TD 2 Gestion des exceptions

Exercice 1:

La classe `Employe` est une classe abstraite qui sert de modèle pour représenter un employé avec un nom, un salaire et des heures travaillées. Elle fournit des constructeurs pour initialiser ces valeurs et des méthodes pour accéder et modifier les informations. La classe inclut une méthode abstraite `getSalaire()` que les classes dérivées doivent implémenter pour calculer le salaire spécifique de l'employé.

```
package fr.tharsanvishnukumar.employe;

public abstract class Employe {
    private String nom;
    // 0 ⇒ salaire
    // 1 ⇒ heur travaille
    private double[] infosSalaire = new double[2];

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

```
    public Employe(String nom,double salaire) {
        this.nom = nom;
        infosSalaire[0] = salaire;
    }
    public Employe(String nom,double salaire,double heursTravaillees) {
        this.nom = nom;
        infosSalaire[0] = salaire;
        infosSalaire[1] = heursTravaillees;
    }

    public String getNom() {
        return nom;
    }

    public double[] getInfosSalaire() {
        return infosSalaire;
    }

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

    public void setInfosSalaire(double... infosSalaire) {
        this.infosSalaire[0] = infosSalaire[0];
        this.infosSalaire[1] = infosSalaire[1];
    }
    public abstract double getSalaire();
}
```

La classe `EmployeHoraire` est une extension de la classe `Employe` qui gère les employés payés à l'heure, incluant les heures supplémentaires. Elle possède un attribut `tauxPourcentageHeuresSup` pour le pourcentage de majoration des heures supplémentaires. Les constructeurs permettent d'initialiser l'employé avec un nom, un tarif horaire, un nombre d'heures travaillées et un taux de majoration. La méthode `getSalaire()` calcule le salaire total en fonction des heures normales et des heures supplémentaires. Si l'employé dépasse 39 heures de travail, les heures supplémentaires sont payées au tarif majoré, sinon, il est payé selon le tarif horaire normal.

```
package fr.tharsanvishnukumar.employe;

public class EmployeHoraire extends Employe {
    public void setTauxPourcentageHeuresSup(double tauxPourcentageHeures
        this.tauxPourcentageHeuresSup = ((double) tauxPourcentageHeuresSup
    }
    public double getTauxPourcentageHeuresSup() {
        return tauxPourcentageHeuresSup;
    }

    private double tauxPourcentageHeuresSup;
   public EmployeHoraire(
       String nom,
       double tarifHoraire,
       int heursTravaillees,
       int tauxPourcentageHeuresSup
     ) {
       super(nom, tarifHoraire, heursTravaillees);
       setTauxPourcentageHeuresSup(tauxPourcentageHeuresSup);
   }
    public EmployeHoraire(String nom) {
       super(nom);
   }


    @Override
    public double getSalaire() {
       double[] salaireInfos = getInfosSalaire();
       double heuresNormalDeTravaillee = 39;
       double salaireParHeures = salaireInfos[0];
       double totalHeurs = salaireInfos[1];

       if(totalHeurs > heuresNormalDeTravaillee){
           double salaireTotalHeurSup =
           (totalHeurs - heuresNormalDeTravaillee) *
           (salaireParHeures * getTauxPourcentageHeuresSup());
           double salaireTatal = heuresNormalDeTravaillee * salaireParHeures;
           return salaireTatal + salaireTotalHeurSup;
```

```
        } else {
            return totalHeurs * salaireParHeures;
        }
    }
}
```

La classe `Commercial` est une extension de la classe `Employe` qui représente un employé travaillant dans le domaine commercial. Elle possède un constructeur qui permet d'initialiser l'employé avec un nom et un salaire de base. La méthode `getSalaire()` surcharge la méthode abstraite de `Employe` pour calculer le salaire total en augmentant le salaire de base de 1% (par exemple, pour prendre en compte des commissions ou des primes). Cette classe permet donc de gérer un employé commercial avec un salaire de base légèrement majoré.

