Les collections

Table des matières

Introduction 2

L' interface `Collection` 3

L'interface `List` 3

L'interface `Set` 4

L'interface `SortedSet` 6

Ordre naturel 6

Comparator 8

Égalité et ordre 9

L'interface `Queue` 10

L'interface Map 10

Interface SortedMap 11

Wrappers méthodes 12

Algorithmes 13

Algorithmes applicables à toute `Collection` 13

Algorithmes propres aux `Lists` 13

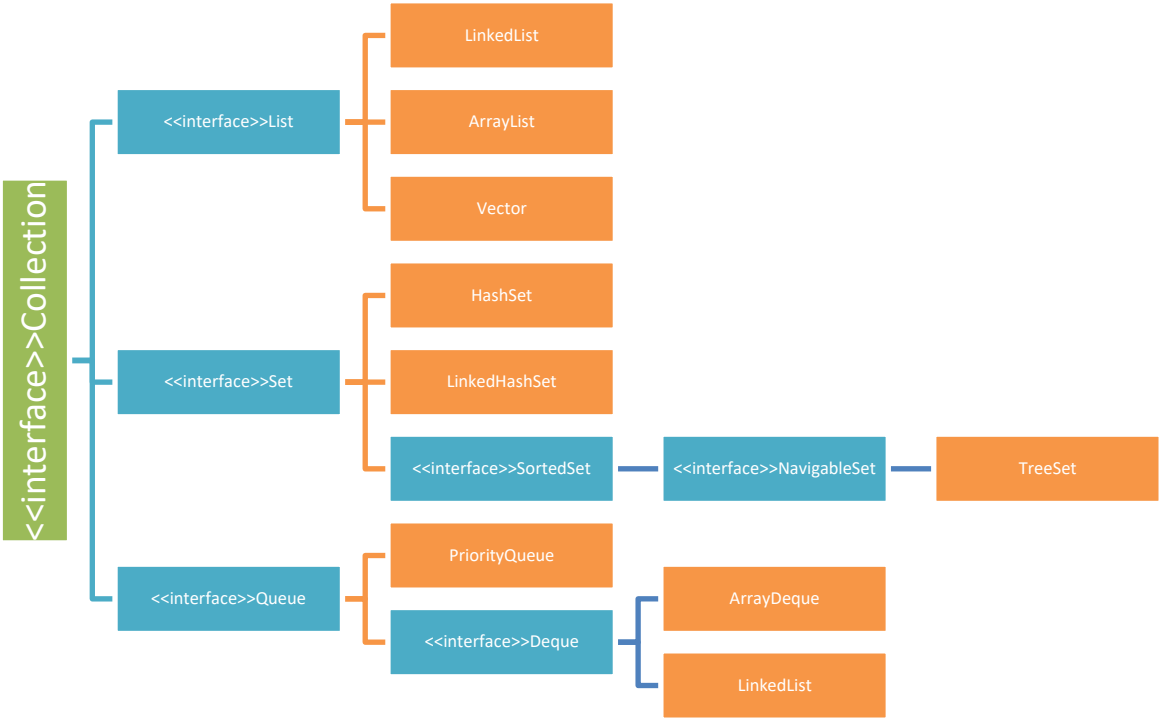
Introduction

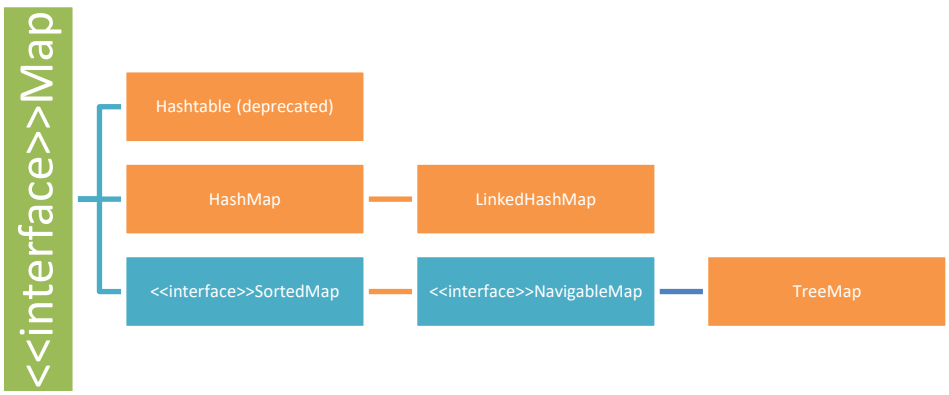
Une collection est une structure de données permettant de regrouper un groupe de données. En Java, un framework a été introduit fournissant déjà les structures de données les plus courantes

