

Héritage, classes internes

Exercice 1 *Surcharge, redéfinition, constructeurs dans les hiérarchies d'héritage*

```
public class Document {
    private String titre;

    public String getNom() {
        return titre;
    }

    public Document(String titre) {
        super();
        this.titre = titre;
    }

    @Override
    public String toString() {
        return "Document [titre=" + titre + "]";
    }
}

public class Livre extends Document{

    private int nbChapitres;

    public int getNbChapitres() {
        return nbChapitres;
    }

    public Livre(String titre , int nbChapitres) {
        super(titre);
        this.nbChapitres = nbChapitres;
    }

    /*      public Livre(){
    }*/
}

public class Biblio {

    private ArrayList<Document> listeReferences=new ArrayList<Document>();

    public Biblio() {
    }

    public void ajoutDocument(Document d){
        listeReferences.add(d);
        System.out.println("ajout doc de " +d);
    }

    public void ajoutDocument(Livre l){
        listeReferences.add(l);
        System.out.println("ajout livre de "+l);
    }
}
```

```

        public boolean contains(Document d){
            return listeReferences.contains(d);
        }

        @Override
        public String toString() {
            return "Biblio [" + listeReferences + "]";
        }
    }

    public class BiblioSansDoublons extends Biblio {

        public void ajoutDocument(Document d){
            if (!contains(d)){
                super.ajoutDocument(d);
            }
        }

        public void ajoutDocument(Livre l){
            if (!contains(l)){
                super.ajoutDocument(l);
            }
        }
    }

    public class Main {

        public static void main(String[] args) {
            Livre l1=new Livre("l1", 3);
            Document l2=new Livre("l2", 4);

            Document d=new Document("d");

            Biblio b =new Biblio();
            BiblioSansDoublons bsd=new BiblioSansDoublons();
            Biblio bsd2=new BiblioSansDoublons();

            // ajout dans b:Biblio
            b.ajoutDocument(l1);
            b.ajoutDocument(l1);
            b.ajoutDocument(l2);
            b.ajoutDocument(d);
            System.out.println(b.toString());
            //      ajout dans bsd:BiblioSansDoublons
            bsd.ajoutDocument(l1);
            bsd.ajoutDocument(l1);
            bsd.ajoutDocument(l2);
            bsd.ajoutDocument(d);
            System.out.println(bsd.toString());
            //      ajout dans bsd2:BiblioSansDoublons
            bsd2.ajoutDocument(l1);
            bsd2.ajoutDocument(l1);
            bsd2.ajoutDocument(l2);
            bsd2.ajoutDocument(d);
            System.out.println(bsd2.toString());
        }
    }
}

```

Question 1. Dans la classe `Document`, à quoi correspond l'appel : `super()` dans le constructeur ? Est-il nécessaire ?

Question 2. Pourquoi a-t-on une erreur de compilation si on décommente le constructeur sans paramètre de la classe `Livre` ?

Question 3. Etudiez les méthodes de la classe `Biblio` et de sa classe fille, puis donnez le résultat de l'exécution du main de la classe `Main`.

Exercice 2 *Expressions arithmétiques*

On considère l'évaluation d'expressions arithmétiques formées à l'aide des quatre opérateurs binaires `+`, `-`, `*`, `/`.

Une expression est définie récursivement de la façon suivante : soit c'est une constante (par exemple 1.5) soit c'est une expression "complexe" de la forme `(a op b)` où `a` et `b` sont des expressions et `op` est l'un des quatre opérateurs. Écrire les classes Java permettant de construire et évaluer des expressions, de façon à ce que l'on puisse écrire par exemple (et par exemple dans une méthode main appartenant à une autre classe) :

```
Constante a = new Constante (5);
Constante b = new Constante (2);
Constante c = new Constante(3);
ExpressionComplexe e1 = new ExpressionComplexe (a, '+', b);

ExpressionComplexe e2 = new ExpressionComplexe (e1, '*', c);
ExpressionComplexe e3 = new ExpressionComplexe(new Constante(4), '*', e2);

System.out.println(a.eval()); // 5.0
System.out.println(e1.eval()); // 7.0
System.out.println(e2.eval()); // 21.0
System.out.println(e3.eval()); // 84.0
```

Exercice 3 *Classes internes*

Question 4. Créez une classe liste chaînée. Cette classe possèdera un attribut privé référençant la racine de la liste, qui sera de type `Node`. La classe `Node` sera une classe interne non statique à la classe liste de manière à faciliter l'accès des noeuds à la liste et de la liste aux noeuds. Un noeud a un nom, et connaît son successeur. Écrire de quoi créer un noeud sans successeur, ajouter un entier en tête de liste, connaître la taille de la liste, et afficher la liste.

Question 5. Ajoutez une méthode renverser qui retourne une nouvelle liste qui est la liste receveur retournée.

Question 6. Les instances de `Node` peuvent-elles être partagées par plusieurs listes ? Pourquoi ? Comment procéder si on souhaite que cela soit le cas ? Quelles en sont les implications ?

Exercice 4 *Classes locales*

Le but de cet exercice est de mettre en place une classe locale. On écrira à cet effet une classe `Personne` qui stocke à des fins d'affichage ultérieur la date de naissance sous forme de chaîne formatée de la forme `jj/mm/aaaa`. Afin de vérifier que la chaîne est bien formée, on introduit une méthode booléenne dans la classe `Personne`. Cette méthode introduira une classe locale `Date` munie d'attributs entiers pour le jour, le mois, et l'année, et d'un constructeur prenant une chaîne en paramètre et qui la ventile en jour, mois, année si elle est syntaxiquement bien formée (on pourra utiliser la méthode `matches` de la classe `String` avec une expression régulière). La classe `Date` introduira également une méthode qui vérifie que les champs jour, mois et année correspondent effectivement à une date possible (on pourra faire une vérification incomplète en vérifiant que le jour est entre 1 et 31, le mois entre 1 et 12 et l'année entre 1900 et l'année courante, ou une vérification plus complète grâce à l'utilisation des `GregorianCalendar` de Java).