Programmation Orientée Objet avancée avec Java

28 octobre 2015

Chapitre 1: Interfaces – Classes abstraites
Chapitre 2: Collections - Généricité
Chapitre 3: Thread
Chapitre 4: Programmation événementielle
Chapitre 5: Lien avec une base de données

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.

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

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

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

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

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

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
```

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

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 v;
v=new C1();
v=new C2();
v=new C3();
```

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

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

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

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 v;
v=new C1();
v=new C2();
v=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

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 v;
v=new C1();
v=new C2();
v=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 v;
v=new C1();
v=new C2();
v=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 v;
v=new C1();
v=new C2();
v=new C3();
```

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

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 aLAfficheDe(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`.

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

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

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

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`

I. Définitions des Interfaces

II. Exemple

III. L'interface : Cloneable

IV. L'interface : Comparable

V. L'interface : Comparator

VI. Les classes abstraites

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`

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

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`

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

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.

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

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)`.

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

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.

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

Exemple

```java
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;i<o1.t.length;j++) s1=s1+o1.t[j];
if(s1<s2) return -1; else if(s2<s1) return 1; else return 0;}} // on compare uniquement la somme
```

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 e1 = EssaiComparableComparator(37, t2);
            if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.print
            if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println(
            }
```

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`.

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

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 e1 = EssaiComparableComparator(37, t2);
            if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.print
            if(tSomComp.compare(e1, e2)<0) System.out.println("e1"); else System.out.println(
            }
```

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 e1 = EssaiComparableComparator(37, t2);
            if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.print
            if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println(
            }
```

L'exécution donne :

e2

e1

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 e1 = EssaiComparableComparator(37, t2);
            if(tComparator.compare(e1,e2)<0) System.out.println("e1"); else System.out.print
            if(tSomComp.compare(e1,e2)<0) System.out.println("e1"); else System.out.println(
            }
```

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`.

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

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)};
}
```

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Chapitre II – Collections - Généricité

- I. Généricité
- II. Collections
- III. Interface Collection
- IV. Les méthodes de l'interface Collection
- V. La classe Collections
- VI. Itérateurs
- VII. Classe ArrayList<T>
- VIII. La classe HashSet<T>

I. Généricité

II. Collections

III. Interface Collection

IV. Les méthodes de l'interface Collection

V. La classe Collections

VI. Itérateurs

VII. Classe ArrayList<T>

VIII. La classe HashSet<T>

IX. La classe TreeSet<T>

X. Interface Map

Généricité

Depuis la version 5.0 Java autorise la définition de classes et d'interfaces contenant un (des) paramètre(s) représentant un *type(s)*. Cela permet de décrire une structure qui pourra être personnalisée au moment de l'*instanciation* à tout type d'objet.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

On veut définir une notion de paire d'objets avec deux attributs de même type.

```
public class PaireEntier {
private int premier;
private int second;
public PaireEntier(int x, int y){
premier =x ; second = y;}
public int getPremier(){return this.premier;}
public void setPremier(int x){this.premier=x;}
public int getSecond(){return this.second;}
public void setSecond(int x){this.second=x;}
public void interchanger(){
int temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

remarque : on a créé une classe spécialement pour des paires d'entiers ; si on veut des paires de booléens il faudrait réécrire une autre classe (avec un autre nom) qui contiendrait les mêmes méthodes.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

```
public class PaireObjet {
private Object premier;
private Object second;
public PaireObjet(Object x, Object y){
premier =x ; second = y;}
public Paire(){}
public Object getPremier(){return this.premier;}
public void setPremier(Object x){this.premier=x;}
public Object getSecond(){return this.second;}
public void setSecond(Object y){this.second=y;}
public void interchanger(){
Object temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

Inconvénients :

- les deux attributs peuvent des instances de classes différentes,
- on peut être amené à faire du transtypage (vers le bas),
- on risque des erreurs de transtypage qui ne se *détecteront qu'à l'exécution.*

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

```
public class PaireObjet {
private Object premier;
private Object second;
public PaireObjet(Object x, Object y){
premier =x ; second = y;}
public Paire(){}
public Object getPremier(){return this.premier;}
public void setPremier(Object x){this.premier=x;}
public Object getSecond(){return this.second;}
public void setSecond(Object y){this.second=y;}
public void interchanger(){
Object temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

Inconvénients :

- les deux attributs peuvent des instances de classes différentes,
- on peut être amené à faire du transtypage (vers le bas),
- on risque des erreurs de transtypage qui ne se *détecteront qu'à l'exécution*.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

A et B étant deux classes, on peut avoir ce genre d'utilisation

```
A a = new A();
B b = new B();
PaireObjet p = new PaireObjet(a,b);
A a2 = (A) p.getPremier(); // downcasting ok
p.interchanger();
A a2 = (A) p.getPremier(); // erreur
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

La solution est l'utilisation de la *généricité*, c'est-à-dire l'usage de *type paramètre*.
La généricité est une notion de *polymorphisme paramétrique*.

```
public class Paire<T> {
private T premier;
private T second;
public Paire(T x, T y){ // en-tête du constructeur sans <T>
premier =x ; second = y;}
public Paire(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public T getSecond(){return this.second;}
public void setSecond(T y){this.second=y;}
public void interchanger(){
T temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

Cette définition permet de définir ici des `Paire` contenant des objets de type (uniforme) mais arbitraire.

utilisation

- une classe générique doit être instanciée pour être utilisée
- on ne peut pas utiliser un type primitif pour l'instanciation, il faut utiliser les classes enveloppantes
- on ne peut pas instancier avec un type générique
- une classe instanciée ne peut pas servir de type de base pour un tableau

```
Paire<String> p = new Paire<String>("bonjour"," Monsieur"); // oui
// le constructeur doit contenir <...> pour l'instanciation
Paire<>p2 = new Paire<>();  // non
Paire<int> p3 = new Paire<int>(1, 2); // non
Paire<Integer> p4 = new Paire<Integer>(1,2);// oui
Paire<Paire> p5 = new Paire<Paire>(); // non
Paire<Paire<String>> p6 = new Paire<Paire<String>>(p,p); // oui
Paire<Integer>[] tab = new Paire<Integer>[10]; // non
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

plusieurs types paramètres

On peut utiliser plusieurs types paramètres

```
public class PaireD<T,U> {
private T premier;
private U second;
public PaireD(T x, U y){ // en-tête du constructeur sans <T,U>
premier =x ; second = y;}
public PaireD(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public U getSecond(){return this.second;}
public void setSecond(U y){this.second=y;}
}
...
PaireD<Integer, String> p = PaireD<Integer, String>(1, "bonjour");
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Utilisation du type paramètre

- le type paramètre peut être utilisé pour *déclarer* des variables (attributs) sauf dans une méthode de classe
- le type paramètre ne peut pas servir à *construire* un objet.

```
public class Paire<T> {
...
T var ; // oui
T var = new T(); //non
T[] tab ; // oui
T[] tab = new T[10];// non
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

méthodes et généricité

Une méthode de classe (`static`) ne peut pas utiliser une variable du type paramètre dans une classe générique.

```
public class UneClasseGenerique<T>{
...
public static void methodeDeClasse(){
        T var ; // erreur à la compilation
...
}}
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

méthodes et généricité

Une méthode (de classe ou d'instance) peut être générique dans une classe non générique. Elle utilise alors son propre type paramètre.

```
public class ClasseA{
...
public <T> T premierElement(T[] tab){
return tab[0];} // méthode d'instance
//<T> est placé après les modificateurs et avant le type renvoyé
public static <T> T dernierElement(T[] tab){
return tab[tab.length-1];} // méthode de classe
//<T> est placé après les modificateurs et avant le type renvoyé
...
}
```

Pour utiliser une telle méthode on doit préfixer le nom de la méthode par le type d'instanciation entre $<$ et $>$.

```
ClasseA a = new ClasseA();
String[] t = {"game", "of", "thrones"};
System.out.println(a.<String> premierElement(t));
System.out.println(ClasseA.<String> dernierElement(t));
```

méthodes et généricité

Une méthode (de classe ou d'instance) peut être générique dans une classe générique. Elle peut utiliser le type paramètre de la classe et son propre type paramètre.

```
public class Paire<T> {
private T premier;
private T second;
public Paire(T x, T y){ // en-tête du constructeur sans <T>
premier =x ; second = y;}
public Paire(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public T getSecond(){return this.second;}
public void setSecond(T y){this.second=y;}
public void interchanger(){
T temp = this.premier;
this.premier = this.second;
this.second = temp;}
public <U> void  voir(U var){
System.out.println("qui est là ?" + var);
System.out.println("le premier est " + this.premier);}
}
...
Paire<Integer> p = new Paire<Integer >(1,2);
p .<String> voir("un ami");
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

exercices

exercice 1 : Réécrire les méthodes `equals` et `toString` pour les deux classes `Paire` et `PaireD`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Limitation du type paramètre

Instancier une classe générique à un type quelconque peut empêcher d'écrire certaines méthodes.
Par exemple pour la classe `Paire`, on voudrait connaître le plus grands des 2 attributs : cela n'a de sens que si l'instanciation se fait avec un type dont les objets sont comparables donc qui implémente l'interface `Comparable` avec sa méthode `compareTo`.
Java permet de préciser que le type paramètre doit être ainsi :

```
public class Paire<T extends Comparable> { ...}
```

On peut limiter le type paramètre T par plusieurs interfaces et une classe au plus.

```
public class Paire<T extends Comparable & Cloneable & UneAutreClasse> { ...}
```

`Comparable` et `Cloneable` sont des interfaces et `UneAutreClasse` est une classe.
A l'instanciation le type choisi pour T devra implémenter les 2 interfaces et être une sous-classe de `UneAutreClasse`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Généricité et héritage

- une classe générique peut étendre une classe (générique ou pas)

```
public class Triplet<T> extends Paire<T>{
T troisieme;
...}
```

- Triplet$< T >$ est une sous classe de Paire$< T >$
- Triplet$< String >$ est une sous classe de Paire$< String >$
- Triplet$< String >$ n'est pas une sous classe de Paire$< T >$
- Triplet$< String >$ n'est pas une sous classe de Paire$< Object >$ bien que `String` soit une sous classe de `Object`
- Triplet$< String >$ n'est pas une sous classe de Triplet$< Object >$ bien que `String` soit une sous classe de `Object`

Ce dernier point interdit donc une affectation du genre

```
Triplet<Integer> t = new Triplet<Short>();
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Collections

Java propose plusieurs moyens de manipuler des ensembles d'objets : on a vu les tableaux dont l'inconvénient est de ne pas être dynamique vis à vis de leur taille.
Java fournit des interfaces qui permettent de gérer des ensembles d'objets dans des structures qui peuvent être parcourues.
Ce chapitre donne un aperçu de ces collections. Elles sont toutes génériques.
Toutes les collections d'objets

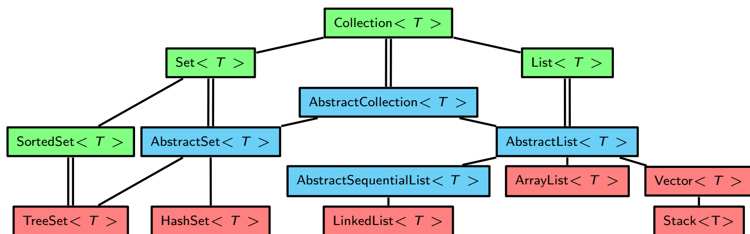
- sont dans le paquetage *java.util*
- implémentent l'interface générique *Collection*

L'interface Set$< T >$ sert à implémenter les collections de type ensemble : les éléments n'y figurent qu'une fois et ne sont pas ordonnés.
L'interface List$< T >$ sert à implémenter les collections dont les éléments sont ordonnées et qui autorisent la répétition.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Interface Collection

les interfaces sont en vert, les classes abstraites en bleu et les classes en rouge, et T est le type paramètre des éléments des collections ; les lignes simples indiquent l'héritage et les lignes doubles l'implémentation. (Le schéma est partiel il existe d'autres classes).



Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

La classe Collections

La classe `java.util.Collections` (notez le pluriel) contient des méthodes *statiques* qui opérent sur des objets `List` ou autre (`Set`, `Map` ...) ou bien renvoie des objets.

- `void sort(List list)` trie le paramètre list
- `void sort(List list, reverseOrder() )` trie le paramètre list en ordre décroissant
- `Object max(Collection coll)` renvoie le plus grand objet
- `Object min(Collection coll)` renvoie le plus petit objet
- ...

On peut utiliser ces méthodes statiques sur des objets de toutes les classes du schéma précédent.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Itérateurs

Un iterateur est un objet utilisé avec une collection pour fournir un accès séquentiel aux éléments de cette collection.
L'interface `Iterator` permet de fixer le comportement d'un itérateur.

- `boolean hasNext()` indique s'il reste au moins un élément à parcourir dans la collection
- `T next()` renvoie le prochain élément dans la collection
- `void remove()` supprime le dernier élément parcouru (celui renvoyé par le dernier appel à la méthode next())

La méthode `next()` lève une exception de type `NoSuchElementException` si elle est appelée alors que la fin du parcours des éléments est atteinte.
La méthode `remove()` lève une exception de type `IllegalStateException` si l'appel ne correspond à aucun appel à `next()`. Cette méthode est optionnelle (exception `UnsupportedOperationException`).
On ne peut pas faire deux appels consécutifs à `remove()`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Remarques

- A sa construction un itérateur doit être lié à une collection.
- A sa construction un itérateur se place tout au début de la collection.
- On ne peut pas « réinitialiser » un itérateur ; pour parcourir de nouveau la collection il faut créer un nouvel itérateur.
- Java utilise un itérateur pour implémenter la boucle `for each` de syntaxe suivante

```
Collection<T> c = new ... ;
for (T element : c) {...} // pour chaque objet element de type T de ma collaction c faire
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

méthode `toString()`

Pour tout objet de type `Collection`, la méthode `print` (ou `println`) appelle itérativement la méthode `toString()` de chacun de ses éléments.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Interface ListIterator

L'interface `ListIterator<T>` étend l'interface `Iterator<T>` et permet de parcourir la collection dans les deux sens.

