ENSET

Compte rendu
TP 2

Source code : https://github.com/YassirJr/jeraidi-yassir-java-oop.git

Elaboré par : **Jeraidi Yassir**

Encadrant pédagogique : **Aminou Loubna**

TP 2, Exercice 1 Aperçu

Classe : Personne

La classe Personne dans le package org.example.ex1 représente une personne avec des attributs tels que le nom, le prénom, l'email, le numéro de téléphone et l'âge. Cette classe encapsule ces attributs et fournit des méthodes pour manipuler et récupérer leurs valeurs. La classe a cinq champs privés : nom, prenom, email, telephone et age. Ces champs stockent le nom, le prénom, l'adresse email, le numéro de téléphone et l'âge de la personne, respectivement.

La classe a cinq champs privés : nom, prenom, email, telephone et age. Ces champs stockent le nom, le prénom, l'adresse email, le numéro de téléphone et l'âge de la personne, respectivement.

```
private String nom;
private String prenom;
private String email;
private String telephone;
private int age;
```

Un constructeur est fourni pour initialiser ces champs lorsqu'un nouvel objet Personne est créé. Le constructeur prend cinq paramètres correspondant aux champs et les attribue aux attributs respectifs.

```
public Personne(String nom, String prenom, String email, String telephone, int age) {
    this.nom = nom;
    this.prenom = prenom;
    this.email = email;
    this.telephone = telephone;
    this.age = age;
}
```

La classe inclut des méthodes getter et setter pour chaque champ, permettant à d'autres parties du programme d'accéder et de modifier les valeurs de ces attributs. Par exemple, les méthodes getNom et setNom sont utilisées pour récupérer et définir la valeur du champ nom.

```
    public String getNom() {
    return nom;
}

public void setNom(String nom) {
    this.nom = nom;
}
```

De même, il existe des méthodes getter et setter pour les champs prenom, email, telephone et age.

De plus, la méthode afficher fournit une représentation de chaîne formatée de l'objet Personne, incluant tous ses attributs. Cette méthode utilise la méthode

String.format pour créer une chaîne qui inclut le nom, le prénom, l'email, l'âge et le numéro de téléphone de la personne.

```
public void afficher() {
    System.out.println(String.format("Nom: %s, Prenom: %s, Email: %s, Age: %d, Telephone: %s
            nom, prenom, email, age, telephone));
}
```

Dans l'ensemble, la classe Personne encapsule les propriétés et les comportements d'une personne, fournissant un moyen structuré de gérer et de manipuler les informations personnelles au sein de l'application

Classe : Adherent

La classe Adherent dans le package org.example.ex1 étend la classe Personne, héritant de ses attributs et méthodes. Cette classe représente un membre avec un attribut supplémentaire, numAdherent, qui est un entier statique qui garde une trace du nombre d'instances Adherent créées.

La classe a un constructeur qui prend cinq paramètres : nom, prenom, email, telephone et age. Ces paramètres sont passés au constructeur de la superclasse pour initialiser les champs hérités. De plus, le constructeur incrémente le champ statique numAdherent chaque fois qu'un nouvel objet Adherent est créé.

```
public Adherent(String nom, String prenom, String email, String telephone, int age) {
    super(nom, prenom, email, telephone, age);
    numAdherent++;
}
```

La méthode getNumAdherent est un getter pour le champ numAdherent, permettant à d'autres parties du programme d'accéder au nombre actuel d'instances Adherent.

```
public static int getNumAdherent() {
    return numAdherent;
}
```

La méthode afficher est redéfinie de la classe Personne pour inclure le numAdherent dans la représentation sous forme de chaîne de l'objet Adherent. Cette méthode appelle la méthode afficher de la superclasse et ajoute la valeur de numAdherent à la chaîne résultante.

```
@Override
public String afficher() {
    return super.afficher() + String.format(" , Adherent : %d", getNumAdherent());
}
```

Dans l'ensemble, la classe Adherent étend la fonctionnalité de la classe Personne en ajoutant un compteur statique pour le nombre de membres et en incluant

cette information dans la représentation sous forme de chaîne de l'objet.

Classe : Auteur

La classe Auteur dans le package org.example.ex1 étend la classe Personne, héritant de ses attributs et méthodes. Cette classe représente un auteur avec un attribut supplémentaire, numAuteur, qui est un entier qui identifie de manière unique l'auteur.