Ce framework est composé de trois choses :

- Interfaces : ce sont des types de données abstraits. **Ce seront toujours eux qu'on utilisera pour déclarer les collections utilisées.**
- Implémentations : ce sont des implémentations concrètes des interfaces.
- Algorithmes : en plus des méthodes proposées dans les interfaces et réalisées dans les implémentations, des algorithmes polymorphiques de tri, de recherche, etc. sont mis à votre disposition dans des classes utilitaires `Collections`, `Arrays`, …

Les schémas ci-dessous reprennent les principales interfaces et classes de ce framework. L'interface `Collection` englobe les structures de données permettant de gérer une collection d'objets tandis que l'interface `Map` englobe les structures de type dictionnaire c.-à-d. celles permettant de gérer des éléments de type paires clé/valeur.





L' interface `Collection`

L'interface `Collection`, qui se trouve la racine de la hiérarchie, regroupe toutes les méthodes communes aux structures permettant de garder une collection d'objets. On y trouve donc la déclaration des méthodes permettant d'ajouter un objet (`add(Object element)`[1]), retirer un objet (`remove(Object element`), de savoir si un objet est ou non dans la collection (`contains(Object element`), …[2] L'interface `Collection` étend l'interface `Iterable`.

Java ne fournit aucune implémentation de cette interface mais seulement des sous interfaces (`List`, `Set`, `Queue`) qui permettront de déterminer la façon dont ces méthodes doivent se comporter.

L'interface `List` représente des collections **séquentielles** d'objets. Elle est implémentée par les classes `ArrayList`, `LinkedList` et `Vector`.

L'interface `Set` représente des collections ne permettant **pas de doublons**. Elle est implémentée par la classe `HashSet` et étendue par l'interface `SortedSet` qui, en plus, garde ses éléments en ordre croissant. `SortedSet` est implémentée par la classe `TreeSet`.

L'interface `Queue` représente des collections sous forme de **file d'attente** ; l'ajout n'est possible qu'en fin de file. Elle est implémentée par la classe `PriorityQueue` et étendue par l'interface `Deque`. `Deque`, qui permet l'ajout en début et en fin de file, est implémentée par `ArrayDeque` et `LinkedList`. Il n'est pas possible d'ajouter un élément **`null`** dans une queue.

L'interface `List`

Une `List` est une `Collection` où les éléments sont gardés dans un ordre séquentiel. Elle peut contenir des doublons. Elle spécifie les contrats suivants pour les méthodes :

- `add(Object element)` doit ajouter l'élément en fin de liste.
- `remove(Object element)` doit supprimer la première occurrence de l'élément. Les autres éléments de la liste doivent rester dans le même ordre.
- `equals` et `hashCode` doivent être définis de sorte que deux listes ayant les mêmes éléments dans le même ordre sont considérées comme égales.

En outre, l'interface List ajoute aussi des méthodes permettant d'accéder à un élément sur base de sa position (`get(int index)`), de gérer l'élément à une position donnée (`add(int index ,Object object)`, …) de récupérer la position d'un élément (`indexOf(Object object)`), … Comme pour les tableaux, le premier élément de la liste a comme position 0.

L'interface List est entre autres implémentée par les classes `ArrayList` (implémentation utilisant un tableau), `LinkedList` (liste doublement chaînée) et `Vector` (implémentation utilisant un tableau). D'un point de vue performance, la classe `ArrayList` est souvent plus intéressante. La classe `Vector`, qui est synchronisée, a été modifiée depuis Java 1.2 afin d'implémenter `List` tout en gardant ses anciennes méthodes.

---

[1] Bien que `Object` soit mis comme type, les collections étant génériques, le type du paramètre passé à la méthode add (et d'autres méthodes) dépendra du type précisé lors de la déclaration de la variable de type Collection (par ex., si on a une variable de type `List<String>`, la méthode `add` prendra un `String` en paramètre).

[2] Pour connaître la liste complète des méthodes des différentes interfaces du framework ainsi que leur spécification, consultez la javadoc.

L'interface `Set`

Cette interface modélise la notion mathématique d'ensemble. Il étend l'interface Collection mais n'y rajoute aucune méthode. Elle redéfinit pourtant le contrat de certaines d'entre elles afin d'éviter la duplication d'éléments. Il n'y a pas d'ordre sur les éléments.