- `T previous()` renvoie l'élément précédent dans la collection
- `boolean hasPrevious()` teste l'existence d'un élément précédent
- `T next()` renvoie l'élément suivant de la liste
- `T previous()` renvoie l'élément précédent de la liste
- `int nextIndex()` renvoie l'indice de l'élément qui sera renvoyé au prochain appel de `next()`
- `int previousIndex()` renvoie l'indice de l'élément qui sera renvoyé au prochain appel de `previous()`
- `void add(T e)` ajoute l'élément e à la liste à l'endroit du curseur (*i.e.* juste avant l'élément retourné par l'appel suivant à `next()`)
- `void remove()` supprime le dernier élément retourné par `next()` ou `previous()`
- `void set(T e)` remplace le dernier élément retourné par `next()` ou `previous()` par e

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Interface ListIterator

remarques :

- `next()` et `previous()` lèvent une exception de type `NoSuchElementException`
- si l'itérateur est en fin de liste alors `nextIndex()` renvoie la taille de la liste
- si l'itérateur est au début de la liste alors `nextIndex()` renvoie `-1`
- `add()`, `remove()` et `set(T e)` lèvent une exception de type `IllegalStateException` si l'appel ne correspond à aucun appel à `next()` ou `previous()`. Elles sont toutes les trois optionnelles.
- `set(T e)` lève une exception de type `ClassCastException` si le type de e ne convient pas.
- dans toutes les classes prédéfinies implémentant `Iterator` ou `ListIterator`, les méthodes `next()` et `previous()` renvoient les *références* des objets de la collection.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Classe ArrayList<T>

Un `ArrayList` est un tableau d'objets dont la taille est dynamique. La classe `ArrayList<T>` implémente en particulier les interfaces `Iterator`, `ListIterator` et `List`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Constructeurs

- `public ArrayList(int initialCapacite)` crée un `arrayList` vide avec la capacité `initialCapacite` (positif)
- `public ArrayList()` crée un `arrayList` vide avec la capacité 10
- `public ArrayList(Collection<? extends T> c)` crée un `arrayList` contenant tous les éléments de la collection c dans le même ordre avec une dimension correspondant à la taille réelle de c et non sa capacité ; le `arrayList` créé contient les références aux éléments de c (copie de surface).

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Méthodes

- `add` et `addAll` ajoute à la fin du tableau
- `void add(int index, T element)` ajoute au tableau le paramètre element à l'indice index en décalant d'un rang vers la droite les éléments du tableau d'indice supérieur
- `void ensureCapacity(int k)` permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
- `T get(int index)` renvoie l'élément du tableau dont la position est précisée
- `T set(int index, T element)` renvoie l'élément à la position index et remplace sa valeur par celle du paramètre element

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Méthodes

- `int indexOf(Object o)` renvoie la position de la première occurrence de l'élément fourni en paramètre
- `int lastIndexOf(Object o)` renvoie la position de la dernière occurrence de l'élément fourni en paramètre
- `T remove(int index)` renvoie l'élément du tableau à l'indice index et le supprime décalant d'un rang vers la gauche les éléments d'indice supérieur
- `void removeRange(int j,int k)` supprime tous les éléments du tableau de la position j incluse jusqu'à la position k exclue
- `void trimToSize()` ajuste la capacité du tableau sur sa taille actuelle

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

On veut gérer un ensemble de personnes connaissant leur age, poids et taille par ordre de risque décroissant de problème cardiaque compte tenu de ces données.
On définit

- la classe `Personne` (nom, prenom)
- la classe `PersonneMedicalise` (étend `Personne` avec age, taille, poids, implémente l'interface `Comparable`)
- la classe `EnsPersonneMedicale` qui utilise `ArrayList`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

```
public class PersonneMedicalise extends Personne implements Comparable {
....
public int compareTo(Object p){
if (this.getAge()> ((PersonneMedicalise)p).getAge()) return −1; else
if (this.getAge()< ((PersonneMedicalise)p).getAge()) return 1; else
if (this.getPoids()> ((PersonneMedicalise)p).getPoids()) return −1; else
if (this.getPoids()< ((PersonneMedicalise)p).getPoids()) return 1; else
return 0;
}}

import java.util.*;
public class EnsPersonneMedicale {
ArrayList <PersonneMedicalise> e;
public EnsPersonneMedicale() {}
...
public PersonneMedicalise quiEstEnDanger(){
Collections.sort(e);
return(e.get(0));}
public int ageMoyen(){
Iterator <PersonneMedicalise> it = e.iterator();
int a=0;
while (it.hasNext()) a= a+ it.next().getAge();
if (e.size()>0) return (a/e.size());else return 0;}}
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exercices

exercice 2 : appliquer le crible d'Eratosthène aux cent premiers entiers puis afficher tous les nombres premiers inférieurs à 100 en utilisant la classe ArrayList et un itérateur.

exercice 3 : Écrire un programme qui accepte, sur la ligne de commande, une suite de nombres et qui stocke dans un ArrayList ceux qui sont positifs.

exercice 4 : Écrire un programme qui accepte, sur la ligne de commande, une suite de chaînes de caractères et qui stocke dans un ArrayList celles qui contiennent au moins une fois le caractère 'a'. Faire afficher à l'écran toutes les chaînes ainsi stockées dans la structure ArrayList.
Ecrire une méthode qui classe le ArrayList par ordre de **longueur de chaînes croissantes** puis de nouveau faire afficher les chaînes dans cet ordre.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exercices

exercice 5 : la princesse Eve a de nombreux prétendants ; elle décide alors de choisir celui qu'elle épousera de la façon suivante :

- les prétendants sont numérotés de 1 à n
- en partant du numéro 1 elle compte par numéro croissant 3 prétendants et élimine le troisième
- elle réitère le procédé en partant du prétendant suivant le dernier éliminé
- lorsque la fin de la liste est atteinte elle compte en recommençant au premier de la liste
- lorsque le début de la liste est atteint elle compte par numéro croissant

Ecrire un programme qui affichera le prétendant restant pour une valeur n quelconque saisie au clavier

`HashSet<T>`

La classe `HashSet<T>` implémente l'interface `Set<T>` et l'interface `Iterator<T>`.
Elle permet de représenter un ensemble ; les éléments ne sont pas ordonnés par leur ordre d'insertion et chaque élément sera en un seul exemplaire : par conséquent la méthode `boolean add(T e)` n'ajoutera pas l'élément e s'il est déjà contenu dans le `HashSet`. Pour cela l'implémentation d'un `HashSet` s'appuie sur une *table de hachage* et sur les méthodes `equals` et `hashCode` de `T`.

Définition

Une *table de hachage* est un tableau indexé par des entiers (en général) contenant les couples *clef-valeur*. L'indexation est réalisée par une *fonction de hachage* qui associe un indice du tableau à chaque clef.

Cette fonction doit être idéalement *injective* pour éviter une *collision* (deux clefs différentes ont le même indice). Elle doit de plus assurer une bonne dispersion des valeurs dans le tableau.

Exemple

Par exemple en Java il existe une fonction de hachage standard des chaînes de caractères :

$$h(s) = \sum_{i=0}^{i=n-1} c(s[i]) \times 31^{n-i-1}$$

où s est de type String, et n est la longueur de s et $c(s[i])$ est le code ASCII du $i+1$ ème caractère de s.
On a par exemple, $h(''toto'') = 3566134$.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

hashCode()

En Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Dans certaines architectures, l'espace d'adressage est beaucoup plus grand que l'intervalle de valeur du type `int` ; il est donc possible que deux objets distincts aient le même `hashCode`.

Donc on constate en pratique que cette fonction de hachage n'est pas injective.

Dans le cas où l'on réécrit la méthode `hashCode()`, on peut toujours revenir à la valeur initiale du `hashCode` de la classe `Object` en utilisant la méthode statique `System.identityHashCode(Object o)`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Difficultés

Donc deux objets distincts peuvent avoir le même `hashCode` ; et de surcroît dans une classe ayant redéfini la méthode `equals` mais pas la méthode `hashCode()`, deux instances peuvent être égales (selon la redéfinition de `equals`) alors que leur `hashCode` sont différents. Cela peut conduire un hashSet a accepté d'ajouter un élément qu'il possède déjà.
En effet puisqu'un `HashSet` doit vérifier si un objet e lui appartient pour exécuter la méthode `add(e)` l'implémentation du `HashSet` va

- calculer `hashCode(e)` modulo la taille de la table et
- à l'indice correspondant à ce calcul vérifier qu'il ne possède pas d'élément `e2` tel que `e.equals(e2)` est vrai.

C'est pourquoi il est indispensable que deux éléments égaux aient le même `hashCode`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```java
public class TestElement{
String nom;
public TestElement(String s){nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
}
public class TestHashSet{
HashSet<TestElement> hs = new HashSet<TestElement>();
void ajouter(String s){hs.add(new TestElement(s));}
void ajouter(TestElement t){ hs.add(t);}
}
public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);}
}
```

Exécution :

```
oui
[TestElement@e76cbf7, TestElement@22998b08]
```

Donc les 2 éléments qui sont égaux au sens de la méthode `equals` de `TestElement` ont été ajoutés ! Cela contredit la définition d'un `HashSet`. Ici le `hashCode` de t1 (et t2) est calculé à partir de son adresse.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```
public class TestElement{
String nom;
public TestElement(String s){nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
}
public class TestHashSet{
HashSet<TestElement> hs = new HashSet<TestElement>();
void ajouter(String s){hs.add(new TestElement(s));}
void ajouter(TestElement t){ hs.add(t);}
}
public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);}
}
```

Exécution :

```
oui
[TestElement@e76cbf7, TestElement@22998b08]
```

Donc les 2 éléments qui sont égaux au sens de la méthode `equals` de `TestElement` ont été ajoutés ! Cela contredit la définition d'un `HashSet`. Ici le `hashCode` de `t1` (et `t2`) est calculé à partir de son adresse.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Réécrire la méthode `hashCode()`

Afin d'utiliser les instances d'une classe comme éléments d'une collection basée sur des tables de hachage (`HashSet`, `HashMap`, ...) il est indispensable de réécrire la méthode `hashCode()` de façon :

- elle soit compatible avec `equals()`
- elle soit rapide
- elle produise le même résultat pour un objet quelque soit le moment de l'appel

ATTENTION : si on change la valeur d'un objet et si son `hashCode` *réécrit* à cause de `equals` est alors modifié *on ne pourra pas le retrouver dans la table grâce à son code*.

I. Généricité

II. Collections

III. Interface Collection

IV. Les méthodes de l'interface Collection

V. La classe Collections

VI. Itérateurs

VII. Classe ArrayList<T>

VIII. La classe HashSet<T>

IX. La classe TreeSet<T>

X. Interface Map

Exemple

```
public class TestElement{
private String nom;
public TestElement(String s){nom=s;}
public void setNom(String s){this.nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
public int hashCode(){return this.nom.hashCode();}
}

public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);
t1.setNom("loulou");
if(ths.hs.contains(t1)) System.out.println("oui"); else System.out.println("non");
}
}
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```java
public class TestElement{
private String nom;
public TestElement(String s){nom=s;}
public void setNom(String s){this.nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
public int hashCode(){return this.nom.hashCode();}
}

public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);
t1.setNom("loulou");
if(ths.hs.contains(t1)) System.out.println("oui"); else System.out.println("non");
}
}
```

Exécution :

