

Introduction aux collections Java

Ensimag 2A 2014-15 - TP POO

Résumé

Ce document présente le concept des collections **Java**, les principales classes et leur utilisation. Avec les exemples donnés, ceci est normalement suffisant pour la réalisation du TP ; vous n'utiliserez d'ailleurs pas tout ce qui est présenté.

Il s'agit ici de *pédagogie inversée* : après avoir d'abord découvert de vous-même ces notions par la pratique, les collections seront ensuite abordées à nouveau plus en profondeur dans la fin du cours.

1 Principe

Les *collections Java* sont un ensemble de classes définissant des structures de données efficaces pour stocker, rechercher et manipuler des objets. De nombreuses structures existent couvrant les besoins les plus courants : séquences d'éléments, ensembles quelconques, ensembles ordonnés, queues (files, piles, files à priorité) ou encore des dictionnaires associatifs entre des clés et des valeurs. Ces « types de données abstraits » sont spécifiés dans des **interface Java** de haut niveau, puis implantés dans différentes classes concrètes. Les collections sont génériques : elles permettent de stocker des objets de tout type (**Object**, **String**, **MaClasse**, ...). Il est aussi possible de gérer des types de base via des classes « wrapper » (**Integer** pour **int**, **Double** pour **double**, ...). Le choix d'une collection dépend de l'utilisation recherchée et des coûts des opérations principalement réalisées.

Il existe en fait deux hiérarchies de classes :

- celle des collections proprement dites, qui sont filles de l'interface **Collection<E>** (figure 1). Elles permettent de stocker des éléments.
- celle des dictionnaires associatifs contenant des couples **<clé,valeur>**, dont la racine est l'interface **Map<K,V>** (figure 2).

Important : avant de se lancer dans le développement éventuel de vos propres structures de données, il est toujours préférable de d'abord chercher parmi les collections celles qui répondent le mieux à vos besoins. Dans la majorité des cas les structures adéquates existent déjà, qui plus est implantées de manière efficace et validées.

Ce document présente les principes d'utilisation des principales collections. De nombreuses ressources sont également disponibles, notamment :

- l'API <http://docs.oracle.com/javase/7/docs/api> (indispensable !). Toutes les classes et interfaces discutées ici sont dans le paquetage **java.util** ;
- les *Java Tutorials* : <http://docs.oracle.com/javase/tutorial/collections/index.html>

2 Parcours d'une collection : itérateur et for each

Le mécanisme d'*itération* permet de parcourir séquentiellement tous les éléments d'une collection, de manière uniforme et efficace.

Toute collection possède une méthode **Iterator<E> iterator()** qui retourne un *itérateur* permettant d'accéder aux éléments un par un en « avançant » dans la collection. Initialement, l'itérateur est placé « avant » le premier élément. A chaque accès, le prochain élément est retourné et l'itérateur avance à l'élément suivant. Lors du parcours complet d'une collection, chaque élément est retourné une et une seule fois, dans un ordre dépendant du type exact de la collection.

Les deux méthodes principales d'un itérateur sont :

- **public boolean hasNext()** qui retourne **true** s'il reste des éléments dans l'itération ;
- **public E next()** qui retourne le prochain élément et avance dans l'itération. S'il n'y a plus d'élément, une **NoSuchElementException** est levée.

Le schéma d'utilisation est toujours le même, quelle que soit la collection :