Les méthodes `equals` et `hashCode` doivent être redéfinies de sorte que deux ensembles ayant les mêmes éléments soient égaux

L'interface `Set` est implémentée par la classe `HashSet`. Comme toutes les implémentations des sous interfaces de `Collection`, cette classe a un constructeur prenant comme paramètre une `Collection`.

Les opérations sur des groupes d'éléments s'interprètent agréablement en termes ensemblistes :

`s1.containsAll(s2)` : $s2 \subseteq s1$

`s1.addAll(s2)` : $s1 = s1 \cup s2$

`s1.retainAll(s2)` : $s1 = s1 \cap s2$

`s1.removeAll(s2)` : $s1 = s1 - s2$

Dans la classe `HashSet`, la méthode `contains` utilise ce que renvoie le `hashCode` pour trouver l'« emplacement » où doit se trouver l'élément s'il est déjà présent. Ensuite, si un ou plusieurs éléments sont trouvés, la méthode `equals` est utilisée pour savoir s'il correspond à l'élément recherché. Par conséquent, quand on redéfinit les méthodes `equals` et `hashCode` pour une classe, il faut :

- faire en sorte que les méthodes `equals` et `hashCode` soient cohérentes c.-à-d. que deux objets qui sont égaux selon la méthode `equals` doivent avoir la même `hashCode` ;
- utiliser, dans la mesure du possible, des attributs immuables (c.-à-d. dont la valeur ne varie pas) pour définir l'égalité.

Prenons, par exemple, la classe `Boisson` ci-dessous :

```
public class Boisson {
    private final String nom;
    private final int contenance;
    private double prix;

    public Boisson(String nom, int contenance, double prix) {
        this.nom = nom;
        this.contenance = contenance;
        this.prix = prix;
    }

    public void setPrix(double prix) {
        this.prix = prix;
    }

    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Boisson boisson = (Boisson) o;
        return contenance == boisson.contenance &&
               nom.equalsIgnoreCase(boisson.nom);
    }

    public int hashCode() {
        return Objects.hash(nom.toLowerCase(),contenance);
    }
```

```
    public String toString() {
        return nom + " (" + contenance + " cl) : " + prix + " euros" ;
    }
}
```

On peut y voir que les méthodes `equals` et `hashCode` sont toutes les deux définies sur base du nom[3] et de la contenance et qu'il n'y a pas moyen de modifier ces attributs après la création d'une boisson. Les deux conditions énoncées ci-dessus sont donc bien respectées et il n'y a pas de risque d'incohérence si on stocke des objets de type `Boisson` dans un `Set`.

Par contre, quand ces conditions ne sont pas respectées, cela peut mener à des incohérences dans les données du `HashSet` comme illustré ci-dessous :

Si, dans l'exemple précédent, on remplace la méthode `hashCode` par celle donnée ci-dessous. Ici, la méthode `hashCode` n'est plus cohérente avec la méthode `equals` car il tient compte de tous les attributs alors que la méthode `equals` ne prend en compte que du nom et de la contenance.

```
    public int hashCode() {
        return Objects.hash(nom.toLowerCase(),contenance,prix);
    }
```

Lors de l'exécution du code ci-dessous, l'appel de la méthode `equals` renverra `true` car `boisson1` et `boisson2` ont les mêmes nom et contenance. Par conséquent, `boisson2` ne devrait pas pouvoir être ajouté au `Set boissons` car il s'y trouve déjà mais, comme son `hashCode` est différent de celui de `boisson1`, il sera quand même ajouté (les deux appels `add` renverront `true`) . Il y aura donc un doublon dans le `Set`, ce qui n'est pas autorisé.

```
Set<Boisson> boissons = new HashSet<>();
Boisson boisson1 = new Boisson("soda",25,2.6);
Boisson boisson2 = new Boisson("soda",25,2.8);
System.out.println("égale : " + boisson1.equals(boisson2));
System.out.println(boissons.add(boisson1));
System.out.println(boissons.add(boisson2));
```

Avoir une méthode `hashCode` qui ne renvoie pas la même valeur lorsque deux objets sont égaux (par la méthode `equals`) est toujours une erreur !

