POO – les collections

Introduction

Les collections représentent les structures de données les plus courantes et vous laissent la possibilité d'en implémenter d'autres. Les premières collections introduites en Java sont les tableaux bien sûr, les `Vector` et les `Hashtable`. Depuis Java 1.2, un framework plus général a été introduit. Au fur et à mesure des versions du Java, ce framework a évolué sans cesse.

Avec le Java 8, il est maintenant possible d'associer des streams à une collection, ceci permettant notamment des facilités propres aux langages fonctionnels.

Le framework

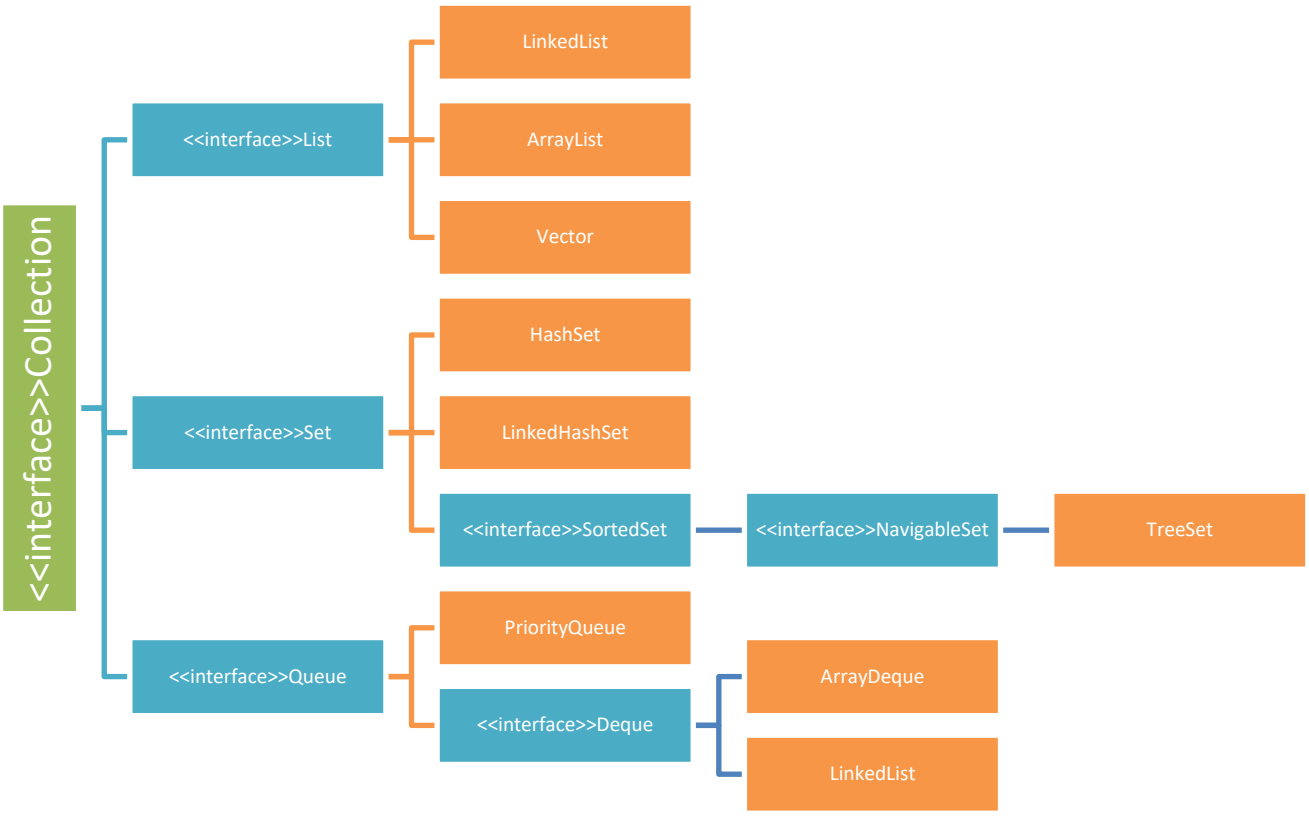
Un tel framework est composé de trois choses :

