Notes Importantes – TP3

1. L'héritage

Ce qu'il faut comprendre

L'héritage permet à une classe (classe enfant) d'hériter des attributs et méthodes d'une autre classe (classe parent).

Pourquoi c'est utile

- **Réutilisation du code**: On évite de réécrire le même code pour chaque classe similaire
- **Organisation logique**: On structure le code selon des relations "est un"
- **Maintenance facilitée**: Une modification dans la classe parent affecte automatiquement toutes les classes enfants

Dans les exercices

- Dans la bibliothèque: `LivreEnfant` et `LivreScolaire` héritent de `Livre`
- Dans la banque: `CompteRemunere` hérite de `CompteBancaire`

Syntaxe essentielle

```
public class LivreEnfant extends Livre {
    // Nouveaux attributs et méthodes spécifiques
}
```

2. Le polymorphisme

Ce qu'il faut comprendre

Le polymorphisme permet d'utiliser une référence de type parent pour manipuler un objet de type enfant. La méthode exécutée sera celle de la classe réelle de l'objet, pas celle du type de la référence.

Pourquoi c'est utile

- **Traitement uniforme**: On peut traiter des objets de différentes sous-classes de manière identique
- **Flexibilité**: On peut ajouter de nouvelles sous-classes sans modifier le code qui les utilise
- **Extensions faciles**: On peut étendre le comportement sans casser le code existant

Dans nos exercices

```
// Dans la classe Biblio
Livre[] livres = new Livre[6];
livres[0] = new Livre(...);
livres[1] = new LivreEnfant(...);
livres[2] = new LivreScolaire(...);

// Même si la référence est de type Livre, la méthode prixTTC() appelée
// sera celle de la classe réelle (LivreEnfant, LivreScolaire...)
for (Livre livre : livres) {
    System.out.println(livre.prixTTC());
}
```

3. La redéfinition de méthodes (Override)

Ce qu'il faut comprendre

La redéfinition permet à une classe enfant de changer le comportement d'une méthode héritée, tout en gardant la même signature (même nom, mêmes paramètres, même type de retour).

Pourquoi c'est utile

- **Spécialisation**: Chaque sous-classe peut adapter le comportement à ses besoins spécifiques
- **Polymorphisme**: C'est ce qui permet au polymorphisme de fonctionner

Dans nos exercices

```
// Dans la classe Livre
public double prixTTC() {
    return prix * (1 + TAUX_TAXE);
}

// Dans la classe LivreEnfant
@Override
public double prixTTC() {
    double prixTTCNormal = super.prixTTC();
    return prixTTCNormal * (1 - REDUCTION); // 50% de réduction
}
```

Points importants

- L'annotation `@Override` n'est pas obligatoire mais fortement recommandée
- `super.prixTTC()` appelle la méthode de la classe parente
- La signature doit être exactement la même que celle de la classe parente

4. Le mot-clé super

Ce qu'il faut comprendre

Le mot-clé `super` permet d'accéder aux membres (attributs et méthodes) de la classe parente.

Pourquoi c'est utile

- **Appel au constructeur parent**: Initialiser les attributs hérités
- **Accès aux méthodes parentes**: Utiliser le comportement de base avant de le personnaliser

Dans nos exercices

```
// Appel au constructeur parent
public CompteRemunere(String code, Personne titulaire, BigDecimal
tauxInteret) {
    super(code, titulaire); // Initialise les attributs hérités
    this.taux = tauxInteret;
}

// Appel à une méthode parente
@Override
public void deposer(BigDecimal montant) {
    BigDecimal bonus = montant.multiply(BONUS_DEPOT);
    super.deposer(montant.add(bonus)); // Utilise la méthode de la classe
parente
}
```

5. L'opérateur instanceof et le casting

Ce qu'il faut comprendre

- `instanceof` vérifie si un objet est d'un type spécifique
- Le casting permet de convertir une référence d'un type à un autre

Pourquoi c'est utile

Permet d'accéder aux méthodes spécifiques d'une sous-classe quand on a une référence de type parent.

Dans nos exercices

```
// Dans la classe Biblio
for (Livre livre : livres) {
    // Vérification du type réel
    if (livre instanceof LivreEnfant) {
        // Conversion (cast) pour pouvoir appeler la méthode spécifique
        LivreEnfant livreEnfant = (LivreEnfant) livre;
        livreEnfant.afficherTrancheAge(); // Méthode spécifique
    }
}
```

Points importants

- Toujours vérifier avec `instanceof` avant de faire un cast pour éviter une `ClassCastException`
- Le cast ne change pas l'objet, seulement le type de la référence

6. Intérêts composés et calcul de puissance avec BigDecimal

Ce qu'il faut comprendre

Pour calculer les intérêts composés sur plusieurs années, on utilise la formule:

```
Solde futur = Solde initial × (1 + taux/100)^n
```

Pourquoi c'est important

- BigDecimal n'a pas de méthode `pow()` pour des exposants variables
- Il faut implémenter le calcul de puissance manuellement avec une boucle

Dans notre exercice bancaire

```
public BigDecimal calculerSoldeFutur(int annees) {
    // Convertir le taux de pourcentage à sa forme décimale (ex: 5% -> 0.05)
    BigDecimal tauxDecimal = taux.divide(new BigDecimal("100"), 10,
RoundingMode.HALF_UP);

    // Calculer (1 + taux/100)
    BigDecimal facteur = BigDecimal.ONE.add(tauxDecimal);

    // Élever à la puissance n: (1 + taux/100)^n
    BigDecimal facteurPuissance = BigDecimal.ONE;
    for (int i = 0; i < annees; i++) {
        facteurPuissance = facteurPuissance.multiply(facteur);
    }

    // Calculer le solde futur: soldeActuel * facteurPuissance
    return getSolde().multiply(facteurPuissance).setScale(2,
RoundingMode.HALF_UP);
}
```