```
oui
[TestElement@366a36]
non
```

Réécrire la méthode `hashCode()`

Voici un procédé donnée par Joshua Block dans *Effective Java* :
On initialise un entier : `int resultat = 17 ;` (un nombre premier)
Pour *chaque attribut* on calcule une valeur entière c selon le tableau suivant

type de l'attribut	calcul de c
boolean boo	c vaut 0 pour boo vrai c vaut 1 pour boo faux
char, byte, short, int n	(int) n
long l	c vaut (int)(l^ (l >>> 32))
float x	c vaut Float.floatToIntBits(x)
double y	on calcule l=Double.doubleToLongBits(y) et c vaut (int)(l^ (l >>> 32))
null	c vaut 0
Objet o (intervenant dans le code de equals)	c vaut o.hashCode()
tableau t	chaque élément de t est traité comme un attribut à part entière

Pour *chaque attribut* on calcule `resultat = resultat * 37 + c`
`resultat` est le `hashCode()`.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Méthodes

Un `HashSet` peut contenir `null`. Les méthodes `add`, `remove`, `contains`, `size` sont exécutées en temps O(1).

- `boolean add(T e)` si e n'appartient pas à l'instance de `HashSet` alors e est ajouté et `true` est renvoyé ; sinon l'instance n'est pas modifiée et `false` est renvoyé.
- `boolean addAll(Collection<? extends T> c)` fait l'union de la collection c avec l'instance de `HashSet` ; si cette instance est modifiée `true` est renvoyé sinon `false` est renvoyé.
- `boolean remove(Object e)` si e appartient à l'instance de `HashSet` alors e est supprimé et `true` est renvoyé ; sinon `false` est renvoyé.
- `boolean removeAll(Collection<? extends T> c)` se comporte de même
- `boolean retainAll(Collection<?> c)` fait l'intersection de la collection c avec l'instance de `HashSet` ; si cette instance est modifiée `true` est renvoyé sinon `false` est renvoyé.
- `void clear()` supprime tous les éléments de la collection

remarque : voir API pour les exceptions possibles

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```
public class PersonneMedicalise extends Personne implements Comparable {
....
public int compareTo(Object p){
if (this.getAge()> ((PersonneMedicalise)p).getAge()) return -1; else
if (this.getAge()< ((PersonneMedicalise)p).getAge()) return 1; else
if (this.getPoids()> ((PersonneMedicalise)p).getPoids()) return -1; else
if (this.getPoids()< ((PersonneMedicalise)p).getPoids()) return 1; else
return 0;
}}

import java.util.*;
public class EnsPersonneMedicale {
HashSet <PersonneMedicalise> e;
public EnsPersonneMedicale() {}
...
public  PersonneMedicalise quiEstEnDanger(){
return(Collections.min(e));
}
public int ageMoyen(){
Iterator <PersonneMedicalise> it = e.iterator();
int a=0;
while (it.hasNext()) a= a+ it.next().getAge();
if (e.size()>0) return (a/e.size());else return 0;}}
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

TreeSet<T>

La classe `TreeSet<T>` implémente l'interface `Set<T>` et l'interface `SortedSet<T>`.

Un `TreeSet` s'appuie sur un arbre (rouge-noir) pour représenter un *ensemble d'objets triés* par ordre croissant (ordre naturel ou précisé par la méthode `compareTo`).

Attention : la classe des éléments d'un `TreeSet` doit redéfinir la méthode `compareTo` et la `equals` de façon cohérente (`equals` vrai doit être équivalent à `compareTo` vaut 0).

Les méthodes `add`, `remove`, `contains`, `size` sont exécutées en temps $O(\log n)$.

Programmation Orientée Objet avancée avec Java

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Méthodes

Outre les méthodes similaires à celles de `HashSet` on peut citer :

- `public SortedSet<T> subSet(T fromElement, T toElement)` renvoie le sous-ensemble des éléments compris entre `fromElement` inclus jusqu'à `toElement` exclu
- `public SortedSet<E> headSet(T toElement)` renvoie le sous-ensemble des éléments strictement inférieurs à `toElement`
- `public SortedSet<E> tailSet(E fromElement)` renvoie le sous-ensemble des éléments supérieurs ou égal à `fromElement`
- `public T first()` renvoie le plus petit élément
- `public T last()` renvoie le plus grand élément
- `public T floor(T e)` renvoie le plus grand élément inférieur ou égal à `e`
- `public T ceiling(T e)` renvoie le plus petit élément supérieur ou égal à `e`

remarque : voir API pour les exceptions possibles

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Utilisation d'un joker

Dans le code suivant il y a une affectation interdite.

```
class A {}
class B extends A{}
...
List<A> lA = new ArrayList<A>();
List<B> lB = new ArrayList<B>();
lA=lB; // interdit
```

Pour contourner l'interdiction précédente, Java permet d'utiliser un joker (ou *wildcard*) noté ?

```
List<B> lb = new ArrayList<B>();
List<? extends A> l =lB; // permis
```

ATTENTION : la liste l ainsi définie ne pourra être utilisée qu'en *lecture*.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Usage du joker

Le *wildcard* est très utile pour étendre le type de paramètre d'une méthode.
Ici la méthode `afficher` n'autorise que les paramètres de type `List<AWild>`

```
public class AWild {...}
public class BWild extends A{... }
public class TestAwildBWild{
        static void afficher(List<AWild> l){
                for(AWild a:l){System.out.println(a);}
        }
                public static void main(String[] arg){
                List<AWild> lA = new ArrayList<AWild>();
                List<BWild> lB = new ArrayList<BWild>();
                lA.add(new AWild(1));
                lA.add(new BWild(2));
                lB.add(new BWild(3));
                afficher(lA);
                afficher(lB);//erreur à la compilation
                // car lB n'est pas d'un sous-type du type de lA
}}
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Usage du joker

Avec le *wildcard* la méthode `afficher` fonctionnera sur toutes listes contenant des sous-types de `AWild` :

```
public class AWild {...}
public class BWild extends A{... }
public class TestAwildBWild{
        static void afficher(List<? extends AWild> l){
                for(AWild a:l){System.out.println(a);}
        }
                public static void main(String[] arg){
                List<AWild> lA = new ArrayList<AWild>();
                List<BWild> lB = new ArrayList<BWild>();
                lA.add(new AWild(1));
                lA.add(new BWild(2));
                lB.add(new BWild(3));
                afficher(lA);
                afficher(lB);// autorisé
}}
```

Remaque : on peut aussi utiliser le *wildcard* pour toute sur-classe d'une classe donnée

```
maMethode(List<? super uneClasse> l) {...}
```

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

L'interface `Map`

Un objet de type `Map` stocke des couples *clef-valeur*. On parle aussi de *dictionnaire*.
La clef doit être unique mais une valeur peut avoir plusieurs clefs.
L'interface `Map<K,V>` fixe les méthodes pour manipuler de tels couples. Les clefs sont de type `K` et sont associées aux valeurs de type `V`.
Toutes les collections de couples clef-valeur

- sont dans le paquetage *java.util*
- implémentent l'interface générique *Map<K,V>*

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- V put(K clef, V valeur) associe *valeur* à *clef*. Si clef est déjà associée à un objet V alors il est remplacé par le paramètre *valeur* puis il est renvoyé (null est renvoyé si *clef* était associée à rien)
- V get(Object clef) renvoie la valeur associée à *clef* si elle existe ou null sinon.
- boolean containsKey(Object clef) indique si un élément est associée au paramètre *clef*
- boolean containsValue(Object valeur) indique si le paramètre *valeur* est associée à au moins une clef
- boolean isEmpty() indique si la collection est vide
- Set<K> keySet() renvoie un ensemble constitué des clefs de l'objet
- Collection<V> values() renvoie la collection constituée des valeurs de l'objet
- Set<Map.Entry<K,V> > entrySet() renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- V put(K clef, V valeur) associe *valeur* à *clef*. Si clef est déjà associée à un objet V alors il est remplacé par le paramètre *valeur* puis il est renvoyé (null est renvoyé si *clef* était associée à rien)
- V get(Object clef) renvoie la valeur associée à *clef* si elle existe ou null sinon.
- boolean containsKey(Object clef) indique si un élément est associée au paramètre *clef*
- boolean containsValue(Object valeur) indique si le paramètre *valeur* est associée à au moins une clef
- boolean isEmpty() indique si la collection est vide
- Set<K> keySet() renvoie un ensemble constitué des clefs de l'objet
- Collection<V> values() renvoie la collection constituée des valeurs de l'objet
- Set<Map.Entry<K,V> > entrySet() renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (set...)
- écrire des accesseurs en lecture (get...) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (set...)
- écrire des accesseurs en lecture (get...) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface `Map`

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (set...)
- écrire des accesseurs en lecture (get...) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Les implémentations

La classe `HashMap<K,V>` implémente l'interface `Map<K,V>` avec une table de hachage. `null` peut être une clef.
La classe `Hashtable<K,V>` implémente l'interface `Map<K,V>` avec une table de hachage. `null` ne peut pas être une clef.
La classe `TreeMap<K,V>` implémente l'interface `Map<K,V>` avec un arbre rouge noir pour un ordre naturel sur les clefs ou bien avec un `Comparator` sur les clefs.
Cette dernière classe implémente l'interface `SortedMap` et a des méthodes spécifiques s'appuyant sur l'ordre des clefs.

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Itérer sur un objet `Map`

On peut utliser plusieurs méthodes :

- utiliser une boucle `foreach` sur l'ensemble des clefs obtenu par la méthode `keySet()`
- utiliser un itérateur sur l'ensemble des clefs obtenu par la méthode `keySet()`
- utiliser une boucle `foreach` sur l'ensemble des couples (clefs,valeurs) obtenu par la méthode `entrySet()`
- utiliser un itérateur sur l'ensemble des couples (clefs,valeurs) obtenu par la méthode `entrySet()`

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```java
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String> s:hm.entrySet())
        System.out.println(s);
}}
```

Exécution :

nom ? : Pierre
+33612121212
Paul=+33614141414
Jacques=+33615151515
Pierre=+33612121212

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String> s:hm.entrySet())
        System.out.println(s);
}}
```

Exécution :

nom ? : Pierre
+33612121212
Paul=+33614141414
Jacques=+33615151515
Pierre=+33612121212

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```java
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String>  s:hm.entrySet())
        System.out.println(s);
}}
```

Exécution :

nom ? : Pierre

+33612121212

Paul=+33614141414

Jacques=+33615151515

Pierre=+33612121212

I. Généricité
II. Collections
III. Interface Collection
IV. Les méthodes de l'interface Collection
V. La classe Collections
VI. Itérateurs
VII. Classe ArrayList<T>
VIII. La classe HashSet<T>
IX. La classe TreeSet<T>
X. Interface Map

Exemple

```
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String> s:hm.entrySet())
        System.out.println(s);
}}
```

Exécution :

nom ? : Pierre

+33612121212

Paul=+33614141414

Jacques=+33615151515

Pierre=+33612121212

Exercice

exercice 6 : Dans le but d'établir des statistiques sur les mots employés dans un document texte de 100000 mots on crée un dictionnaire de couples (int, liste de (mots,flottant))) de la façon suivante : la longueur d'un mot constitue une clef d'une liste ne contenant que des mots de cette longueur avec leur fréquence.
Ecrire une classe interne `Paire` caractérisée par deux attributs `String` et `Double`
Ecrire une méthode `void setMot(String s)` qui place la chaine de lettres `s` dans le dictionnaire en mettant à jour sa fréquence.
Ecrire une méthode `double frequenceMot(String s)` qui renvoie la fréquence de la chaine `s`

I. notion de processus

II. Les threads

III. Gestion des threads

IV. Exercices

Chapitre III – Thread

- I. notion de processus
- II. Les threads
- III. Gestion des threads
- IV. Exercices

I. notion de processus
II. Les threads
III. Gestion des threads
IV. Exercices

Chapitre III – Thread

- I. notion de processus
- II. Les threads
- III. Gestion des threads
- IV. Exercices

Chapitre III – Thread

- I. notion de processus
- II. Les threads
- III. Gestion des threads
- IV. Exercices

Chapitre III – Thread

- I. notion de processus
- II. Les threads
- III. Gestion des threads
- IV. Exercices

I. notion de processus
II. Les threads
III. Gestion des threads
IV. Exercices

Les Threads : notion de processus

Un processus est un programme en cours d'exécution. Le système d'exploitation alloue à chaque processus une partie de mémoire (pour stocker ses instructions, variables, ...) et il lui associe des informations (identifieur, priorités, droits d'accès ...).
Un processus s'exécute sur un processeur du système. Il a besoin de ressources : le processeur qui l'exécute, de la mémoire, des entrées sorties. Certaines ressources ne possèdent qu'un point d'accès et ne peuvent donc être utilisées que par un processus à la fois (par exemple, une imprimante).
On dit alors que les processus sont en *exclusion mutuelle* s'ils partagent une même ressource qui est dite *critique*. Il est nécessaire d'avoir une politique de synchronisation pour de telle ressource partagée.

I. notion de processus
II. Les threads
III. Gestion des threads
IV. Exercices

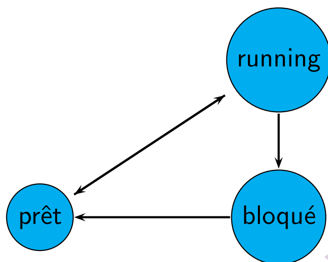
notion de processus

Par exemple concernant l'impression, il y a un processus qui gère les demandes et la file d'attente de ces demandes. Un tel processus - ≪ invisible ≫ - toujours en fonctionnant tant que le système fonctionne est appelé un *démon*.

Un démon - deamon - est un processus qui s'exécute en tâche de fond comme le service d'impression. En général un démon est lancé par le système d'exploitation au démarrage et stoppe à l'arrêt du système.

Un processus peut être :

- en cours d'exécution - *running*
- prêt à s'exécuter mais sans processeur libre pour l'exécuter
- bloqué (par manque de ressources)



I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

Les threads

Un *thread* ou *processus léger* est un processus à l'intérieur d'un processus.
En Java, lorsqu'on lance la machine virtuelle pour exécuter un programme, on lance un processus ; ce processus est composé de plusieurs threads : le thread principal (qui correspond au `main`) et d'autres threads comme le ramasse-miettes.
Donc un processus est composé de plusieurs threads (ou tâches) et va devoir partager ses ressources entre ses différents threads.
En Java on a deux moyens de créer un thread :

1. étendre la classe `Thread` : on aura alors un objet qui *contrôle une tâche*,
2. implémenter l'interface `Runnable` : on aura lors un objet qui *représente le code à exécuter*.

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

Hériter de la classe Thread

Lorsque l'on hérite de la classe `Thread`, il faut réécrire la méthode `void run()` pour qu'elle exécute le comportement attendu.
Puis une fois un objet de type `Thread` créé, cet objet peut invoquer la méthode `void start()` de la classe `Thread` qui elle-même invoque la méthode `run()` réécrite.

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

Exemple

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){this.s=s; }
   public void run() {
      while (true) {
         System.out.println(s);
         try { sleep(100);}
         catch(InterruptedException e){}
} }
}
public class TicTac {
   public static void main(String arg[]){
      Thread tic=new TestThread("TIC");
      Thread tac=new TestThread("TAC");
      tic.start();
      tac.start();
   }
}
```

A l'exécution on a un affichage continu de TIC TAC .... qui fait régulièrement une pause de 100 millisecondes.