La classe a un constructeur qui prend cinq paramètres : nom, prenom, email, telephone et age. Ces paramètres sont passés au constructeur de la superclasse pour initialiser les champs hérités.

```
public Auteur(String nom, String prenom, String email, String telephone, int age) {
    super(nom, prenom, email, telephone, age);
}
```

La méthode getNumAuteur est un getter pour le champ numAuteur, permettant à d'autres parties du programme d'accéder au numéro unique de l'auteur.

```
public int getNumAuteur() {
    return numAuteur;
}
```

La méthode setNumAuteur est un setter pour le champ numAuteur, permettant à d'autres parties du programme de modifier le numéro unique de l'auteur.

```
public void setNumAuteur(int numAuteur) {
    this.numAuteur = numAuteur;
}
```

La méthode afficher est redéfinie de la classe Personne pour fournir une représentation sous forme de chaîne de l'objet Auteur. Cette méthode retourne une chaîne formatée qui inclut le numéro unique de l'auteur.

```
@Override
public String afficher() {
    return String.format("Auteur %d", getNumAuteur());
}
```

Dans l'ensemble, la classe Auteur étend la fonctionnalité de la classe Personne en ajoutant un identifiant unique pour les auteurs et en incluant cette information dans la représentation sous forme de chaîne de l'objet.

Classe : Livre

La classe Livre dans le package org.example.ex1 représente un livre avec des attributs tels que l'ISBN, le titre et l'auteur. Cette classe encapsule ces attributs et fournit des méthodes pour manipuler et récupérer leurs valeurs. La classe a trois champs privés : ISBN, titre et auteur. Ces champs stockent le numéro

ISBN du livre, le titre et l'auteur, respectivement. L'auteur est représenté par un objet Personne.

La classe a trois champs privés : ISBN, titre et auteur. Ces champs stockent le numéro ISBN du livre, le titre et l'auteur, respectivement. L'auteur est représenté par un objet Personne.

```
private int ISBN;
private String titre;
private Personne auteur;
```

Un constructeur est fourni pour initialiser ces champs lorsqu'un nouvel objet Livre est créé. Le constructeur prend trois paramètres correspondant aux champs et les assigne aux attributs respectifs.

```
public Livre(int ISBN, String titre, Personne auteur) {
    this.ISBN = ISBN;
    this.titre = titre;
    this.auteur = auteur;
}
```

La classe inclut des méthodes getter et setter pour chaque champ, permettant à d'autres parties du programme d'accéder et de modifier les valeurs de ces attributs. Par exemple, les méthodes getISBN et setISBN sont utilisées pour récupérer et définir la valeur du champ ISBN.

```
public int getISBN() {
    return ISBN;
}

public void setISBN(int ISBN) {
    this.ISBN = ISBN;
}
```

De même, il existe des méthodes getter et setter pour les champs titre et auteur.

De plus, la méthode afficher fournit une représentation sous forme de chaîne de caractères de l'objet Livre, incluant tous ses attributs. Cette méthode utilise la méthode String.format pour créer une chaîne qui inclut le numéro ISBN du livre, le titre et les informations de l'auteur.

```
public String afficher() {
    return String.format("ISBN: %d, titre: %s , nom autheur : %s , prenom autheur : %s , ema
            getISBN(), getTitre(), auteur.getNom(), auteur.getPrenom(), auteur.getEmail());
}
```

Dans l'ensemble, la classe Livre encapsule les propriétés et les comportements d'un livre, fournissant un moyen structuré de gérer et de manipuler les informations sur les livres au sein de l'application.

Classe : Main

La classe Main dans le package org.example.ex1 sert de point d'entrée pour l'application. Elle démontre la création et la manipulation des objets Adherent et Livre.

Dans la méthode main, un objet Adherent est instancié avec des détails tels que le nom, le prénom, l'email, le numéro de téléphone et l'âge. Cela se fait en utilisant le constructeur Adherent :

```
Adherent adherent = new Adherent("Yassir", "Jeraidi", "yassir.jeraidi@gmail.com", "060000000
```

Ensuite, un objet Livre est créé, représentant un livre. Le livre est initialisé avec un ISBN, un titre et l'objet Adherent précédemment créé en tant qu'auteur :

```
Livre livre = new Livre(1, "Livre 1", adherent);
```

La méthode afficher est ensuite appelée sur les objets Adherent et Livre pour imprimer leurs représentations sous forme de chaîne de caractères dans la console. La méthode afficher dans la classe Adherent inclut les détails du membre, tandis que la méthode afficher dans la classe Livre inclut les détails du livre ainsi que les informations de l'auteur :

```
System.out.println(adherent.afficher());
System.out.println(livre.afficher());
```

Dans l'ensemble, cette classe Main démontre l'utilisation de l'héritage et de la composition en programmation orientée objet en créant des objets de différentes classes et en interagissant avec eux pour obtenir la fonctionnalité souhaitée.

TP 2, Exercice 2 Aperçu

Classe : Employe

La classe `Employe` dans le package `org.example.ex2` est une classe abstraite qui représente un employé avec des attributs tels que le nom, le prénom, l'email, le numéro de téléphone et le salaire. Cette classe encapsule ces attributs et fournit des méthodes pour les manipuler et les récupérer.