Comme second exemple, on considère toujours la classe Boisson mais dans laquelle les méthodes `equals` et `hashCode` sont toutes les deux définies sur base du `nom` et du `prix` mais ce dernier peut varier.

```
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Boisson boisson = (Boisson) o;
        return Double.compare(boisson.prix, prix) == 0 &&
                nom.equalsIgnoreCase(boisson.nom);
    }

    public int hashCode() {
        return Objects.hash(nom.toLowerCase(),prix);
    }
```

[3] Le fait d'utiliser `equalsIgnoreCase` fait que des noms qui ne différeraient qu'au niveau de la casse seront considérés aussi comme égaux. Afin que deux boissons ayant même contenance et dont les noms différeraient uniquement au niveau de la casse aient le même hashcode, le nom est mis entièrement en minuscule dans le calcul du hashcode.

Lors de l'exécution du code qui suit, comme le prix de `boisson` a été modifié après l'avoir ajouté dans le `HashSet boissons`, son `hashCode` ne sera plus le même lors de l'appel de la méthode `contains` et, par conséquent, celle-ci renverra `false` alors qu'elle devrait renvoyer `true` puisque `boisson` est dans le `HashSet`.

```
Boisson boisson = new Boisson("soda",25,2.5);
Set<Boisson> boissons = new HashSet<>();
boissons.add(boisson);
boisson.setPrix(2.8);
System.out.println(boissons.contains(boisson));
```

Il est parfois nécessaire de définir, dans une classe, `equals` et `hashCode` sur des attributs qui peuvent varier mais, si c'est le cas et qu'on vaut garder un `HashSet` avec des objets de cette classes, il faut toujours veiller à ne jamais modifier les attributs sur lesquels `equals` et `hashCode` ont été définis pour les objets qui sont stockés dans le `HashSet`.

L'interface `SortedSet`

`SortedSet` est une extension de l'interface `Set`. La différence est que, dans un `SortedSet`, les éléments sont triés de façon croissante selon un ordre fourni à l'aide d'un `Comparator` ou, s'il n'y a pas de `Comparator` fourni, selon l'odre naturel si celui-ci existe. Dans le cas où il n'y a pas de `Comparator` fourni et pas d'ordre naturel, une exception se produit lorsqu'on essaie d'ajouter un élément dans le `SortedSet`.

L'itérateur permettra de parcourir les éléments dans l'ordre croissant. De plus, l'interface SortedSet définit des méthodes supplémentaire pour, par exemple, récupérer l'élément le plus petit de l'ensemble (`first()`), récupérer une vue contenant tous les éléments stricetment inférieur à un élément donné (`headSet(Object toElement)`), … Les vues sur des parties de l'ensemble sont des vues sur le `SortedSet` et donc une modification faite à une de ces parties, est répercutée sur le `SortedSet` de départ.

L'interface `SortedSet` est implémentée par la classe `TreeSet`. Cette classe possède trois constructeurs :

- Un constructeur sans paramètre. Il faut que les éléments ajoutés aient un ordre naturel
- Un constructeur qui prend en paramètre un `Comparator` indiquant l'ordre du tri.
- Un constructeur qui prend en paramètre un `SortedSet`. Ce `SortedSet` servira à remplir celui construit. Mais de plus, l'ordre du `SortedSet` passé sera préservé dans le `SortedSet` construit.

Remarque : La classe `TreeSet` implémente également l'interface `NavigableSet` qui est une extension de `SortedSet` définissant des méthodes supplémentaires pour, par exemple, récupérer un itérateur permettant de parcourir les éléments dans l'ordre décroissant (`descendingIterator()`), …

Ordre naturel

Pour qu'une classe possède un ordre naturel, il faut qu'elle implémente l'interface `Comparable<T>` où T correspond à la classe pour laquelle on veut définir cet ordre. Cette interface ne contient qu'une seule méthode :

```
public int compareTo(T o)
```

Cette méthode doit renvoyer un entier négatif, nul ou positif, selon que l'objet courant est inférieur, égal ou supérieur à l'objet `o` passé en paramètre.

Si on veut définir un ordre naturel pour la classe `Boisson` donnée dans le premier exemple de sorte qu'une boisson est strictement inférieure à une autre si son nom est avant celui de l'autre selon l'ordre lexicographique sans prendre en compte la casse ou si elles ont le même nom mais que sa contenance est strictement inférieure à celle de l'autre, on peut le faire comme suit :

```
//Le fait d'implémenter Comparable<Boisson> indique qu'on va définir un
//ordre pour les Boissons
public class Boisson implements Comparable<Boisson>{
    …

    //Implémentation de la méthode compareTo permettant de définir l'ordre
    public int compareTo(Boisson boisson) {
        int compare = this.nom.compareToIgnoreCase(boisson.nom);
        if (compare!= 0) return compare;
        return Integer.compare(this.contenance,boisson.contenance);
    }
}
```

Dans l'implémentation de la méthode `compareTo`, on compare d'abord les noms[4]. Si le résultat obtenu est différent de 0, alors les noms sont différents et il n'est pas nécessaire de comparer les contenances. Sinon, on renvoie le résultat de la comparaison des deux contenances.