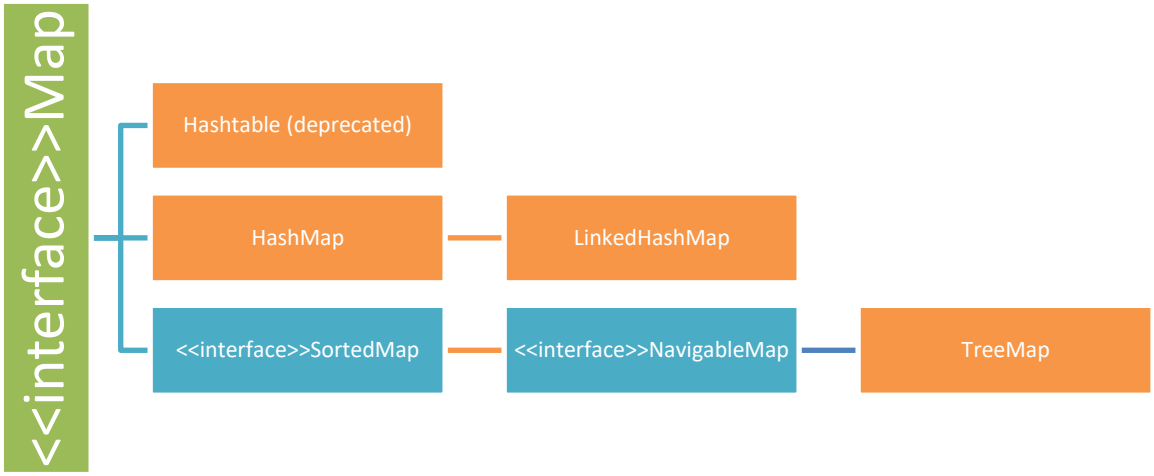
- Interfaces : ce sont des types de données abstraits. Ce seront toujours eux qu'on utilisera pour déclarer les collections utilisées
- Implémentations : ce sont des implémentations concrètes des interfaces. On les utilisera uniquement derrière l'instruction `new`.
- Algorithmes : en plus des méthodes 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`, etc.

Etendre ce framework

Si vous voulez rajouter d'autres structures de données, vous ne devez pas partir de rien et implémenter vous-même toutes les méthodes décrites dans l'interface à implémenter. Java fournit en plus de ce qui est décrit dans le framework, toute une série de classes abstraites qu'il suffit d'étendre et qui implémentent déjà pas mal des méthodes déclarées dans les interfaces.

Le schéma suivant résume l'arborescence des collections. Il existe bien d'autres types fournis par Java. Nous ne nous attarderons pas à les présenter tous ici. Référez-vous à la doc si besoin.





La plupart des collections présentées ici sont non synchronisées (sauf `Vector` qui est une classe relativement « vieille », de moins en moins utilisée). Elles sont relativement efficaces mais en contrepartie elles ne peuvent être utilisées telles quelles dans un contexte d'accès concurrents. Pour gérer les accès concurrents, soit il faudra utiliser la synchronisation que nous verrons plus tard soit utiliser directement les collections synchronisées offertes par le package `java.util.concurrent`.

Les Interfaces Collection et Map

Description générale

L'interface `Collection` est la racine de la hiérarchie. Elle déclare les méthodes communes. Java ne fournit aucune implémentation de cette interface mais seulement 3 sous interfaces :

- `List`
- `Set`
- `Queue`

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

L'interface `Set` représente des collections ne permettant **pas de doublons**. Il est implémenté par la classe `HashSet`. L'interface `SortedSet` est un `Set` qui garde ses éléments en ordre croissant. Il est implémenté 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. Il est implémenté par la classe `PriorityQueue` et étendu par l'interface `Deque`. `Deque` permet l'ajout en début et en fin de file. Il n'est pas possible d'ajouter un élément **`null`** dans une queue.

L'interface `Map` (qui ne dérive pas de `Collection`) représente des objets **associant des clés à des valeurs**. Il est implémenté par la classe `HashMap`.

L'interface `SortedMap` est une `Map` qui garde ses éléments en **ordre croissant des clés**. Il est implémenté par la classe `TreeMap`.

Dans le cas des `SortedSet` et des `SortedMap`, on conserve les éléments en ordre croissant. Cela signifie que les éléments sont membres d'une classe qui implémente l'interface `Comparable` ou que la structure a été construite en lui passant un objet implémentant l'interface `Comparator`. Si les objets sont membres d'une classe qui implémente `Comparable`, on dit souvent que la structure les garde triés selon leur *ordre naturel*.