La classe a cinq champs privés : `nom`, `prenom`, `email`, `telephone` et `salaire`. Ces champs stockent respectivement le nom, le prénom, l'adresse email, le numéro de téléphone et le salaire de l'employé.

```
private String nom;
private String prenom;
private String email;
private String telephone;
private double salaire;
```

Un constructeur est fourni pour initialiser ces champs lorsqu'un nouvel objet Employe est créé. Le constructeur prend cinq paramètres correspondant aux

champs et les assigne aux attributs respectifs.

```
public Employe(String nom, String prenom, String email, String telephone, double salaire) {
    this.nom = nom;
    this.prenom = prenom;
    this.email = email;
    this.telephone = telephone;
    this.salaire = salaire;
}
```

La classe inclut des méthodes getter et setter pour chaque champ, permettant à d'autres parties du programme d'accéder et de modifier les valeurs de ces attributs. Par exemple, les méthodes `getNom` et `setNom` sont utilisées pour récupérer et définir la valeur du champ `nom`.

```
public String getNom() {
    return nom;
}

public void setNom(String nom) {
    this.nom = nom;
}
```

De même, il existe des méthodes getter et setter pour les champs prenom, email, telephone et salaire.

De plus, la classe Employe déclare une méthode abstraite calculerSalaire, qui doit être implémentée par toute sous-classe. Cette méthode est destinée à calculer le salaire de l'employé, mais l'implémentation spécifique est laissée aux sous-classes.

```
public abstract double calculerSalaire();
```

Dans l'ensemble, la classe Employe fournit un moyen structuré de gérer les informations des employés, encapsulant leurs attributs et comportements tout en permettant à une logique de calcul de salaire spécifique d'être définie dans les sous-classes.

Classe : Ingenieur

La classe Ingenieur dans le package org.example.ex2 étend la classe abstraite Employe, représentant un ingénieur avec un attribut supplémentaire pour sa spécialité. Cette classe hérite des attributs et méthodes de Employe et ajoute des fonctionnalités spécifiques aux ingénieurs.

La classe a un champ privé specialite pour stocker la spécialité de l'ingénieur. Ce champ est unique à la classe Ingenieur et n'est pas présent dans la classe Employe.

```
private String specialite;
```

Le constructeur de la classe Ingenieur appelle le constructeur de la superclasse Employe pour initialiser les champs hérités : nom, prenom, email, telephone et salaire. Cela garantit que tous les attributs communs d'un employé sont correctement initialisés.

```
public Ingenieur(String nom, String prenom, String email, String telephone, double salaire)
    super(nom, prenom, email, telephone, salaire);
}
```

La classe inclut des méthodes getter et setter pour le champ specialite, permettant à d'autres parties du programme d' accéder et de modifier la spécialité de l'ingénieur.

```
public String getSpecialite() {
    return specialite;
}

public void setSpecialite(String specialite) {
    this.specialite = specialite;
}
```

La classe Ingenieur redéfinit la méthode abstraite calculerSalaire de la classe Employe. Cette méthode calcule le salaire de l'ingénieur en ajoutant une prime de 15% au salaire de base. L'implémentation spécifique de cette méthode est unique à la classe Ingenieur et démontre le polymorphisme.

```
@Override
public double calculerSalaire() {
    return getSalaire() + getSalaire() * 0.15;
}
```

Dans l'ensemble, la classe Ingenieur étend les fonctionnalités de la classe Employe en ajoutant un attribut de spécialité et en fournissant une implémentation spécifique pour le calcul du salaire, encapsulant les propriétés et comportements d'un ingénieur au sein de l'application.

Classe : Manager

La classe Manager dans le package org.example.ex2 étend la classe abstraite Employe, représentant un manager avec un attribut supplémentaire pour son service. Cette classe hérite des attributs et méthodes de Employe et ajoute des fonctionnalités spécifiques aux managers.

La classe a un champ privé service pour stocker le service du manager. Ce champ est unique à la classe Manager et n'est pas présent dans la classe Employe.

```
private String service;
```

Le constructeur de la classe Manager appelle le constructeur de la superclasse

Employe pour initialiser les champs hérités : nom, prenom, email, telephone et salaire. Cela garantit que tous les attributs communs d'un employé sont correctement initialisés.

```
public Manager(String nom, String prenom, String email, String telephone, double salaire) {
    super(nom, prenom, email, telephone, salaire);
}
```

La classe inclut des méthodes getter et setter pour le champ service, permettant à d'autres parties du programme d' accéder et de modifier le service du manager.

```
public String getService() {
    return service;
}

public void setService(String service) {
    this.service = service;
}
```

La classe Manager redéfinit la méthode abstraite calculerSalaire de la classe Employe. Cette méthode calcule le salaire du manager en ajoutant une prime de 20% au salaire de base. L'implémentation spécifique de cette méthode est unique à la classe Manager et démontre le polymorphisme.

```
@Override
public double calculerSalaire() {
    return getSalaire() + getSalaire() * 0.2;
}
```

Dans l'ensemble, la classe Manager étend les fonctionnalités de la classe Employe en ajoutant un attribut de service et en fournissant une implémentation spécifique pour le calcul du salaire, encapsulant les propriétés et comportements d'un manager au sein de l'application.