Ci-dessous, le code de la méthode `main` :

```
SortedSet<Boisson> boissons = new TreeSet<>();
Boisson boisson1 = new Boisson("soda",25,2.6);
Boisson boisson2 = new Boisson("soda",50,4.2);
Boisson boisson3 = new Boisson("jus d'oranges",30,3.5);
Boisson boisson4 = new Boisson("soda",25,2.8);
Boisson boisson5 = new Boisson("jus d'ananas",25,2.8);
boissons.add(boisson1);
boissons.add(boisson2);
boissons.add(boisson3);
boissons.add(boisson4);
boissons.add(boisson5);
for (Boisson boisson : boissons) {
    System.out.println(boisson);
}
```

L'exécution de ce code aura comme résultat :

jus d'ananas (25 cl) : 2.8 euros
jus d'oranges (20 cl) : 2.6 euros
soda (25 cl) : 2.6 euros
soda (50 cl) : 4.2 euros

`boisson4` ne sera pas ajoutée car la méthode `compareTo` ne prend pas en compte le prix et qu'il a le même nom et la même contenance que `boisson1` et, par conséquent `boisson1.compareTo(boisson4)` renvoie 0. Pour le reste, les boissons apparaissent dans l'ordre voulu.

Quand on implémente la méthode `compareTo`, elle doit respecter certaines conditions afin qu'elle définisse bien un ordre :

- `x.compareTo(y)` et `y.compareTo(x)` doivent être de signes opposés pour tout x,y (cela revient à dire que x est supérieur ou égal à y si et seulement si y est inférieur ou égal à x).

[4] La méthode `compareToIgnoreCase` de la classe String est une méthode qui permet de définir l'ordre lexicographiqe sans prendre en compte la casse.

- Quels que soient x,y,z, si `x.compareTo(y)>0` et `y.compareTo(z)>0` alors il faut que `x.compareTo(z)>0` (cela revient à dire que si x est strictement supérieur à y et y est strictement supérieur à z, alors x doit être strictement supérieur à z.
- Quels que soient x,y,z, si `x.compareTo(y)==0`, alors `x.compareTo(y)` et `x.compareTo(z)` doivent être de même signe (Cela revient à dire que si x et y sont égaux, alors x est inférieur ou égal à z si et seulement si y est inférieur ou égal à z)

Remarque : De nombreuses classes java définissent un ordre naturel qui correspond à . C'est le cas, par exemple, de toutes les classes « wrappers » (`Byte`, `Integer`, `Double`, …), des classes `String`, `LocalDate`, `LocalDateTime`, …

Comparator

Lorsqu'il n'y a pas d'ordre naturel ou si on veut que les éléments soient triés dans le `SortedSet` dans un ordre qui ne correspond pas à l'ordre naturel, alors il faut passer, lors de la construction du `SortedSet`, un `Comparator`<T> où T correspond au type des éléments à ajouter dans le `SortedSet`. `Comparator<T>` est en fait une interface qui ne contient qu'une seule méthode :

```
public int compare(T o1,T o2)
```

Cette méthode doit renvoyer un entier négatif, nul ou positif, selon que le premier objet passé en paramètre (`o1`) est inférieur, égal ou supérieur au deuxième objet passé en paramètre (`o2`).

Par exemple, si on veut un `SortedSet` de `Boisson` dans lequel une boisson est inférieure à une autre si sa contenance est strictement inférieure à celle de l'autre ou si elles ont la même contenance mais que le nom est celui de l'autre selon l'ordre lexicographique sans prendre en compte la casse. Cela ne correspond plus à l'ordre qu'on a défini sur `Boisson`. Il faut donc définir un `Comparator<Boisson>` et le passer lors de la création du `TreeSet`.

Définition du `Comparator<Boisson>` :

```
public class ComparatorBoisson implements Comparator<Boisson> {
    @Override
    public int compare(Boisson o1, Boisson o2) {
        int compare =
                Integer.compare(o1.getContenance(),o2.getContenance());
        if (compare != 0) return compare;
        return o1.getNom().compareToIgnoreCase(o2.getNom());
    }
}
```

Dans le `main`, il faut simplement remplacer la première ligne par :

```
SortedSet<Boisson> boissons = new TreeSet<>(new ComparatorBoisson());
```

L'exécution du main affichera alors :

```
jus d'ananas (25 cl) : 2.8 euros
soda (25 cl) : 2.6 euros
jus d'oranges (30 cl) : 3.5 euros
soda (50 cl) : 4.2 euros
```

L'ordre est bien celui voulu.