Les implémentations (`HashSet`, `ArrayList`, `LinkedList`, `HashMap`, `TreeSet` et `TreeMap`) implémentent `Serializable` et `Cloneable`. Elles ont donc toutes une méthode `clone()` en plus de celles prévues par l'interface implémentée.

Les opérations « optionnelles »

Dans les interfaces qui vont suivre, certaines opérations sont marquées optionnelles. Il s'agit de toutes les méthodes modifiant la structure. Cela ne veut pas dire qu'une Classe implémentant l'interface ne doit pas définir ces méthodes mais qu'elle peut se contenter de le faire comme suit, s'il est impossible de l'implémenter réellement ou si on ne désire pas le faire pour des raisons de sécurité.

```
public void méthodeOptionnelle()
                        throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
}
```

Collection

```
public interface Collection {
     // Opérations de base
     int size();
     boolean isEmpty();
     boolean contains(Object element);
     boolean add(Object element);    // Optionnel
     boolean remove(Object element); // Optionnel
     Iterator iterator();

     // Opérations de groupes
     boolean containsAll(Collection c);
     boolean addAll(Collection c);    // Optionnel
     boolean removeAll(Collection c); // Optionnel
     boolean retainAll(Collection c); // Optionnel
     void clear();                    // Optionnel

     // Opérations de tableaux
     Object[] toArray();
     Object[] toArray(Object[] a);
}
```

Parmi ces méthodes la plupart font ce qu'on en attend. La méthode `retainAll` ne garde dans la `Collection` en cours que les objets qui sont aussi présents dans la `Collection` `c`.

La deuxième méthode `toArray` n'utilise son paramètre que pour indiquer le type des objets à mettre dans le tableau (attention pas le type du tableau : il faudra caster). On l'utilise sous la forme suivante :

```
String[] t = (String[])c.toArray(new String[0]);
```

Toutefois, si le tableau passé en paramètre est assez grand, c'est là que la méthode stockera le résultat (qui sera de plus renvoyé). Sinon, la méthode alloue un autre tableau et retourne celui-ci.

Iterator

Pour parcourir une `Collection` sans en dévoiler la structure interne, on a recours à un itérateur. L'interface `Iterator` est le suivant :

```
public interface Iterator {
     boolean hasNext();
     Object next();
     void remove();    // Optionnel
}
```

Les noms des méthodes sont plus simples que ceux employés dans l'interface `Enumeration` et de plus Iterator déclare une méthode `remove()`.

Quand on parcourt une `Enumeration`, il n'y a pas de moyen sûr de supprimer des éléments de la structure traversée. Avec un `Iterator`, ce moyen existe : appeler `remove()` sur l'`Iterator`.

Attention, ne pas appeler `remove()` directement sur la `Collection`. La méthode `remove()` de `Iterator` supprime le dernier élément renvoyé par `next()`. On ne peut l'appeler qu'une seule fois après chaque `next()`. Sinon on aura une exception.

`List` : implémentation : `ArrayList` et `LinkedList`

Une `List` est une `Collection` où les éléments sont gardés dans un certain ordre (ce qui ne signifie pas triés). Une `List` peut contenir des doublons. Il existe deux implémentations (complètes) de l'interface List : la classe `ArrayList` et la classe `LinkedList` (qui est doublement chaînée). La classe `Vector` a été modifiée depuis Java 1.2 afin d'implémenter `List` tout en gardant ses anciennes méthodes. D'un point de vue performance, la classe `ArrayList` est souvent plus intéressante.

Méthodes héritées de Collection

Certaines de ces méthodes nécessitent une explication car leur contrat est modifié :
`add(Object element)` et `addAll(Collection c)` ajoutent à la fin de la liste.
`remove(Object element)` supprime la première occurrence d'`element`.
Les méthodes `equals()` et `hashCode()` sont également modifiées :
deux `List`s seront `equals` si elles ont les mêmes éléments dans le même ordre.
La méthode `hashCode()` est définie comme suit :

```
int hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
return hashCode;
```

La méthode `iterator()` renvoie les éléments dans l'ordre de la List.

