Programmation Orientée Objet avancée avec Java

17 décembre 2019

Ce document est un support de cours qui ne se prétend ni exhaustif, ni exempt de toute erreur.
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 d'une classe.

Chapitre I – Interfaces – Classes abstraites

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

Chapitre I – Interfaces – Classes abstraites

Chapitre I – Interfaces – 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

Chapitre I – Interfaces – Classes abstraites

Définitions des Interfaces

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

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

Syntaxe :

```
interface  MonInterface
class  MaClasse implements MonInterface
```

Remarques

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

```
1 MonInterface x;
2 x = new C1();
3 x = new C2();
4 x = 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

```
1 MonInterface x;
2 x = new C1();
3 x = new C2();
4 x = 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

```
1 MonInterface x;
2 x = new C1();
3 x = new C2();
4 x = 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

```
1 MonInterface x;
2 x = new C1();
3 x = new C2();
4 x = 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

```
1 MonInterface x;
2 x = new C1();
3 x = new C2();
4 x = 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

```
1 MonInterface x;
2 x = new C1();
3 x = new C2();
4 x = 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

```
1 MonInterface v;
2 v=new C1();
3 v=new C2();
4 v=new C3();
```

Exemple d'usage

Supposons que l'on a défini trois classes `Film`, `Anime`, `Documentaire`.
Dans une *autre classe* trois méthodes sont écrites avec un argument de chacune de ces classes : `int duree(Anime a); String aLAfficheDe(Film 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`, `Anime`, `Documentaire`
- à écrire des classes implémentant respectivement `Film`, `Anime`, `Documentaire`, `Film et Documentaire`, `...`

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

L'interface : `Cloneable`

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

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

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

Par exemple, si une classe A contient :

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

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

- 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 que l'attribut t de l'instance a

Exemple

```
1  class EssaiClone implements Cloneable{
2  int n;
3  int[] t = {1,2,3};
4
5  public Object clone() throws CloneNotSupportedException {
6  return super.clone();}
7
8  public static void main(String[] arg)
9      throws CloneNotSupportedException{
10     EssaiClone o= new EssaiClone();
11     o.n=0;
12     EssaiClone oc= (EssaiClone)o.clone();
13     System.out.println(oc.n+" "+ oc.t);
14         oc.n = 1;
15         System.out.println("original="+o.n + " "
16                            + o.t+ " copie= "+oc.n+" "+ oc.t);
17 }}
```

Exécution :

0 [I@eb42cbf

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

Exemple

```
1  class EssaiClone implements Cloneable{
2  int n;
3  int[] t = {1,2,3};
4
5  public Object clone() throws CloneNotSupportedException {
6  return super.clone();}
7
8  public static void main(String[] arg)
9      throws CloneNotSupportedException{
10     EssaiClone o= new EssaiClone();
11     o.n=0;
12     EssaiClone oc= (EssaiClone)o.clone();
13     System.out.println(oc.n+" "+ oc.t);
14         oc.n = 1;
15         System.out.println("original="+o.n + " "
16                             + o.t+ " copie= "+oc.n+" "+ oc.t);
17 }}
```

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

```
1  class EssaiCloneProfondeur implements Cloneable{
2  int n;
3  int[] t = {1,2,3};
4  public Object clone() throws CloneNotSupportedException {
5  EssaiCloneProfondeur o =(EssaiCloneProfondeur) super.clone();// cast
6   o.t = new int[this.t.length];
7  for (int k=0;k<this.t.length;k++) o.t[k]=this.t[k];
8  return o;}
9
10 public static void main(String[] arg)
11     throws CloneNotSupportedException{
12 EssaiCloneProfondeur o= new EssaiCloneProfondeur();
13 o.n=0;
14 EssaiCloneProfondeur oc= (EssaiCloneProfondeur)o.clone();
15 oc.t[0]=12;
16 System.out.println("oc.n="+oc.n+"o.n="+o.n+" oc[0]="+
17   oc.t[0]+"  o[0] ="+ o.t[0]);
18 }}
```

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

requis de la méthode clone()

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

requis de la méthode clone()