Afin que la méthode `compare` définisse bien un ordre, son implémentation doit respecter des conditions analogues à celles du `compareTo`.

On aurait pu passer par une classe anonyme (pour ce qu'est une classa anonyme, consultez la théorie sur les classes internes) pour passer le `Comparator` au lieu de faire une classe à part. Remplacer la première ligne du `main` par le code ci-dessous, où le `Comparator` a été instancié à l'aide d'une classe anonyme, produirait le même résultat que précédemment.

```
Comparator <Boisson> comparatorBoisson = new Comparator<Boisson>() {
    public int compare(Boisson o1, Boisson o2) {
        int compare =
                Integer.compare(o1.getContenance(),o2.getContenance());
        if (compare != 0) return compare;
        return o1.getNom().compareToIgnoreCase(o2.getNom());
    }
};
SortedSet<Boisson> boissons = new TreeSet<>(comparatorBoisson);
```

Égalité et ordre

Comme dans un `Set`, il ne peut pas y avoir de doublons dans un `SortedSet`. Cependant, dans un `SortedSet`, c'est, selon le cas, la méthode `compareTo` ou la méthode `compare` qui permet de déterminer si deux éléments sont égaux ou non (si la méthode renvoie 0, les éléments sont considérés comme égaux). Il est donc fortement recommandé de définir la méthode `compareTo` (ou `compare`) de façon cohérente avec la méthode `equals` c.-à-d. de sorte que :

`o1.equals(o2)` si et seulement si `o1.compareTo(o2)==0` (ou `compare(o1,o2) == 0`)

C'était bien le cas dans les exemples présentés dans les sections précédentes. Si une classe définit une méthode `compareTo` (ou `compare`) qui n'est pas cohérente avec la méthode `equals`, il faut l'indiquer explicitement dans la javadoc car, le fait d'avoir cette incohérence fera qu'un `SortedSet` utilisant cette comparaison aura moins ou plus d'éléments qu'attendu.

SI on définit un `Comparator` dont la méthode `compare` renvoie parfois 0 pour deux objets qui sont différents selon la méthode `equals`, les deux objets ne pourront pas se trouver en même temps dans un `SortedSet`. Pour illustrer cela, considérons `Comparator` suivant pour la classe `Boisson` dont la méthode compare ne tient compte que du nom.

```
public class ComparatorBoissonNom implements Comparator<Boisson> {
    public int compare(Boisson o1, Boisson o2) {
        return o1.getNom().compareToIgnoreCase(o2.getNom());
    }
}
```

Si, dans le `main`, on passe un `ComparatorBoissonNom` en paramètre au `TreeSet` lors de sa création, son exécution aura maintenant comme résultat :

jus d'ananas (25 cl) : 2.8 euros
jus d'oranges (20 cl) : 2.6 euros
soda (25 cl) : 2.6 euros

Bien que `boisson1.equals(boisson2)` renvoie `false`, `boisson2` n'a pas été ajoutée car elle à la même nom que `boisson1` et dans ce cas `compare(boisson1,boisson2)` renvoie 0.

Si on utilise un `Comparator` qui renvoie quelque chose de différents de 0 pour deux objets qui sont égaux solon la méthode `equals`, alors les deux objets pourront quand même se trouver en même temps dans le `SortedSet`. Pour tester cela, vous pouvez écrire vous-même un `Comparator` pour la classe `Boisson` qui trie d'abord sur selon le nom et, en cas de nom identique, trie sur la contenance et, en cas de nom et de contenance identiques, trie sur le prix. Si, dans le main, vous passez ce

comparator lors de la création du `TreeSet`, toutes les boissons seront ajoutées même si `boisson1.equals(boisson4)`.

L'interface Queue

Une file d'attente est une collection normale avec quelques différences sémantiques : l'utilisation classique d'une file d'attente est de servir de tampon entre une source d'objets et un consommateur de ces mêmes objets.

Traditionnellement, une file d'attente expose trois types de méthodes :

- ajout d'un objet dans la file ;
- récupération et retrait de l'objet suivant disponible (le premier ayant été ajouté dans la file),
- examen de l'objet suivant disponible.

