibilité dans la gestion des informations des employés tout en offrant la possibilité de personnaliser les calculs de salaire en fonction du type d'employé.

Détails du Code par Classe

1. Classe Abstraite Employe

- **Rôle** : Employe est une classe abstraite qui représente un employé avec des informations générales (nom, prénom, email, téléphone, et salaire). Elle sert de classe de base pour les différents types d'employés (ingénieurs et managers) qui ont des caractéristiques communes mais peuvent différer dans la manière de calculer leur salaire.
- **Attributs** :
    - nom, prenom, email, telephone, et salaire sont des attributs privés. L'encapsulation garantit que l'accès à ces informations se fait uniquement via des méthodes dédiées.
- **Méthodes** :
    - **Constructeurs** :
        - Le constructeur avec paramètres initialise les attributs d'un employé lors de la création d'une instance.
        - Le constructeur sans paramètres permet la création d'un employé sans initialiser les valeurs d'attributs.
    - **Getters et Setters** : Permettent de récupérer et de modifier les informations de chaque employé de manière contrôlée.
    - **Méthode abstraite calculerSalire()** : Cette méthode est déclarée abstraite, ce qui oblige les sous-classes à implémenter leur propre

version du calcul de salaire. Elle est laissée non définie ici, car chaque type d'employé a sa propre logique de calcul du salaire.

- **Méthode toString()** : Redéfinie pour afficher les informations de l'employé, incluant le salaire calculé via calculerSalire(). Cette méthode facilite l'affichage des informations, en rendant le code de la classe Main plus lisible.

- **Concepts appliqués** :
  - **Classe Abstraite** : Employe ne peut pas être instanciée directement, elle fournit une base pour les classes dérivées. Cela permet d'assurer que seuls les types spécifiques d'employés, comme Ingenieur et Manager, seront utilisés.
  - **Méthode Abstraite** : calculerSalire() impose aux sous-classes d'implémenter leur propre version, permettant une spécialisation du calcul du salaire en fonction du type d'employé.

2. Classe Ingenieur

- **Rôle** : Ingenieur hérite de la classe Employe et représente un ingénieur dans l'entreprise, avec un attribut supplémentaire specialite pour indiquer son domaine de spécialisation.
- **Attributs** :
  - specialite est un attribut privé propre à Ingenieur, permettant de spécifier la spécialité de l'ingénieur.
- **Méthodes** :
  - **Constructeur avec paramètres** : Utilise super() pour initialiser les attributs hérités d'Employe, puis initialise la spécialité de l'ingénieur.
  - **Constructeur sans paramètres** : Permet de créer un ingénieur sans initialiser ses informations personnelles.
  - **Getter et Setter** : Les méthodes getSpecialite() et setSpecialite() permettent d'accéder et de modifier la spécialité de l'ingénieur.
  - **Redéfinition de calculerSalire()** : Cette méthode ajoute une augmentation de 15 % au salaire de base de l'ingénieur. La redéfinition permet de fournir une logique spécifique pour le calcul du salaire.

    - **Redéfinition de toString()** : La méthode toString() redéfinie inclut les informations générales de l'employé (via super.toString()) ainsi que la spécialité de l'ingénieur.

- **Concepts appliqués** :
    - **Héritage** : Ingenieur hérite de Employe, permettant de réutiliser les attributs et méthodes de base pour un ingénieur.
    - **Polymorphisme** : La redéfinition de calculerSalire() et toString() permet à Ingenieur de fournir son propre comportement tout en respectant la structure définie par Employe.
    - **Utilisation de super** : **super()** est utilisé dans le constructeur pour initialiser les attributs hérités et dans toString() pour inclure les informations générales d'un Employe.

