Programmation Orientée Objet avancée avec Java

21 septembre 2015

Chapitre 1: Interfaces – Classes abstraites

Dans ce document, la description des classes de l'API ne prétend aucunement être exhaustive. Reportez-vous à l'API en question pour connaître tous les détails de cette classe.

Chapitre I – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. Exemple
- III. L'interface : `Cloneable`
- IV. L'interface : `Comparable`
- V. L'interface : `Comparator`
- VI. Les classes abstraites

Chapitre I – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. Exemple
- III. L'interface : `Cloneable`
- IV. L'interface : `Comparable`
- V. L'interface : `Comparator`
- VI. Les classes abstraites

Chapitre I – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. Exemple
- III. L'interface : `Cloneable`
- IV. L'interface : `Comparable`
- V. L'interface : `Comparator`
- VI. Les classes abstraites

Chapitre I – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. Exemple
- III. L'interface : `Cloneable`
- IV. L'interface : `Comparable`
- V. L'interface : `Comparator`
- VI. Les classes abstraites

Chapitre I – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. Exemple
- III. L'interface : `Cloneable`
- IV. L'interface : `Comparable`
- V. L'interface : `Comparator`
- VI. Les classes abstraites

Chapitre I – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. Exemple
- III. L'interface : `Cloneable`
- IV. L'interface : `Comparable`
- V. L'interface : `Comparator`
- VI. Les classes abstraites

Définitions des Interfaces

Une interface est la description d'un ensemble de méthodes que les classes Java peuvent mettre en oeuvre. Par nature les interfaces sont abstraites, elles ne contiennent que des prototypes de méthodes et/ou des constantes (`final static`).
Une classe **implémente** une interface : chaque méthode de l'interface est implémentée dans la classe.
Cela peut être vu comme un contrat entre la classe et l'interface.
Une interface est définie au même niveau qu'une classe : elle contient **uniquement**

- des définitions de constantes (`final static`)
- des déclarations de méthodes (prototype uniquement).

Syntaxe :

```
interface MonInterface

class MaClasse implements MonInterface
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Remarques

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)
- on peut déclarer une variable avec comme type une interface
- si `MonInterface` est une interface implémentée par trois classes C1, C2, C3 alors les instructions suivantes sont valides

```
MonInterface v1,v2,v3;
v1=new C1();
v2=new C2();
v3=new C3();
```

Exemple d'usage

Supposons que l'on a défini trois classes `Film`, `LongMetrage`, `Documentaire`.
Dans une *autre classe* trois méthodes sont écrites avec un argument de chacune de ces classes : `int duree(Film a); String aLAffecheDe(LongMetrage b); String sujet(Documentaire c);`
Comment faire pour passer à `duree` un argument de type `Film` ou de type `Documentaire` ?
Une solution consiste

- à créer trois interfaces `Film`, `LongMetrage`, `Documentaire`
- à écrire des classes implémentant respectivement `Film`, `LongMetrage`, `Documentaire`, `Film` et `Documentaire`, ...

Si la classe `UnLongMetrageDocumentaire` implémente `LongMetrage et Documentaire` alors une instance de `UnLongMetrageDocumentaire` pourra être l'argument de `duree` et l'argument de `sujet`.

L'interface : `Cloneable`

`public Object clone() throws CloneNotSupportedException` est une méthode d'instance de la classe `Object` ; elle est conçue pour effectuer une opération de clonage (duplication) :

- `clone()` lance l'exception `CloneNotSupportedException` lorsque la classe de l'objet n'implémente pas l'interface `Cloneable`
- sinon une *nouvelle instance* de la classe `Object` est créée avec les attributs initialisés avec ceux de l'objet cloné

La méthode `clone()` de la classe `Object` duplique tous les attributs d'une classe et renvoie une instance `Object`.

Par exemple, si une classe A contient :

- un attribut entier n
- un attribut t référençant un tableau,

la méthode `clone` de la classe `Object` appliquée à une instance a de A construit une nouvelle instance a2 de `Object` avec :

- l'entier n qui a, au moment de la construction de la copie, la même valeur que l'attribut n de l'instance a ; si on change la valeur de n dans la copie, on ne change pas la valeur de n dans l'original ;
- l'attribut t qui référence le même tableau l'attribut t de l'instance a

Exemple