Les files d'attente ont une capacité limitée. Il se peut que l'ajout d'un objet dans une file d'attente échoue. Les files d'attente proposent plusieurs méthodes pour chaque sont le comportement en cas d'échec d'un ajout : génération d'exception, ou retour d'une valeur spéciale

	throws exception	returns special value
Ajout d'un élément dans la file	`add(e)`	`offer(e)`
Retrait d'un élément de la file	`remove()`	`poll()`
Examen d'un élément de la file	`element()`	`peek()`

L'interface Map

L'interface `Map` ne fait pas partie des Collection à proprement parler. Il s'agit ici d'un nouveau type de structure de données, qui stocke les données sous forme de dictionnaire : Clef – Valeur.

Pensez à un dictionnaire Français-Anglais par exemple. Vous y trouvez tous des couples de mots. Un mot en Français (la clef), et son équivalent en Anglais (la valeur). Vous cherchez la traduction du mot « Clavier », vous trouvez directement le mot « Keyboard ».

Voici un extrait des méthodes disponibles dans l'interface `Map` de Java :

```
public interface Map {

    // Opérations de base
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Vues de la Map
    public Set keySet();
    public Collection values();
    public Set entrySet();

}
```

Pour une liste plus complète, nous vous invitons à consulter la Javadoc de votre version de Java. Effectivement, ces interfaces évoluent également dans le temps, et de nouvelles méthodes apparaissent au fil des versions de Java pour vous simplifier la vie.

Diverses implémentations sont fournies par Java : `HashMap`, `TreeMap` (qui implémente la sous interface `SortedMap`) et `Hashtable`, plus ancien, qui a été modifié afin d'implémenter `Map`.

Les opérations de base font ce qu'on en attend. Par exemple `get(key)` renvoie la valeur associée à la clé passée et `remove(key)` supprime l'entrée de clé donnée.

Les vues sont la seule manière de parcourir une `Map`. On notera que `values()` renvoie une `Collection` et non un `Set` : c'est parce que les valeurs, contrairement aux clés, peuvent être dupliquées.

Le parcours se fera, par exemple, en utilisant l'itérateur du `Set` renvoyé par `keySet()` :

```
Map<Object,Object> maMap = new HashMap<>();

Iterator<Object> iterator = maMap.keySet().iterator();
while (iterator.hasNext()) {
    Object key = iterator.next();
    Object val = maMap.get(key);
}
```

Dans l'exemple ci-dessus, on a un type de clef `Object`, et un type de Valeur `Object`. La clef peut être n'importe quel autre type, tout comme la valeur. Le type générique de l'itérateur sera évidemment le même que celui de la valeur de la Map.

De façon simplifiée, on peut également utiliser un `foreach` avec la méthode `entrySet()` :

```
Map<Object,Object> maMap = new HashMap<>();

for (Map.Entry<Object, Object> entry : maMap.entrySet()) {
    Object key = entry.getKey();
    Object val = entry.getValue();
}
```

Cette technique permet de récupérer toutes les `Entry` d'une `Map`. Une `Entry` est un couple clé valeur de la `Map`. Pour faire le parallèle avec le dictionnaire Français Anglais, une `Entry` serait un mot avec sa traduction. En appelant la méthode `getKey()` dessus, on récupère le mot en Français, en appelant la méthode `getValue()`, on récupère la traduction.

Les vues, comme leur nom l'indique, sont des vues sur la `Map` et non des copies de parties de celle-ci. C'est pourquoi un appel à `remove()` sur l'`Iterator` d'une des vues supprime en réalité dans la `Map`. De plus si on utilise `entrySet()`, il est même possible de changer la valeur associée à une clé grâce à la méthode `setValue()` de `Map.Entry`.

Par ailleurs, toute opération de modification effectuée directement sur une vue affecte directement la `Map`. C'est le cas des méthodes `remove()`, `removeAll()`, `retainAll()`, `clear()`. Bien sûr, ceci suppose que l'implémentation de la `Map` utilisée permet les suppressions ! C'est le cas des implémentations fournies par Java, mais ce n'est pas obligatoire pour des implémentations personnalisées.

En aucun cas il n'est possible d'ajouter des éléments à une `Map` lors d'un parcours ou via une des vues.

L'implémentation la plus utilisée de cette interface est la fameuse `HashMap`. Lorsqu'on a besoin que les clefs de la `Map` soient triées, on se dirige alors plutôt vers une implémentation de `SortedMap`, à savoir la `TreeMap`.