```
package fr.tharsanvishnukumar.employe;

public class Commercial extends Employe {
    public Commercial(String nom, double salaire) {
        super(nom, salaire);
    }

    @Override
    public double getSalaire() {
        double[] salaireInfos = getInfosSalaire();
        double salaire =  salaireInfos[0];
        return  salaire * 1.01;
    }

}
```

La classe `Paie` est le point d'entrée d'une application qui gère le calcul des salaires pour différents types d'employés. Elle utilise une `ArrayList` pour stocker des objets de type `Employe`, qui peuvent être des instances de `EmployeHoraire` ou de `Commercial`.

- **Création d'employés :**
    - Un employé horaire est ajouté avec tous les paramètres (nom, tarif horaire, heures travaillées, et taux de majoration des heures

supplémentaires).

- Un autre employé horaire est ajouté en créant un objet avec uniquement le nom, puis en définissant ses informations de salaire et son taux de majoration après la création.
- Un employé commercial est ajouté avec un salaire de base défini au moment de la création.

- **Affichage des salaires :**
  - La boucle `for` parcourt la liste d'employés et affiche pour chacun le nom et le salaire calculé, formaté à deux décimales.

```
package fr.tharsanvishnukumar.employe;

import java.util.ArrayList;

public class Paie {
    public static void main(String[] args) {
        ArrayList<Employe> employes = new ArrayList<>();

        // Employé horaire créé avec tous les paramètres
        employes.add(new EmployeHoraire("Dupond", 20, 40, 30));

        // Employé horaire créé avec seulement le nom, informations ajoutées en
        EmployeHoraire employe1 = new EmployeHoraire("Martin");
        employe1.setInfosSalaire(25, 45); // Salaire horaire : 25, heures travaillées
        employe1.setTauxPourcentageHeuresSup(50); // Taux pour heures supple
        employes.add(employe1);

        // Employé commercial avec salaire défini au constructeur
        employes.add(new Commercial("Luis", 1000));

        for (Employe employe: employes) {
            System.out.printf("Monsieur %s a gagné %.2f € cette semaine\n",emp
        }
    }
}
```

1) Création de la class entreprise

La classe `EmployeNonTrouveException` est une exception personnalisée qui hérite de `Exception`. Elle sert à signaler qu'un employé n'a pas été trouvé dans une entreprise, notamment lors de la suppression d'un employé. Son constructeur accepte un message explicatif pour décrire l'erreur. Cette classe améliore la gestion ciblée des erreurs liées aux employés.

```
package fr.tharsanvishnukumar.employe;

public class EmployeNonTrouveException extends Exception {
    public EmployeNonTrouveException(String message) {
        super(message);
    }
}
```

La classe `Entreprise` représente une entreprise qui peut gérer ses employés. Elle possède un attribut `nom` pour stocker le nom de l'entreprise et un `ArrayList<Employe>` pour conserver la liste des employés. Le constructeur permet d'initialiser l'entreprise avec un nom et de créer une liste vide d'employés. Les méthodes `getNom()` et `getEmployes()` permettent de récupérer respectivement le nom de l'entreprise et la liste des employés. La méthode `ajouterEmploye(Employe employe)` ajoute un employé à l'entreprise s'il n'est pas déjà présent, sinon elle affiche un message de duplication. La méthode `enleverEmploye(Employe employe)` retire un employé de la liste et lance une exception `EmployeNonTrouveException` si l'employé n'existe pas. Cette classe permet donc d'ajouter, de retirer et de gérer les employés dans une entreprise de manière sécurisée.

```
package fr.tharsanvishnukumar.employe;

import java.util.ArrayList;

public class Entreprise {
    private String nom;
    private ArrayList<Employe> employes;

    public Entreprise(String nom) {
        this.nom = nom;
        this.employes = new ArrayList<>();
```