Classe : Main

La classe Main dans le package org.example.ex2 sert de point d'entrée pour l'application. Elle démontre la création et la manipulation d'objets Ingenieur et Manager, qui sont des sous-classes de la classe abstraite Employe.

Dans la méthode main, un objet Ingenieur est instancié avec des détails spécifiques tels que le nom, le prénom, l'email, le numéro de téléphone et le salaire. Cela se fait en utilisant le constructeur Ingenieur :

```
Ingenieur ingenieur = new Ingenieur("Yassir", "Jeraidi", "yassir@g.com", "0600000000", 10000
```

De même, un objet Manager est créé avec son propre ensemble de détails :

```
Manager manager = new Manager("Yacer", "Jr", "y@g.c", "0600000000", 12000);
```

Le programme imprime ensuite les détails de l'objet Ingenieur en utilisant la méthode System.out.printf. Cette méthode formate la sortie pour inclure le nom, le prénom, l'email, le numéro de téléphone et le salaire de l'ingénieur :

```
System.out.printf("nom: %s, prenom: %s, email: %s, telephone: %s, salaire: %f%n",
    ingenieur.getNom(), ingenieur.getPrenom(), ingenieur.getEmail(), ingenieur.getTelephone
```

La même approche est utilisée pour imprimer les détails de l'objet Manager :

```
System.out.printf("nom: %s, prenom: %s, email: %s, telephone: %s, salaire: %f%n",
    manager.getNom(), manager.getPrenom(), manager.getEmail(), manager.getTelephone(), manag
```

Dans l'ensemble, la classe Main démontre comment instancier et interagir avec des objets des classes Ingenieur et Manager, mettant en avant l'utilisation de l'héritage et du polymorphisme en Java.

Vue d'ensemble du TP 2, Exercice 3

Ce projet (TP) est une application Java qui gère des produits informatiques (`Ordinateur`), des clients (`Client`) et des commandes (`Commande`). L'application est structurée en plusieurs classes, chacune encapsulant des fonctionnalités et des attributs spécifiques.

Classe : `Ordinateur`

La classe `Ordinateur` représente un ordinateur avec divers attributs et méthodes pour manipuler et récupérer ses données. La classe a six champs privés : `nom`, `marque`, `prix`, `description`, `stock` et `categorie`. Ces champs stockent respectivement le nom, la marque, le prix, la description, la quantité en stock et la catégorie de l'ordinateur.

```
private String nom;
private String marque;
private double prix;
private String description;
private int stock;
private Categorie categorie;
```

La classe a également un constructeur qui initialise les attributs de l'ordinateur et une méthode toString qui renvoie une représentation sous forme de chaîne des données de l'ordinateur.

```
public Ordinateur(String nom, String marque, double prix, String description, int stock, Cat
    this.nom = nom;
    this.marque = marque;
    this.prix = prix;
    this.description = description;
    this.stock = stock;
```

```
    this.categorie = categorie;
}
```

La classe Ordinateur fournit des méthodes pour obtenir et définir les attributs de l'ordinateur, telles que getNom, getMarque, getPrix, getDescription, getStock, getCategorie, setNom, setMarque, setPrix, setDescription, setStock et setCategorie.

```
public String getNom() {
    return nom;
}
```

Méthode : prixForQuantity

De plus, la classe inclut une méthode prixForQuantity qui calcule le prix total pour une quantité donnée d'ordinateurs. Cette méthode multiplie le champ prix par la quantité fournie et renvoie le résultat sous forme d'entier.

```
public int prixForQuantity(int quantity) {
    return (int) (prix * quantity);
}
```

Dans l'ensemble, la classe Ordinateur encapsule les propriétés et les comportements d'un ordinateur, fournissant un moyen structuré de gérer et de manipuler les données des ordinateurs au sein de l'application.

Classe : `Client`

La classe `Client` représente un client avec divers attributs et méthodes pour manipuler et récupérer ses données. La classe a trois champs privés : `nom`, `adresse` et `email`. Ces champs stockent respectivement le nom, l'adresse et l'adresse e-mail du client.

```
private String nom;
private String prenom;
private String email;
private String adresse;
private String ville;
private String telephone;
private final List<Commande> commandes = new ArrayList<>();
```

La classe a également un constructeur qui initialise les attributs du client et une méthode toString qui renvoie une représentation sous forme de chaîne des données du client.

```
public Client(String nom, String prenom, String email, String adresse, String ville, String
    this.nom = nom;
    this.prenom = prenom;
    this.email = email;
```

```
    this.adresse = adresse;
    this.ville = ville;
    this.telephone = telephone;
}
```

La classe Client fournit des méthodes pour obtenir et définir les attributs du client, telles que getNom, getPrenom, getEmail, getAdresse, getVille, getTelephone, setNom, setPrenom, setEmail, setAdresse, setVille et setTelephone.

```
public String getNom() {
    return nom;
}
```

La classe inclut également une méthode ajouterCommande pour ajouter une nouvelle commande à la liste des commandes du client.