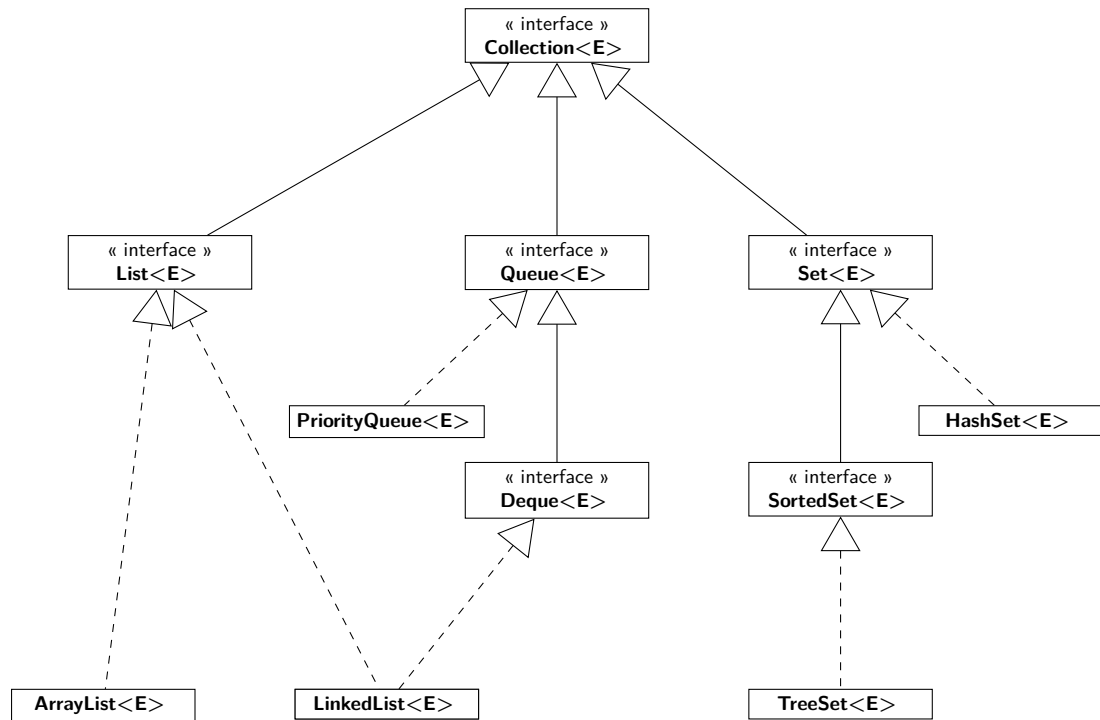


FIGURE 1 – Hiérarchie des principales collections Java : des interfaces (spécifications purement abstraites) et les classes concrètes utilisables en pratique : **ArrayList<E>**, **LinkedList<E>**, **PriorityQueue<E>**, **HashSet<E>** et **TreeSet<E>**.

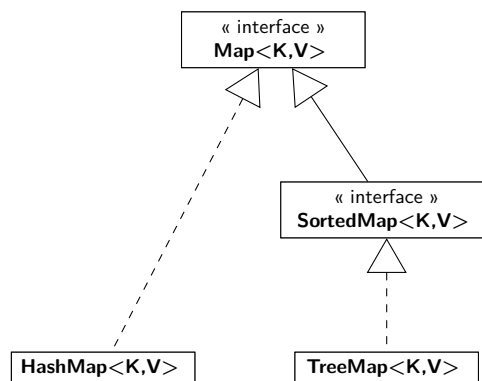


FIGURE 2 – Hiérarchie des principaux dictionnaires associatifs, définissant essentiellement deux classes concrètes : **HashMap<K,V>** et **TreeMap<K,V>**.

```

1 Collection<E> coll = ...;           // Collection existante, de type quelconque
2
3 Iterator<E> it = coll.iterator();   // Crée un nouvel itérateur,
4                                     // initialisé AVANT le 1er élément de coll
5 while (it.hasNext()) {              // Tant qu'il reste des éléments
6     E e = it.next();                 // Récupère le prochain élément et avance
7     ...                              // Traiter e
8 }

```

Les collections, comme les tableaux, peuvent aussi être parcourues à l'aide du mécanisme de « *for each* » :

```

1 for (E e : coll) {                  // Pour tout élément e de coll
2     ...                              // Traiter e
3 }

```

Par rapport au parcours avec un itérateur, un *for each* traite systématiquement *tous* les éléments de la collection. L'itérateur permet aussi d'enlever un élément pendant le parcours (la classe `Iterator` possède une méthode `remove()`, optionnelle, qui retire de la collection le dernier élément retourné par `next()`).