Interface SortedMap

Lorsqu'on à besoin d'une `Map`, triée par ordre de clefs, on se dirige plutôt vers la `SortedMap`. Son implémentation habituelle est la `TreeMap`.

Voici un extrait des méthodes disponibles dans l'interface `SortedMap` de Java :

```
public interface SortedMap extends Map {

    // vues sur parties
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    // extrémités
    Object firstKey();
    Object lastKey();

    // Comparator utilisé
    Comparator comparator();

}
```

Cette interface est tout à fait semblable à `SortedSet` et pratiquement les mêmes remarques s'imposent.

Des méthodes héritées de `Map` ont un contrat différent : Ce sont les vues `keySet()`, `values()` et `entrySet()`. Elles diffèrent en deux points :

- leur méthode `iterator()` renvoie les clés, valeurs ou entrées respectivement, dans l'ordre des clés triées.
- leurs méthodes `toArray()` place dans un tableau les clés, valeurs ou entrées respectivement, dans l'ordre des clés.

Constructeurs : Toute classe implémentant `SortedMap` doit fournir deux constructeurs supplémentaires :

- un constructeur qui prend en paramètre un `Comparator` indiquant l'ordre du tri des clés. Ce `Comparator` doit être cohérent avec `equals()` si vous utilisez comme type `Map` au lieu de `SortedMap`.
- un constructeur qui prend un paramètre une `SortedMap`. Comme pour le constructeur prenant une `Map` en paramètre, cette `SortedMap` servira à remplir celle construite. Mais de plus, l'ordre de la `SortedMap` passée sera préservé dans la `SortedMap` construite.

Pour le reste rapportez-vous à `SortedSet`.

N'hésitez pas à consulter la Javadoc de votre version de Java si vous souhaitez plus d'informations. La plupart des méthodes et des contraintes des classes et interfaces expliquées ici y sont expliquées largement.

Wrappers méthodes

La classe `Collections` fournit aussi une série de méthodes dites "wrappers" :
Une première série de méthodes transforment une structure en la même structure rendue immuable. Ces méthodes sont bien utiles quand on veut renvoyer une collection sans qu'elle puisse être modifiée

```
public static Collection unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
public static List unmodifiableList(List list);
public static Map unmodifiableMap(Map m);
public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);
```

Une deuxième série renvoie une structure synchronisée. Nous nous attarderons plus sur cette notion dans le chapitre consacré aux Threads. Il faut savoir que `Vector` et `Hashtable` sont synchronisées mais qu'aucune des collections et maps ne l'est. Pour les rendre synchronisées, on utilisera la méthode appropriée parmi :

```
public static Collection synchronizedCollection(Collection c);
public static Set synchronizedSet(Set s);
public static List synchronizedList(List list);
public static Map synchronizedMap(Map m);
public static SortedSet synchronizedSortedSet(SortedSet s);
public static SortedMap synchronizedSortedMap(SortedMap m);
```

Algorithmes

Algorithmes applicables à toute `Collection`

La classe `Collections` fournit une série de méthodes permettant d'appliquer des algorithmes classiques à des `Collections`.

```
static Object max(Collection coll);
```

```
static Object max(Collection coll, Comparator comp);
```

```
static Object min(Collection coll);
```

```
static Object min(Collection coll, Comparator comp);
```

Les méthodes n'ayant pas de `Comparator` en paramètre nécessite que la classe des objets contenus dans la collection implémente l'interface `Comparable`.

Algorithmes propres aux `Lists`

```
static int binarySearch(List list, Object key);
```

```
static int binarySearch(List list, Object key, Comparator c);
```

La `List list` est supposée triée (dans le deuxième cas dans l'ordre du `Comparator c`). La méthode renvoie la position de l'objet s'il est trouvé et un nombre négatif s'il n'est pas trouvé. Ce nombre négatif (`n`) est tel que `-n -1` est l'endroit où il faut insérer l'élément pour conserver le tri.

```
static void copy(List dest, List src);
```

La `List dest` doit être au moins aussi longue que `src`. Les éléments correspondants sont écrasés. Les éléments restants ne sont pas modifiés.

```
static void fill(List list, Object o);
```

```
static void reverse(List l);
```

```
static void shuffle(List list);
```

```
static void shuffle(List list, Random rnd);
```

```
static void sort(List list);
```

```
static void sort(List list, Comparator c);
```

Dans ces deux dernières méthodes, le tri est stable et sa performance est en `n*log(n)`.