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

Aperçu de la classe Thread

- constantes
  - static int MAX_PRIORITY : priorité maximale = 10
  - static int MIN_PRIORITY : priorité minimale = 1
  - static int NORM_PRIORITY : priorité normale = 5

- constructeurs
  - Thread() ;
  - Thread(Runnable cible, String nom)
  - ...
- méthodes
  - void start() invoque la méthode run
  - void run() : exécute le thread et peut lancer InterruptedException
  - int getPriority() : renvoie le niveau de priorité
  - static Thread currentThread() : renvoie le thread en cours d'exécution
  - static void sleep(long millis) : suspend l'activité du thread en cours d'exécution pendant millis millseconde
  - static void yield() : suspend l'activité du thread en cours d'exécution et permet aux autres threads de s'exécuter
  - void join() : attend la fin de l'objet Thread qui l'invoque
  - void interrupt() : interrompt `this`
  - static boolean interrupted() : teste si le thread courant a été interrompu
  - booleanisInterrupted() : test si `this` a été interrompu
  - boolean isAlive() : renvoie vrai si le thread n'a pas fini
  - String toString() : renvoie le nom du thread, sa priorité, son groupe de la classe ThreadGroup

Exemple

```
public class MaTache{
        public static void main(String[] arg) throws InterruptedException{
                System.out.println(Thread.currentThread());
                Thread tacheInitiale = Thread.currentThread();
                tacheInitiale.setName("tache initiale");
                Thread.sleep(1000); // sleep est une méthode de classe
                System.out.println(tacheInitiale);
                // le nom du thread a été changé mais pas le nom de son groupe
                System.out.println(tacheInitiale.isAlive());
                Thread maTache = new Thread();
                maTache.setName("ma tache");
                System.out.println(maTache);
                System.out.println(maTache.isAlive());
        }
}
```

A l'exécution on a :

```
Thread[main,5,main]
Thread[tache initiale,5,main]
true
Thread[ma tache,5,main]
false
```

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

implémenter l'interface Runnable

L'interface `Runnable` contient uniquement la méthode `run()` qui est à implémenter.
Pour lancer un thread avec une classe implémentant `Runnable` on utilise le constructeur de la classe `Thread` qui prend en paramètre un objet `Runnable`.

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

Exemple

```
class TestThread2 implements Runnable{
   String s;
   public TestThread2(String s){this.s=s; }
   public void run() {
      while (true) {
         System.out.println(s);
         try { Thread.sleep(100);}
         catch(InterruptedException e){}
}}}
public class TicTac2 {
   public static void main(String arg[]){
      Thread tic=new Thread(new TestThread2("TIC"));
      Thread tac=new Thread(new TestThread2("TAC"));
      tic.start();
      tac.start();
   }
}
```

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

la méthode join()

La méthode `join()` de la classe `Thread` invoquée par un objet `Thread` t met en attente le thread en cours d'exécution jusquà ce que t soit terminé.
La méthode `join()` lance une exception de type `InterruptedException` il faut donc l'utiliser dans un bloc `try catch`.

la méthode join()

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){this.s=s; }
   public void run() {
     for (int i =1; i<=2; i++) {
         System.out.print(s+" ");
         try { sleep(100);}
         catch(InterruptedException e){}}
} }
}
public class TicTac3 {
   public static void main(String arg[]){
      Thread tic=new TestThread("TIC");
      Thread tac=new TestThread("TAC");
      tic.start();
      tac.start();
      try{tic.join();}
       catch(InterruptedException e){}
       System.out.println("c'est fini ");
   }
}
```

Exécution TIC TAC TIC TAC c'est fini

remarque : sans l'instruction join() l'affichage de « c'est fini » se fait avant

I. notion de processus
II. Les threads
1. Hériter de la classe Thread
2. implémenter l'interface Runnable
III. Gestion des threads
IV. Exercices

la méthode join()

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){this.s=s; }
   public void run() {
     for (int i =1; i<=2; i++) {
         System.out.print(s+" ");
         try { sleep(100);}
         catch(InterruptedException e){}}
} }
}
public class TicTac3 {
   public static void main(String arg[]){
      Thread tic=new TestThread("TIC");
      Thread tac=new TestThread("TAC");
      tic.start();
      tac.start();
      try{tic.join();}
       catch(InterruptedException e){}
       System.out.println("c'est fini ");
   }
}
```

Exécution TIC TAC TIC TAC c'est fini
remarque : sans l'instruction join() l'affichage de« c'est fini » se fait avant

I. notion de processus
II. Les threads
III. Gestion des threads
1. synchronized
2. wait() et notify()
IV. Exercices

Gestion des threads avec `synchronized`

Les threads peuvent partager des ressources. Il faut alors s'assurer que cette ressource ne sera utilisée que par un seul thread en même temps.

Pour cela on utilise un mécanisme de *verrou* : tout objet (ou tableau) possède un verrou qui peut être ouvert ou fermé. Un thread t1 peut fermer un verrou sur un objet (si le verrou n'est pas déjà fermé) et lorsqu'il termine la portion de code verrouillée il rouvre le verrou.

Pour éviter qu'un autre thread t2 ne puisse exécuter une portion de code sur l'objet verrouillé il faut que cette portion de code ait le *même mécanisme* de verrou sur cet objet.

On parle alors de *synchronisation* entre t1 et t2 et on utilise le mot `synchronized` à cet effet.

On peut synchroniser

- une méthode m : `synchronized void m()` - ici `this` est l'objet sur lequel le verrou est posé
- un objet o : `synchronized(o)`...instructions ... - ici o est l'objet sur lequel le verrou est posé

Gestion des threads

Pendant l'exécution d'une portion de code marqué `synchronized` par un thread t1, tout autre thread t2 tentant d'exécuter une portion de code marquée `synchronized` relative au même objet est suspendu.

Remarques :

- attention une méthode non synchronisée peut modifier `this` même si `this` est verrouillée par une méthode synchronisée
- une méthode statique peut être synchronisée, elle verrouille alors sa classe empêchant une autre méthode statique de s'exécuter pendant sa propore exécution
- une méthode statique synchronisée n'empêche pas les modifications sur les instances de la classe par des méthodes d'instance.

Exemple sans synchronisation

```
public class  CompteJoint{
String nomH;
String nomF;
String numCompte;
int solde=0;
public CompteJoint(String sH, String sF, String numC){
numCompte=numC ; nomH= sH; nomF = sF;}
public String toString(){return ("le compte de " + nomH + " et "+ nomF + " numéro :"+numCompte )
public void depot(int somme){
        int resultat = solde;
        try{Thread.sleep(100);} // temps de traitement
        catch(Exception e){}
        solde= somme +  resultat;
        System.out.println("depot de " + somme);
}////
public class GuichetBanque extends Thread {
CompteJoint cj;
int id;
public GuichetBanque(CompteJoint cj,int n){this.cj=cj;this.id=n;}
public void run(){
        System.out.println("début de la transaction sur "+
                cj + " au guichet numéro " + id);
       cj.depot(100);
       System.out.println("fin de la transaction sur "+
                cj + " au guichet numéro " + id);
}}
```

Exemple sans synchronisation

```
public class TestGuichetCompteJoint{
        public static void main(String[] arg){
                CompteJoint  unCompte = new CompteJoint("Paul", "Eve", "00100100");
                GuichetBanque gb1 = new GuichetBanque(unCompte, 1);
                GuichetBanque gb2 = new GuichetBanque(unCompte, 2);
                gb1.start();
                gb2.start();
                try{gb1.join();
                        gb2.join();}
                catch(InterruptedException e){}
                System.out.println("votre solde est "+ unCompte.solde);
                }
}
```

Exécution :

début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 1
début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 2
depot de 100
depot de 100
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 1
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 2
votre solde est 100

Exécution :
début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 1
début de la transaction sur le compte de Paul et Eve numéro 00100100 au guichet numéro 2
depot de 100
depot de 100
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 1
fin de la transaction sur le compte compte de Paul et Eve numéro 00100100 au guichet numéro 2
votre solde est 100

I. notion de processus
II. Les threads
III. Gestion des threads
1. synchronized
2. wait() et notify()
IV. Exercices

Les deux objets `GuichetBanque` lisent le solde du compte avant de le créditer de 100. Donc chacun d'eux part d'un solde de 0.

I. notion de processus
II. Les threads
III. Gestion des threads
1. synchronized
2. wait() et notify()
IV. Exercices

Exemple avec synchronisation

```
public class  CompteJointSync{
String nomH;
String nomF;
String numCompte;
int solde=0;
public CompteJointSync(String sH, String sF, String numC){
numCompte=numC ; nomH= sH; nomF = sF;}
public String toString(){return ("le compte de " + nomH + " et "+ nomF + " numéro :"+numCompte )
public synchronized void depot(int somme){
        int resultat = solde;
        try{Thread.sleep(100);} // temps de traitement
        catch(Exception e){}
        solde= somme +  resultat;
        System.out.println("depot de " + somme);
}////
public class GuichetBanqueSync extends Thread {
CompteJoint cj;
int id;
public GuichetBanqueSync(CompteJointSync cj,int n){this.cj=cj;this.id=n;}
public void run(){
        System.out.println("début de la transaction sur "+
                cj + " au guichet numéro " + id);
       cj.depot(100);
       System.out.println("fin de la transaction sur "+
                cj + " au guichet numéro " + id);
}}
```

Exemple avec synchronisation

```java
public class TestGuichetCompteJointSync{
        public static void main(String[] arg){
                CompteJointSync  unCompte = new CompteJointSync("Paul", "Eve", "00100100");
                GuichetBanqueSync gb1 = new GuichetBanqueSync(unCompte, 1);
                GuichetBanqueSync gb2 = new GuichetBanqueSync(unCompte, 2);
                gb1.start();
                gb2.start();
                try{gb1.join();
                        gb2.join();}
                catch(InterruptedException e){}
                System.out.println("votre solde est "+ unCompte.solde);
                }
}
```

Exécution :
début de la transaction sur le compte compte de Paul et Eve au guichet numéro 1
début de la transaction sur le compte compte de Paul et Eve au guichet numéro 2
depot de 100
fin de la transaction sur le compte compte de Paul et Eve au guichet numéro 1
depot de 100
fin de la transaction sur le compte compte de Paul et Eve au guichet numéro 2
votre solde est 200

Ici l'objet `GuichetBanqueSync gb1` invoque la méthode synchronisée `depot` et donc verrouille `cj`.
Pendant la pause de `gb1`, l'objet `gb2` ne peut pas accéder au compte `cj` par la méthode synchronisée `depot` donc il doit attendre la fin de `gb1` pour exécuter la méthode synchronisée `depot`.
Quand il lit le solde de l'objet `cj` ce solde a été crédité par `gb1`.

wait() et notify()

La classe `Object` a les méthodes suivantes :

- `public final void wait()`
- `public final void wait(long maxMilli)`
- `public final void wait(long maxMill, int maxNano)`
- `public final void notify()`
- `public final void notifyAll()`

Les méthodes `wait` mettent en attente le thread en cours d'exécution et les méthodes `notify`, `notifyAll` interrompent cette attente.
Les `wait` doivent être invoqués dans une portion de code synchronisée ; le thread mis en attente déverouille alors l'objet qui a invoqué `wait`.

Exemple `wait notify`

Dans cet exemple on a 4 classes :

- une classe Producteur qui place les objets dans un entrepôt
- une classe Consommateur qui prend les objets dans un entrepôt
- une classe Entrepot
- une classe Test

- un Producteur ne peut mettre un objet que si l'entrepôt n'est pas plein ; il doit donc attendre qu'un Consommateur ait vidé l'entrepôt,
- un Consommateur ne peut pas prendre un objet si l'entrepôt est vide ; il doit donc attendre qu'un Producteur ait rempli l'entrepôt.