Méthode : ajouterCommande

```
public void ajouteCommande(Commande commande) {
    if (!commandes.contains(commande)) {
        commandes.add(commande);
    }
}
```

De plus, la classe inclut une méthode getCommandes qui renvoie la liste des commandes associées au client. Cette méthode récupère le champ commandes, qui stocke la liste des commandes, et la renvoie sous forme de collection.

Méthode : getCommandes

```
public List<Commande> getCommandes() {
    return commandes;
}
```

La classe inclut également une méthode supprimerCommande qui supprime une commande spécifique de la liste des commandes du client. Cette méthode prend une commande en paramètre et la supprime de la liste commandes si elle existe.

Méthode : supprimerCommande

```
public void supprimerCommande(Commande commande) {
    commandes.remove(commande);
}
```

Dans l'ensemble, la classe Client encapsule les propriétés et les comportements d'un client, fournissant un moyen structuré de gérer et de manipuler les données des clients au sein de l'application.

Classe : `Commande`

La classe `Commande` représente une commande passée par un client pour un ou plusieurs ordinateurs. La classe a trois champs privés : `client`, `ordinateurs` et `date`. Ces champs stockent respectivement le client associé à la commande, la liste des ordinateurs commandés et la date de la commande.

```
private String reference;
private Client client;
private LocalDateTime dateCommande;
private CommandeEtat etat;
```

La classe a également un constructeur qui initialise les attributs de la commande et une méthode toString qui renvoie une représentation sous forme de chaîne des données de la commande.

```
public Commande(String reference, Client client, LocalDateTime dateCommande, CommandeEtat et
    this.reference = reference;
    this.client = client;
    this.dateCommande = dateCommande;
    this.etat = etat;
}
```

La classe Commande fournit des méthodes pour obtenir et définir les attributs de la commande, telles que getReference, getClient, getDateCommande, getEtat, setReference, setClient, setDateCommande et setEtat.

```
public String getReference() {
    return reference;
}

public void setReference(String reference) {
    this.reference = reference;
}
```

De même, les méthodes getClient et setClient gèrent le champ client, les méthodes getDateCommande et setDateCommande gèrent le champ dateCommande, et les méthodes getEtat et setEtat gèrent le champ etat.

Dans l'ensemble, la classe Commande encapsule les propriétés et les comportements d'une commande, fournissant un moyen structuré de gérer et de manipuler les données des commandes au sein de l'application.

Enum : `CommandeEtat`

L'énumération CommandeEtat dans le package org.example.ex3 définit les états possibles d'une commande. Les énumérations en Java sont un type spécial de classe qui représente un groupe de constantes (variables immuables). Cette énumération a trois constantes : OK, PENDING et REFUSED.

```
public enum CommandeEtat {
    OK,
    PENDING,
    REFUSED
}
```

Chaque constante de l'énumération CommandeEtat représente un état spécifique dans lequel une commande peut se trouver. L'état OK indique que la commande a été traitée avec succès. L'état PENDING signifie que la commande est en cours de traitement et n'a pas encore été complétée. L'état REFUSED signifie que la commande a été rejetée. Les énumérations sont utiles pour représenter un ensemble fixe de constantes liées, rendant le code plus lisible et moins sujet aux erreurs. En utilisant l'énumération CommandeEtat, le code peut se référer de manière claire et cohérente aux différents états d'une commande dans toute l'application.

Classe : `Categorie`

La classe Categorie représente une catégorie d'ordinateurs avec divers attributs et méthodes pour gérer et récupérer les informations de la catégorie. La classe a deux champs privés : nom et description. Ces champs stockent respectivement le nom et la description de la catégorie.

```
private String nom;
private String description;
private List<Ordinateur> ordinateurs = new ArrayList<>();
```

La classe a également un constructeur qui initialise les attributs de la catégorie et une méthode toString qui renvoie une représentation sous forme de chaîne des données de la catégorie.

```
public Categorie(String nom, String description) {
    this.nom = nom;
    this.description = description;
}
```

La classe Categorie fournit des méthodes pour obtenir et définir les attributs de la catégorie, telles que getNom, getDescription, setNom et setDescription.

```
public String getNom() {
    return nom;
}

public void setNom(String nom) {
    this.nom = nom;
}
```

Méthode : ajouterOrdinateur

La classe inclut également une méthode ajouterOrdinateur pour ajouter un ordinateur à la liste des ordinateurs de la catégorie.

```
public void ajouterOrdinateur(Ordinateur ordinateur) {
    if (!ordinateurs.contains(ordinateur)) {
        ordinateurs.add(ordinateur);
    }
}
```