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

requis de la méthode clone()

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

méthode `equals`

La réécriture de la méthode `public boolean equals(Object obj)` dans une classe `A` doit suivre les étapes suivantes :

- si obj==null on renvoie faux
- si obj.getClass()!=this.getClass() on renvoie faux
- si obj==this on renvoie vrai
- maintenant on peut effectuer un transtypage de `obj` sur la classe `A` par A objA=(A) obj;
- on teste alors l'égalité
  - avec equals pour chaque attribut objet de `this` et de `objA`
  - avec == pour chaque attribut primitif de `this` et de `objA`

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

Exemple

```
1  class EssaiCloneCompProfondeur implements Cloneable,
2                                                  Comparable{
3  int n;
4  int[] t = {1,2,3};
5  // voir EssaiCloneProfondeur
6  int compTableau(int[] tab) {
7  int lThis = this.t.length;
8  int lTab = tab.length;
9  int l;
10 if(lThis<lTab) l=lThis; else l=lTab;
11 for (int k=0;k<l;k++) {if (this.t[k]<tab[k]) return -1;
12 else {if (tab[k]<this.t[k]) return 1; };}
13 return 0;
14 }
15 public int compareTo (Object obj){
16   if (((EssaiCloneCompProfondeur)obj).n< this.n) return 1;
17   else {if (((EssaiCloneCompProfondeur)obj).n> this.n) return -1;
18   else return this.compTableau(((EssaiCloneCompProfondeur) obj).t);}
19 }}
20 // equals doit être compatible avec return 0 de compareTo
```

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

L'interface : `Comparator< >`

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

Exemple I

```
1  public class EssaiComparableComparator implements Comparable{
2  int n;
3  int[] t ;
4  public EssaiComparableComparator(int a, int[] tab){n=a;t=tab;}
5  int compTableau(int[] tab) {
6  int lThis = this.t.length;
7  int lTab = tab.length;
8  int l;
9  if(lThis<lTab) l=lThis; else l=lTab;
10 for (int k=0;k<l;k++) {if (this.t[k]<tab[k]) return -1;
11 else {if (tab[k]<this.t[k]) return 1; };}
12 return 0;
13 }
14 public int compareTo (Object obj){
15   if (((EssaiComparableComparator)obj).n< this.n) return 1;
16   else {
17   if (((EssaiComparableComparator)obj).n> this.n) return -1;
18   else return this.compTableau(((EssaiComparableComparator) obj).t);}
19 }}
20
21 public class TabComparator implements
22    Comparator<EssaiComparableComparator>{
23 public int compare(EssaiComparableComparator o1,
24                            EssaiComparableComparator o2){
25 if(o1.t.length<o2.t.length) return -1;
26 else if(o1.t.length>o2.t.length) return 1;
27 else return 0;}} // on compare uniquement les longueurs
28
29 public class TabSommeComparator implements
30    Comparator<EssaiComparableComparator>{
31 public int compare(EssaiComparableComparator o1,
32                             EssaiComparableComparator o2){
33 int s1=0; int s2=0;
34 for(int i=0;i<o1.t.length;i++) s1=s1+o1.t[i];
35 for(int j=0;i<o2.t.length;j++) s2=s2+o1.t[j];
36 if(s1<s2) return -1; else if(s2<s1) return 1; else return 0;}}
```

Exemple II

```
37  // on compare uniquement la somme des éléments
```

Exemple

```
1  class Test{
2     public static void main(String[] args) {
3       TabComparator tComparator = new TabComparator();
4       TabSommeComparator tSomComp = new TabSommeComparator();
5       int[] t1 ={1,2,3,4,5};
6       int[] t2 = {7,8,9};
7       EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
8       EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
9       if(tComparator.compare(e1,e2)<0) System.out.println("e1");
10      else System.out.println("e2");
11      if(tSomComp.compare(e1,e2)<0) System.out.println("e1");
12      else System.out.println("e2");
13    }
```

