2 — Héritage & super

🏦 Exercice Java — CompteCourant & CompteEpargne

🎯

Objectifs pédagogiques

- Construire une **hiérarchie de classes** à partir d'un modèle parent (Compte).
- Utiliser le mot-clé **super** pour appeler le constructeur du parent.
- **Surcharger ou redéfinir** des méthodes (polymorphisme).
- Manipuler une **liste polymorphe** (List) dans Banque.
- Introduire la notion de **comportement spécialisé** dans les sous-classes.

🧠

Contexte

Tu travailles toujours sur ton application bancaire.

Désormais, ta banque doit gérer **plusieurs types de comptes** :

- Des **comptes courants** : utilisables au quotidien, avec un **découvert autorisé**.
- Des **comptes épargne** : destinés à faire fructifier l'argent, avec un **taux d'intérêt annuel**.

Tu vas donc faire évoluer la classe Compte en une classe **mère** commune,

et créer deux sous-classes spécialisées : CompteCourant et CompteEpargne.

🧩

Étape 1 — Classe mère Compte

Cette classe représente les caractéristiques communes à tous les comptes.

Tu peux **reprendre** ton code du PALIER 1 et l'adapter.

✳️ Attributs protégés

```
protected String numero;
protected double solde;
```

> 🔎 Pourquoi protected ?

> Cela permet aux classes enfants d'y accéder directement,

> sans briser totalement l'encapsulation.

✳️ Constructeurs

- Compte(String numero, double solde) → à appeler depuis les sous-classes avec super(numero, solde).

✳️ Méthodes de base

- public void crediter(double montant)
- public void debiter(double montant)
- public double getSolde()
- public String getNumero()
- public String toString() (⚠️ méthode héritée et personnalisable)
- public double calculInteret() → **méthode à redéfinir dans les sous-classes**

💡 **Indice :**

Tu peux la laisser vide ou retourner 0.0 dans Compte.

Elle sera ensuite redéfinie ("overridden") par chaque type de compte.

🧩

Étape 2 — Classe fille CompteCourant

✳️ Attribut spécifique

```
private double decouvertAutorise;
```

✳️ Constructeur

Appelle le constructeur parent :

```
public CompteCourant(String numero, double solde, double decouvertAutorise)
{
    super(numero, solde);
    this.decouvertAutorise = decouvertAutorise;
}
```

✳️ Méthodes spécifiques

- Redéfinis debiter(double montant) → autorise le débit jusqu'à solde + decouvertAutorise.
- Redéfinis calculInteret() pour un taux **faible ou nul**. (par ex : retourne solde * 0.001)
- Redéfinis toString() pour inclure le découvert.

💡 **Indice :**

Utilise super.toString() pour réutiliser la logique du parent :

```
return super.toString() + " (Découvert : " + decouvertAutorise + " €)";
```

🧩

Étape 3 — Classe fille CompteEpargne

✳️ Attribut spécifique

```
private double tauxInteret;
```

✳️ Constructeur

Appelle le parent :

```
public CompteEpargne(String numero, double solde, double tauxInteret) {
    super(numero, solde);
```

```
    this.tauxInteret = tauxInteret;
}
```

✳️ Méthodes spécifiques

- Redéfinis calculInteret() → retourne solde * tauxInteret.
- Redéfinis toString() pour afficher le taux d'intérêt.

💡 **Indice :**

Tu peux utiliser String.format("%.2f", tauxInteret * 100) pour afficher le taux en %.

🧩

Étape 4 — Classe Banque

Ta classe Banque évolue : elle doit maintenant gérer **des comptes polymorphes**.

✳️ Attribut

```
private List<Compte> comptes = new ArrayList<>();
```

✳️ Méthodes

- public void ajouterCompte(Compte c)
- public void afficherTous() → affiche chaque compte via toString() (⚡ polymorphisme !)
- public void afficherBilan() → affiche le solde total et la somme des intérêts de tous les comptes.

💡 **Indice :**

Pas besoin de instanceof ici si tu appelles calculInteret() :

chaque objet utilisera automatiquement sa version spécifique (polymorphisme).

Exemple :

```
for (Compte c : comptes) {
    totalInterets += c.calculInteret();
}
```

🧩

Étape 5 — Classe Main (tests)

🧪 Scénario de test

1. Crée une Banque.
2. Ajoute :
   - 1 CompteCourant : BE1001, solde 500 €, découvert autorisé 200 €.
   - 1 CompteEpargne : BE2002, solde 1000 €, taux 2% (0.02).
3. Appelle :
   - afficherTous()
   - afficherBilan()
   - teste debiter() sur le compte courant pour vérifier le découvert.
   - appelle calculInteret() sur chaque compte et affiche le résultat.

💡 **Indices de progression :**

- Ton code compile mais tu n'as pas encore redéfini calculInteret() ? → tu verras 0.0 partout. C'est normal.
- Tu vois deux affichages différents pour CompteCourant et CompteEpargne ? → c'est le **polymorphisme** en action.

✅

Critères de réussite

- Utilisation correcte de super dans les constructeurs.
- calculInteret() redéfini dans les sous-classes.
- toString() exploite super.toString().
- Banque gère une List polymorphe.
- Code lisible et bien structuré.

⚡

Variante bonus : afficherBilan()

Affiche un résumé clair du patrimoine de la banque :

```
=== Bilan Banque ===
Nombre de comptes : 2
Solde total : 1500 €
Intérêts totaux : 20 €
```

💡 **Indice :**

Utilise StringBuilder pour composer proprement la chaîne de sortie.

```
Dataview (inline field '== Bilan Banque ===
Nombre de comptes : 2
Solde total : 1500 €
Intérêts totaux : 20 €'): Error:
-- PARSING FAILED ------------------------------------------------
-

> 1 | == Bilan Banque ===
    | ^
  2 | Nombre de comptes : 2
  3 | Solde total : 1500 €

Expected one of the following:

'(', 'null', boolean, date, duration, file link, list ('[1, 2,
3]'), negated field, number, object ('{ a: 1, b: 2 }'), string,
variable
```