3. Classe Manager

- **Rôle** : Manager hérite de Employe et représente un manager dans l'entreprise, avec un attribut service indiquant le service qu'il supervise.
- **Attributs** :
    - service est un attribut privé propre à Manager, qui permet de stocker le service sous la responsabilité du manager.
- **Méthodes** :
    - **Constructeur avec paramètres** : Utilise super() pour initialiser les attributs hérités d'Employe et initialise le service du manager.
    - **Constructeur sans paramètres** : Permet de créer un manager sans initialiser ses informations personnelles.
    - **Getter et Setter** : Les méthodes getService() et setService() permettent d'accéder et de modifier le service du manager.
    - **Redéfinition de calculerSalire()** : Applique une augmentation de 20 % au salaire de base du manager. La méthode est redéfinie pour intégrer une logique de calcul spécifique pour les managers.
    - **Redéfinition de toString()** : Inclut les informations générales de l'employé (via super.toString()) et le service du manager, permettant d'afficher des informations complètes sur le manager.

- **Concepts appliqués** :
    - **Héritage** : Manager hérite d'Employe, ce qui simplifie la définition des attributs et méthodes en partageant une base commune avec Ingenieur.
    - **Polymorphisme** : La redéfinition de calculerSalire() et toString() permet au Manager de fournir un comportement adapté tout en s'alignant sur la structure de Employe.
    - **Utilisation de super** : super() est utilisé dans le constructeur pour initialiser les attributs hérités et dans toString() pour inclure les informations d'un Employe.

4. Classe Main

- **Rôle** : Main sert de point d'entrée pour tester les fonctionnalités de l'application, en créant des instances d'ingénieurs et de managers et en affichant leurs informations.
- **Fonctionnalités** :
    - Création d'un ingénieur (Ingenieur) avec nom, prénom, email, téléphone, salaire, et spécialité.
    - Création d'un manager (Manager) avec nom, prénom, email, téléphone, salaire, et service.
    - Utilisation de System.out.println() pour afficher les informations de l'ingénieur et du manager. Les appels à toString() sur chaque objet démontrent l'utilisation du polymorphisme et la redéfinition des méthodes dans les sous-classes.
- **Concepts appliqués** :
    - **Polymorphisme** : En appelant toString() sur des objets Ingenieur et Manager, le programme utilise le polymorphisme pour exécuter la version redéfinie de chaque classe, démontrant ainsi le comportement spécifique de chaque type d'employé.

Conclusion

Cet exercice illustre comment les classes abstraites et le polymorphisme peuvent être utilisés pour structurer une application flexible, où les comportements spécifiques des sous-classes sont définis indépendamment tout en partageant une

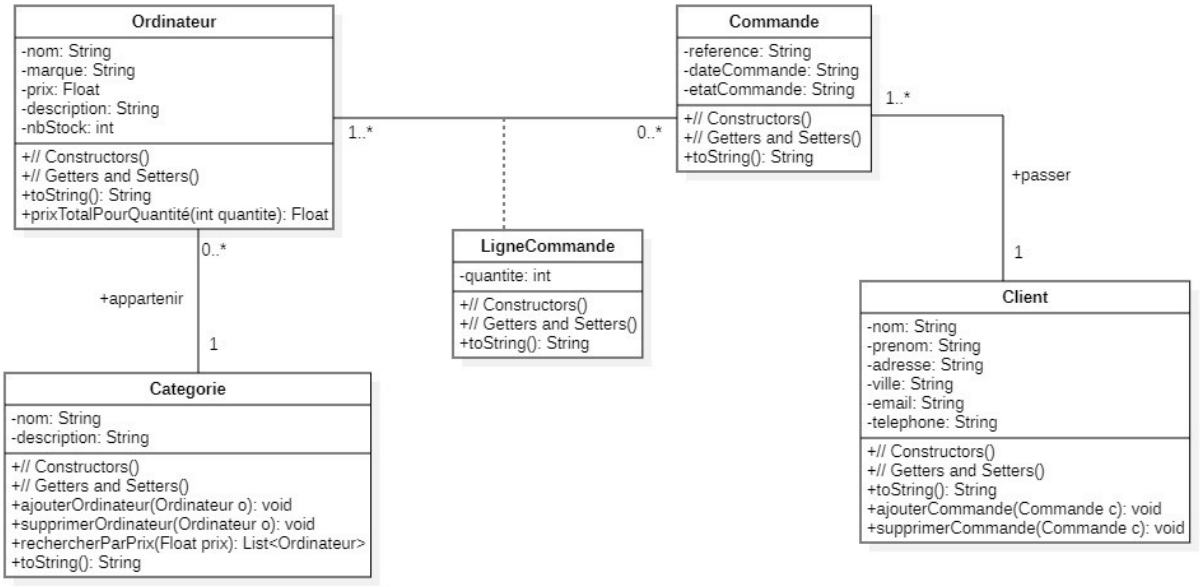
structure de base commune. L'utilisation de super dans les constructeurs assure une initialisation cohérente, tandis que le polymorphisme et l'héritage permettent de créer des méthodes de calcul de salaire adaptées à chaque type d'employé. En appliquant ces concepts, l'application garantit une gestion organisée et évolutive des employés.

