

2 — Héritage & super



Exercice Java — Héritage & super

PALIER 2 — CompteCourant & CompteEpargne

(modélisation orientée héritage et polymorphisme)



Objectifs

- Comprendre la hiérarchie **parent** ↔ **enfants** entre classes.
 - Appeler et étendre les comportements d'une **classe mère**.
 - Reconnaître le rôle du mot-clé **super** (constructeurs et méthodes).
 - Manipuler une **liste polymorphe** (traiter plusieurs types d'objets comme un seul type commun).
-



Contexte

Ta banque s'agrandit.

Elle gère désormais **plusieurs types de comptes** ayant chacun leurs règles spécifiques.

L'objectif :

reprendre la classe Compte du PALIER 1,

et en faire une **classe de base** pour différents types de comptes spécialisés.



Diagramme UML simplifié

```
classDiagram
    class Compte {
        - numero : String
        - solde : double
        + crediter(montant : double) : void
        + debiter(montant : double) : void
        + calculInteret() : double
        + toString() : String
    }

    class CompteCourant {
        - decouvertAutorise : double
        + calculInteret() : double
        + debiter(montant : double) : void
        + toString() : String
    }

    class CompteEpargne {
        - tauxInteret : double
        + calculInteret() : double
        + toString() : String
    }

    class Banque {
        - comptes : List<Compte>
        + ajouterCompte(c : Compte) : void
        + afficherTous() : void
        + afficherBilan() : void
    }

    Compte <|-- CompteCourant
    Compte <|-- CompteEpargne
    Banque "1" --> "n" Compte
```



Consignes de conception



Étape 1 – Classe mère

Compte

- Contient les **attributs communs**.
- Définit les **méthodes génériques** (crediter, debiter, calculInteret, etc.).
- Sert de **point de départ** pour les sous-classes.

🧩 *Indice UML :*

- Méthodes + → publiques
 - Attributs - → privés (ou protégés si nécessaire pour héritage)
 - Tu décideras si la classe est abstract ou non selon ta logique.
-

🏦 Étape 2 – Classe fille

CompteCourant

- Hérite de Compte.
- Ajoute un **découvert autorisé**.
- Redéfinit certaines méthodes pour **adapter le comportement** (ex: calcul d'intérêts ou débit).

🧩 *Indice UML :*

- Redéfinition (override) = même signature que la méthode parent.
 - Penser au constructeur parent : relier les attributs communs avec super.
-

💰 Étape 3 – Classe fille

CompteEpargne

- Hérite aussi de Compte.
- Ajoute un **taux d'intérêt** spécifique.
- Calcule les intérêts selon ce taux.

🧩 *Indice UML :*

- Penser à comment le polymorphisme permettra à Banque de manipuler ce compte comme un simple Compte.



Étape 4 – Classe

Banque

- Gère **une collection de comptes** (courants et épargne confondus).
- Utilise une structure polymorphe (ex: List).
- Affiche un **bilan global** à partir des comptes.



Indice UML :

- Composition ou agrégation entre Banque et Compte (1 → n).



Diagramme d'interaction (vue d'ensemble)

```
sequenceDiagram
    participant M as Main
    participant B as Banque
    participant CC as CompteCourant
    participant CE as CompteEpargne

    M->>B: créer une Banque
    M->>B: ajouter CompteCourant
    M->>B: ajouter CompteEpargne
    M->>B: afficherTous()
    B->>CC: toString()
    B->>CE: toString()
    M->>B: afficherBilan()
    B->>CC: calculInteret()
    B->>CE: calculInteret()
```



Scénario attendu

1. Une banque gère plusieurs comptes (courants + épargne).
2. Chaque type calcule ses intérêts selon sa propre logique.

3. Banque affiche les informations et le total des soldes/intérêts.
4. Le polymorphisme fait que Banque ne sait pas si un compte est “courant” ou “épargne” — elle les traite uniformément.



Variante bonus — Méthode afficherBilan()

Affiche un résumé général :

```
=== Bilan Banque ===  
Nombre de comptes : X  
Solde total : Y €  
Intérêts totaux : Z €
```

🧠 *Indice UML :*

Cette méthode appartient à Banque.

Elle interagit avec les objets de type Compte, sans connaître leur type exact.



Mini quiz pour te guider (à répondre avant de coder)

1. Quelle est la différence entre *héritage* et *composition* ?
2. Pourquoi choisir super plutôt que this dans un constructeur ?
3. Qu'implique le fait d'utiliser List pour stocker des CompteEpargne et CompteCourant ?
4. Si tu ajoutes une méthode calculInteret() dans Compte, mais que chaque sous-classe la redéfinit, que se passe-t-il quand tu appelles c.calculInteret() dans une boucle ?
5. Que veut dire instanceof dans ce contexte, et pourquoi n'en as-tu peut-être pas besoin ici ?

```
Dataview (inline field '=== Bilan Banque ===  
Nombre de comptes : X  
Solde total : Y €  
Intérêts totaux : Z €'): Error:  
-- PARSING FAILED -----  
-
```

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

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
```