```
    }

    public String getNom() {
        return nom;
    }

    public ArrayList<Employe> getEmployes() {
        return employes;
    }

    public void ajouterEmploye(Employe employe) {
        if (!employes.contains(employe)) {
            employes.add(employe);
        } else {
            System.out.println("L'employé est déjà dans l'entreprise.");
        }
    }

    public void enleverEmploye(Employe employe) throws EmployeNonTrouveE
        if (employes.contains(employe)) {
            employes.remove(employe);
        } else {
            throw new EmployeNonTrouveException("L'employé n'existe pas dans
        }
    }
}
```

La classe `TestEntreprise` est un programme de test qui démontre l'utilisation de la classe `Entreprise` pour gérer des employés. Une entreprise nommée "SlamCorp" est créée, et deux employés sont ajoutés : un `EmployeHoraire` nommé Alice et un `Commercial` nommé Bob. L'exemple montre comment utiliser la méthode `ajouterEmploye()` pour ajouter des employés à l'entreprise. Ensuite, l'employé Alice est supprimé avec `enleverEmploye()` et un message de confirmation est affiché. Un second appel à `enleverEmploye()` tente de supprimer Alice une deuxième fois, ce qui déclenche l'exception `EmployeNonTrouveException`, affichant un message d'erreur approprié.

```
package fr.tharsanvishnukumar.employe;

public class TestEntreprise {
    public static void main(String[] args) {
        Entreprise entreprise = new Entreprise("SlamCorp");

        EmployeHoraire employe1 = new EmployeHoraire("Alice", 15, 40, 25);
        Commercial employe2 = new Commercial("Bob", 1200);

        // Ajouter des employés
        entreprise.ajouterEmploye(employe1);
        entreprise.ajouterEmploye(employe2);

        // Enlever un employé existant
        try {
            entreprise.enleverEmploye(employe1);
            System.out.println("Employé Alice supprimé.");
        } catch (EmployeNonTrouveException e) {
            System.out.println(e.getMessage());
        }

        // Tenter de supprimer un employé non existant
        try {
            entreprise.enleverEmploye(employe1);
        } catch (EmployeNonTrouveException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

2) Création d’un association bidirectionnelle entre entreprise et employe

La classe `Employe` a été modifiée pour inclure une référence à l'entreprise à laquelle l'employé appartient, établissant ainsi une association bidirectionnelle. Un attribut `entreprise` a été ajouté, accompagné de méthodes `getEntreprise()` et `setEntreprise()` pour accéder et modifier cette référence. Une méthode

`rejoindreEntreprise(Entreprise entreprise)` a été introduite pour permettre à un employé de rejoindre une entreprise tout en gérant la mise à jour de la référence de l'employé et de l'entreprise de manière bidirectionnelle. Si l'employé appartient déjà à une autre entreprise, celle-ci est mise à jour pour refléter ce changement. Dans la classe `Entreprise`, la méthode `ajouterEmploye()` a été ajustée pour établir le lien bidirectionnel en appelant `setEntreprise()` sur l'employé. De même, `enleverEmploye()` supprime la référence de l'employé à l'entreprise lorsqu'il est retiré. Cette modification garantit que l'association entre l'entreprise et ses employés est cohérente et synchronisée des deux côtés.

```
package fr.tharsanvishnukumar.employe;

public abstract class Employe {
    private String nom;
    // 0 ⇒ salaire
    // 1 ⇒ heur travaille
    private double[] infosSalaire = new double[2];

    private Entreprise entreprise;
    public Entreprise getEntreprise() {
        return entreprise;
    }
    public void rejoindreEntreprise(Entreprise entreprise) throws EmployeNonTr
        if (this.entreprise == entreprise) {
            return;
        }
        // Si l'employé appartient déjà à une entreprise, la retirer de cette entrepr
        if (this.entreprise != null) {
            this.entreprise.enleverEmploye(this);
        }
        // Ajouter l'employé à la nouvelle entreprise
        this.entreprise = entreprise;
        entreprise.ajouterEmploye(this);
    }
    public Employe(String nom) {
        this.nom = nom;
    }
    public Employe(String nom,double salaire) {
```

```
        this.nom = nom;
        infosSalaire[0] = salaire;
    }
    public Employe(String nom,double salaire,double heursTravaillees) {
        this.nom = nom;
        infosSalaire[0] = salaire;
        infosSalaire[1] = heursTravaillees;
    }

    public String getNom() {
        return nom;
    }

    public double[] getInfosSalaire() {
        return infosSalaire;
    }

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

    public void setInfosSalaire(double... infosSalaire) {
        this.infosSalaire[0] = infosSalaire[0];
        this.infosSalaire[1] = infosSalaire[1];
    }
    public abstract double getSalaire();
}
```

3) création d'une méthode `toString` pour l'entreprise.

