Notes Importantes – TP4

Ce document résume les concepts clés que vous devez retenir des trois exercices du TP4.

Classes abstraites - Pourquoi et quand les utiliser?

Définition

Une classe abstraite est une classe qui:

- Ne peut pas être instanciée directement
- Peut contenir des méthodes abstraites (sans implémentation)
- Peut aussi contenir des méthodes concrètes (avec implémentation)

Quand utiliser une classe abstraite?

1. **Quand on peut identifier une classe "générique"** dont les instances n'ont pas de sens:
    - `Employe` sans type spécifique n'a pas de sens (comment calculer son salaire?)
    - `Formateur` générique n'est pas suffisant (comment calculer sa rémunération?)
    - `Forme` sans type précis est un concept abstrait (quelle formule de volume utiliser?)

2. **Quand plusieurs classes partagent:**
    - Des attributs communs (nom, prénom, heures, etc.)
    - Des comportements communs (bouger, toString, etc.)
    - Mais ont des comportements spécifiques (calcul du salaire, calcul du volume)

Syntaxe essentielle

```
// Déclaration d'une classe abstraite
public abstract class ClasseAbstraite {
    // Attributs normaux
    private String attributCommun;

    // Méthodes concrètes (avec implémentation)
    public void methodeCommune() {
        // code ici
    }

    // Méthodes abstraites (sans implémentation)
    public abstract double methodeSpecifique();
}
```

Méthodes abstraites vs méthodes concrètes

Méthode abstraite

- Déclarée avec le mot-clé `abstract`
- Sans implémentation (se termine par `;`)
- Doit être implémentée par toutes les sous-classes non abstraites
- Exemple: `public abstract double calculerSalaire();`

Méthode concrète dans une classe abstraite

- Code réutilisable par toutes les sous-classes
- Peut être redéfinie (override) si nécessaire
- Exemple: la méthode `getNom()` dans `Employe`
- Exemple: la méthode `calculerPoids()` dans `Forme`

La classe final - Empêcher l'héritage

Définition

Une classe déclarée `final` ne peut pas être étendue (aucune classe ne peut en hériter).

Pourquoi l'utiliser?

- Pour garantir qu'un comportement spécifique ne sera pas modifié
- Pour des raisons de sécurité ou d'intégrité des données

Exemple

```
public final class Cube extends Brique {
    // Implémentation
}
```

Hiérarchie d'héritage - Relations "est un"

Principe fondamental

L'héritage représente une relation "est un":

- Un `Commercial` **est un** `Employe`
- Un `FormateurInterne` **est un** `Formateur`
- Un `Cube` **est une** `Brique` spéciale (où largeur = longueur = hauteur)

Avantages de l'héritage

1. **Réutilisation du code**:
    - Les attributs et méthodes communs ne sont définis qu'une fois

- Les sous-classes héritent automatiquement de ces éléments

2. **Polymorphisme**:
   - Une référence de type parent peut désigner un objet de type enfant
   - `Employe e = new Commercial()`
   - `Formateur f = new FormateurExterne()`
   - `Forme f = new Boule()`

3. **Extensibilité**:
   - Facile d'ajouter de nouveaux types sans modifier le code existant
   - Exemple: ajouter un nouveau type d'employé ou une nouvelle forme

Composition vs Héritage

Composition: relation "a un"

- Une `Forme` **a un** `Point3D` comme centre de gravité
- Un `Formateur` **a des** compétences

Héritage: relation "est un"

- Un `Commercial` **est un** `Employe`
- Une `Boule` **est une** `Forme`

Quand choisir l'une ou l'autre?

- Utilisez l'héritage pour spécialiser un concept
- Utilisez la composition pour combiner des fonctionnalités

Conception de classes abstraites - Bonnes pratiques

1. **Placer les attributs communs dans la classe abstraite**
   - Nom, prénom, âge pour tous les employés
   - Centre de gravité, densité pour toutes les formes

2. **Définir des méthodes abstraites pour les comportements variables**
   - `calculerSalaire()` varie selon le type d'employé
   - `calculerSurface()` et `calculerVolume()` varient selon la forme

3. **Fournir des implémentations par défaut quand c'est possible**
    - `getNom()` dans `Employe`
    - `calculerPoids()` dans `Forme`

Points à retenir pour le contrôle

1. **Une classe abstraite ne peut pas être instanciée**

```
// Impossible - erreur de compilation
Employe e = new Employe(...);
```

4. **Une classe avec au moins une méthode abstraite doit être déclarée abstraite**

```
// Impossible si calculerSalaire() est abstraite
public class Employe { ... }
```

7. **Toute classe non abstraite qui hérite d'une classe abstraite doit implémenter toutes ses méthodes abstraites**

```
public class Commercial extends Employe {
    // Doit implémenter calculerSalaire()
    @Override
    public double calculerSalaire() { ... }
}
```

13. **Une classe finale ne peut pas être étendue**

```
// Impossible - erreur de compilation
public class SuperCube extends Cube { ... }
```

16. **Le polymorphisme se produit à l'exécution**

```
Employe[] employes = new Employe[3];
employes[0] = new Commercial(...);
employes[1] = new Technicien(...);

// La méthode appelée dépend du type réel
for (Employe e : employes) {
    e.calculerSalaire(); // Polymorphisme
}
```