Méthode : getOrdinateurs

De plus, la classe inclut une méthode getOrdinateurs qui renvoie la liste des ordinateurs associés à la catégorie. Cette méthode récupère le champ ordinateurs, qui stocke la liste des ordinateurs, et la renvoie sous forme de collection.

```
public List<Ordinateur> getOrdinateurs() {
    return ordinateurs;
}
```

Méthode : supprimerOrdinateur

La classe inclut également une méthode supprimerOrdinateur qui supprime un ordinateur spécifique de la liste des ordinateurs de la catégorie. Cette méthode prend un ordinateur en paramètre et le supprime de la liste ordinateurs si il existe.

```
public void supprimerOrdinateur(Ordinateur ordinateur) {
    ordinateurs.remove(ordinateur);
}
```

Méthode : rechercherParPrix

La classe inclut une méthode rechercherParPrix qui recherche des ordinateurs dans la catégorie ayant un prix égal à la valeur spécifiée. Cette méthode prend un prix en paramètre et renvoie une liste d'ordinateurs correspondant au prix donné.

```
public List<Ordinateur> rechercherParPrix(double prix) {
    return ordinateurs.stream()
            .filter(ordinateur -> ordinateur.getPrix() == prix)
            .toList();
}
```

Dans l'ensemble, la classe Categorie encapsule les propriétés et les comportements d'une catégorie, fournissant un moyen structuré de gérer et de manipuler les informations des catégories au sein de l'application.

Classe : `LigneCommande`

La classe LigneCommande représente un article dans une commande, contenant des informations sur un ordinateur spécifique et la quantité commandée. La classe a deux champs privés : ordinateur et quantite. Ces champs stockent respectivement l'ordinateur commandé et la quantité de cet ordinateur.

```
private int quantite;
private Commande commande;
private Ordinateur ordinateur;
```

La classe a également un constructeur qui initialise les attributs de la ligne de commande et une méthode toString qui renvoie une représentation sous forme de chaîne des données de la ligne de commande.

```
public LigneCommande(int quantite, Commande commande, Ordinateur ordinateur) {
    this.quantite = quantite;
    this.commande = commande;
    this.ordinateur = ordinateur;
}
```

La classe LigneCommande fournit des méthodes pour obtenir et définir les attributs de la ligne de commande, telles que getQuantite, getCommande, getOrdinateur, setQuantite, setCommande et setOrdinateur.

```
public int getQuantite() {
    return quantite;
}

public void setQuantite(int quantite) {
    this.quantite = quantite;
}
```

Dans l'ensemble, la classe LigneCommande encapsule les propriétés et les comportements d'une ligne de commande, fournissant un moyen structuré de gérer et de manipuler les données des lignes de commande au sein de l'application.

Classe : `Main`

La classe Main est le point d'entrée de l'application, contenant la méthode principale qui exécute le programme. La classe démontre les fonctionnalités de l'application en créant des instances d'ordinateurs, de clients et de commandes, et en effectuant diverses opérations sur eux.

Dans la méthode principale, une liste d'objets Ordinateur est récupérée en appelant la méthode getOrdinateurs. Cette méthode crée et renvoie une liste d'instances Ordinateur, chacune initialisée avec des attributs spécifiques tels que le nom, la marque, le prix, la description, la quantité en stock et la catégorie.

```
List<Ordinateur> ordinateurs = getOrdinateurs();
```

Un objet Client est ensuite instancié avec des détails tels que le nom, le prénom, l'email, l'adresse, la ville et le numéro de téléphone.

```
Client client = new Client("yacer", "jr", "y@g.c", "casa", "casa/maroc", "0600000000");
```

Ensuite, un objet Commande est créé, représentant une commande passée par le client. La commande est initialisée avec une référence, l'objet client, la date et l'heure actuelles, et un état de commande OK.

```
Commande commande = new Commande("ref1", client, LocalDateTime.now(), CommandeEtat.OK);
```

Une liste d'objets LigneCommande est ensuite créée, chacun représentant une ligne de commande. Ces objets sont initialisés avec une quantité, l'objet commande et un objet Ordinateur de la liste des ordinateurs.

```
List<LigneCommande> ligneCommandes = List.of(
        new LigneCommande(2, commande, ordinateurs.get(0)),
        new LigneCommande(5, commande, ordinateurs.get(1)),
        new LigneCommande(10, commande, ordinateurs.get(2))
);
```

La méthode forEach est utilisée pour itérer sur la liste des objets LigneCommande, en affichant les détails de chaque ligne de commande, y compris le nom du client, le nom de l'ordinateur et la quantité commandée.

```
ligneCommandes.forEach(ligneCommande -> {
    System.out.println("Client: " + ligneCommande.getCommande().getClient().getNom());
    System.out.println("Ordinateur: " + ligneCommande.getOrdinateur().getNom());
    System.out.println("Quantite: " + ligneCommande.getQuantite());
    System.out.println();
});
```