La méthode `toString()` a été ajoutée à la classe `Entreprise` pour permettre une description complète de l'entreprise et de ses employés. Cette méthode utilise un `StringBuilder` pour construire la chaîne de caractères de manière efficace. Elle affiche le nom de l'entreprise suivi de la liste des employés, avec un préfixe `-` devant chaque nom d'employé. Si aucun employé n'est présent, la méthode affiche "Aucun employé dans l'entreprise". La logique de la méthode parcourt la liste `employes` et ajoute chaque nom d'employé à la chaîne de description. Cette

amélioration permet un affichage clair et structuré des détails de l'entreprise, facilitant le suivi et la présentation des informations sur les employés associés.

```
package fr.tharsanvishnukumar.employe;

import java.util.ArrayList;

public class Entreprise {
    private String nom;
    private ArrayList<Employe> employes;

    public Entreprise(String nom) {
        this.nom = nom;
        this.employes = new ArrayList<>();
    }

    public String getNom() {
        return nom;
    }

    public ArrayList<Employe> getEmployes() {
        return employes;
    }

    public void ajouterEmploye(Employe employe) {
        if (!employes.contains(employe)) {
            employes.add(employe);
        } else {
            System.out.println("L'employé est déjà dans l'entreprise.");
        }
    }

    public void enleverEmploye(Employe employe) throws EmployeNonTrouveE
        if (employes.contains(employe)) {
            employes.remove(employe);
        } else {
            throw new EmployeNonTrouveException("L'employé n'existe pas dans
        }
```

```
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Entreprise : ").append(nom).append("\n");
        sb.append("Employés :\n");

        if (employes.isEmpty()) {
            sb.append("Aucun employé dans l'entreprise.");
        } else {
            for (Employe employe : employes) {
                sb.append("- ").append(employe.getNom()).append("\n");
            }
        }
        return sb.toString();
    }
}
```

4) Pour tester la classe Entreprise.

La classe `TestEntreprise` teste les fonctionnalités de la classe `Entreprise` et de ses associations avec les employés. Trois employés sont créés : deux employés horaires (`EmployeHoraire`) et un commercial (`Commercial`), chacun ayant des caractéristiques spécifiques. Deux entreprises, `TechCorp` et `InnovateLtd`, sont instanciées. Les employés sont ensuite assignés aux entreprises respectives en utilisant la méthode `ajouterEmploye()`, garantissant que l'association bidirectionnelle entre les employés et leurs entreprises est correctement gérée. Enfin, les détails de chaque entreprise, y compris le nom de l'entreprise et les noms des employés, sont affichés en utilisant la méthode `toString()` de la classe `Entreprise`. Cette classe de test valide les fonctionnalités principales du système, notamment l'ajout d'employés aux entreprises et l'affichage des informations associées.

```
package fr.tharsanvishnukumar.employe;

public class TestEntreprise {
    public static void main(String[] args) {
```

```
        // Création de deux entreprises
        Entreprise entreprise1 = new Entreprise("TechCorp");
        Entreprise entreprise2 = new Entreprise("InnovateLtd");

        // Création d'employés
        Employe employe1 = new EmployeHoraire("Alice", 20, 40, 30);
        Employe employe2 = new EmployeHoraire("Bob", 23, 40, 30);
        Employe employe3 = new Commercial("Diane",2900);

        // Ajout des employés à l'entreprise 1
        entreprise1.ajouterEmploye(employe1);
        entreprise1.ajouterEmploye(employe2);

        // Ajout des employés à l'entreprise 2
        entreprise2.ajouterEmploye(employe3);

        // Affichage des détails des entreprises et de leurs employés
        System.out.println(entreprise1);
        System.out.println(entreprise2);
    }
}
```

Exercice 2:

Création des Exceptions

Les classes `NonProfitException` et `SecretMissionException` sont des exceptions personnalisées qui héritent de la classe `Exception`.