Exercice 3 : Gestion des Commandes d'Ordinateurs avec Associations de Classes et Composition

Introduction

Cet exercice consiste à concevoir un système de gestion de commandes pour une entreprise vendant des ordinateurs. Le but est de structurer le code en appliquant des principes de programmation orientée objet, notamment les associations de classes, la composition, et l'encapsulation. Chaque classe du système représente une entité (ordinateur, catégorie, client, commande, ligne de commande) avec ses attributs et méthodes spécifiques, permettant de modéliser le fonctionnement d'une gestion de commandes de manière modulaire.

Diagramme des classes



Détails du Code par Classe

1. Classe Categorie

- **Rôle** : La classe Categorie regroupe des ordinateurs en fonction de leurs caractéristiques communes (comme les ordinateurs portables, de bureau, etc.).
- **Attributs** :
    - nom et description sont des attributs qui définissent le nom et la description de la catégorie.
    - ordinateurs est une liste d'objets de type Ordinateur associée à la catégorie.
- **Méthodes** :
    - **Constructeurs** : Permettent d'initialiser une catégorie avec ou sans ordinateurs.
    - **ajouterOrdinateur()** : Ajoute un ordinateur à la catégorie après avoir vérifié qu'il n'existe pas déjà (basé sur le nom et la marque). Cela assure l'unicité des ordinateurs dans une catégorie.
    - **supprimerOrdinateur()** : Supprime un ordinateur de la liste, en comparant les noms des ordinateurs.
    - **rechercherParPrix()** : Filtre la liste des ordinateurs pour retourner ceux ayant un prix donné.
    - **toString()** : Renvoie une représentation textuelle détaillée de la catégorie et des ordinateurs qu'elle contient.
- **Concepts appliqués** :
    - **Association** : Categorie est associée à Ordinateur par la liste ordinateurs. Cette relation illustre une composition où la catégorie possède plusieurs ordinateurs.
    - **Encapsulation** : Les méthodes d'accès et de modification protègent les données internes de la catégorie.

2. Classe Client

- **Rôle** : La classe Client représente un client, en regroupant ses informations personnelles et la liste des commandes qu'il a passées.
- **Attributs** :
    - nom, prenom, adresse, ville, email, et telephone stockent les informations du client.
    - commandes est une liste des commandes passées par le client.
- **Méthodes** :
    - **Constructeurs** : Initialisent un client avec ou sans liste de commandes.
    - **ajouterCommande()** : Ajoute une commande au client après avoir vérifié que cette commande n'existe pas déjà (par comparaison des références).
    - **supprimerCommande()** : Retire une commande de la liste basée sur sa référence.
    - **toString()** : Renvoie une représentation textuelle détaillée des informations du client et de ses commandes.
- **Concepts appliqués** :
    - **Association** : Client est associé à Commande, modélisant la relation entre un client et ses commandes. La liste commandes permet de stocker les commandes associées.
    - **Encapsulation** : Les attributs sont protégés par des getters et setters, ce qui assure la sécurité des données.