Méthodes ajoutées

L'interface `List` ajoute toute une série de méthodes à l'interface `Collection`. Elle est définie comme suit :

```
public interface List extends Collection {
    // Accès via un indice
    Object get(int index);
    Object set(int index, Object element);            // Optionnel
    void add(int index, Object element);              // Optionnel
    Object remove(int index);                         // Optionnel

    abstract boolean addAll(int index, Collection c); // Optionnel

    // Recherche
    int indexOf(Object o);
    int lastIndexOf(Object o);
```

```
    // Tri (depuis Java 8)
    void sort(Comparator c);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Partie
    List subList(int from, int to);
}
```

Les indices, comme pour les tableaux, commencent à 0.

- `set(int index, Object element)` remplace l'élément à l'indice indiqué.
- `add(int index, Object element)` insère un élément à l'indice passé et décale les éléments suivants vers l'arrière.
- `remove(int index)` supprime l'élément indiqué et décale les suivants vers l'avant.
- `addAll(int index, Collection c)` insère tous les éléments de la `Collection` en commençant à l'indice fourni dans l'ordre donné par l'`Iterator` de la `Collection`; elle décale ensuite les éléments restants vers l'arrière.
- `indexOf` – `lastIndexOf` – renvoie l'indice de la première – dernière – occurrence de l'objet cherché et `-1` s'il n'est pas présent.
- `subList(int from, int to)` renvoie une sous liste de la liste reprenant les éléments partant de l'indice `from` (inclus) jusqu'à l'indice `to` (exclus). Toute modification faite sur la sous liste sera répercutée sur la `List`. Cependant, si on modifie la `List` directement, la sous liste n'est plus valide et l'accès à cette sous liste engendrera une exception.

Depuis Java 10, l'interface `List` fournit aussi la méthode

```
static List copyOf(Collection coll)
```

qui renvoie une liste immuable contenant tous les éléments de la collection passée en paramètre.

`Set` : implémentation : `HashSet`

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. Elle modifie aussi le contrat de `equals()` et de `hashCode()` :

La méthode `equals(Object o)` renverra `true` si `o` est un `Set` de même taille et contenant les mêmes éléments.

La méthode `hashCode()` renvoie la somme des `hashCode()` de ses éléments ce qui garantit que deux `Set`s égaux ont le même `hashCode()`.

L'interface `Set` est implémentée (complètement) par la classe `HashSet`. Comme toutes les implémentations des sous interfaces de `Collection`, cette classe a un constructeur prenant comme paramètre une `Collection`. Cette interface est également implémentée par `TreeSet` qui implémente en réalité la sous interface `SortedSet` et garantit dès lors, l'ordre des éléments lors d'une itération.

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

```
s1.containsAll(s2)    :    s2 ⊆ s1
s1.addAll(s2)         :    s1 = s1 ∪ s2
s1.retainAll(s2)      :    s1 = s1 ∩ s2
s1.removeAll(s2)      :    s1 = s1 – s2
```

Il n'est pas difficile à partir de là d'écrire des méthodes renvoyant l'union, l'intersection ou la différence de deux ensembles sans modifier le premier. Il suffit de recopier le premier ensemble dans un autre. Par exemple, l'union se calcule comme suit :

```
Set union = new HashSet(s1);
union.addAll(s2);
return union;
```

L'utilisation de `Set` permet de supprimer les doubles dans toute `Collection` :

```
Collection sansDoubles = new HashSet(collection);
```

Il est alors facile de recréer à partir de ce `HashSet` le type de `Collection` désiré.

Depuis Java 10, l'interface `Set` fournit aussi la méthode

```
static Set copyOf(Collection coll)
```

qui renvoie un ensemble immuable contenant tous les éléments (sans doublon) de la collection passée en paramètre.

Eléments immuables

Les objets membres d'un `Set` ont intérêt à être immuables. Si on les modifiait alors qu'ils sont déjà dans le `Set`, on risquerait de briser le contrat du `Set` qui n'admet pas de doublons.

hashCode()

Si vous redéfinissez la méthode `equals()` pour les membres d'un `Set`, vous devez aussi redéfinir la méthode `hashCode()` afin que deux objets equals aient le même `hashCode()`.

SortedSet : implémentation : `TreeSet`

```
public interface SortedSet extends Set {
    // vues sur parties
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);

    // extrémités
    Object first();
    Object last();

    // le Comparator  utilisé
    Comparator comparator();
}
```

Parmi les méthodes héritées de `Set`, seules deux ont un contrat modifié :

- `iterator()` renvoie un `Iterator` qui parcourt les éléments de l'ensemble dans l'ordre du tri.
- `toArray()` construit un tableau dont les éléments sont triés.