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

```
1  class Test{
2     public static void main(String[] args) {
3       TabComparator tComparator = new TabComparator();
4       TabSommeComparator tSomComp = new TabSommeComparator();
5       int[] t1 ={1,2,3,4,5};
6       int[] t2 = {7,8,9};
7       EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
8       EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
9       if(tComparator.compare(e1,e2)<0) System.out.println("e1");
10      else System.out.println("e2");
11      if(tSomComp.compare(e1,e2)<0) System.out.println("e1");
12      else System.out.println("e2");
13    }
```

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

```
1  class Test{
2     public static void main(String[] args) {
3       TabComparator tComparator = new TabComparator();
4       TabSommeComparator tSomComp = new TabSommeComparator();
5       int[] t1 ={1,2,3,4,5};
6       int[] t2 = {7,8,9};
7       EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
8       EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
9       if(tComparator.compare(e1,e2)<0) System.out.println("e1");
10      else System.out.println("e2");
11      if(tSomComp.compare(e1,e2)<0) System.out.println("e1");
12      else System.out.println("e2");
13    }
```

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

```
1  class Test{
2     public static void main(String[] args) {
3       TabComparator tComparator = new TabComparator();
4       TabSommeComparator tSomComp = new TabSommeComparator();
5       int[] t1 ={1,2,3,4,5};
6       int[] t2 = {7,8,9};
7       EssaiComparableComparator e1 = EssaiComparableComparator(10, t1);
8       EssaiComparableComparator e2 = EssaiComparableComparator(37, t2);
9       if(tComparator.compare(e1,e2)<0) System.out.println("e1");
10      else System.out.println("e2");
11      if(tSomComp.compare(e1,e2)<0) System.out.println("e1");
12      else System.out.println("e2");
13    }
```

L'exécution donne :
e2
e1
Remarque : pour trier une collection (cf. chapitre suivant) on peut passer en 2ème argument des méthodes `sort`, `min`, `max` ... une classe qui implémente `Comparator`.
Par exemple `Collections.sort(c, tComparator)` où c serait une collection (cf chapitre suivant) d'instances de la classe `EssaiComparableComparator`.

Les classes abstraites

Une classe est abstraite si

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

Une méthode abstraite

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

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

Exemple

On définit une classe abstraite `Quadrilatère` avec

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

.

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

```
1  public abstract class Quadrilatere {
2  double a,b,c,d;
3  public abstract double surface();// prototype
4  public  double perimetre(){return (this.a+this.b+this.c+this.d)};}
5  public class Trapèze extends Quadrilatere {
6  double h;
7  public Trapèze(double x, double petiteBase, double z,
8                          double grandeBase, double haut)
9  {a=x;b=petiteBase;c=z; d= grandeBase; h=haut;}
10 public  double surface(){return (h*(b+d)/2);};
11 }
12 public class Rectangle extends Quadrilatere {
13 public Rectangle(double larg, double longueur){
14    a=longueur; b=larg;c=a;d=b
15 }
16 public  double surface(){return (a*b)};
17 }
```

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

- 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 – Genericite

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

Chapitre II – Collections – Genericite

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.

Exemple

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

```
1  public class PaireEntier {
2  private int premier;
3  private int second;
4  public PaireEntier(int x, int y){
5  premier =x ; second = y;}
6  public int getPremier(){return this.premier;}
7  public void setPremier(int x){this.premier=x;}
8  public int getSecond(){return this.second;}
9  public void setSecond(int x){this.second=x;}
10 public void interchanger(){
11 int temp = this.premier;
12 this.premier = this.second;
13 this.second = temp;}
14 }
```

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.

```
1  public class PaireObjet {
2  private Object premier;
3  private Object second;
4  public PaireObjet(Object x, Object y){
5  premier =x ; second = y;}
6  public Paire(){}
7  public Object getPremier(){return this.premier;}
8  public void setPremier(Object x){this.premier=x;}
9  public Object getSecond(){return this.second;}
10 public void setSecond(Object y){this.second=y;}
11 public void interchanger(){
12 Object temp = this.premier;
13 this.premier = this.second;
14 this.second = temp;}
15 }
```

Inconvénients :

- les deux attributs peuvent être 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.

```
1  public class PaireObjet {
2  private Object premier;
3  private Object second;
4  public PaireObjet(Object x, Object y){
5  premier =x ; second = y;}
6  public Paire(){}
7  public Object getPremier(){return this.premier;}
8  public void setPremier(Object x){this.premier=x;}
9  public Object getSecond(){return this.second;}
10 public void setSecond(Object y){this.second=y;}
11 public void interchanger(){
12 Object temp = this.premier;
13 this.premier = this.second;
14 this.second = temp;}
15 }
```

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*.