La méthode getOrdinateurs est une méthode auxiliaire qui crée et renvoie une liste d'objets Ordinateur. Chaque Ordinateur est initialisé avec des attributs tels que le nom, la marque, le prix, la description, la quantité en stock et la catégorie.

```
private static List<Ordinateur> getOrdinateurs() {
    Categorie etudeCategorie = new Categorie("etude", "etude categorie");
    Categorie gamingCategorie = new Categorie("gaming", "gaming categorie");
    Categorie editingCategorie = new Categorie("editing", "editing categorie");

    return List.of(
            new Ordinateur("pc 1", "dell", 10000.00, "bon pc", 20, etudeCategorie),
            new Ordinateur("pc 2", "msi", 15000.00, "bon pc", 10, gamingCategorie),
            new Ordinateur("mac 2024", "apple", 30000.00, "bon pc", 5, editingCategorie)
    );
}
```

La classe Main crée des instances d'ordinateurs dans différentes catégories, un client et une commande. Elle crée ensuite des lignes de commande pour la

commande, en associant chaque ligne de commande à un ordinateur et à une quantité. Enfin, la classe affiche des informations sur chaque ligne de commande, y compris le nom du client, le nom de l'ordinateur et la quantité commandée.

TP 2, Exercice 4 Aperçu

Classe : Produit

La classe `Produit` dans le package `org.example.ex4` représente un produit avec divers attributs tels que l'ID, le nom, la marque, le prix, la description et le stock. Cette classe encapsule ces attributs et fournit des méthodes pour manipuler et récupérer leurs valeurs.

La classe a six champs privés : `id`, `nom`, `marque`, `prix`, `description` et `stock`. Ces champs stockent l'ID du produit, le nom, la marque, le prix, la description et la quantité en stock, respectivement.

```
private int id;
private String nom;
private String marque;
private double prix;
private String description;
private int stock;
```

Un constructeur est fourni pour initialiser ces champs lorsqu'un nouvel objet Produit est créé. Le constructeur prend six paramètres correspondant aux champs et les assigne aux attributs respectifs.

```
public Produit(int id, String nom, String marque, double prix, String description, int stock
    this.id = id;
    this.nom = nom;
    this.marque = marque;
    this.prix = prix;
    this.description = description;
    this.stock = stock;
}
```

La classe inclut des méthodes getter et setter pour chaque champ, permettant à d'autres parties du programme d'accéder et de modifier les valeurs de ces attributs. Par exemple, les méthodes getId et setId sont utilisées pour récupérer et définir la valeur du champ id.

```
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
```

```
}
```

De même, il existe des méthodes getter et setter pour les champs nom, marque, prix, description et stock, permettant la manipulation de ces attributs. Dans l'ensemble, la classe Produit fournit un moyen structuré de gérer les informations sur les produits, encapsulant leurs attributs et comportements tout en permettant un accès et une modification faciles via des méthodes getter et setter.

Interface : IMetierProduit

L'interface IMetierProduit dans le package org.example.ex4 définit un contrat pour la gestion des objets Produit. Cette interface décrit plusieurs méthodes que toute classe implémentant doit fournir, assurant une manière cohérente de gérer les opérations liées aux produits.
La méthode add est destinée à ajouter un nouveau Produit au système. Elle prend un objet Produit en paramètre et retourne le Produit ajouté.

```
public Produit add(Produit produit);
```

La méthode getAll récupère une liste de tous les objets Produit. Cette méthode ne prend aucun paramètre et retourne une List.

```
public List<Produit> getAll();
```

La méthode getAllByNom récupère une liste d'objets Produit qui correspondent à un nom donné. Elle prend un paramètre String représentant le nom du produit et retourne une List.

```
public List<Produit> getAllByNom(String nom);
```

La méthode getById récupère un Produit par son identifiant unique. Elle prend un paramètre int représentant l'ID du produit et retourne le Produit correspondant.

```
public Produit getById(int id);
```

La méthode delete supprime un Produit spécifié du système. Elle prend un objet Produit en paramètre et ne retourne aucune valeur.

```
public void delete(Produit p);
```

Dans l'ensemble, l'interface IMetierProduit fournit un moyen structuré de gérer les produits, définissant des opérations essentielles telles que l'ajout, la récupération et la suppression d'objets Produit. Cela garantit que toute classe implémentant cette interface fournira ces fonctionnalités fondamentales.

Classe : MetierProduitImpl

La classe MetierProduitImpl dans le package org.example.ex4 implémente l'interface IMetierProduit, fournissant des implémentations concrètes pour la

gestion des objets Produit. Cette classe utilise une ArrayList pour stocker les produits.

