TD n°5 - Correction

Classes et Héritage

Exercice 1 (Valeurs et références, surcharge)
Qu'affiche le programme suivant ?

```
1  class A{
2    private int val=0;
3
4    public static void incremente(int a){
5      a++;
6      System.out.println(a);
7    }
8
9    public static void incremente(A a){
10     a.val++;
11     System.out.println(a.val);
12   }
13
14   public static void main (String[] args) {
15     A objet = new A();
16     A autreObjet = new A();
17     incremente(objet.val);
18     incremente(objet.val);
19     incremente(objet);
20     incremente(objet);
21     incremente(autreObjet);
22     incremente(autreObjet);
23     if (objet==autreObjet) System.out.println("Egales");
24     else System.out.println("Differentes");
25   }
26 }
```

Exercice 2 (Classification)
On prend les classes suivantes : Etudiant, Personne, EtudiantTravailleur, Enseignant, EtudiantSportif et Travailleur.

1. dessinez une arborescence cohérente pour ces classes en la justifiant,
2. où se situeront les champs suivants : salaire, emploiDuTemps, anneDEtude, nom, age et sportPratique.

Exercice 3 (Surcharge, polymorphisme)
Considérez les classes Personne, Etudiant, Travailleur mentionnées ci-dessus. Pour chaque classe écrivez une méthode superieur qui compare un objet à un autre. Une personne est « supérieure » à une autre, si elle est plus âgée que l'autre. Un étudiant est « supérieur » à un autre, s'il étudie depuis plus longtemps. Un travailleur est « supérieur » à un autre, si son salaire est plus grand. Qu'est-ce qui se passe quand on compare un étudiant avec un travailleur ?

Exercice 4 (Constructeurs)
Les exemples suivants sont-ils corrects ? Justifiez.

```
1  class A{
2    private int a;
3    public A() {System.out.println(a);}
4    public A(int a) {this.a = a; this();}
5  }
```

Correction : Non : dans le constructeur à un paramètre l'appel au constructeur sans paramètre doit être effectué en premier.

```
1  class A{
2    int a;
3  }
4  class B extends A{
5    int b;
6    B(int a, int b) {this.a = a; this.b = b;}
7  }
```

Correction : Oui.

```
1  class A{
2    int a;
3    A(int a) {this.a = a;}
4  }
5  class B extends A{
6    int b;
7    B(int a,int b) {this.a = a; this.b = b;}
8  }
```

Correction : Non : En l'absence d'appel explicite à un constructeur de la super classe dans le constructeur de la classe B, un appel implicite au constructeur par défaut de la super classe est tenté ; or la définition d'un constructeur quelconque annihile l'existence du constructeur par défaut. La correction est donc d'appeler explicitement dans la sous-classe le constructeur de la super classe : **super**(a);.

```
1  class A{
2    private int a;
3    A() {this.a=0;}
4  }
5  class B extends A{
6    private int b;
7    B() {this.a=0; this.b=0;}
8  }
```

Correction : Non : La classe B n'a pas accès au champ a. Il faut utiliser le constructeur de la super classe **super**(); **this**.b=0;

Exercice 5 (Redéfinition)
Les exemples suivants sont-ils corrects ? Justifiez.

```
1  class A{
2    public void f() {System.out.println("Bonjour.");}
3  }
4  class B extends A{
5    private void f() {System.out.println("Bonjour les amis.");}
6  }
```

Correction : Non : on ne peut redéfinir une méthode en restreignant sa visibilité.

```
1  class A{
2    public int f(int a) {return a++;}
3  }
4  class B extends A{
5    public boolean f(int a) {return a==0;}
6  }
```

Correction : Non : on ne peut redéfinir une méthode et changer le type du résultat.

```
1  class A{
2    public int f(int a) {return a++;}
3  }
4  class B extends A{
5    public int f(int a, int b) {return a+b;}
6  }
7  class Test{
8    B obj = new B();
9    int x = obj.f(3);
10   int y = obj.f(3,3);
11 }
```

Correction : Oui. L'appel f(3) utilise la méthode de la super classe.

```
1  class A{
2    public int f(int a) {return a++;}
3  }
4  class B extends A{
5    public int f(int a, int b) {return a+b;}
6  }
7  class Test{
8    A obj = new B();
9    int x = obj.f(3);
10   int y = obj.f(3,3);
11 }
```

Correction : Non, le complilateur va réjecter le programme car obj a été declaré de la classe A pour le quel f est défini sur un seul argument.