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. De plus, contrairement à ce qui se passait pour `subList`, si on modifie le `SortedSet` directement, la partie reste valide. Comme toujours en Java, l'objet `fromElement` est inclus dans la partie et `toElement` ne l'est pas.

Toute implémentation de cette interface doit posséder deux constructeurs supplémentaires :

- un constructeur qui prend en paramètre un `Comparator` indiquant l'ordre du tri. Ce `Comparator` doit être cohérent avec `equals()`.
- un constructeur qui prend un paramètre un `SortedSet`. Comme pour le constructeur prenant un `Set` en paramètre, ce `SortedSet` servira à populer celui construit. Mais de plus, l'ordre du `SortedSet` passé sera préservé dans le `SortedSet` construit.

La méthode `comparator()` renvoie le `Comparator` utilisé lors de la construction du `SortedSet` ou `null`, si l'ordre naturel est employé.

Si `iterator()` permet de parcourir le `SortedSet` dans l'ordre, il n'y a pas de méthode permettant de le parcourir dans l'ordre inverse. Pour le faire, on peut partir de `last()` et faire comme suit :

```
Object o = ensembleTrié.last();
Object prec = ensembleTrié.headSet(o).last();
```

et ainsi de suite, jusqu'à obtenir un `headSet()` vide.

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 ;
- retrait de l'objet suivant disponible, et retrait de 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 différents comportements en cas d'échec d'un ajout : génération d'exception, ou retour d'un booléen à `false` :

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

`Map` : implémentation : `HashMap`

Présentation

Une `Map` associe des clés à des valeurs. Les clés ne peuvent pas être dupliquées. L'interface est la suivante :

```
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();

    // Opérations de groupe
    void putAll(Map t);
    void clear();

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

    // Interface interne pour les éléments du Set renvoyé par
    // la méthode entrySet();
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Depuis Java 10, l'interface `Map` fournit aussi la méthode

```
static Map copyOf(Map map)
```

qui renvoie une `Map` immuable contenant les mêmes entrées que la map passée en paramètre.

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 bases 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.

Dans les opérations de groupe, `putAll(map)` effectue un `put(key, map.get(key))` pour toutes les clés de `map`, remplaçant les valeurs pour les clés qui existaient déjà et ajoutant les autres.

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<>();

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

De façon simplifiée, on peut également utiliser un `foreach` :

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

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.

Eléments immuables

Les objets servant de clés à une `Map` ont intérêt à être immuables. Si on les modifiait alors qu'ils sont déjà dans la `Map`, on risquerait de briser le contrat de cette `Map` qui n'admet pas de doublons pour ses clés.

hashCode()

Si vous redéfinissez la méthode `equals()` pour les clés d'une `Map`, vous devez aussi redéfinir la méthode `hashCode()` afin que deux objets equals aient le même `hashCode()`.

SortedMap : implémentation : `TreeMap`

```
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.

Java fournit une implémentation de `SortedMap` : `TreeMap`.
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()`.
- un constructeur qui prend un paramètre une `SortedMap`. Comme pour le constructeur prenant une `Map` en paramètre, cette `SortedMap` servira à populer celle construite. Mais de plus, l'ordre de la `SortedMap` passée sera préservé dans la `SortedMap` construite.

Pour le reste rapportez-vous à `SortedSet`.

Egalite et ordre

Il y a deux manières d'ordonner les objets afin de les garder triés dans un `SortedSet` (implémentation : `TreeSet`) ou une `SortedMap` (implémentation : `TreeMap`).

- Les éléments de la collection sont des objets d'une classe qui implémente l'interface `Comparable`.
- Le constructeur utilisé pour la collection est celui qui prend en paramètre un objet d'une classe implémentant l'interface `Comparator`.