3. Classe Commande

- **Rôle** : La classe Commande représente une commande spécifique passée par un client. Elle regroupe des informations comme la référence, le client, la date, et l'état de la commande.
- **Attributs** :
    - reference est un identifiant unique pour chaque commande.

- client est un objet de type Client, représentant le client associé à la commande.
- dateCommande et etatCommande stockent respectivement la date et l'état de la commande.

- **Méthodes** :
  - **Constructeurs** : Permettent de créer une commande avec ou sans informations.
  - **toString()** : Retourne une description détaillée de la commande, incluant les informations du client.
- **Concepts appliqués** :
  - **Association** : Commande est associée à Client, illustrant que chaque commande appartient à un client.
  - **Encapsulation** : Les attributs sont protégés, et l'accès se fait par des méthodes publiques.

4. Classe LigneCommande

- **Rôle** : La classe LigneCommande relie une commande et un ordinateur, représentant une ligne spécifique de la commande avec la quantité commandée d'un ordinateur.

- **Attributs** :
  - quantite est la quantité de l'ordinateur commandée.
  - commande est une référence à l'objet Commande associée à la ligne de commande.
  - ordinateur est une référence à l'Ordinateur commandé.
- **Méthodes** :
  - **Constructeurs** : Permettent de créer une ligne de commande avec ou sans détails.
  - **toString()** : Fournit une description détaillée de la ligne de commande.
- **Concepts appliqués** :

- **Composition** : La LigneCommande relie Commande et Ordinateur, illustrant que chaque ligne fait partie intégrante de la commande en associant des ordinateurs commandés.
- **Encapsulation** : Les attributs sont protégés et manipulés uniquement via les méthodes d'accès.

5. Classe Ordinateur

- **Rôle** : La classe Ordinateur représente un produit (ordinateur) vendu dans le système de gestion de commandes. Elle contient des détails sur chaque ordinateur, tels que son nom, sa marque, son prix, et la catégorie à laquelle il appartient.
- **Attributs** :
  - nom, marque, prix, description, et nbStock sont des attributs décrivant les caractéristiques de l'ordinateur.
  - categorie est une référence à la Categorie associée.
- **Méthodes** :
  - **Constructeurs** : Initialisent un ordinateur avec ou sans détails.
  - **prixTotalPourQuantité()** : Calcule le coût total pour une quantité donnée d'ordinateurs.
  - **toString()** : Renvoie une représentation détaillée de l'ordinateur.
- **Concepts appliqués** :
  - **Composition** : L'ordinateur est lié à Categorie, reflétant la relation entre un ordinateur et sa catégorie.
  - **Encapsulation** : Les attributs sont protégés, ce qui rend l'accès et la modification des informations de chaque ordinateur contrôlés et sécurisés.

6. Classe Main

- **Rôle** : La classe Main teste les fonctionnalités du système en créant des instances d'ordinateurs, de catégories, de clients, de commandes, et de lignes de commandes. Elle sert de démonstration de l'interaction entre les classes.

- **Fonctionnalités** :
    - Crée trois ordinateurs (Ordinateur) avec des caractéristiques distinctes.
    - Crée une catégorie (Categorie) pour regrouper les ordinateurs sous le nom "Laptop".
    - Crée un client (Client) avec ses informations personnelles et une commande (Commande) associée.
    - Crée trois lignes de commande (LigneCommande) pour commander plusieurs quantités d'ordinateurs.
    - Affiche les informations de la commande, ce qui inclut les détails des lignes de commande et des ordinateurs associés.
- **Concepts appliqués** :
    - **Association et Composition** : La classe Main illustre les relations entre les différentes entités, montrant comment elles interagissent pour former un système de gestion de commandes complet.
    - **Encapsulation et Modularité** : La classe Main interagit avec les objets uniquement via leurs méthodes publiques, respectant ainsi l'encapsulation et assurant la modularité du code.