Entrepot fait la sunchronisation

```
public class Entrepot{
private static final int NB_MAX = 3;
private int nbObjet = 0;
String s;
public Entrepot(String s){this.s=s;}
public int getNbObjet(){return this.nbObjet;}
public boolean estVide(){return nbObjet==0;}
public boolean estPlein(){return nbObjet==NB_MAX;}
public String toString(){return (this.s +" (" + this.nbObjet + " objets)");}
public synchronized void mettre(){
        try{ while(estPlein()) {wait();
                System.out.println("prod endormi");}} //fin try
        catch(Exception e){}
        nbObjet++;
        notifyAll();
}//fin mettre()
public synchronized void prendre(){
        try{while(estVide()){ wait();
        System.out.println("cons endormi");}}
        catch(Exception e){}
        nbObjet--;
        notifyAll();
}//fin prendre()
}
```

```java
public class Producteur extends Thread{
        Entrepot e;
        String nom;
        public Producteur(Entrepot e, String s){this.e=e; this.nom=s;}
        public void run(){
                for (int i=1; i<=4; i++)
                {
                        System.out.println("avant prod  reste "+e);
                        e.mettre();
                System.out.println(" apres prod il y a " + e);
         }
}}
///
public class Consommateur extends Thread{
        Entrepot e;
        String nom;
        public Consommateur(Entrepot e, String s){this.e=e; this.nom=s;}
        public void run(){
                for (int i=1; i<=4; i++)
                {
                System.out.println("avant cons "+e);
                e.prendre();
                        System.out.println("apres cons reste "+e);
                }
}}///
public class TestEntrepot{
                public static void main(String[] arg){
                Entrepot e= new Entrepot("entrepot");
                Producteur p= new Producteur(e, "prod");
                Consommateur c= new Consommateur(e, "cons");
                 p.start();
                 c.start();
                 try{p.join();c.join();}
                 catch(Exception exc){}
                 System.out.println("final "+e);
                }}
```

Exécution

avant prod reste entrepot (0 objets)
apres prod il y a entrepot (1 objets)
avant prod reste entrepot (1 objets)
apres prod il y a entrepot (2 objets)
avant prod reste entrepot (2 objets)
apres prod il y a entrepot (3 objets)
avant prod reste entrepot (3 objets)
prod endormi
avant cons entrepot (3 objets)
apres cons reste entrepot (2 objets)
avant cons entrepot (2 objets)
apres cons reste entrepot (1 objets)
avant cons entrepot (1 objets)
apres cons reste entrepot (0 objets)
avant cons entrepot (0 objets)
cons endormi
apres prod il y a entrepot (1 objets)
apres cons reste entrepot (0 objets)
final entrepot (0 objets)

I. notion de processus
II. Les threads
III. Gestion des threads
1. synchronized
2. wait() et notify()
IV. Exercices

Exécution

```
avant prod reste entrepot (0 objets)
apres prod il y a entrepot (1 objets)
avant prod reste entrepot (1 objets)
apres prod il y a entrepot (2 objets)
avant prod reste entrepot (2 objets)
apres prod il y a entrepot (3 objets)
avant prod reste entrepot (3 objets)
prod endormi
avant cons entrepot (3 objets)
apres cons reste entrepot (2 objets)
avant cons entrepot (2 objets)
apres cons reste entrepot (1 objets)
avant cons entrepot (1 objets)
apres cons reste entrepot (0 objets)
avant cons entrepot (0 objets)
cons endormi
apres prod il y a entrepot (1 objets)
apres cons reste entrepot (0 objets)
final entrepot (0 objets)
```

Producteur et Consommateur font la synchronsation

Dans cette version la classe `Entrepot` n'organise pas la synchronisation qui est laissée à chacun des deux threads.

```
public class Entrepot{
//....
public void mettre(){nbObjet++;}
public void prendre(){nbObjet--;}
}
public class Producteur extends Thread{
        Entrepot e;
        String nom;
        public Producteur(Entrepot e, String s){this.e=e; this.nom=s;}
        public void run(){
                try{for (int i=1; i<=4; i++)
                synchronized(e){
                while(e.estPlein()){e.wait();} // e verrouille l'accès au Producteur
                System.out.println("avant prod "+e);
                e.mettre();
                System.out.println(" prod " + e);
                e.notifyAll(); } }
                catch(Exception exc){} } }
public class Consommateur extends Thread{
        Entrepot e;
        String nom;
        public Consommateur(Entrepot e, String s){this.e=e; this.nom=s;}
        public void run(){
                try{for (int i=1; i<=4; i++)
            synchronized(e)    {
                while(e.estVide()){e.wait();}// e verrouille l'accès au Consommateur
                System.out.println("avant cons "+e);
                e.prendre();
                System.out.println("cons "+e);
                e.notifyAll(); } }
                catch(Exception exc){} } }
```

I. notion de processus

II. Les threads

III. Gestion des threads

1. synchronized

2. wait() et notify()

IV. Exercices

Exécution

avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant prod entrepot (1 objets)
prod entrepot (2 objets)
avant prod entrepot (2 objets)
prod entrepot (3 objets)
avant cons entrepot (3 objets)
cons entrepot (2 objets)
avant cons entrepot (2 objets)
cons entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
final entrepot (0 objets)

Exécution

avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant prod entrepot (1 objets)
prod entrepot (2 objets)
avant prod entrepot (2 objets)
prod entrepot (3 objets)
avant cons entrepot (3 objets)
cons entrepot (2 objets)
avant cons entrepot (2 objets)
cons entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
avant prod entrepot (0 objets)
prod entrepot (1 objets)
avant cons entrepot (1 objets)
cons entrepot (0 objets)
final entrepot (0 objets)

Exercices

exercice 7 : Ecrire une classe `Compteur` qui hérite de la classe `Thread` ; elle a un attribut de type `String` ; sa méthode `run()` compte de 1 à n en faisant une pause aléatoire de 0 à 5s entre deux incrémentations, affiche chaque valeur incrémentée avec son nom puis affiche un message de fin.
Tester cette classe dans une classe `TestCompteur` qui lance plusieurs `Compteur`.
Modifier la méthode `run()` de la classe `Compteur` pour que le thread affiche le message de fin avec son ordre d'arrivée.
Tester la modification.

Exercices

exercice 8 : Quel est le problème du programme suivant

```
class MonObjet {
public MonObjet () {}
public synchronized void action1 (MonObjet o) {
try{Thread.currentThread().sleep(200);}
catch (InterruptedException ex) { return ; }
o.action2(this); }
public synchronized void action2 (MonObjet o) {
 try{Thread.currentThread().sleep(200);}
catch (InterruptedException ex) { return ; }
o.action1(this); } }
class MonThread extends Thread {
private MonObjet obj1 , obj2;
public Thread(MonObjet o1, MonObjet o2) {
obj1 = o1;
obj2 = o2; }
public void run() {
obj1.action1(obj2); } }
class Deadlock {
public static void main (String[] args) {
MonObjet o1 = new MonObjet (); MonObjet o2 = new MonObjet ();
MonThread t1 = new MonThread(o1,o2); t1.setName("t1");
MonThread t2 = new MonThread(o2,o1); t2.setName("t2");
t1.start(); t2.start();
}}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Chapitre IV – Programmation événementielle

- I. Introduction
- II. Notions d'événements et de gestionnaires d'événements
- III. Evénements
- IV. Conteneur
- V. Les gestionnaires de répartition
- VI. Composant

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Chapitre IV – Programmation événementielle

- I. Introduction
- II. Notions d'événements et de gestionnaires d'événements
- III. Evénements
- IV. Conteneur
- V. Les gestionnaires de répartition
- VI. Composant

Chapitre IV – Programmation événementielle

- I. Introduction
- II. Notions d'événements et de gestionnaires d'événements
- III. Evénements
- IV. Conteneur
- V. Les gestionnaires de répartition
- VI. Composant

Chapitre IV – Programmation événementielle

- I. Introduction
- II. Notions d'événements et de gestionnaires d'événements
- III. Evénements
- IV. Conteneur
- V. Les gestionnaires de répartition
- VI. Composant

Chapitre IV – Programmation événementielle

- I. Introduction
- II. Notions d'événements et de gestionnaires d'événements
- III. Evénements
- IV. Conteneur
- V. Les gestionnaires de répartition
- VI. Composant

Chapitre IV – Programmation événementielle

- I. Introduction
- II. Notions d'événements et de gestionnaires d'événements
- III. Evénements
- IV. Conteneur
- V. Les gestionnaires de répartition
- VI. Composant

IHM

Une interface homme machine (IHM) permet à une personne d'interagir avec un système.
Pour concevoir une IHM il faut considérer les trois aspects suivants :

- le modèle est le système manipulé, il doit conserver son autonomie
- la vue est ce qui est fourni graphiquement ou textuellement pour représenter le modèle auprès de l'utilisateur
- le contrôle est proposé dans la vue par des commandes textuelles ou des boutons ou des menus

C'est ce que l'on appelle le modèle MVC.
Le modèle doit être indépendant de toutes les IHM (VC) qui permettent de le manipuler.

On peut avoir plusieurs IHM sur un même modèle : par exemple vous pouvez créer un nouveau dossier par la ligne de commande `mkdir` ou par le menu `Fichier` puis `Nouveau Dossier` et dans les deux cas le dossier créé sera visible par la commande `ls` ou comme icone dans une fenêtre.
Le principe de base d'une IHM est que l'utilisateur ne peut voir ou agir sur le modèle que par la vue.
Par exemple devant une machine à café, vous ne pourrez obtenir un café qu'en introduisant une pièce ... pas en ouvrant la machine pour y prendre l'eau et le café.
Les dispositifs logiques permettant une action de l'utilisateur sont appelés *widgets* (pour **wi**ndow ga**dget**) par exemple boutons, ascenceurs, ...
Une interface graphique est donc un assemblage de *widgets*. Il y a deux types de widgets

- les *widgets atomiques* : ceux qui permettent à l'utilisateur d'interagir
- les *widgets* composés d'autres widgets

En Java

Java propose deux niveaux de composants graphiques :

- Abstract Window Toolkit (AWT) est une bibliothèque graphique pour Java, faisant partie de JFC (Java Foundation Classes) dans le paquetage `java.awt`.
- swing présente des composants graphiques de conception différente dans le paquetage `javax.swing`. Swing partage certaines classes avec AWT. Les noms de ses composants commencent par J.

La différence principale réside dans la représentation des fenêtres natives qui dans AWT prennent beaucoup de place en mémoire, n'ont pas le même aspect selon l'environnement alors que dans la mesure du possible les composants swing auront le même aspect sur toutes les plates-formes.

Structure d'une IHM

En résumé une IHM sera structurée par :

- ses conteneurs
- ses composants atomiques
- son gestionnaires de disposition
- ses gestionnaires d'événements

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Notion d'événements

Les actions faites par l'utilisateur seront des événements (`event`) et le programme réagira à ces actions par des gestionnaires d'événements (`event handler`).
La gestion d'un événement passe par l'utilisation d'*écouteur d'événements* (les `xxxListener`) liée à un objet source d'événements par une méthode addXxxListener()).
La source d'événement envoie donc l'événement (de type `XxxEvent`) à tous les écouteurs auxquels il est lié ; un de ces écouteurs exécute alors une action - le gestionnaire d'événements.

Exemple

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class EcouteurDeFin implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}//////////
import javax.swing.JFrame;
import javax.swing.JButton;
public class EssaiSwing1 {
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
    public static void main(String[] args)
    {
        JFrame fenetre1 = new JFrame( "Essai Swing");
        fenetre1.setSize(WIDTH, HEIGHT);

        fenetre1.setDefaultCloseOperation(
                               JFrame.DO_NOTHING_ON_CLOSE);

        JButton boutonFin = new JButton("Cliquez pour terminer le programme");
        EcouteurDeFin boutonEcouteur = new EcouteurDeFin( );
        boutonFin.addActionListener(boutonEcouteur);
        fenetre1.getContentPane().add(boutonFin);

        fenetre1.setVisible(true);
    }
}
```

Ici le source d'événement est l'objet `boutonFin`; il est lié à un écouteur `boutonEcouteur`. Lorsqu'on clique sur le bouton `boutonFin` l'événement « cliquer » est envoyé à `boutonEcouteur` qui exécute la méthode `actionPerformed` de sa classe.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Evénements et écouteurs

L'utilisateur effectue une action au niveau de l'interface utilisateur (clic souris, sélection d'un item, etc) alors un événement graphique est émis. Lorsqu'un événement se produit

- il est reçu par le composant avec lequel l'utilisateur interagit (par exemple un bouton, un curseur, un champ de texte, etc.) ;
- ce composant transmet cet événement à un autre objet, un *écouteur* qui possède une méthode pour traiter l'événement ;
- cette méthode reçoit l'objet événement généré de façon à traiter l'interaction de l'utilisateur.

L'événement agit comme un signal envoyé à un écouteur qui lui-même accomplit une action en réponse à cet événement.
La gestion des de tous les événements s'exécute dans un thread unique :*event-dsipatching thread*. De cette façon, un seul gestionnaire (comme la méthode `actionPerformed(ActionEvent e)`) peut s'exécuter à la fois.