3 Principales collections

Nous ne décrivons ici que les collections usuelles principales.

3.1 Séquences : List

L'interface `List<E>` définit une séquence d'éléments, indexés du 1^{er} au dernier par leur position dans la séquence (un entier de 0 à `size() - 1`). Les méthodes permettent d'insérer au début, à la fin, à une position précise. Lors d'une itération, les éléments sont naturellement parcourus dans l'ordre de la séquence. Les deux principales classes réalisant cette interface sont :

1. `LinkedList<E>` : basée sur une liste doublement chaînée, cette classe est intéressante en cas d'ajouts et suppressions en début ou milieu de séquence ;

```

1 // Exemple : une LinkedList de points
2 LinkedList<Point> lPoints = new LinkedList<Point>();
3 lPoints.add(new Point(0, 0)); // par défaut, ajout en fin
4 lPoints.addFirst(new Point(1, 1));
5 lPoints.add(new Point(2, 2));
6 System.out.println(lPoints); // [(1,1), (0,0), (2,2)]
7 lPoints.remove(1);           // enlève la valeur en position 1
8 System.out.println(lPoints); // [(1,1), (2,2)]

```

2. `ArrayList<E>` : basée sur un tableau redimensionnable et donc à cout constant pour les accès direct à un élément en fonction de sa position (i^{ème}). Par contre, les couts des insertions/suppressions en position quelconque sont linéaires (décalage des valeurs aux indices suivants).

```

1 // Exemple : un ArrayList de Character (auto-boxing sur des char)
2 ArrayList<Character> lettres = new ArrayList<Character>();
3 for (char c = 'a'; c <= 'z'; c++) {
4     lettres.add(c); // ajout en fin
5 }
6 lettres.set(4, 'E'); // remplace la 5ème lettre par sa majuscule
7 System.out.println("La 25eme lettre est " + lettres.get(24));
8 lettres.remove(6); // cout! (décalages)
9
10 // On affiche les 10 premières lettres
11 Iterator<Character> it = lettres.iterator();
12 int n = 0;
13 while (it.hasNext() && n++ < 10) {
14     char c = it.next();
15     System.out.print(c + " ");
16 } // affichage : a b c d E f h i j k

```

3.2 Files et piles : Queue, Deque

L'interface `Queue<E>` définit typiquement une file (FIFO), avec deux méthodes `add(E e)` qui ajoute en fin et `remove()` qui enlève et retourne le premier élément.

L'interface `Deque<E>` (*Double Ended Queue*) spécialise une `Queue<E>` avec des méthodes d'insertion, d'accès et de retrait en tête et queue de liste : `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, `getLast`. Ces méthodes permettent notamment d'utiliser `Deque` comme une pile (LIFO), en considérant par exemple la tête comme le sommet de la pile.

La classe de référence réalisant ces interfaces est `LinkedList<E>`. Les opérations principales sont à coût constant $O(1)$.

```
1 // Ex : file (FIFO) de valeurs entières
2 //   notez le Queue<Integer> et pas Queue<int>
3 //   => pour les types de base, il faut utiliser des classes "wrapper"
4 //   (Integer, Float, Double ...)
5 Queue<Integer> file = new LinkedList<Integer> ();
6 file.add(1);
7 file.add(2);
8 file.add(3);
9 System.out.println(file.remove()); // affiche 1
10 file.add(4);
11 while (!file.isEmpty()) {
12     System.out.println(file.remove()); // affiche 2, 3 puis 4
13 }
```

3.3 File à priorités

La classe `PriorityQueue<E>` est une file à priorité : l'ordre de sortie des éléments ne dépend plus de leur date d'insertion, comme une FIFO ou LIFO, mais d'une *priorité* entre éléments. En pratique les éléments doivent être munis d'une *relation d'ordre*, et l'élément le plus prioritaire (le prochain à sortir) est le plus petit selon cet ordre.