```
class EssaiClone implements Cloneable{
int n;
int[] t = {1,2,3};

public Object clone() throws CloneNotSupportedException {
return super.clone();}

public static void main(String[] arg)
    throws CloneNotSupportedException{
                EssaiClone o= new EssaiClone();
                o.n=0;
                EssaiClone oc= (EssaiClone)o.clone();
                System.out.println(oc.n+" "+ oc.t);
        oc.n = 1;
        System.out.println("original="+o.n + " "
                           + o.t+ "  copie= "+oc.n+" "+ oc.t);
}}
```

Exécution :

0 [I@eb42cbf

original=0 [I@eb42cbf copie= 1 [I@eb42cbf

Exemple

```
class EssaiClone implements Cloneable{
int n;
int[] t = {1,2,3};

public Object clone() throws CloneNotSupportedException {
return super.clone();}

public static void main(String[] arg)
    throws CloneNotSupportedException{
                EssaiClone o= new EssaiClone();
                o.n=0;
                EssaiClone oc= (EssaiClone)o.clone();
                System.out.println(oc.n+" "+ oc.t);
        oc.n = 1;
        System.out.println("original="+o.n + " "
                            + o.t+ "  copie= "+oc.n+" "+ oc.t);
}}
```

Exécution :
0 [I@eb42cbf
original=0 [I@eb42cbf copie= 1 [I@eb42cbf

copie de surface/en profondeur

La méthode `clone()` réalise une copie de surface (shallow copy) : les références des attributs de type non primitifs sont copiés.
Pour réaliser une copie en profondeur (deep copy), on doit :

- récupérer l'objet à renvoyer en appelant la méthode super.clone() (copie de surface),
- cloner les attributs non immuables afin de passer d'une copie de surface à une copie en profondeur de l'objet.

Exemple

```
class EssaiCloneProfondeur implements Cloneable{
int n;
int[] t = {1,2,3};

public Object clone() throws CloneNotSupportedException {
EssaiCloneProfondeur o =(EssaiCloneProfondeur) super.clone();// cast
 o.t = new int[this.t.length];
for (int k=0;k<this.t.length;k++) o.t[k]=this.t[k];
return o;}

public static void main(String[] arg)
                    throws CloneNotSupportedException{
EssaiCloneProfondeur o= new EssaiCloneProfondeur();
o.n=0;
EssaiCloneProfondeur oc= (EssaiCloneProfondeur)o.clone();
oc.t[0]=12;
System.out.println("oc.n="+oc.n+"o.n="+o.n+"  oc[0]="+
        oc.t[0]+"   o[0] ="+ o.t[0]);
}}
```

Remarque : on peut changer le prototype de `clone()` :

```
public EssaiCloneProfondeur clone() ...
```

requis de la méthode clone()

- x.clone() != x ; doit renvoyer `true`
- x.clone().getClass() == x.getClass() ; doit renvoyer `true`
- x.clone().equals(x) ; doit renvoyer `true`

requis de la méthode clone()

- x.clone() != x ; doit renvoyer `true`
- x.clone().getClass() == x.getClass() ; doit renvoyer `true`
- x.clone().equals(x) ; doit renvoyer `true`

requis de la méthode clone()

- x.clone() != x ; doit renvoyer `true`
- x.clone().getClass() == x.getClass() ; doit renvoyer `true`
- x.clone().equals(x) ; doit renvoyer `true`

L'interface : `Comparable`

L'interface `Comparable` du paquetage `java.lang` permet de définir une méthode de comparaison sur toute classe d'objets que l'on peut ordonner selon un ordre total (deux objets quelconques sont toujours comparables) et de façon transitive (si un objet a est avant un objet b lui-même avant un objet c alors l'objet a est avant l'objet c).
Cela permet d'utiliser les méthodes de tri standard en Java.
L'interface java.lang.Comparable est définie ainsi