Ecouteurs

Les écouteurs sont des interfaces. Donc une même classe peut implémenter plusieurs interfaces écouteur. Par exemple une classe héritant de `Frame` implémentera les interfaces `MouseMotionListener` (pour les déplacements souris) et `MouseListener` (pour les clics souris).
Chaque composant de l'AWT est conçu pour être la source d'un ou plusieurs types d'événements particuliers. Cela se voit notamment grâce à la présence dans la classe de composant d'une méthode nommée addXxxListener().
Tous les écouteurs seront dans le paquetage `java.awt.event`

Voici quelques interfaces de ce paquetage (liste non exhaustive) :

- `ActionListener` : action sur un bouton, retour chariot, ...
- `WindowListener` : fermeture, iconisation, ...
- `TextListener` : changement de valeur dans une zone de texte
- `ItemListener` : sélection d'un item dans une liste
- `MouseListener` : clic, relachement du clic
- `MouseMotionListener` : déplacement de la souris, selectionner et glisser, ...
- `ComponentListener` : état caché, visible, ...
- `ContainerListener` : ajout d'un composant
- `FocusListener` : savoir si un élément a le focus (s'il est actif)
- `KeyListener` : gestion du clavier

I. Introduction

II. Notions d'événements et de gestionnaires d'événements

III. Evénements

IV. Conteneur

V. Les gestionnaires de répartition

VI. Composant

`ActionListener`

L'interface `ActionListener` a une méthode
`void actionPerformed (ActionEvent e)`

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

WindowListener

L'interface `WindowListener` a les méthodes

- `void windowClosing (WindowEvent e)`
- `void windowOpened(WindowEvent e)`
- `void windowIconified (WindowEvent e)`
- `void windowDeiconified (WindowEvent e)`
- `void windowClosed (WindowEvent e)`
- `void windowActivated (WindowEvent e)`
- `void windowDeactivated (WindowEvent e)`

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

TextListener

L'interface `TextListener` a la méthode
`void textValueChanged(TextEvent e)`

I. Introduction

II. Notions d'événements et de gestionnaires d'événements

III. Evénements

IV. Conteneur

V. Les gestionnaires de répartition

VI. Composant

`ItemListener`

L'interface `ItemListener` a une méthode
`void itemStateChanged(ItemEvent e)`

MouseListener

L'interface `MouseListener` a les méthodes

- `void mouseClicked(MouseEvent e)` (appuyer et relacher)
- `void mousePressed(MouseEvent e)` (appuyer)
- `void mouseReleased(MouseEvent e)` (relacher)
- `void mouseEntered (MouseEvent e)`
- `void mouseExited (MouseEvent e)`

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

MouseMotionListener

L'interface `MouseMotionListener` a les méthodes

- `void mouseDragged(MouseEvent e)` (appuyer et déplacer)
- `void mouseMoved(MouseEvent e)` (déplacer)

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

ComponentListener

L'interface `ComponentListener` a les méthodes

- `void componentResized(ComponentEvent e)`
- `void componentMoved(ComponentEvent e)`
- `void componentShown(ComponentEvent e)`
- `void componentHidden (ComponentEvent e)`

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

ContainerListener

L'interface `ContainerListener` a les méthodes

- `void componentAdded(ContainerEvent e)`
- `void componentRemoved(ContainerEvent e)`

FocusListener

L'interface `FocusListener` a les méthodes

- `void focusGained(FocusEvent e)`
- `void focusLost(FocusEvent e)`

`KeyListener`

L'interface `KeyListener` a les méthodes

- `void keyTyped(KeyEvent e)` (appuyer et lacher)
- `void keyPressed(KeyEvent e)` (appuyer)
- `void keyReleased(KeyEvent e)` (lacher)

Adaptateurs

L'inconvénient de ces interfaces est qu'il faut implémenter toutes les méthodes, même si on n'en a pas besoin.
Java fournit des classes *adaptateur* qui implémentent certaines (pas toutes) de ces interfaces ayant plus d'une méthode avec un code vide pour chaque méthode.
Il suffit donc d'étendre ces adaptateurs - XxxAdapter - et de réécrire la méthode qui nous est utile.
On ne peut bien sûr étendre qu'une unique classe *adaptateur*.

```
public class Terminer extends WindowAdapter // implémente WindowListener
{ public void windowClosing(WindowEvent e) { // seule méthode utile
 System.exit(0); }
 }
 // dans un main
Terminer term = new Terminer() ;
JFrame fenetre = new JFrame( ) ;
fenetre.addWindowListener(Term) ;
```

Ici le programme s'arrête à la fermeture de la fenêtre.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Composant et conteneur

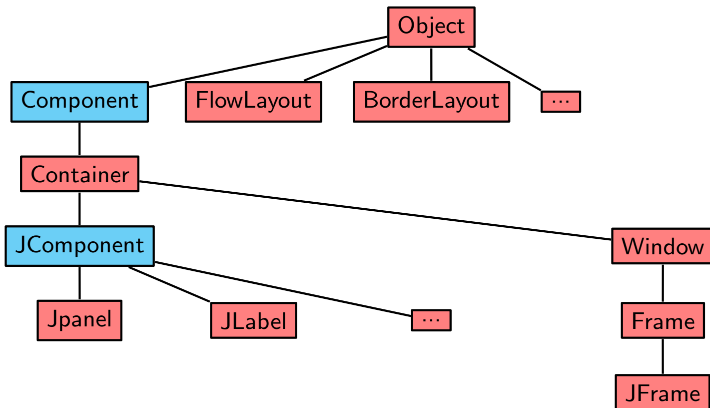
Un composant est une partie visible de l'interface. Ce sont des sous-classes de la classe abstraite `java.awt.Component` ou de `javax.swing.JComponent`.
On peut distinguer :

- les composants atomiques : `Button, JButton, JLabel, JTextField, JList`, ...
- les composants structurés : `Panel, JPanel, JScrollPane`, ...; les composants structurés pourront contenir d'autres composants. On les désignera comme conteneur

Tout composant a (notamment) :

- une taille préférée
- une taille minimum
- une taille maximum

On peut changer les valeurs par défaut de la taille d'un composant en utilisant un objet de type `Dimension` instancié par le constructeur `Dimension (int largeur, int hauteur)` où `largeur` et `hauteur` expriment un nombre de pixels.


Object
Component
FlowLayout
BorderLayout
...
Container
JComponent
Window
Jpanel
JLabel
...
Frame
JFrame

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Conteneur

Un conteneur est un espace dans lequel on peut positionner plusieurs composants.

Ce sont des sous-classes de la classe abstraite `java.awt.Container`.

La classe `Container` est elle-même une sous-classe de la classe `Component` : donc un conteneur est une instance particulière d'un composant.

Dans `AWT`, les deux conteneurs les plus courants sont les `Frame` et les `Panel`. Il y a aussi `Window` et `Dialog`.

Dans `swing`, les deux conteneurs les plus courants sont les `JPanel` et les `JFrame`.

De façon générale, on ajoute un *composant* dans un *conteneur*, avec la méthode `add()` :

```
Panel p = new Panel();
Button b = new Button();
p.add(b);
```

De manière similaire, un composant est retiré de son conteneur par la méthode `remove()` : `p.remove(b);`

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Frame

Un `Frame` représente une fenêtre de haut niveau avec un titre, une bordure et des angles de redimensionnement. La classe `Frame` hérite de la classe `Window` qui définit une fenêtre sans bordure.
La plupart des applications utilisent au moins un `Frame` comme point de départ de leur interface graphique.
Constructeur :

- `Frame()`;
- `Frame(String)` : précise le nom de la fenêtre

Méthodes :

- `String getTitle()` : renvoie le titre de la fenêtre
- `void setTitle(String)` : définit le titre de la fenêtre
- `boolean isResizeable()` : renvoie si la taille est modifiable
- `void setResizeable(boolean)` : définit si la taille peut être modifiée
- `void setSize(int l, int h)` : définit la largeur l et la hauteur h de la fenêtre (unité : pixel)
- `void setBackground( Color)` : donne une couleur de fond à la fenêtre
- `void setBounds(int x, int y, int largeur, int hauteur)` : fixe la largeur, hauteur et position (x,y) de la fenêtre

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

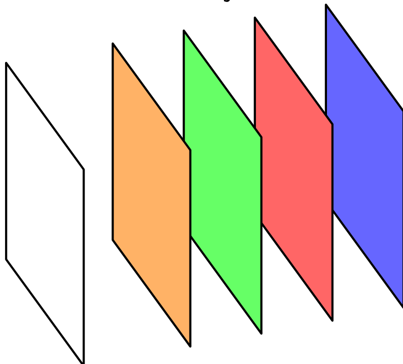
Exemple

```
import java.awt.*;
public class EssaiFenetre1 {
public static void main(String[] args) {
        Frame f = new Frame("ma fenetre");
        Button b =  new Button("coucou");
        f.setBackground(Color.red);
        f.add(b);
        f.setVisible(true);
}}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Jframe

La structure d'un objet `JFrame` est la suivante :



GlassPane, Content Pane, Layered Pane, Root Pane, Frame

Contrairement à un objet `Frame`, on n'ajoute pas direcement un composant à un objet `JFrame` mais à son `contentPane` associé que l'on obtient par la méthode `getContentPane()` qui renvoie un objet `JPanel`.
De plus un `JFrame` ne peut pas contenir une aute `JFrame`
Constructeurs

- `public JFrame()` : crée un objet fenêtre vide de la classe JFrame
- `public JFrame(String titre)` : crée un objet fenêtre de la classe JFrame avec le titre en argument

Jframe

Quelques méthodes :

- `public void setDefaultCloseOperation(int operation)` : règle le comportement de la fenêtre lorsqu'on clique sur le bouton de fermeture de la fenêtre. L'argument doit être une des constantes suivantes :
  - `JFrame.DO_NOTHING_ON_CLOSE` : ne fait rien ; mais s'il y a des `WindowListener` enregistrés, alors ils sont invoqués (les listeners font partie des événements)
  - `JFrame.HIDE_ON_CLOSE` : cache la fenêtre après avoir invoqué tout `WindowListener` enregistré.
  - `JFrame.DISPOSE_ON_CLOSE` : cache et supprime la fenêtre après avoir invoqué tout `WindowListener` enregistré. La fenêtre est éliminée mais le programme ne termine pas. Pour terminer le programme, on utilise plutôt l'argument suivant
  - `JFrame.EXIT_ON_CLOSE` : termine l'application en utilisant la méthode `System.exit` (A ne pas utiliser pour des fenêtres dans des applets)
  - l'action par défaut est `JFrame.HIDE_ON_CLOSE`.
  - lance l'exception `IllegalArgumentException` en cas d'argument illégal

Jframe

- `public Container getContentPane()` : renvoie le `contentPane` de l'objet
- `public void dispose()` : élimine l'objet fenêtre et tous ses composants
- `public void setLayout(LayoutManager manager)` : définit le gestionnaire de répartition de l'objet fenêtre.
- `public void setJMenuBar(JMenuBar menubar)` : définit le menu de l'objet fenêtre.

Exemple Jframe

```
import javax.swing.* ;
public class EssaiFenetre2 {
public static void main(String[] args){
JFrame f =new JFrame("Ma deuxieme fenetre");
 f.setVisible(true) ;
//f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE);
 }}
```

Cette fenêtre se ferme en cliquant sur le coin mais cela ne ferme pas l'application, cele ne fait que rendre la fenêtre invisible.
Il faudrait utiliser l'instruction en commentaire pour stopper l'application à la fermeture de la fenêtre.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Exemple

```
import java.awt.*;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class EssaiSwing3 extends JFrame implements ActionListener
{    public static final int WIDTH = 300;
     public static final int HEIGHT = 200;
     public   String m="Cliquez pour changer";
     private int c=0;
     private JButton bouton ;
     public static void main(String[] args)
     {
        EssaiSwing3 fenetre = new EssaiSwing3();
         fenetre.setVisible(true);
     }
     public EssaiSwing3() {
         super("Essai Swing 3");
         this.setSize(WIDTH,HEIGHT);
         this.setDefaultCloseOperation(
                                  JFrame.EXIT_ON_CLOSE);
      bouton = new JButton(m);
      bouton.setBackground( Color.RED);
      bouton.addActionListener(this);
      this.setLayout(new BorderLayout());
      this.getContentPane().setBackground(Color.BLUE);
      this.getContentPane().add(bouton, BorderLayout.SOUTH);
      }
     public void actionPerformed(ActionEvent e)
     {
         getContentPane().setBackground(Color.RED);
         c+=1;
         if (c%2==0)
         { bouton.setText(m);getContentPane().setBackground(Color.PINK); }
  else {bouton.setText("Hello!");getContentPane().setBackground(Color.YELLOW); }
     }}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

JPanel

Un `JPanel` n'a pas une apparence propre et ne peut pas être utilisé comme fenêtre autonome.
Les `JPanel` sont créés et ajoutés aux autres conteneurs de la même façon que les composants tels que les boutons.
Les `JPanel` peuvent ensuite redéfinir une présentation qui leur soit propre pour contenir eux-mêmes d'autres composants.
Ils sont à la fois conteneur et composant.
Constructeurs :

- `JPanel()` Création d'un panneau avec `FlowLayout` comme gestionnaire de placement par défaut.
- `JPanel(LayoutManager MonGestionnaire)` Création d'un panneau avec MonGestionnaire comme gestionnaire de répartition.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

JPanel

Méthodes :

- `void add(Component c)` : ajoute le composant c à la suite dans le panneau.
- `void add(Component c, int i)` : ajoute le composant c en position i dans la liste des composants du panneau. L'indice du premier composant est 0.
- `void add(Component c, Object o)` : ajoute c dans le panneau. L'ajout se fait à une position écran qui dépend de o et du gestionnaire de placement.
- `void add(Component c, Object o, int i)` : ajoute le composant c en position i dans la liste des composants du panneau. L'ajout se fait à une position écran qui dépend de o et du gestionnaire de placement.
- `void add(String s, Component c)` : ajoute c dans le panneau. L'ajout se fait à une position écran qui dépend de la chaîne s et du gestionnaire de placement.