1. **`NonProfitException`** : utilisée pour signaler des erreurs liées à des opérations incompatibles avec une organisation à but non lucratif. Elle accepte un message décrivant l'erreur via son constructeur.
2. **`SecretMissionException`** : utilisée pour gérer des erreurs ou restrictions spécifiques à des missions ou activités confidentielles. Elle prend également un message pour préciser la nature du problème.

Ces classes permettent une gestion ciblée des exceptions, offrant des messages explicatifs pour faciliter le suivi et le débogage.

```
package fr.tharsanvishnukumar.entreprise;

public class NonProfitException extends Exception {
    public NonProfitException(String message) {
        super(message);
    }
}
```

```
package fr.tharsanvishnukumar.entreprise;

public class SecretMissionException extends Exception {
    public SecretMissionException(String message) {
        super(message);
    }
}
```

Création des classes `Entreprises`

La classe `Entreprise` représente une entreprise avec des attributs comme le nom, le nombre d'employés, le capital et une description de sa mission. Elle inclut un constructeur pour initialiser ces attributs lors de la création d'une instance. La méthode `mission()` retourne la description de la mission de l'entreprise et peut lancer une exception personnalisée `SecretMissionException` pour signaler des missions confidentielles. La méthode `capital()` renvoie le capital de l'entreprise sous forme entière et peut lever une exception `NonProfitException` pour les cas où le capital est incompatible avec une organisation à but non lucratif. La méthode `getNom()` fournit le nom de l'entreprise. Cette classe combine gestion d'informations basiques et contrôle d'accès à certaines données critiques grâce à des exceptions spécifiques.

```
package fr.tharsanvishnukumar.entreprise;

public class Entreprise {
    private int nombreEmployes;
    private double capital;
    private String nom;
    private String missionDescription;
```

```
    public Entreprise(String nom, int nombreEmployes, double capital, String mi
        this.nom = nom;
        this.nombreEmployes = nombreEmployes;
        this.capital = capital;
        this.missionDescription = mission;
    }

    public String mission() throws SecretMissionException {
        return missionDescription;
    }

    public int capital() throws NonProfitException {
        return (int)capital;
    }

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

La classe `EntrepriseSansProfit` est une sous-classe de `Entreprise` qui représente une organisation à but non lucratif. Son constructeur initialise les attributs en appelant le constructeur de la classe parente avec une mission par défaut définie comme "Mission caritative". La méthode `capital()` est redéfinie pour toujours lancer une exception personnalisée `NonProfitException`, signalant que l'entreprise ne réalise pas de profit. Cette implémentation reflète le caractère distinct des entreprises sans but lucratif en empêchant l'accès au capital comme indicateur de profit.

```
package fr.tharsanvishnukumar.entreprise;

public class EntrepriseSansProfit extends Entreprise {
    public EntrepriseSansProfit(String nom, int nombreEmployes, double capita
        super(nom, nombreEmployes, capital, "Mission caritative");
    }

    @Override
```

```
    public int capital() throws NonProfitException {
        throw new NonProfitException("L'entreprise " + getNom() + " ne fait pas
    }
}
```

La classe `EntrepriseSecrete` est une sous-classe de `Entreprise` qui modélise une organisation ayant des activités confidentielles. Le constructeur initialise les attributs en appelant le constructeur de la classe parente avec une mission prédéfinie intitulée "Mission secrète". La méthode `mission()` est redéfinie pour lever systématiquement une exception personnalisée `SecretMissionException`, indiquant que la mission de l'entreprise est strictement confidentielle.

```
package fr.tharsanvishnukumar.entreprise;

public class EntrepriseSecrete extends Entreprise {
    public EntrepriseSecrete(String nom, int nombreEmployes, double capital) {
        super(nom, nombreEmployes, capital, "Mission secrète");
    }

    @Override
    public String mission() throws SecretMissionException {
        throw new SecretMissionException("La mission de " + getNom() + " est t
    }
}
```

Création de test d'entreprise

La classe `TestEntreprise` teste différentes sous-classes et fonctionnalités associées à la classe `Entreprise`. Elle contient une méthode statique `afficherEntreprises()` qui affiche les informations des entreprises fournies dans un tableau. Pour chaque entreprise, son nom est affiché, suivi de sa mission et de son capital. Les exceptions `SecretMissionException` et `NonProfitException` sont gérées pour signaler les missions confidentielles et les organisations sans profit. La méthode principale crée un tableau d'entreprises comprenant des instances de `Entreprise`, `EntrepriseSecrete` et `EntrepriseSansProfit`, représentant différents types d'organisations. Ce programme illustre l'héritage et la gestion des exceptions en manipulant divers types d'entreprises tout en respectant leurs spécificités.