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

```
1  A a = new A();
2  B b = new B();
3  PaireObjet p = new PaireObjet(a,b);
4  A a2 = (A) p.getPremier(); // downcasting ok
5  p.interchanger();
6  A a2 = (A) p.getPremier(); // erreur
```

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*.

```
1  public class Paire<T> {
2  private T premier;
3  private T second;
4  public Paire(T x, T y){ // en-tête du constructeur sans <T>
5  premier =x ; second = y;}
6  public Paire(){}
7  public T getPremier(){return this.premier;}
8  public void setPremier(T x){this.premier=x;}
9  public T getSecond(){return this.second;}
10 public void setSecond(T y){this.second=y;}
11 public void interchanger(){
12 T temp = this.premier;
13 this.premier = this.second;
14 this.second = temp;}
15 }
```

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

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

plusieurs types paramètres

On peut utiliser plusieurs types paramètres

```
1  public class PaireD<T,U> {
2  private T premier;
3  private U second;
4  public PaireD(T x, U y){ // en-tête du constructeur sans <T,U>
5  premier =x ; second = y;}
6  public PaireD(){}
7  public T getPremier(){return this.premier;}
8  public void setPremier(T x){this.premier=x;}
9  public U getSecond(){return this.second;}
10 public void setSecond(U y){this.second=y;}
11 }
12 ...
13 PaireD<Integer, String> p = PaireD<Integer, String>(1, "bonjour");
```

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.

```
1 public class Paire<T> {
2 ...
3 T var ; // oui
4 T var = new T(); //non
5 T[] tab ; // oui
6 T[] tab = new T[10];// non
```

méthodes et généricité

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

```
1 public class UneClasseGenerique<T>{
2 ...
3 public static void methodeDeClasse(){
4   T var ; // erreur à la compilation
5 ...
6 }}
```

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.

```
1  public class ClasseA{
2  ...
3  public <T> T premierElement(T[] tab){
4  return tab[0];} // méthode d'instance
5  //<T> est placé après les modificateurs et avant le type renvoyé
6  public static <T> T dernierElement(T[] tab){
7  return tab[tab.length-1];} // méthode de classe
8  //<T> est placé après les modificateurs et avant le type renvoyé
9  ...
10 }
```

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

```
1  ClasseA a = new ClasseA();
2  String[] t = {"game", "of", "thrones"};
3  System.out.println(a.<String> premierElement(t));
4  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.

```
1  public class Paire<T> {
2  private T premier;
3  private T second;
4  public Paire(T x, T y){ // en-tête du constructeur sans <T>
5  premier =x ; second = y;}
6  public Paire(){}
7  public T getPremier(){return this.premier;}
8  public void setPremier(T x){this.premier=x;}
9  public T getSecond(){return this.second;}
10 public void setSecond(T y){this.second=y;}
11 public void interchanger(){
12 T temp = this.premier;
13 this.premier = this.second;
14 this.second = temp;}
15 public <U> void  voir(U var){
16 System.out.println("qui est là ?" + var);
17 System.out.println("le premier est " + this.premier);}
18 }
19 ...
20 Paire<Integer> p = new Paire<Integer>(1,2);
21 p .<String> voir("un ami");
```

exercices

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

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 :

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

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

```
1 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`.