JPanel

Méthodes :

- `int getComponentCount()` : retourne le nombre de composants du panneau.
- `Component getComponent(int i)` : retourne le ième composant.
- `Component getComponentAt(int x, int y)` : retourne le composant qui se trouve au point (x, y).
- `Component getComponentAt(Point p)` : retourne le composant qui se trouve au point p.

JPanel

- `Component[] getComponents()` : retourne un tableau de tous les composants.
- `void remove(Component c)` : retire le composant c.
- `void remove(int i)` : Retire le composant de rang i.
- `void removeAll()` : retire tous les composants.

gestionnaires de répartition

Ils sont dans le paquetage `java.awt` et permettent de structurer les composants d'un conteneur.

Chaque conteneur a un gestionnaire de position par défaut avec sa stratégie qui lui est propre ; le gestionnaire de position gère le positionnement, le dimensionnement.

Il y a un réagencement des composants lorsque :

- la taille du conteneur est modifiée
- un composant est ajouté ou retiré
- un composant modifie sa taille

Les principaux gestionnaires sont

- `FLowLayout` pour les `Panel` et ses descendants
- `BorderLayout` pour les `Window` et ses descendants (`Frame`, `Dialog`,...)
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

On pourra changer de gestionnaire par la méthode `setLayout()`

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Exemple sans gestionnaire de répartition

```
import java.awt.*;
public class Fenetre3Boutons extends Frame{
public static final int LARGEUR = 300 ;
public static final int HAUTEUR = 250 ;
 public Fenetre3Boutons(String titre){
 super(titre) ;
setSize(LARGEUR, HAUTEUR) ;
setBackground (Color.red) ;
add(new Button("bouton 1")) ;
add(new Button("bouton 2")) ;
add(new Button("bouton 3")) ;
setVisible(true);
} }
////
public class EssaiFenetre3Boutons{
public static void main(String[] args) {
Fenetre3Boutons f = new Fenetre3Boutons("ou sont les 3 boutons?");
} }
```

FLowLayout

C'est le plus simple :

- il respecte la taille préférée des composants
- il place les composants autant que faire ce peut horizontalement dans le conteneur
- il commence une nouvelle rangée lorsqu'il ne peut plus placer de composants dans la première rangée
- il ne gère pas le cas où tous les composants ne peuvent tenir dans le conteneur et dans ce cas des composants peuvent ne pas apparaître.

Ces constructeurs sont

- FlowLayout()
- FlowLayout(int align) : permet de préciser l'alignement des composants dans le conteneur (FlowLayout.CENTER, FlowLayout.LEFT, FlowLayout.RIGHT ...) ; par défaut, align vaut CENTER
- FlowLayout(int align, int hgap, int vgap) : permet de préciser l'alignement et les écarts horizontaux et verticaux entre les composants dont la valeur par défaut est 5.

Exemple

On place cinq boutons dans un `Panel` dans une fenêtre et on redimensionne la fenêtre pour faire déplacer puis disparaître des boutons.

```
import java.awt.*;
public class EssaiFlowLayout extends Frame {
private Button b1,b2,b3,b4, b5;
// constructeur insérant 5 boutons
public EssaiFlowLayout() {
FlowLayout maLigne = new FlowLayout(FlowLayout.LEFT,10,10);
setLayout(maLigne);
b1 = new Button ("1");
b2 = new Button ("2");
b3 = new Button ("3");
b4 = new Button ("4");
b5 = new Button ("5");
this.add("prem",b1);
this.add("deux",b2);
this.add("trois",b3);
this.add("quatre",b4);
this.add("cinq",b5);
}
public static void main (String args []) {
EssaiFlowLayout essai = new EssaiFlowLayout();
essai.pack (); essai.setVisible(true) ; }}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

BorderLayout

Ce gestionnaire distingue 5 zones dans le conteneur : nord, est, sud, ouest et centre.
Les constantes `BorderLayout.NORTH`, `BorderLayout.EAST`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.CENTER` sont à utiliser avec la méthode add en tant que deuxième argument.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant

Exemple

```
import java.awt.*;
public class EssaiBorderLayout extends Frame {
private Button b1,b2,b3,b4, b5;
// constructeur insérant 5 boutons
public EssaiBorderLayout() {
//setLayout(new BorderLayout());// par défaut
b1 = new Button ("Nord");
 b2 = new Button ("Sud");
  b3 = new Button ("Est");
  b4 = new Button ("Ouest");
  b5 = new Button ("Centre");
this.add(b1, BorderLayout.NORTH);
this.add(b2 , BorderLayout.SOUTH);
this.add(b3, BorderLayout.EAST);
this.add(b4, BorderLayout.WEST);
this.add(b5, BorderLayout.CENTER);
}
public static void main (String args []) {
EssaiBorderLayout essai = new EssaiBorderLayout();
essai.pack (); essai.setVisible(true) ; }}
```

GridLayout

La disposition des composants est fixée par une grille dont toutes les cellules ont la même taille et dont on peut fixer le nombre par les dimensions (nbreLignes,nbreColonnes) données en argument au constructeur de la classe `GridLayout`.
Lors d'un redimensionnement, les composants changent de taille mais pas de position relative.

Exemple

```
import java.awt.*;
public class EssaiGridLayout extends Frame {
private Button b1,b2,b3,b4, b5,b6;
public EssaiGridLayout() {
GridLayout maGrille = new GridLayout(2,3);
setLayout(maGrille);
b1 = new Button ("1");
 b2 = new Button ("2");
  b3 = new Button ("3");
  b4 = new Button ("4");
  b5 = new Button ("5");
  b5 = new Button ("6");
this.add(b1);
this.add(b2);
this.add(b3);
this.add(b4);
this.add(b5);
this.add(b6);
}
public static void main (String args []) {
EssaiGridLayout essai = new EssaiGridLayout();
essai.pack (); essai.setVisible(true) ; }}
```

CardLayout

Le `CardLayout` n'affiche qu'un composant à la fois : les composants sont considérées comme mis à la façon d'un tas de cartes (le nouvelle se plaçant derrière la précédente).
La présentation `CardLayout` permet à plusieurs composants de partager le même espace d'affichage de telle sorte que seul l'un d'entre eux soit visible aÌ la fois.
Pour ajouter un composant aÌ un conteneur utilisant un `CardLayout` il faut utiliser la méthode `add(String cle, Component monComposant)`; le paramètre `cle` permettra d'identifier le composant `monComposant` dans le gestionnaire `CardLayout`.
Pour passer de l'affichage d'un composant à un autre, le gestionnaire `CardLayout` a les méthodes `first(Container leContenant)`, `last(Container leContenant)`, `next(Container leContenant)`, `previous(Container leContenant)` ou `show(Container leContenant, String cle)`.

Exemple

```
import java.awt.*;
public class EssaiCardLayout extends Frame {
private Button b1,b2,b3,b4, b5;
// constructeur insérant 5 boutons
public EssaiCardLayout() {
CardLayout mesCartes = new CardLayout();
setLayout(mesCartes);
b1 = new Button ("1");
b2 = new Button ("2");
b3 = new Button ("3");
b4 = new Button ("4");
b5 = new Button ("5");
this.add("prem",b1);
this.add("deux",b2);
this.add("trois",b3);
this.add("quatre",b4);
this.add("cinq",b5);
mesCartes.show(this,"cinq");
}
public static void main (String args []) {
EssaiCardLayout essai = new EssaiCardLayout();
essai.pack (); essai.setVisible(true) ; }}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Composant `AbstractButton`

Les classes `JButton` et `JMenuItem` héritent de cette classe abstraite.

- `public void setBackground(Color theColor)` : donne une couleur au composant objet.
- `public void addActionListener(ActionListener listener)` : ajoute un écouteur (ActionListener).
- `public void removeActionListener(ActionListener listener)` : supprime un écouteur (ActionListener).
- `public void setActionCommand(String actionCommand)` : envoie la chaîne de caractères paramètre à l'événement, par défaut actionCommand est la chaîne écrite sur le bouton ou bien le menu.
- `public String getActionCommand()` : renvoie la chaîne de caractères the action command
- `public void setText(String texte)` : règle le texte du composant.
- `public String getText()` : renvoie le texte écrit sur le composant (texte sur un bouton, . . .)

- `public void setPreferredSize(Dimension preferredSize)` donne une dimension préférée au composant bouton ou label. Mais ce n'est qu'une suggestion pour le gestionnaire de répartition (layout manager). Ce gestionnaire n'est pas forcé d'utiliser cette dimension préférée.
- `public void setPreferredSize( new Dimension(int largeur, int hauteur))` : même chose, largeur et hauteur en pixels
- `public void setMaximumSize(Dimension maximumSize)` donne la dimension maximale du bouton ou du label. Mais ce n'est qu'une suggestion pour le gestionnaire de répartition (layout manager). Ce gestionnaire n'est pas obligé de respecter cette dimension maximale. The int values give the width and height in pixels.
- `public void setMaximumSize( new Dimension(int width, int height))` : même chose, largeur et hauteur en pixels
- méthodes duales `setMinimumSize`(..)

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

JButton

Constructeurs :

- `JButton()`
- `JButton(String)` : précise le texte du bouton
- `JButton(Icon)` : précise une icône

```
ImageIcon img = new ImageIcon("unDessin.gif");
JButton bouton = new JButton("Mon bouton",img);
```

la classe `JToggleButton` définit un bouton à deux états.
la classe `ButtonGroup` permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné.
Voir aussi les classes `JCheckBox` et `JRadioButton`.

I. Introduction

II. Notions d'événements et de gestionnaires d'événements

III. Evénements

IV. Conteneur

V. Les gestionnaires de répartition

VI. Composant

1. les boutons
2. les menus
3. Composants de saisie de texte

les menus

- `JMenuBar` : encapsule une barre de menu
- `JMenu` : encapsule un menu
- `JMenuItem` : encapsule un élément d'un menu

↬ La classe `JMenuBar` encapsule une barre de menu qui contient zéro ou plusieurs menus. Elle hérite de `Jcomponent`.
Quelques méthodes :

- `JMenu add(JMenu)` : ajoute un menu à la barre de menu
- `JMenu getMenu(int)` : obtient le menu dont l'indice est fourni en paramètre
- `int getMenuCount()` : obtient le nombre de menu de la barre de menu
- `MenuElement[] getSubElements()` : obtient un tableau de tous les menus

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

les menus - suite

↬La classe `JMenuItem` encapsule les données d'un élément de menu.
Elle hérite de la classe `AbstractButton`.
Le comportement diffère de celui d'un bouton : avec la classe `JMenuItem`, le composant est considéré comme sélectionné dès que le curseur de la souris passe dessus.
↬La classe `JMenu` encapsule un menu qui est attaché à un objet de type `JMenuBar` ou à un autre objet de type `JMenu`.
Dans ce second cas, l'objet est un sous menu. `JMenu` hérite de la classe `JMenuItem`.

Exemple

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TestMenuSwing extends JMenuBar {
  public TestMenuSwing(){
ActionListener afficherMenuListener = new ActionListener() {
     public void actionPerformed(ActionEvent event) {
        System.out.println("Elément de menu [" + event.getActionCommand()
            + "] utilisé.");
    }};
// Création du menu Fichier
    JMenu fichierMenu = new JMenu("Fichier");
    JMenuItem item = new JMenuItem("Nouveau", 'N');
// le deuxième paramètre est un raccourci clavier
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
    item = new JMenuItem("Ouvrir", 'O');
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
    item = new JMenuItem("Sauver", 'S');
    item.addActionListener(afficherMenuListener);
    fichierMenu.insertSeparator(1);
    fichierMenu.add(item);
    item = new JMenuItem("Quitter");
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);}}
```

I. Introduction

II. Notions d'événements et de gestionnaires d'événements

III. Evénements

IV. Conteneur

V. Les gestionnaires de répartition

VI. Composant

1. les boutons
2. les menus
3. Composants de saisie de texte

Exemple

```java
import java.awt.*;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class EssaiSwing4 extends JFrame {
 public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    TestMenuSwing menu = new TestMenuSwing();
    frame.getContentPane().add(menu, BorderLayout.SOUTH);
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
 }}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Composants de saisie de texte

La classe abstraite `JTextComponent` est la classe mère de tout les composants permettant la saisie de texte.