```
package fr.tharsanvishnukumar.entreprise;

public class TestEntreprise {
    public static void afficherEntreprises(Entreprise[] entreprises) {
        for (Entreprise entreprise : entreprises) {
            System.out.println("Entreprise : " + entreprise.getNom());

            try {
                System.out.println("Mission : " + entreprise.mission());
            } catch (SecretMissionException e) {
                System.out.println("Mission : " + e.getMessage());
            }

            try {
                System.out.println("Capital : " + entreprise.capital());
            } catch (NonProfitException e) {
                System.out.println("Capital : " + e.getMessage());
            }

            System.out.println("---");
        }
    }

    public static void main(String[] args) {
        Entreprise[] entreprises = {
                new Entreprise("Ford", 200000, 1000000000, "Construire des véhic
                new EntrepriseSecrete("CIA", 10000, 500000000),
                new EntrepriseSecrete("Spectre", 5000, 250000000),
                new EntrepriseSansProfit("CroixRouge", 50000, 0),
                new Entreprise("Microsoft", 220000, 2500000000.0, "Développer d
        };

        afficherEntreprises(entreprises);
    }

}
```

Exercice 3:

Création de la classe `Cinema`

La classe `Cinema` représente un cinéma composé d'une liste de salles. Le constructeur initialise cette liste vide. La méthode `placer(Spectateur spectateur)` tente de placer un spectateur dans la dernière salle de la liste. Si cette salle est pleine, elle attrape l'exception `SallePleineException` et crée une nouvelle salle avec une capacité par défaut de 100 places, dans laquelle le spectateur est ajouté. Ensuite, la nouvelle salle est ajoutée à la liste des salles. La méthode `getSalles()` retourne la liste des salles du cinéma. Cette classe gère donc l'ajout de spectateurs de manière dynamique et crée de nouvelles salles lorsque la dernière salle est pleine, garantissant ainsi un placement de spectateurs sans interruption.

```
package fr.tharsanvishnukumar.cinema;

import java.util.ArrayList;
import java.util.List;

public class Cinema {
    private List<Salle> salles;

    public Cinema() {
        this.salles = new ArrayList<>();
    }

    public void placer(Spectateur spectateur) throws SallePleineException  {
        try {
            Salle salle = salles.getLast();
            salle.ajouterSpectateur(spectateur);
        } catch (Exception e) {
            Salle nouvelleSalle = new Salle(100); // Capacité par défaut de 100 plac
            nouvelleSalle.ajouterSpectateur(spectateur);
            salles.add(nouvelleSalle);
        }
    }
```

```
    public List<Salle> getSalles() {
        return salles;
    }
}
```

Création de la classe `Salle` de Cinéma

La classe `Salle` représente une salle de cinéma avec une capacité maximale de spectateurs. Elle possède un constructeur qui initialise la capacité et crée une liste vide de spectateurs. La méthode `ajouterSpectateur(Spectateur spectateur)` permet d'ajouter un spectateur à la salle, mais vérifie d'abord si la taille de la liste des spectateurs est inférieure à la capacité maximale. Si la salle est pleine, une exception `SallePleineException` est lancée. Les méthodes `getCapacite()` et `getSpectateurs()` permettent de récupérer respectivement la capacité de la salle et la liste des spectateurs présents. Cette classe permet de gérer les spectateurs et de s'assurer que la salle ne dépasse pas sa capacité maximale.

