Exercices Java - Gestion de Cachets de Concerts

École Nationale d'Ingénieurs de Tarbes - POO : Encapsulation, Héritage, Polymorphisme

---

Vérification de JShell (à faire avant de commencer)

Ouvrez https://onecompiler.com/jshell et tapez :

System.out.println("JShell OK !");

int test = 42;

System.out.println("Valeur:" + test);

Vous devez voir :

JShell OK ! Valeur: 42

Si c'est le cas, vous pouvez commencer les exercices.

Exercice 1 : Encapsulation du Cachet

Objectif

Protéger le cachet d'un groupe de musique et contrôler son accès.

Contexte

Une salle de concert engage des groupes. Le cachet (prix de la prestation) est confidentiel : seul le groupe lui-même et l'organisateur doivent pouvoir le consulter.

Consigne

Créez la classe Groupe avec :

nom (String, public)

cachet (double, privé)

nbMembres (int, public)

Un constructeur Groupe(String nom, int nbMembres)

Getter et setter pour cachet avec validation (montant > 0)

Méthode partParMusicien() qui retourne le cachet divisé par le nombre de membres

toString() pour afficher les informations

Code à compléter dans JShell

```
public class Groupe {
public String nom;
private double cachet;
public int nbMembres;
// TODO: constructeur
// TODO: getter getCachet()
// TODO: setter setCachet(double c) avec validation
// TODO: partParMusicien()
// TODO: toString()
}

// Tests
Groupe g = new Groupe("Les Rockers", 4);
g.setCachet(2000);
System.out.println(g);
System.out.println("Part par musicien: " + g.partParMusicien() + "€");
```

Questions de réflexion

Que se passe-t-il si vous essayez d'accéder directement à g.cachet depuis JShell ?

Pourquoi est-il important de valider le cachet dans le setter plutôt que de laisser l'attribut public ?

Que se passerait-il si nbMembres était à 0 lors du calcul de partParMusicien() ?

Exercice 2 : Héritage et Spécialisation

Objectif

Créer des types de groupes spécialisés avec des règles de répartition différentes.

Contexte

Groupe amateur : Cachet divisé équitablement entre musiciens (déjà fait)

Groupe pro avec manager : Le manager prend 20% de commission, le reste est divisé entre les musiciens

Consigne

Créez GroupePro qui hérite de Groupe :

Attribut manager (String, public)

Constructeur GroupePro(String nom, int nbMembres, String manager)

Redéfinissez partParMusicien() pour prendre en compte la commission

Ajoutez commissionManager() qui retourne le montant de la commission

Code à compléter dans JShell

```
// Réutilisez votre classe Groupe de l'exercice 1, ou recopiez-la

public class GroupePro extends Groupe {
public String manager;
// TODO: constante COMMISSION = 0.20
// TODO: constructeur avec super(...)
// TODO: commissionManager()
// TODO: override partParMusicien()
// TODO: toString() enrichi
}

// Tests
Groupe amateurs = new Groupe("Les Debutants", 3);
amateurs.setCachet(1500);
```

GroupePro pros = new GroupePro("The Stars", 5, "John Smith");

pros.setCachet(10000);

System.out.println("Amateurs: " + amateurs.partParMusicien() + "€/musicien");

System.out.println("Pros: " + pros.partParMusicien() + "€/musicien");

System.out.println("Commission manager: " + pros.commissionManager() + "€");

Questions de réflexion

Pourquoi utilise-t-on super(...) dans le constructeur de GroupePro ?

Que se passerait-il si Groupe n'était pas conçu pour l'héritage (attributs privés sans getters) ?

Peut-on appeler commissionManager() sur une variable de type Groupe ? Pourquoi ?

Exercice 3 : Polymorphisme et Organisation

Objectif

Manipuler des groupes de différents types de manière uniforme.

Contexte

L'organisateur de la salle gère une programmation variée (amateurs et pros mélangés). Il doit pouvoir calculer les dépenses totales et établir les fiches de paie sans se soucier du type de chaque groupe.

Consigne

Créez la classe Organisateur :

Liste de Groupe (utilisez ArrayList)

Méthode ajouter(Groupe g) pour ajouter un groupe

Méthode cachetTotal() qui calcule la somme de tous les cachets

Méthode fichePaie() qui affiche pour chaque groupe :

Le nom

Le cachet brut

Si c'est un pro : la commission du manager

La part par musicien

Code à compléter dans JShell

```
import java.util.ArrayList;
import java.util.List;
public class Organisateur {
private List<Groupe> programmation = new ArrayList<>();
// TODO: ajouter(Groupe g)
// TODO: cachetTotal()
// TODO: fichePaie() avec instanceof pour détecter les GroupePro
}

// Scénario
Organisateur orga = new Organisateur();
Groupe g1 = new Groupe("Garage Band", 3);
g1.setCachet(1200);
GroupePro g2 = new GroupePro("Super Star", 4, "Big Boss");
g2.setCachet(15000);
Groupe g3 = new Groupe("Les Copains", 5);
g3.setCachet(800);
orga.ajouter(g1);
orga.ajouter(g2);
orga.ajouter(g3);
orga.fichePaie();
System.out.println("Total dépenses: " + orga.cachetTotal() + "€");
```

Questions de réflexion

Pourquoi List<Groupe> peut-il contenir à la fois des Groupe et des GroupePro ?

Comment le compilateur sait-il quelle version de partParMusicien() appeler ?

L'utilisation de instanceof est-elle élégante ? Comment pourrait-on l'éviter ?

Exercice 4 : Élimination du instanceof (Bonus)

Objectif

Utiliser le polymorphisme pur pour éviter les tests de type.

Contexte

L'instanceof dans fichePaie() est une "code smell". On peut faire mieux en déléguant l'affichage des détails à chaque classe.

Consigne

Ajoutez une méthode detailsRepartition() dans Groupe :

Dans Groupe : retourne "Part égale: X€ par musicien"

Dans GroupePro : retourne "Manager (nom): Y€, Musiciens: X€ chacun"

Modifiez fichePaie() pour utiliser uniquement detailsRepartition() sans instanceof

Code à compléter dans JShell

```
// Dans Groupe
public String detailsRepartition() {
// TODO
}

// Dans GroupePro (override)
@Override
public String detailsRepartition() {
// TODO
}
```

```
// Nouvelle version de fichePaie() dans Organisateur
public void fichePaieV2() {
for (Groupe g : programmation) {
System.out.println(g.nom + ": " + g.detailsRepartition());
}
}
```