Comparable

L'interface `Comparable` contient une seule méthode :

```
int compareTo(Object 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.

Cette interface est implémentée par de nombreuses classes dont toutes les classes "wrappers" (`Byte`, `Integer`, `Double`, …), les classes `String`, `File`, `Date`, `BigInteger`, `BigDecimal`, …

Elle fournit sur les classes qui l'implémentent, un ordre dit "naturel".
C'est cet ordre qui est utilisé si on essaie de trier une `List` avec `Collections.sort(liste)` ou directement la méthode `sort` de la liste ou un tableau avec `Arrays.sort(table)`.
C'est aussi lui qui sert à comparer les éléments pour les gardés triés dans une collection triée, si le constructeur de cette collection n'a pas reçu de `Comparator` en paramètre.

Pour les classes usuelles, cet ordre est le suivant:

Classe	**Ordre Naturel**
Byte	numérique signé
Character	numérique non-signé
Short	numérique signé
Integer	numérique signé
Long	numérique signé
Float	numérique signé
Double	numérique signé
BigInteger	numérique signé
BigDecimal	numérique signé
String	lexicographique
File	lexicographique du pathname (dépend de l'O.S.)
Date	chronologique

L'exemple suivant est typique de la façon dont il faut écrire `equals()` et `compareTo()` :

```
import java.util.*;

public class Name implements Comparable<Name> {
    private String  firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName==null || lastName==null)
            throw new NullPointerException();
        this.firstName = firstName;
```

```
        this.lastName = lastName;
    }

    public String firstName() {return firstName;}
    public String lastName()  {return lastName;}

    public boolean equals(Object o) {
       if (o == null) return false;
       if (o == this) return true;
        if (o.getClass() != this.getClass())
            return false;
        Name n = (Name)o;
        return n.firstName.equals(firstName) &&
               n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {return firstName + " " + lastName;}

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return lastCmp!=0  ? lastCmp
                           : firstName.compareTo(n.firstName);
    }
}
```

On remarquera que `compareTo()` renvoie `0` si et seulement si `equals()` renvoie `true`.
Quand c'est le cas on dit que *`compareTo()` est cohérent avec `equals()`*. C'est fondamental, car `equals()` est utilisé pour détecter les doublons dans un *`Set`* alors que dans un *`SortedSet`* c'est `compareTo()` qui est utilisé. Si les méthodes étaient incohérentes, cela conduirait à des ensembles triés contenant moins ou plus d'éléments qu'attendu.
La cohérence a une exception : `elem.equals(null)` renvoie `false` tandis que `elem.compareTo(null)` lance une `NullPointerException`.
Toutes les classes usuelles implémentent `Comparable` de façon cohérente avec `equals()` sauf `BigDecimal` où `compareTo()` renvoie `0` pour des nombres de même valeur mais de précision différente (ex. : `17.5` et `17.50`).

De plus `compareTo()` doit définir un ordre, et donc vérifier les propriétés suivantes, y compris au niveau des exceptions :

```
signe(x.compareTo(y)) == -signe(y.compareTo(x))

(x.compareTo(y)>0 && y.compareTo(z)>0) => x.compareTo(z)>0
```

`x.compareTo(y)==0 => signe(x.compareTo(z)) == signe(y.compareTo(z))`, pour tout `z`.

equals

Dans la méthode `equals` présentée ci avant, on a utilisé `getClass()` pour vérifier que l'objet passé est de même classe que celui avec lequel on le teste. De cette manière, un objet d'une sous-classe ne sera jamais `equals` à un objet de la classe parent.

Au cas où on désirerait pourvoir identifier des objets d'une sous classe à certains objets de la classe parent, deux cas se présentent :

1. On ne désire pas redéfinir `equals` dans la classe enfant. Par exemple si la classe `Employe` dérive de `Personne`. Deux employés seront égaux, s'ils le sont en tant que personne.
2. On désire redéfinir equals dans la classe enfant et identifier certains enfants avec leur parent. Par exemple la classe Point3D est une sous classe de Point2D. Elle rajoute un champ z donnant la troisième coordonnée du point. ON désire identifier un Point2D avec un point3D où z = 0.

En général, il faudra convenir de valeurs par défaut pour les champs ajoutés dans la classe enfant.

Veillez toujours à ce qu'`equals` et `hashCode` soient cohérents !

Comparator

Si on désire trier dans un ordre différent de l'ordre naturel ou si on désire trier des objets d'une classe qui n'implémente pas `Comparable`, il faut utiliser un `Comparator`. L'interface `Comparator` est constituée d'une seule méthode :

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

La méthode `compare()` doit vérifier des propriétés analogues à celle de `compareTo()` :

```
signe(compare(x, y)) == -signe(compare(y, x))

((compare(x, y)>0) && (compare(y, z)>0)) => compare(x, z)>0

compare(x, y)==0 => signe(compare(x, z)) == signe(compare(y, z))
```

pour tout `z`

Ici aussi, on dira que *`compare()` est cohérent avec `equals()`* si