```
package fr.tharsanvishnukumar.cinema;

import java.util.ArrayList;
import java.util.List;

public class Salle {
    private int capacite;
    private List<Spectateur> spectateurs;

    public Salle(int capacite) {
        this.capacite = capacite;
        this.spectateurs = new ArrayList<>();
    }

    public void ajouterSpectateur(Spectateur spectateur) throws SallePleineExc
        if (spectateurs.size() < capacite) {
            spectateurs.add(spectateur);
        } else {
            throw new SallePleineException(spectateur);
        }
    }
```

```
    public int getCapacite() {
        return capacite;
    }

    public List<Spectateur> getSpectateurs() {
        return spectateurs;
    }
}
```

Création de la classe `Spectateur`

La classe `Spectateur` représente un spectateur de cinéma. Elle contient deux attributs privés : `nom` et `age`, qui sont définis par le constructeur. Celui-ci initialise les valeurs de `nom` et `age` lors de la création de l'objet. Les méthodes `getNom()` et `getAge()` sont des accesseurs permettant de récupérer les valeurs des attributs. Cette classe est utilisée pour identifier et stocker les informations d'un spectateur, facilitant ainsi la gestion des spectateurs dans les salles et le cinéma.

```
package fr.tharsanvishnukumar.cinema;

public class Spectateur {
    private String nom;
    private int age;

    public Spectateur(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    public String getNom() {
        return nom;
    }

    public int getAge() {
        return age;
```

```
    }
}
```

Création de l'exception `SallePleineException`

La classe `SallePleineException` est une exception personnalisée qui est lancée lorsqu'une salle de cinéma atteint sa capacité maximale et ne peut plus accueillir de spectateurs supplémentaires. Elle hérite de la classe `Exception` et contient un attribut privé `spectateur` qui représente le spectateur pour lequel la salle est pleine. Le constructeur initialise cet attribut lors de la création de l'exception. La méthode `getSpectateur()` est un accesseur permettant de récupérer le spectateur concerné par l'exception. Cette classe permet de signaler et de gérer des erreurs spécifiques liées à la gestion de la capacité des salles de cinéma.

```
package fr.tharsanvishnukumar.cinema;

public class SallePleineException extends Exception {
    private Spectateur spectateur;

    public SallePleineException(Spectateur spectateur) {
        this.spectateur = spectateur;
    }

    public Spectateur getSpectateur() {
        return spectateur;
    }
}
```

Création de la classe de test pour tester le Cinema

La classe `TestCinema` sert de point d'entrée pour tester le fonctionnement de l'application de gestion de cinéma. Elle commence par créer une instance de la classe `Cinema`. Ensuite, elle crée plusieurs objets `Spectateur` représentant des personnes avec des noms et des âges différents. Ces spectateurs sont ensuite placés dans le cinéma à l'aide de la méthode `placer()` de la classe `Cinema`, qui gère l'ajout des spectateurs à des salles, en tenant compte de leur capacité. Si une salle est pleine, une nouvelle salle est créée automatiquement. Le

programme affiche ensuite les détails des salles, y compris leur capacité et les spectateurs qu'elles contiennent. Si une exception se produit lors du placement des spectateurs (comme la salle étant pleine), elle est capturée et imprimée sur la console.

```
package fr.tharsanvishnukumar.cinema;

public class TestCinema {
    public static void main(String[] args) {
        Cinema cinema = new Cinema();

        // Créer quelques spectateurs
        Spectateur spectateur1 = new Spectateur("Jean Dupont", 35);
        Spectateur spectateur2 = new Spectateur("Marie Dupont", 28);
        Spectateur spectateur3 = new Spectateur("Pierre Leblanc", 42);
        Spectateur spectateur4 = new Spectateur("Camille Mercier", 19);
        Spectateur spectateur5 = new Spectateur("Julien Rousseau", 24);

        // Placer les spectateurs dans le cinéma
        try {
            cinema.placer(spectateur1);
            cinema.placer(spectateur2);
            cinema.placer(spectateur3);
            cinema.placer(spectateur4);
            cinema.placer(spectateur5);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Afficher les salles et leurs spectateurs
        System.out.println("Salles du cinéma :");
        for (Salle salle : cinema.getSalles()) {
            System.out.println("Salle - Capacité : " + salle.getCapacite());
            for (Spectateur spectateur : salle.getSpectateurs()) {
                System.out.println("- " + spectateur.getNom() + " (" + spectateur.ge
            }
            System.out.println();
        }
```

```
    }
}
```