Deux solutions sont possibles pour définir cette relation d'ordre (un exemple est donné en annexe A) :

1. la classe `E` des éléments peut réaliser l'interface `Comparable<E>`, qui définit l'ordre « naturel » entre des instances de `E`. Elle doit donc redéfinir la méthode `public int compareTo(E e)` qui retourne une valeur négative/nulle/positive si `this` est plus petit/égal/plus grand que `e`.
2. il est aussi possible de déléguer la comparaison de deux objets `E` à une tierce classe qui réalise l'interface `Comparator<E>`. Celle-ci définit une seule méthode : `public int compare(E e1, E e2)` qui retourne une valeur négative/nulle/positive si `e1` est plus petit/égal/plus grand que `e2`. Si une classe fille de `Comparator` est donnée à la `PriorityQueue`, c'est elle qui effectue la comparaison des éléments même si le type `E` réalise `Comparable<E>`.

Cette seconde approche est surtout utilisée pour pouvoir comparer des éléments selon des critères différents. Il est par exemple possible de créer deux classes qui comparent des étudiants selon leur nom ou selon leur note de POO.

L'implantation de la classe `PriorityQueue<E>` repose sur un tas binaire. Les coûts des opérations d'insertion et d'accès/retrait de l'élément le plus prioritaire sont en $O(\log(n))$. Attention par contre, la recherche ou la suppression d'un élément quelconque sont possibles mais en $O(n)$.

3.4 Ensembles : Set

A la différence des séquences ou queues, un ensemble définit par l'interface `Set<E>` n'admet pas de doublons : un élément ne peut pas être présent deux fois dans la collection. Cette égalité entre éléments est testée via la méthode `boolean equals(Object o)` héritée de la super classe `Object`, qui doit de ce fait être correctement redéfinie dans la classe `E` des éléments du `Set`.

Ensemble quelconque La classe `HashSet<E>` réalise l'interface `Set<E>` avec une implémentation de type table de hachage. Le cout amorti des opérations principales (ajout, retrait, recherche) est en $O(1)$. L'itération retourne bien toutes les valeurs mais dans un ordre quelconque.

En plus de redéfinir `equals`, les éléments doivent aussi redéfinir une autre méthode de la classe `Object` : `public int hashCode()`, qui doit retourner une clé de hachage entière. Cette méthode doit être cohérente avec la redéfinition de l'égalité : si deux objets sont égaux au sens de `equals`, alors leur méthode `hashCode` doit retourner la même valeur.

Ensemble ordonné La classe `TreeSet<E>` définit un ensemble dont les valeurs sont *ordonnées*. Il est donc nécessaire de munir les éléments d'une *relation d'ordre* pour pouvoir les comparer. Comme précédemment, deux solutions sont possibles :

1. la classe `E` des éléments peut réaliser l'interface `Comparable<E>`. C'est donc la méthode `compareTo` qui est utilisée pour ordonner les éléments dans la collection.
2. un objet de type `Comparator<E>` peut être donné au `TreeSet`. C'est alors lui qui réalise la comparaison des éléments même si la classe `E` réalise `Comparable<E>`.

Quelle que soit la méthode utilisée, la comparaison doit être compatible avec l'égalité : si `e1.equals(e2)` (respectivement `!equals`) alors `e1.compareTo(e2)` et/ou `compare(e1, e2)` doivent retourner 0 (respectivement une valeur non nulle).

L'implantation de cette classe repose sur un arbre équilibré (de type rouge-noir), avec des coûts en $O(\log(n))$ pour les opérations principales.