```
public abstract interface Comparable {public int compareTo (Object obj);}
```

Cette méthode renvoie 0 en cas « d'égalité » -1 si l'objet considéré est avant le paramètre obj et $+1$ sinon.

Exemple

```
class EssaiCloneCompProfondeur implements Cloneable, Comparable{
int n;
int[] t = {1,2,3};
// voir EssaiCloneProfondeur
int compTableau(int[] tab) {
int lThis = this.t.length;
int lTab = tab.length;
int l;
if(lThis<lTab) l=lThis; else l=lTab;
for (int k=0;k<l;k++) {if (this.t[k]<tab[k]) return -1;
else {if (tab[k]<this.t[k]) return 1; };}
return 0;
}
public int compareTo (Object obj){
if (((EssaiCloneCompProfondeur)obj).n< this.n) return 1;
else {if (((EssaiCloneCompProfondeur)obj).n> this.n) return -1;
else return this.compTableau(((EssaiCloneCompProfondeur) obj).t);}
}}
// equals doit être compatible avec return 0 de compareTo
```

Application : pour trier un tableau de `EssaiCloneCompProfondeur` on utilise la méthode statique sort qui se trouve dans la classe java.util.Arrays de prototype **public static void** sort(Object[] tableau).

L'interface : `Comparator< >`

Dans l'exemple précédent le principe de comparaison prenanit en compte tout d'abord la valeur de l'attribut `n` puis les valeurs contenues dans l'attribut du `t`.
On pourrait avoir besoin d'un autre ordre mais on ne peut pas avoir plusieurs versions de `compareTo`.
L'interface `Comparator< >` nous permet de définir des ordres variés et de les coupler avec les méthodes de la classe `Collections`.
Elle est générique et contient deux méthodes `public int compare(T o1, T o2); public boolean equals(Object obj);`
Elle sera utilisée dans une classe externe qui définira un ordre particulier sur les objets à comparer.

Exemple

```
public class EssaiComparableComparator implements Comparable{
int n;
int[] t ;
public EssaiComparableComparator(int a, int[] tab){n=a;t=tab;}
int compTableau(int[] tab) {
int lThis = this.t.length;
int lTab = tab.length;
int l;
if(lThis<lTab) l=lThis; else l=lTab;
for (int k=0;k<l;k++) {if (this.t[k]<tab[k]) return -1;
else {if (tab[k]<this.t[k]) return 1; };}
return 0;
}
public int compareTo (Object obj){
if (((EssaiComparableComparator)obj).n< this.n) return 1;
else {if (((EssaiComparableComparator)obj).n> this.n) return -1;
else return this.compTableau(((EssaiComparableComparator) obj).t);}
}}
public class TabComparator implements Comparator<EssaiComparableComparator>{
public int compare(EssaiComparableComparator o1, EssaiComparableComparator o2){
if(o1.t.length<o2.t.length) return -1;
else if(o1.t.length>o2.t.length) return 1;
else return 0;}} // on compare uniquement les longueurs
public class TabSommeComparator implements Comparator<EssaiComparableComparator>{
public int compare(EssaiComparableComparator o1, EssaiComparableComparator o2){
int s1=0; int s2=0;
for(int i=0;i<o1.t.length;i++) s1=s1+o1.t[i];
for(int j=0;j<o2.t.length;j++) s2=s2+o2.t[j];
if(s1<s2) return -1; else if(s2<s1) return 1; else return 0;}}
// on compare uniquement la somme des éléments
```

Exemple

```
class Test{
    public static void main(String[] args) {
        TabComparator tComparator = new TabComparator();
        TabSommeComparator tSomComp = new TabSommeComparator();
        int[] t1 ={1,2,3,4,5};
        int[] t2 = {7,8,9};
        EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
        EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
        if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
        if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
                }
```

L'exécution donne :

e2

e1

Remarque : pour trier une collection (cf. chapitre suivant) on peut passer en 2ème argument des méthodes `sort`, `min`, `max` ... une classe qui implémente `Comparator`.

Par exemple `Collections.sort(c, tComparator)` où c serait une collection (cf chapitre suivant) d'instances de la classe `EssaiComparableComparator`.

Exemple