Conclusion

Cet exercice démontre comment structurer une application de gestion de commandes en utilisant des associations de classes, la composition, et l'encapsulation. Chaque classe a des responsabilités bien définies, et leurs interactions permettent de modéliser le processus de commande d'ordinateurs dans une entreprise. L'utilisation de collections pour gérer des listes d'objets et la composition entre les classes renforcent la modularité et la lisibilité du code, rendant l'application flexible et extensible pour des ajouts futurs.

Exercice 4 : Gestion des Produits avec Interface et Implémentation

Introduction

Cet exercice consiste à créer un système de gestion des produits en utilisant une interface pour définir les opérations de base sur les produits, avec une classe de service pour implémenter cette interface. En utilisant des concepts comme les interfaces, la programmation orientée objet permet de séparer la définition des opérations de leur implémentation, ce qui rend le code flexible et extensible.

Détails du Code par Classe et Interface

1. Interface IMetierProduit

- **Rôle** : IMetierProduit définit les opérations de base de gestion des produits que toute implémentation devra fournir, notamment les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer).
- **Méthodes** :
    - **add(Produit p)** : Ajoute un produit à la collection, ou le met à jour s'il existe déjà.
    - **getAll()** : Récupère tous les produits de la collection.
    - **findByNom(String motCle)** : Recherche des produits par un mot-clé dans leur nom.
    - **findById(long id)** : Recherche un produit par son identifiant unique.
    - **delete(long id)** : Supprime un produit par son identifiant.

- **Concepts appliqués** :
    - **Interface** : En définissant uniquement les méthodes sans fournir d'implémentation, l'interface IMetierProduit établit un contrat que les classes implémentant l'interface doivent respecter. Cela permet de maintenir une flexibilité pour différentes implémentations.

2. Classe MetierProduitImpl

- **Rôle** : MetierProduitImpl implémente l'interface IMetierProduit et fournit une gestion complète des produits via une liste dynamique.
- **Attributs** :
    - produits est une ArrayList de Produit qui stocke tous les produits de l'application.
- **Méthodes** :
    - **add(Produit p)** : Ajoute un produit à la liste ou met à jour le produit s'il existe déjà (en fonction de l'ID). Si un produit est mis à jour, un message de confirmation est affiché.
    - **getAll()** : Retourne une copie de la liste des produits pour éviter toute modification externe de la liste d'origine.
    - **findByNom(String motCle)** : Retourne une liste de produits dont le nom contient le mot-clé, en ignorant la casse.
    - **findById(long id)** : Recherche un produit spécifique par son identifiant.
    - **delete(long id)** : Supprime un produit de la liste à partir de son identifiant et affiche un message indiquant si la suppression a réussi.
- **Concepts appliqués** :
    - **Implémentation d'Interface** : En implémentant IMetierProduit, MetierProduitImpl est contrainte d'inclure toutes les méthodes définies dans l'interface, assurant une cohérence fonctionnelle.
    - **Encapsulation et Sécurité des Données** : getAll() retourne une copie de la liste pour empêcher les modifications externes, préservant ainsi l'intégrité des données.
    - **Utilisation de ArrayList** : ArrayList permet une gestion dynamique de la collection, facilitant l'ajout et la suppression de produits.

3. Classe Produit

- **Rôle** : Produit représente un produit dans le système de gestion, avec des informations détaillées telles que l'identifiant, le nom, la marque, le prix, la description, et le stock.

- **Attributs** :
    - id : Identifiant unique pour chaque produit.
    - nom, marque, prix, description, et nombreEnStock sont des attributs pour caractériser chaque produit.
- **Méthodes** :
    - **Constructeur avec paramètres** : Permet d'initialiser un produit avec toutes ses caractéristiques.
    - **Getters et Setters** : Permettent un accès contrôlé et sécurisé aux attributs.
    - **toString()** : Fournit une représentation textuelle du produit pour faciliter son affichage.
- **Concepts appliqués** :
    - **Encapsulation** : Les attributs privés sont accessibles uniquement via des méthodes publiques, assurant ainsi la sécurité et la gestion des données.
    - **Méthode toString()** : Simplifie l'affichage des informations de chaque produit, ce qui est particulièrement utile dans la classe Application.