Exercice 6 (Liaison dynamique - 1)
Qu'affiche le programme suivant ?

```
1  class A{
2    public String f() {return "A";}
3  }
4  class B extends A{
5    public String f() {return "B";}
6  }
7  class Test{
8    public static void main (String[] args){
9      A a1 = new A();
10     A a2 = new B();
11     B b = new B();
12     System.out.println(a1.f());
13     System.out.println(a2.f());
14     System.out.println(b.f());
15   }
16 }
```

Correction :

```
1 A
2 B
3 B
```

Le choix de la méthode est fait dynamiquement pendant l'exécution.

Exercice 7 (Liaison dynamique - 2)
Qu'affiche le programme suivant ?

```
1  class A{
2    public String f(B obj) {return "A et B";}
3    public String f(A obj) {return "A et A";}
4  }
5  class B extends A{
6    public String f(B obj) {return "B et B";}
7    public String f(A obj) {return "B et A";}
8  }
9  class Test{
10   public static void main (String[] args){
11     A a1 = new A();
12     A a2 = new B();
13     B b = new B();
14     System.out.println(a1.f(a1));
15     System.out.println(a1.f(a2));
16     System.out.println(a2.f(a1));
17     System.out.println(a2.f(a2));
18     System.out.println(a2.f(b));
19     System.out.println(b.f(a2));
20   }
21 }
```

Correction :

```
1 A et A
```

```
2 A et A
3 B et A
4 B et A
5 B et B
6 B et A
```

Le choix de la méthode est fait dynamiquement pendant l'exécution, tandis que le type de l'argument est déterminé à la compilation.

Exercice 8 (Polymorphisme)
Qu'affiche le programme suivant ?

```
1  class A{
2    public String f(D obj) {return "A et D";}
3    public String f(A obj) {return "A et A";}
4  }
5  class B extends A{
6    public String f(B obj) {return "B et B";}
7    public String f(A obj) {return "B et A";}
8  }
9  class C extends B{
10 }
11 class D extends B{
12 }
13 class Test{
14   public static void main (String[] args){
15     A a1 = new A();
16     A a2 = new B();
17     B b = new B();
18     C c = new C();
19     D d = new D();
20     System.out.println(a1.f(b));
21     System.out.println(a1.f(c));
22     System.out.println(a1.f(d));
23     System.out.println(a2.f(b));
24     System.out.println(a2.f(c));
25     System.out.println(a2.f(d));
26     System.out.println(b.f(b));
27     System.out.println(b.f(c));
28     System.out.println(b.f(d));
29   }
30 }
```

Correction :

```
1 A et A
2 A et A
3 A et D
4 B et A
5 B et A
6 A et D
7 B et B
8 B et B
9 A et D
```

Les premiers trois appels se comportent d'une façon polymorphe comme on s'y attend.

Les trois appels suivants semblent un peu « bizarres » cela depend du fait que la *signature* de la méthode est choisie à la compilation, tandis que le *code* à exécuter est choisi à l'exécution. À la compilation, l'objet a2 est supposé être de type A. Dans la classe A, la méthode f n'est pas définie pour un objet de type B. Ainsi, l'appel a2.f(b) utilise le polymorphisme et considère b comme un objet de type A. Donc, la signature pour cet appel exige un argument de type A. À l'exécution, a2 est un objet de type B. Le code utilisé est celui de la méthode de B qui a la même signature, même si B pourrait avoir une méthode plus « précise »L'appel a2.f(d) choisit la signature de f qui exige objets de type D. À l'exécution, a2 est de type B. La classe B n'a pas une méthode f avec la bonne signature. Donc le code de la superclasse est utilisé.

Parmi les trois derniers appels, le plus interessant est le dernier : B n'a pas une méthode pour D. Est-ce que D va etre vu comme un B, ou est-ce que la méthode de la superclasse A va être appelée ? En faite, l'héritage a la priorité sur le polymorphisme.