Un exemple avancé d'ensemble ordonné est disponible en annexe [A](#).

4 Dictionnaires : Map

L'interface `Map<K,V>` spécifie des associations entre une clé de type `K` et une valeur de type `V`. Un `Map` ne peut pas contenir des clés identiques, et chaque clé n'est associée qu'à une et une seule valeur. Les opérations principales sont l'ajout d'un couple (`put(K key, V value)`), l'accès à une valeur via sa clé (`V get(K key)`), la recherche de clé ou de valeur, la suppression d'une valeur, etc.

L'interface `SortedMap<K,V>` étend `Map<K,V>`, en rajoutant une relation d'ordre sur les clés du dictionnaire. Deux classes principales existent :

1. `HashMap<K,V>` : une table avec hachage sur les clés ;
2. `TreeMap<K,V>` : un ensemble ordonné sur les clés (implanté là encore avec un arbre rouge-noir équilibré).

Comme pour les `Set`, les méthodes `equals`, `hashCode` ainsi qu'une relation d'ordre doivent être correctement définies, en particulier sur le type `K` des clés¹.

Parcours Un `Map` peut être parcouru de différentes manières. En fait trois méthodes renvoient les clés et/ou les valeurs dans des collections, qui peuvent à leur tour être itérées :

- la méthode `Collection<V> values()` retourne une collection (dont on ne connaît pas le type dynamique, mais ce n'est pas nécessaire) contenant toutes les valeurs.
- `Set<K> keySet()` retourne un ensemble contenant toutes les clés.
- `Set<Map.Entry<K,V>> entrySet()` retourne un ensemble de tous les couples `<clé,valeur>`. Ces couples sont de type `Map.Entry<K,V>`, où `Entry<K,V>` est une classe *interne* à `Map` fournissant principalement deux méthodes `K getKey()` et `V getValue()`.

Pour un `SortedMap`, les collections ci-dessus sont ordonnées suivant l'ordre défini sur les clés.

Un exemple de `Map` est disponible en annexe [B](#).

1. Même si elles ne sont pas utilisées par toutes les collections, il est recommandé de toujours redéfinir `hashCode` en même temps que `equals`, pour garantir la cohérence en cas d'utilisation d'une classe dans plusieurs collections.