Généricité et héritage

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

```
1 public class Triplet<T> extends Paire<T>{
2 T troisieme;
3 ...}
```

- 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

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

Collections

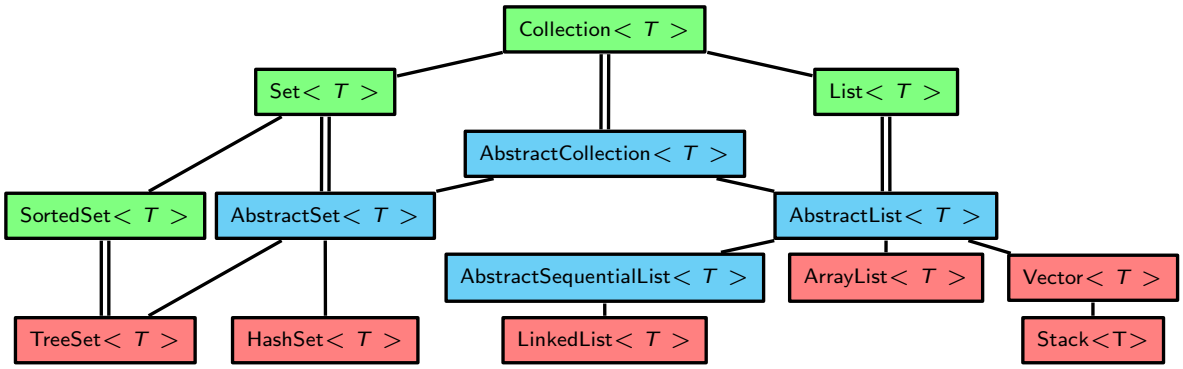
Java propose plusieurs moyens de manipuler des ensembles d'objets : on a vu les tableaux dont l'inconvénient est de ne pas être dynamique vis à vis de leur taille.
Java fournit des interfaces qui permettent de gérer des ensembles d'objets dans des structures qui peuvent être parcourues.
Ce chapitre donne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

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

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

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`.

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`.

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`.

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.

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()`.

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

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

méthode `toString()`

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

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

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.

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`.

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).

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

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

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`.

```
1  public class PersonneMedicalise extends Personne
2                    implements Comparable {
3  ....
4  public int compareTo(Object p){
5  if (this.getAge()> ((PersonneMedicalise)p).getAge()) return -1; else
6  if (this.getAge()< ((PersonneMedicalise)p).getAge()) return 1; else
7  if (this.getPoids()> ((PersonneMedicalise)p).getPoids()) return -1; else
8  if (this.getPoids()< ((PersonneMedicalise)p).getPoids()) return 1;
9  else return 0;
10 }}
11
12 import java.util.*;
13 public class EnsPersonneMedicale {
14 ArrayList <PersonneMedicalise> e;
15 public EnsPersonneMedicale() {}
16 ...
17 public  PersonneMedicalise quiEstEnDanger(){
18 Collections.sort(e);
19 return(e.get(0));}
20 public int ageMoyen(){
21 Iterator <PersonneMedicalise> it = e.iterator();
22 int a=0;
23 while (it.hasNext()) a= a+ it.next().getAge();
24 if (e.size()>0) return (a/e.size());else return 0;}}
```

Exercices I

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.

Exercices I

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

exercice 6 :

1. Définir une classe `Boite` qui a pour attributs deux entiers `poids` et `volume`.

Exercices II

2. Définir une classe `Caisse` qui a pour attributs
   - une collection de type `ArrayList` `listeBoite` contenant des objets de type `Boite`,
   - deux entiers `poidsMax` et `VolumeMax` qui représentent respectivement le poids maximal que la `Caisse` peut supporter et le volume maximal que la `Caisse` peut contenir,
   - deux entiers `poidsContenu` et `volumeRestant` qui représentent respectivemement le poids total des `Boite` dans `listeBoite` et le volume non occupé par les `Boite` de `listeBoite`.

   Dans cette classe vous écrirez
   - une méthode `public boolean ajouter(Boite b)` qui ajoute `b` à `listeBoite` à condition que le poids de `b` plus le poids des `Boite` de `listeBoite` ne dépasse pas `poidsMax` et que le volume restant soit suffisant pour mettre `b` dans la `Caisse this` - le booléen retourné vaut `true` si on a pu ajouter `b` et `false` sinon,
   - une méthode `public void retirer(Boite b)` qui retire `b` de `listeBoite`