La classe a un champ privé produits, qui est une ArrayList contenant des objets Produit.

```
private final List<Produit> produits = new ArrayList<>();
```

La méthode add ajoute un nouveau Produit à la liste s'il n'est pas déjà présent. Elle vérifie si la liste ne contient pas le produit avant de l'ajouter.

```
public Produit add(Produit p) {
    if (!produits.contains(p)) {
        produits.add(p);
    }
    return p;
}
```

La méthode getAll retourne la liste de tous les objets Produit stockés dans la liste produits.

```
public List<Produit> getAll() {
    return produits;
}
```

La méthode getAllByNom filtre la liste des produits par leur nom. Elle utilise un flux pour filtrer les produits dont les noms contiennent la chaîne spécifiée.

```
public List<Produit> getAllByNom(String nom) {
    return produits.stream().filter(produit -> produit.getNom().contains(nom))
            .toList();
}
```

La méthode getById récupère un Produit par son identifiant unique. Elle utilise un flux pour trouver le premier produit avec l'ID correspondant ou retourne null si aucun match n'est trouvé.

```
public Produit getById(int id) {
    return produits.stream().filter(produit -> produit.getId() == id).findFirst().orElse(nul
}
```

La méthode delete supprime un Produit spécifié de la liste.

```
public void delete(Produit p) {
    produits.remove(p);
}
```

Dans l'ensemble, la classe MetierProduitImpl fournit une implémentation concrète pour la gestion des produits, y compris l'ajout, la récupération, le filtrage et la suppression d'objets Produit, garantissant que ces opérations sont effectuées de manière cohérente.

Classe : Application

La classe Application dans le package org.example.ex4 sert de point d'entrée principal pour l'application, fournissant une interface pilotée par menu pour la gestion des objets Produit. Elle utilise un Scanner pour les entrées utilisateur et une instance de MetierProduitImpl pour effectuer des opérations sur les produits.

La méthode menu affiche une liste d'options à l'utilisateur, telles que l'affichage de tous les produits, la recherche de produits par mot-clé, l'ajout d'un nouveau produit, la récupération d'un produit par ID, la suppression d'un produit par ID et la sortie du programme. Les éléments du menu sont stockés dans une List et imprimés à la console.

```
public static void menu() {
    List<String> menuItems = List.of(
            "1 - Afficher la liste des produits.",
            "2 - Rechercher des produits par mot clé.",
            "3 - Ajouter un nouveau produit dans la liste.",
            "4 - Récupérer et afficher un produit par ID.",
            "5 - Supprimer un produit par id.",
            "6 - Quitter ce programme."
    );
    menuItems.forEach(System.out::println);
}
```

La méthode addNewProduct invite l'utilisateur à entrer les détails d'un nouveau produit, tels que le nom, la marque, le prix, la description et la quantité en stock. Elle crée ensuite et retourne un nouvel objet Produit avec ces détails.

```
public static Produit addNewProduct(int id) {
    System.out.println("Veuillez saisir le nom de produit :");
    String nom = scanner.next();
    System.out.println("Veuillez saisir la marque de produit :");
    String marque = scanner.next();
    System.out.println("Veuillez saisir le prix de produit :");
    double prix = scanner.nextDouble();
    System.out.println("Veuillez saisir la description de produit :");
    scanner.nextLine();
    String description = scanner.nextLine();
    System.out.println("Veuillez saisir le nombre de stock de produit :");
    int stock = scanner.nextInt();
    return new Produit(id, nom, marque, prix, description, stock);
}
```

Dans la méthode main, une instance de MetierProduitImpl est créée pour gérer les produits. La méthode entre dans une boucle où elle affiche le menu et traite le choix de l'utilisateur. En fonction de l'entrée de l'utilisateur, elle effectue

diverses opérations telles que l'affichage de tous les produits, la recherche de produits par nom, l'ajout d'un nouveau produit, la récupération d'un produit par ID et la suppression d'un produit par ID. Par exemple, pour afficher tous les produits, la méthode getAll de MetierProduitImpl est appelée et les produits sont imprimés à la console.

```
List<Produit> produits = metierProduitImpl.getAll();
if (!produits.isEmpty()) {
    produits.forEach(System.out::println);
} else {
    System.out.println("Aucun produit maintent");
}
```

Pour ajouter un nouveau produit, la méthode addNewProduct est appelée pour recueillir les détails du produit auprès de l'utilisateur, et la méthode add de MetierProduitImpl est utilisée pour ajouter le produit à la liste.

```
int id = metierProduitImpl.getAll().stream().mapToInt(Produit::getId).sum();
metierProduitImpl.add(addNewProduct(id));
System.out.println("Le produit ajoute avec success");
```

Dans l'ensemble, la classe Application fournit une interface conviviale pour la gestion des produits, tirant parti de la classe MetierProduitImpl pour effectuer des opérations CRUD sur les objets Produit.