```
class Test{
    public static void main(String[] args) {
        TabComparator tComparator = new TabComparator();
        TabSommeComparator tSomComp = new TabSommeComparator();
        int[] t1 ={1,2,3,4,5};
        int[] t2 = {7,8,9};
        EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
        EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
        if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
        if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
                }
```

L'exécution donne :

e2

e1

Remarque : pour trier une collection (cf. chapitre suivant) on peut passer en 2ème argument des méthodes `sort`, `min`, `max` ... une classe qui implémente `Comparator`.

Par exemple `Collections.sort(c, tComparator)` où c serait une collection (cf chapitre suivant) d'instances de la classe `EssaiComparableComparator`.

Exemple

```
class Test{
    public static void main(String[] args) {
        TabComparator tComparator = new TabComparator();
        TabSommeComparator tSomComp = new TabSommeComparator();
        int[] t1 ={1,2,3,4,5};
        int[] t2 = {7,8,9};
        EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
        EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
        if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
        if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
                }
```

L'exécution donne :

e2

e1

Remarque : pour trier une collection (cf. chapitre suivant) on peut passer en 2ème argument des méthodes `sort`, `min`, `max` ... une classe qui implémente `Comparator`.

Par exemple `Collections.sort(c, tComparator)` où c serait une collection (cf chapitre suivant) d'instances de la classe `EssaiComparableComparator`.

Exemple

```
class Test{
    public static void main(String[] args) {
        TabComparator tComparator = new TabComparator();
        TabSommeComparator tSomComp = new TabSommeComparator();
        int[] t1 ={1,2,3,4,5};
        int[] t2 = {7,8,9};
        EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
        EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
        if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
        if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println("e2");
                }
```

L'exécution donne :

e2

e1

Remarque : pour trier une collection (cf. chapitre suivant) on peut passer en 2ème argument des méthodes `sort`, `min`, `max` ... une classe qui implémente `Comparator`.

Par exemple `Collections.sort(c, tComparator)` où c serait une collection (cf chapitre suivant) d'instances de la classe `EssaiComparableComparator`.

Les classes abstraites

Une classe est abstraite si

- elle est marquée par le modificateur `abstract`
- elle contient des (au moins 1) méthodes abstraites.

Une méthode abstraite

- est marquée par le modificateur `abstract`
- se déclare seulement par son prototype.

Elle n'est pas instanciable.
Une classe est abstraite peut être étendue par une classe qui devra alors définir *toutes* les méthodes abstraites héritées pour pouvoir, elle, être instanciée.
Cette notion est utile pour factoriser du code et laisser des méthodes abstraites qui peuvent être implémentées dans des sous-classes.

Exemple

On définit une classe abstraite `Quadrilatère` avec

- 4 attributs pour les longueurs des 4 côtés,
- une méthode d'instance `périmètre` renvoyant le périmètre d'un objet
- une méthode abstraite `surface` qui devra renvoyer la surface d'un objet.

.

Puis on définira des sous-classes Trapèze, Rectangle qui implémenteront la méthode `surface` différement mais qui pourront utiliser la méthode `périmètre` de leur super classe.

```
public abstract class Quadrilatere {
double a,b,c,d;
public abstract double surface();// prototype
public  double perimetre(){return (this.a+this.b+this.c+this.d)};}
public class Trapèze extends Quadrilatere {
double h;
public Trapèze(double x, double petiteBase, double, z, double grandeBase, double haut)
{a=x;b=petiteBase;c=z; d= grandeBase; h=haut;}
public  double surface(){return (h*(b+d)/2);};
}
public class Rectangle extends Quadrilatere {
public Rectangle(double larg, double longueur){a=longueur; b=larg;c=a;d=b}
public  double surface(){return (a*b)};
}
```