Quelques méthodes :

- `void copy()` copie le contenu du texte et le mettre dans le presse papier système
- `void cut()` coupe le contenu du texte et le mettre dans le presse papier système
- `String getSelectectedText()` renvoie le texte sélectionné dans le composant
- `int getSelectionEnd()` renvoie la position de la fin de la sélection
- `int getSelectionStart()` renvoie la position du début de la sélection
- `String getText()` renvoie le texte saisie
- `String getText(int, int)` renvoie une portion du texte incluse à partir de la position donnée par le premier paramètre et la longueur donnée dans le second paramètre

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Composants de saisie de texte

- `void paste()` colle le contenu du presse papier système dans le composant
- `void select(int,int)` sélectionne une portion du texte dont les positions de début et de fin sont fournies en paramètres
- `void setCaretPosition(int)` déplace le curseur à la position dans le texte précisé en paramètre
- `void setSelectionEnd(int)` modifie la position de la fin de la sélection
- `void setSelectionStart(int)` modifie la position du début de la sélection
- `void setText(String)` modifie le contenu du texte

JTextField

La classe `JTextField` est un composant qui permet la saisie d'une seule ligne de texte simple.

```
import javax.swing.*;
public class EssaiSwing5 {
  public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel panneau = new JPanel();
    JTextField  testField = new JTextField ("mon texte");
    panneau.add(testField);
    f.getContentPane().add(panneau);
    f.setVisible(true);
  }
}
```

JPasswordField

La classe JPasswordField permet la saisie d'un texte dont tous les caractères saisis seront affichés sous la forme d'un caractère particulier ('*' par défaut). Cette classe hérite de la classe JTextField.

```
import java.awt.Dimension;
import java.awt.event.*;
import javax.swing.*;
public class EssaiSwing6 implements ActionListener {
   JPasswordField passwordField = new JPasswordField("");
   JPanel pannel = new JPanel();
   public static void main(String arg[]) {
     EssaiSwing6 jpf2 = new EssaiSwing6();
     jpf2.init();
  }
  public EssaiSwing6() {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    passwordField.setPreferredSize(new Dimension(100, 20));
    pannel.add(passwordField);
    JButton bouton = new JButton("Afficher");
    bouton.addActionListener(this);
    pannel.add(bouton);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
  public void actionPerformed(ActionEvent e) {
    System.out.println("texte saisi = " +
    String.copyValueOf(passwordField.getPassword()));
  }}
```

JTextArea

La classe `JTextArea` est un composant qui permet la saisie de texte simple en mode multi-lignes. Il ne peut contenir que du texte brut sans éléments multiples de formatage.
`JTexteArea` propose plusieurs méthodes pour ajouter du texte dans son modèle :

- soit fournir le texte en paramètre du constructeur utilisé
- soit utiliser la méthode `setText()` qui permet d'initialiser le texte du composant
- soit utiliser la méthode `append()` qui permet d'ajouter du texte à la fin de celui contenu dans le texte du composant
- soit utiliser la méthode `insert()` permet d'insérer un texte à une position données en caractères dans le texte du composant

La zone de texte augmente au fur et à mesure que l'on tape du texte à l'intérieur.
Si on veut fixer la dimension de la zonr de texte sans limiter sa capacité on doit ajouter cette zone à un contenant `JScrollPane`.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exemple

```
import java.awt.Dimension;
import javax.swing.*;
public class EssaiSwing8 {
public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();
    JTextArea  textArea = new JTextArea ("");
    JScrollPane scrollPane = new JScrollPane(textArea);
    scrollPane.setPreferredSize(new Dimension(200,70));
    pannel.add(scrollPane);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exemple : tic tac toe

```java
import java.awt.*;
import java.awt.Color;
import java.awt.event.*;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.BorderFactory;


public class TicTacToe {

public static final int LARGEUR = 320;
public static final int LARGEUR_PANNEAU = 100;
public static final int HAUTEUR_PANNEAU = 100;
public static final  int HAUTEUR = 320;
// attribut tableau de jeu
// qui reflète l'état du jeu
// 0 pour un case libre
//1 pour le premier joueur (rouge)
//2 pour le deuxième joueur (bleu)
int [ ][ ] tabJeu = new int [ 3 ][3 ];
int nbreEssai =0;
JFrame fenetreJeu = new JFrame("la fenetre pour jouer") ;
////////
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exemple : tic tac toe

```
////// classe interne pour définir un panneau
class PanneauCouleur extends JPanel implements ActionListener {
//attribut coordonnées du panneau
        int abs; int ord;
//attribut bouton pour choisir ce panneau
         JButton boutonJeu = new JButton("cliquez");
// constructeur
public PanneauCouleur(int i,int j){
        super();
        abs = i;ord = j;
Dimension dim= new Dimension(LARGEUR_PANNEAU,HAUTEUR_PANNEAU);
this.setBackground(Color.LIGHT_GRAY);
this.setLayout(new BorderLayout());
this.setSize(dim);
 this.setBorder(BorderFactory.createLineBorder(Color.BLACK));
 // bouton de jeu source de l'événement
 boutonJeu.addActionListener(this);
 //placement du bouton en bas du panneau
    this.add(boutonJeu, BorderLayout.SOUTH);
    }
```

Exemple : tic tac toe

```
////// verification du tableau de jeu selon le dernier coup
///// on vérifie la ligne ou la colonne ou la diagonale de celui
///// qui vient de jouer

  boolean verifLigne() {
boolean b = true;
for (int col=0; col<tabJeu.length; col++)
     { b=b && (tabJeu [ abs ][ col ]== tabJeu [ abs ][ ord ]);}
return b; }

 boolean verifColonne() {
boolean b = true;
for (int lig=0; lig<tabJeu.length; lig++)
     { b=b && (tabJeu [ lig ][ ord]== tabJeu [ abs ][ ord ]);}
return b; }

boolean verifDiagonale() {
boolean b = true;
if (abs!= ord) { return false; } // le joueur n'a pas joué sur la diagonale montante
else {for (int j=0; j<tabJeu.length; j++)
          { b=b && (tabJeu [ j ][ j ]== tabJeu [ abs ][ ord ]) ;}
      return b;  }
}
// autre diagonale
boolean verifDiagonale2() {
boolean b = true;
if ((abs+ord+1)!= tabJeu.length) // le joueur n'a pas joué sur la diagonale descendante
 { return false; }
else {for (int j=0; j<tabJeu.length; j++)
          { b=b && (tabJeu [ j ][tabJeu.length-j-1]== tabJeu [ abs ][ord]);}
      return b;}
}

// synthèse des vérifications
 boolean verif() {
 return ( verifLigne() || verifColonne() || verifDiagonale() || verifDiagonale2()); }
```

I. Introduction

II. Notions d'événements et de gestionnaires d'événements

III. Evénements

IV. Conteneur

V. Les gestionnaires de répartition

VI. Composant

1. les boutons
2. les menus
3. Composants de saisie de texte

Exemple : tic tac toe

```java
// méthode actionPerformed réalisée par l'écouteur panneau
public void actionPerformed(ActionEvent e){
// met la couleur du joueur dans le panneau correspondant
// et met le numéro du joueur dans le tableau de jeu
// et verifie si le joueur a gagne
int numJoueur;// variable pour affichage du gagnant
    nbreEssai+=1;
    if (nbreEssai%2== 1) {setBackground(Color.RED);tabJeu[abs][ ord ]=1;numJoueur=1;}
    else {setBackground(Color.BLUE);tabJeu[abs][ ord ]=2;numJoueur=2;}
     boutonJeu.setVisible(false); // le bouton devient invisible et indisponible
     if (verif()) {System.out.println("le joueur " + numJoueur + " a gagné");System.exit(0);}
      else if (nbreEssai==9)
      {System.out.println("le jeu est fini , il n'y a pas de gagnant");}
}
} // fin de le classe interne PanneauCouleur
```

Exemple : tic tac toe

```
/////// constructeur de la classe TicTacToe
  public TicTacToe() {
  fenetreJeu.setLayout(new GridLayout(3,3));
  fenetreJeu.setBackground(Color.BLACK);
  Dimension dim0= new Dimension(LARGEUR,HAUTEUR);
  fenetreJeu.setSize(dim0);
  fenetreJeu.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  //création et placement des 9 cases du jeu
  PanneauCouleur panneau;
  for(int i=0; i<tabJeu.length; i++){
  for (int j=0; j<tabJeu.length; j++){
  panneau = new PanneauCouleur(i,j);
  fenetreJeu.getContentPane().add(panneau);}
  fenetreJeu.setVisible(true);}
  // initialisation du tableau de jeu, tous les éléments valent 0 (aucun joueur)
        for (int i=0; i<tabJeu.length; i++){ for (int j=0; j<tabJeu.length; j++){
        tabJeu [ i ] [ j ] = 0;
   }}}
//////// méthode main
    public static void main(String[] args) {
        TicTacToe unJeu = new TicTacToe();
 }}
```

Exemple de menus

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
 class Fjframe extends JFrame implements ActionListener
 {  JPanel panel;
     JMenuBar mbar;
     JMenu m1;
     JMenu m2;
  public Fjframe()
    { setTitle("Test de JFrame");
       setSize(300,200);
       // gestion evenementielle de la fermeture de la fenêtre
       addWindowListener(new WindowAdapter()
           {public void windowClosing(WindowEvent e)
                  {System.exit(0);
                  } } ) ;
// insertion d'objets graphiques sur la fenêtre à partir d'un panneau
        panel = new JPanel();
        Container contentPane = getContentPane();
        panel.setBackground(Color.blue);
        contentPane.add(panel);
        // Ajout d'une barre de menus à la fenêtre
        mbar = new JMenuBar();
        m1 = new JMenu("Couleur fond");
        JMenuItem m11 = new JMenuItem("Jaune");
        m11.addActionListener(this); // installation d'un écouteur d'action
        m1.add(m11);  // ajout d'une option à un menu
        JMenuItem m12 = new JMenuItem("Rouge");
        m12.addActionListener(this); m1.add(m12);
        m2 = new JMenu("Couleur menu");
        JMenuItem m21 = new JMenuItem("Bleu");
        m21.addActionListener(this); m2.add(m21);
        JMenuItem m22 = new JMenuItem("Vert");
        m22.addActionListener(this); m2.add(m22);
        mbar.add(m1);  // ajout de menus à la barre de menus
        mbar.add(m2);
        setJMenuBar(mbar); }
```

I. Introduction

II. Notions d'événements et de gestionnaires d'événements

III. Evénements

IV. Conteneur

V. Les gestionnaires de répartition

VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exemple de menus

```
public void actionPerformed(ActionEvent evt)
  {  if (evt.getSource() instanceof JMenuItem)
           // gestion des événements liés aux menus
         { String ChoixOption = evt.getActionCommand();
            if (ChoixOption.equals("Jaune"))
            panel.setBackground(Color.YELLOW);
            else if (ChoixOption.equals("Rouge"))
            panel.setBackground(Color.RED);
            else if (ChoixOption.equals("Bleu"))
                         {mbar.setBackground(Color.BLUE);
                           m1.setBackground(Color.BLUE);
                           m2.setBackground(Color.BLUE);
                         }
            else if (ChoixOption.equals("Vert"))
                        {mbar.setBackground(Color.GREEN);
                          m1.setBackground(Color.GREEN);
                          m2.setBackground(Color.GREEN);
                         }; };}
public static void main(String[] args)
  {JFrame f =new Fjframe();
   f.setVisible(true);
   }
}
```

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exercices

exercice 9 : Refaire le jeu Tic Tac Toe en séparant clairement en deux classes le jeu et l'interface pour jouer.

exercice 10 : Concevoir une IHM pour un jeu de mémoire : dans une partie de la fenêtre une série de chiffres est affichée : chaque chiffre n'est visible qu'une fraction de seconde. Puis dans une autre partie de la fenêtre qui ne sera visible qu'une fois la série finie, l'utilisateur doit écrire cette série de chiffres proposée et on doit afficher s'il a réussi ou sinon combien d'erreurs il a commis.

exercice 11 : Concevoir une IHM pour un jeu de « métronome » : un affichage annonce que le joueur doit compter un certain nombre `nbrSec` de secondes (entre 5 et 12 par exemple) ; un chronomètre caché va être démarré et arrêté par le clic d'un bouton ; on affiche alors la durée comptée par le joueur. (La méthode `System.currentTimeMillis()` renvoie le nombre de secondes écoulées depuis le 1er Janvier 1970 et renvoie un type `long`)

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exercices

exercice 12 : Concevoir une IHM avec soit trois panneaux verticaux soit trois panneaux horizontaux qui afficheront les couleurs de drapeaux suivants : France, Pays-Bas, Belgique, Italie, Allemagne, Irlande, Arménie. Le choix du pays sera dans un menu. Vous pourrez utiliser `java.awt.Color` : cette classe contient des constantes pour les couleurs.

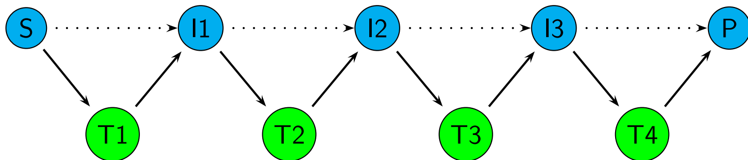
exercice 13 : Concevoir une IHM pour un jeu de master mind avec 6 couleurs. Vous pourrez utiliser

- `JComboBox` : cette classe propose un constructeur `JComboBox(Object[] items)` qui crée un bouton menant à un menu déroulant selon le tableau `items`
- `JOptionPane` : cette classe propose une fenêtre pop-up avec un texte ; elle peut être utilisée pour donner le résultat après chaque coup.

I. Introduction
II. Notions d'événements et de gestionnaires d'événements
III. Evénements
IV. Conteneur
V. Les gestionnaires de répartition
VI. Composant
1. les boutons
2. les menus
3. Composants de saisie de texte

Exercices

exercice 14 : On veut simuler une chaîne de traitement de 5 threads en parallèle selon le schéma suivant :



La tâche Source émet des messages, la tâche Puit reçoit des messages et les tâches intermédiaires recoivent de leur voisin de gauche et transmettent à leur voisin de droite. Chaque tâche se met en sommeil pour une durée aléatoire de 2s maximum après chaque réception ou émission.

Les tampons servent de stockage des messages devant être échangés.