4. Classe Application

- **Rôle** : La classe Application fournit une interface utilisateur en ligne de commande pour interagir avec le système de gestion des produits. Elle propose un menu pour ajouter, afficher, rechercher et supprimer des produits.
- **Méthodes** :
    - **menu()** : Affiche le menu principal des opérations disponibles.
    - **main()** :
        - **Boucle principale** : Le programme continue de tourner jusqu'à ce que l'utilisateur choisisse de quitter.
        - **Gestion des choix de l'utilisateur** :
            - 1 : Affiche tous les produits. Si aucun produit n'est enregistré, un message est affiché.

        - 2 : Recherche des produits par mot-clé dans leur nom.
        - 3 : Permet d'ajouter un nouveau produit. Les informations sont saisies par l'utilisateur.
        - 4 : Recherche un produit par son identifiant.
        - 5 : Supprime un produit par son identifiant.
        - 6 : Quitte le programme.
    - **Traitement des erreurs** : Utilise des blocs try-catch pour gérer les erreurs de saisie de l'utilisateur.

- **Concepts appliqués** :
    - **Interface Utilisateur en Ligne de Commande** : Cette interface console permet une interaction simple avec l'utilisateur, facilitant les tests et l'utilisation de l'application.
    - **Boucle et Structure Conditionnelle** : La boucle do-while assure la répétition du menu tant que l'utilisateur n'a pas choisi de quitter. Les structures conditionnelles switch-case permettent de gérer efficacement chaque option du menu.
    - **Encapsulation et Modularité** : Application interagit uniquement avec IMetierProduit, ce qui la rend indépendante de la classe d'implémentation spécifique (MetierProduitImpl). Cela améliore la flexibilité et facilite le changement d'implémentation si nécessaire.

Conclusion

Cet exercice démontre l'utilisation des interfaces pour définir des comportements, ainsi que la manière dont une classe d'implémentation concrétise ces comportements pour gérer les produits. En séparant les opérations des données, l'application bénéficie d'une meilleure modularité et flexibilité, tout en restant sécurisée et extensible. L'interface utilisateur en ligne de commande facilite l'interaction et les tests, permettant de vérifier le bon fonctionnement de toutes les opérations CRUD.

Conclusion Générale

Ce projet de programmation orientée objet en Java, réparti en plusieurs exercices, illustre efficacement des concepts fondamentaux tels que l'héritage, la redéfinition de méthodes, les classes abstraites, le polymorphisme, les interfaces, et la composition. Chaque exercice a été conçu pour aborder un aspect spécifique de la POO, appliqué à des scénarios concrets de gestion d'information, permettant ainsi de construire des applications modulaires et extensibles.

À travers l'**héritage** et la **redéfinition de méthodes** dans les exercices 1 et 2, nous avons vu comment la réutilisation du code et la spécialisation des classes permettent de simplifier la gestion d'objets aux caractéristiques similaires. L'implémentation des **classes abstraites** et du **polymorphisme** a démontré l'utilité de ces concepts pour définir un comportement commun tout en autorisant des variantes spécifiques dans les sous-classes.

L'**association de classes** et la **composition** dans les exercices 3 et 4 ont souligné l'importance de structurer les relations entre les objets, ce qui rend le code plus clair et les données plus faciles à gérer. Les **interfaces** dans l'exercice 4 ont permis de définir des opérations sans dépendre d'une implémentation spécifique, garantissant ainsi la flexibilité du code et la possibilité de changer facilement d'implémentation en fonction des besoins futurs.

En appliquant ces principes, ce projet aboutit à des applications orientées objet robustes, où les bonnes pratiques de la POO permettent non seulement de structurer et organiser les données, mais aussi de simplifier la maintenance et l'évolution du code. En fin de compte, cette approche modulaire et sécurisée démontre comment la POO en Java peut être utilisée pour créer des solutions logicielles puissantes, évolutives, et bien organisées.