A Exemple d'ordonnancement

L'exemple ci-dessous, contenu dans `ExempleOrdonnancement.java`, montre l'utilisation d'un ensemble ordonné `TreeSet`, avec notamment la redéfinition des méthodes `equals` et `hashCode` ainsi que l'utilisation des interfaces `Iterable` et `Iterator`

```
1  /*
2   * Exemple d'utilisation de Comparable et Comparator.
3   *
4   * On travaille sur les Daltons, définis par leur nom et leur taille.
5   * Ils sont stockés dans un ensemble ordonné, en utilisant différents
6   * moyens de définir la relation d'ordre.
7   *
8   * 1. la classe Dalton peut réaliser l'interface Comparable<Dalton>. Elle doit
9   *    donc définir une méthode compareTo(Dalton) qui compare une instance de
10  *    Dalton à un autre Dalton donné en paramètre.
11  *    Ceci définit "l'ordre naturel" pour comparer deux objets de type Dalton
12  *    (ici on ordonne sur la taille, puis sur le nom à taille égale)
13  *
14  * 2. on peut aussi déléguer la comparaison à une tierce classe, qui définit
15  *    simplement comment comparer deux Daltons.
16  *    Cette classe doit réaliser l'interface Comparator, et redéfinir une seule
17  *    méthode compare(Dalton, Dalton) qui compare deux objets.
18  *    On peut créer un comparateur pour chaque ordre souhaité. Ici on en crée
19  *    deux : comparaison sur le nom uniquement et sur la taille uniquement.
20  *
21  * Les méthodes de comparaisons entre deux objet o1 et o2 retournent une
22  * valeur entière négative si o1 < o2, , nulle si o1 == o2
23  * ou positive si o1 > o2.
24  */
25
26
27 import java.util.*;
28
29
30 public class ExempleOrdonnancement{
31
32     public static void main(String [] args) {
33
34         // Trois ensembles ordonnés suivant un ordre différent :
35         // - dans les deux premiers cas, on délègue la comparaison à un objet
36         //   externe de type Comparator
37         // - dans le dernier, on utilise l'ordre naturel de la classe Dalton
38         SortedSet<Dalton> daltonsParNom    = new TreeSet<Dalton> (new
39             CompareurNom());
40         SortedSet<Dalton> daltonsParTaille = new TreeSet<Dalton> (new
41             CompareurTaille());
42         SortedSet<Dalton> daltons         = new TreeSet<Dalton> ();    // ordre
43                                     naturel
44
45         Dalton joe = new Dalton("Joe", Taille.PETIT);
46         Dalton jack = new Dalton("Jack", Taille.MOYEN);
47         Dalton william = new Dalton("William", Taille.MOYEN);
48         Dalton averell = new Dalton("Averell", Taille.BETE);
49
50         // insertion ds un ordre quelconque, de toutes facons c'est trié
51         daltonsParNom.add(william);
52         daltonsParNom.add(joe);
53         daltonsParNom.add(averell);
54         daltonsParNom.add(jack);
55     }
56 }
```

```

53     daltonsParTaille.add(william);
54     daltonsParTaille.add(joe);
55     daltonsParTaille.add(averell);
56     daltonsParTaille.add(jack);
57
58     daltons.add(william);
59     daltons.add(joe);
60     daltons.add(averell);
61     daltons.add(jack);
62
63     System.out.println("Les Daltons, ordonnés par nom : ");
64     System.out.println(daltonsParNom);
65     System.out.println("    -> Bof...\n");
66
67     System.out.println("Les Daltons, ordonnés par taille : " +
68         daltonsParTaille);
69     System.out.println("    -> Argh, il en manque un! (ben oui, c'est un Set!)
70         \n");
71
72     System.out.println("Les Daltons, tout court : " + daltons);
73     System.out.println("    -> Là c'est mieux!\n");
74
75     System.out.println("\n(I'm a poor lonesome teacher...)");
76 }
77
78
79
80 enum Taille {
81     PETIT,
82     MOYEN,
83     BETE
84 }
85
86 /**
87  * Class Dalton, munie d'un ordre naturel.
88  */
89 class Dalton implements Comparable<Dalton> {
90     private String nom;
91     private Taille taille;
92
93     public Dalton(String nom, Taille taille) {
94         this.nom = nom;
95         this.taille = taille;
96     }
97
98     public String getNom() {
99         return nom;
100     }
101
102     public Taille getTaille() {
103         return taille;
104     }
105
106     @Override
107     public String toString() {
108         return nom + (" (" + taille.toString() + ")");
109     }
110
111     /**