Exercice 9 (Champs de classe)
Qu'affiche le programme suivant ?

```
1  class A{
2    int i;
3    int f() {return i;}
4    static String g() {return "A";}
5    String h() {return g();}
6  }
7  class B extends A{
8    int i=2;
9    int f() {return -i;}
10   static String g() {return "B";}
11   String h() {return g();}
12 }
13 class Test{
14   public static void main(String[] args) {
15     B b = new B();
16     System.out.println(b.i);
17     System.out.println(b.f());
18     System.out.println(b.g());
19     System.out.println(b.h());
20     A a = b;
21     System.out.println(a.i);
22     System.out.println(a.f());
23     System.out.println(a.g());
24     System.out.println(a.h());
25   }
26 }
```

Correction :

```
1 2
2 -2
3 B
4 B
5 0
6 -2
7 A
8 B
```

La « surprise » est en ligne 5, où l'on s'attend à lire `2` (comportement des méthodes). En réalité, vu q'il n'y a pas de dynamisme pour l'accès aux champs, le type de l'objet est déterminé à la compilation (ici `A`). Donc c'est bien le champ de la « structure » `A` auquel on accède, alors même que l'objet est dynamiquement un `B`.

Exercice 10 Dans cet exercice, on veut écrire une classe EnsembleDEntiers fournissant des méthodes permettant d'ajouter ou enlever un entier de l'ensemble et d'afficher cet ensemble :

1. donnez la liste des prototypes des constructeurs et méthodes de la classe EnsembleDEntiers,

2. implémentez cette classe,
3. on veut maintenant étendre cette classe en EnsembleOrdonneDEntiers dans laquelle l'ensemble apparaîtra comme toujours ordonné (ses éléments seront visiblement ordonnés).

Correction :

```
1   class EnsembleDEntiers{
2     private int [] elements;
3     private int nElements;
4     private void retaille(){
5       if (nElements<elements.length) return;
6       int [] tmp = new int[elements.length+10];
7       for (int i=0; i<nElements; i++)
8         tmp[i] = elements[i];
9       elements = tmp;
10    }
11    public EnsembleDEntiers() {
12      elements = new int[10];
13      nElements = 0;
14    }
15    public EnsembleDEntiers(EnsembleDEntiers e) {
16      this();
17      for (int i=0; i<e.nombreDElements(); i++) ajoute(e.element(i));
18    }
19    public void ajoute(int element) {
20      retaille();
21      elements[nElements++] = element;
22    }
23    private int cherche(int element) {
24      for (int i=0; i<nombreDElements(); i++)
25      if (element(i)==element) return i;
26      return -1;
27    }
28    public int element(int indice) {
29      return elements[indice];
30    }
31    public void enleve(int element) {
32      int indice = cherche(element);
33      if (indice==-1) return;
34      for (int i=indice+1; i<nombreDElements(); i++)
35      elements[i-1] = elements[i];
36      nElements--;
37    }
38    public int nombreDElements() {
39      return nElements;
40    }
41    public String toString() {
42      StringBuffer tmp = new StringBuffer("{");
43        if (nombreDElements()>0) {
44          for (int i=0; i<nombreDElements()-1; i++)
45          tmp.append(element(i) + ",");
46          tmp.append(element(nombreDElements()-1));
47        }
48        tmp.append("}");
49      return tmp.toString();
```

```
50      }
51      /* protected : méthode visible uniquement de cette classe
52         et des classes qui l'"extend"ent */
53      protected void modifie(int indice,int valeur) {
54        elements[indice] = valeur;
55      }
56    }
57
58    class EnsembleOrdonneDEntiers extends EnsembleDEntiers {
59      private int cherchePremierPlusGrandOuEgal(int element) {
60        for (int i=0; i<nombreDElements()-1; i++)
61        if (element(i)>=element) return i;
62        return nombreDElements()-1;
63      }
64      public void ajoute(int element) {
65        super.ajoute(element);
66        int ppg = cherchePremierPlusGrandOuEgal(element);
67        int valeur = element(nombreDElements()-1);
68        for (int i=nombreDElements()-1; i>ppg; i--)
69        modifie(i,element(i-1));
70        modifie(ppg,valeur);
71      }
72    }
73
74    class EnsembleFaineantOrdonneDEntiers extends EnsembleDEntiers {
75      private boolean ordonne;
76      public void ajoute(int element) {
77        ordonne = false;
78        super.ajoute(element);
79      }
80      public String toString() {
81        if (!ordonne) trie();
82        return super.toString();
83      }
84      public int element(int indice) {
85        if (!ordonne) trie();
86        return super.element(indice);
87      }
88      private void trie() {
89        if (ordonne) return;
90        // tri quelconque du tableau (code à rajouter !)...
91        ordonne = true;
92      }
93    }
```