```
(compare(x, y) == 0) == x.equals(y)
```

Cette propriété n'est pas exigée mais est fortement conseillée et indispensable pour trier des collections basées sur le hashing comme `TreeSet` et `TreeMap`. Au cas où un `Comparator` ne

respecterait pas cette propriété, il est indispensable de l'indiquer dans la documentation. On saura ainsi qu'on ne pourra l'utiliser que pour trier des tableaux ou des `Lists`.

Java 8 a introduit des nouvelles méthodes dans l'interface `Comparator` : `comparing`, `reversed`,… et surtout fournit des implémentations par défaut ou statiques. Voir l'API : http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparing-java.util.Comparator-

Il devient alors possible d'utiliser les expressions Lambda. Supposant une classe `Point` doté d'une méthode `getX()`, l'expression lambda suivante permet le tri des points sur base de leur `x` :

```
(p1,p2) ->((Integer)(p1.getX())).compareTo(p2.getX())
```

La conversion de `p1.getX()` en `Integer` est indispensable car `compareTo` s'applique à des objets.
Grace à la méthode `comparing`, il suffit d'écrire :

```
Comparator.comparing(p -> p.getX())
```

Ou l'ordre inverse :

```
Comparator.comparing(p -> p.getX()).inversed
```

Si on veut comparer les points ensuite sur leur y :

```
Comparator.comparing(p -> p.getX()).thenComparing(p -> p.getY())
```

Java Internationalization: Collator - Sorting Strings

Chaque langue peut avoir ses propres règles sur la façon dont les chaînes et les lettres sont triées. Ainsi, L'usage de la méthode `String.compareTo()` peut ne pas toujours fournir le résultat escompté.

Pour information, la comparaison des chaînes de caractères se fait à partir de la valeur Unicode de leurs caractères. Cet ordre est le suivant : espaces < chiffres < lettres majuscules < lettres minuscules.

Dans le cas où on veut un tri insensible à la casse, la classe `String` définit un `Comparator`, `public static final Comparator CASE_INSENSITIVE_ORDER`, qui permet de trier des collections comme le fait la méthode `compareToIgnoreCase()`.

Les choses se compliquent lorsque l'on souhaite faire une comparaison plus évoluée sans tenir compte de l'accentuation des caractères. Par exemple, on souhaite que les chaînes de caractères « aout » et « Août » soient considérées identiques.

La solution est alors de passer par l'utilisation de la classe `java.text.Collator`, la région communiquée par la variable de type `Locale` et de l'attribut `strength` qui permet de déterminer la manière avec laquelle 2 caractères seront considérés identiques ou différents.

La classe `Locale` permet de représenter une région géographique, politique ou culturelle. Notre variable est `Locale.FRENCH`.

L'attribut `strength` détermine le niveau de comparaison. Il se décline selon 3 niveaux :

- Le niveau *primaire* : la comparaison de 2 caractères prend en compte uniquement les caractères de base (pas de prise en compte de l'accentuation ni de la casse). "à" est différent de "b" mais équivalent à "a".
- Le niveau *secondaire* : la comparaison de 2 caractères prend en compte les caractères de base ET les accents. "à" est différent de "a".
- Le niveau *tertiaire*: la comparaison de 2 caractères prend en compte les caractères de base ET les accents ET la casse. "A" est différent de "a".

Petit exemple de code:

```
public class CompareTest implements Comparator<String> {
private Collator compareOperator;
public CompareTest() {
	compareOperator = Collator.getInstance(Locale.FRENCH);
	compareOperator.setStrength(Collator.PRIMARY);
}
@Override
public int compare(String o1, String o2) {
	return compareOperator.compare(o1, o2);
}
public static void main(String args[]) {
List<String> test = {de Wasseige", "Desemberg", "degreef", "dégreef","Dewa","
degreef","âttention","attention","De Greef","dewa"};
test.sort();
}}
```

ConcurrentModificationException

Pour rappel, les collections présentées dans ce chapitre ne gèrent pas la concurrence. Leur `iterator()` et `listIterator()` sont **fail-fast**, c'est à dire qu'ils détectent une modification faite directement sur la collection en cours de parcours et terminent immédiatement en lançant une `ConcurrentModificationException`.

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.

```
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 synchronisée mais qu'aucune des collections et maps introduites en Java 2 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);
```

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.

```
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)`.