```

```

112     * Ordre naturel : comparaison sur la taille d'abord, puis
113     * sur le nom.
114     * @return une valeur négative / nulle / positive si this est
115     *         plus petit / égal / plus grand que d.
116     */
117     @Override
118     public int compareTo(Dalton d) {
119         if (d == null)
120             throw new NullPointerException();
121
122         int dt = this.taille.ordinal() - d.taille.ordinal(); // ordre ds l'enum
123         if (dt < 0) { // this < d
124             return -1;
125         } else if (dt > 0) { // this > d
126             return 1;
127         } else { // même taille, on compare sur le nom
128             // (String realise Comparable<String>)
129             return this.nom.compareTo(d.nom);
130         }
131     }
132
133
134     // A partir du moment où une classe réalise Comparable, il est fortement
135     // recommandé de redéfinir equals (et hashCode) de manière cohérente
136     // par rapport à cet ordre :
137     //     o1.compareTo(o2) == 0 doit être identique à o1.equals(o2)
138     //     si o1.equals(o2) == true, alors o1.hashCode() == o2.hashCode()
139
140     @Override
141     public boolean equals(Object o) {
142         if (o instanceof Dalton == false) // vérifie aussi o != null
143             return false;
144
145         Dalton d = (Dalton) o;
146         return this.nom.equals(d.nom) && this.taille == d.taille;
147     }
148
149     // Deux instance identiques, au sens d'equals et de compareTo,
150     // doivent retourner un même hashCode
151     @Override
152     public int hashCode() {
153         // ici on délègue au nom (hashCode de String est déjà redéfinie)
154         return nom.hashCode();
155     }
156 }
157
158
159 /**
160  * Classe servant à comparer deux objets Dalton sur leur nom uniquement.
161  */
162 class CompareurNom implements Comparator<Dalton> {
163
164     @Override
165     public int compare(Dalton d0, Dalton d1) {
166         return d0.getNom().compareTo(d1.getNom());
167     }
168 }
169
170 /**
171  * Classe servant à comparer deux objets Dalton sur leur taille uniquement.
172  */

```



```

173 class CompareurTaille implements Comparator<Dalton> {
174
175     @Override
176     public int compare(Dalton d0, Dalton d1) {
177         return d0.getTaille().ordinal() - d1.getTaille().ordinal();
178     }
179 }

```

Et voici la trace du programme :

Les Daltons, ordonnés par nom :

[Averell (BETE), Jack (MOYEN), Joe (PETIT), William (MOYEN)]

-> Bof...

Les Daltons, ordonnés par taille : [Joe (PETIT), William (MOYEN), Averell (BETE)]

-> Argh, il en manque un ! (ben oui, c'est un Set donc pas de doublons !)

Les Daltons, tout court : [Joe (PETIT), Jack (MOYEN), William (MOYEN), Averell (BETE)]

-> Là c'est mieux !

(I'm a poor lonesome teacher...)

B Exemple d'utilisation d'un dictionnaire

Un exemple d'utilisation d'un HashMap.

```
1 import java.util.*;
2
3
4 public class ExempleMap {
5
6     public static void main(String[] args) {
7
8         // ex: dictionnaire associant des étudiants (clé, chaîne)
9         // à des notes (valeurs entières)
10        Map<String, Integer> annuaire = new HashMap<String, Integer> ();
11
12        String mc = new String("Matthieu");
13        String sb = new String("Sylvain");
14        String nc = new String("Nicolas");
15        annuaire.put(mc, 4);
16        annuaire.put(sb, 18);
17        annuaire.put(nc, 12);
18        annuaire.put(mc, 14);    // pas de doublons,
19                                // mais remplace l'ancienne valeur associée a mc
20
21        // et quelle est la note de Catherine?
22        Integer note = annuaire.get("Catherine");
23        System.out.println("La note de Catherine est : " + note);    // null!
24
25
26        // affichage avec toString();    ordre?
27        System.out.println("L'annuaire contient : " + annuaire);
28
29        // ensemble des clés, et des valeurs
30        Set<String> cles = annuaire.keySet();
31        System.out.println("Les clés sont : " + cles);
32
33        // collection des valeurs
34        Collection<Integer> notes = annuaire.values();
35        System.out.println("Les valeurs sont : " + notes);
36
37        // parcours avec un itérateur sur les couples
38        // (prévoir une aspirine pour la syntaxe...)
39        System.out.println("Les couples sont :");
40        Set<Map.Entry<String, Integer>> couples = annuaire.entrySet();
41        Iterator<Map.Entry<String, Integer>> itCouples = couples.iterator();
42        while (itCouples.hasNext()) {
43            Map.Entry<String, Integer> couple = itCouples.next();
44            System.out.println("\t" + couple.getKey() + " a la note " + couple.
45                               getValue());
46        }
47    }
48 }
```