Gestion des exceptions en Java

Jean-Marie Lagniez

LPDIOC : Algorithmique et Programmation
IUT Lens

Département Informatique

Table des matières

Sommaire

Dans ce cours nous allons voir :

- Les collections
- Les ensembles
- Les ensembles triés
- Les listes
- Les itérateurs

Pré-requis :

- Héritage, création d'objets, polymorphisme, interfaces, classes abstraites et templates

C'est quoi une collection ?

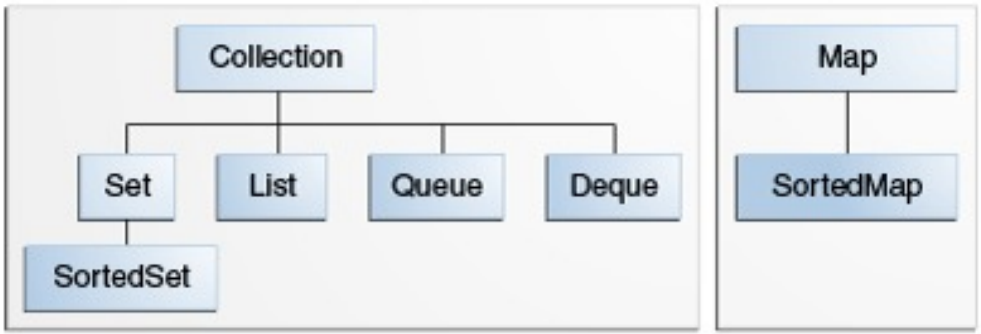
- Une **collection**, aussi appelé **conteneur**, est un objet qui regroupe une multitude d'éléments dans une seule unité
- Les collections sont utilisées afin d'enregistrer, rechercher, manipuler, et communiquer des données
- Ce sont des données qui **forment un groupe d'éléments**. Par exemple :
  - une main au poker (une collection de carte)
  - un répertoire téléphonique (un *mapping* entre les noms et les numéros)
  - un ensemble de lettres (une collection de lettres)
- Toutes les collections s'appuient sur :
  - **Interfaces** : es interfaces permettent de manipuler les collections indépendamment de la manière dont ces informations sont représentées. Ces interfaces forment une hiérarchie
  - **Implémentations** : il y a de nombreuses implémentations de l'interface collection disponible de l'API Java
  - **Algorithmes** : de nombreux algorithmes réutilisable et applicable avec des collections (`sort`, `search`, ...)

Pourquoi utiliser les collections de l'API Java ?

- **Réduire l'effort de programmation** : en fournissant des algorithmes et des structures de données, l'utilisation des collections permet de se concentrer uniquement sur l'architecture globale du programme
- **Augmenter la rapidité et la qualité des programmes** : les classes implémentées dans l'API sont souvent très performantes (utilisation de structures de données et d'algorithmes efficaces) et très bien documentées (`https://docs.oracle.com/javase/7/docs/api/`)
- **Permet l'interopérabilité entre les différentes classes de l'API Java** : cela permet de connecter facilement les nombreuses classes de l'API
- **Réduire l'effort pour apprendre à utiliser de nouvelles APIs** : de nombreuses classes de l'API Java utilisent des collections en entrée/sortie
- **Réduire l'effort pour créer de nouvelle API** : le programmeur n'a plus besoin de réinventer la roue chaque fois qu'il veut créer une nouvelle API qui manipule des collections d'objets
- **Forcer la réutilisation de codes** : les nouvelles structures de données qui sont conformes à l'interface `Collection` sont par nature réutilisable

Core Collection Interfaces

- Java propose un ensemble d'interfaces pour plusieurs types de collections
- Ces interfaces permettent de manipuler des collections sans se soucier de la manière dont ces dernières sont implémentées
- **Nous avons une hiérarchie de collections**



- Un `Set` est une `Collection` particulière, un `SortedSet` est un `Set` particulier, ...
- Les interfaces sont génériques : `public interface Collection<E>`
- Afin de conserver un nombre d'interface faible, Java ne fournit pas une interface pour chaque variante de collection (mutable, ajout uniquement, ...)

  → **`UnsupportedOperationException`**

Listes des interfaces disponibles

- `Collection` : une collection représente un groupe d'objets. C'est le dénominateur commun entre toutes les collections
- `Set` : c'est une collection qui ne contient pas de doublons
- `List` : c'est une collection ordonnée d'élément (aussi appelé séquence)
- `Queue` : c'est une file d'élément classique (FIFO - first-in, first-out)
- `Deque` : c'est une structure permettant gérer des piles ou des files (LIFO - last-in, first-out)
- `Map` : permet de sauvegarder des couples clef/valeur. Une map ne contient pas deux fois la même clef !

- `SortedSet` : c'est un ensemble d'élément qui conserve ses éléments triés dans l'ordre croissant
- `SortedMap` : c'est une map qui conserve un ordre croissant sur ces clefs

Table des matières

L'interface `Collection`

- Elle est utilisée dans le cas où l'on souhaite un maximum de généralité
- Supposons que l'on souhaite utiliser une collection de `String` :

```
Collection<String> c;
```

- Par convention, toutes les collections (`Set`, `Map`, ...) implémentent un constructeur qui prend en paramètre un objet de type `Collection`
- Ainsi, il est possible d'initialiser une liste avec `c` (vie l'utilisation de la classe `ArrayList` qui est une implémentation d'un liste) :

```
List<String> list = new ArrayList<>(c);
```

- L'interface `Collection` contient les opérations basiques suivantes :
  - ⇒ int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element), void clear() et Iterator<E> iterator()
- Elle contient aussi des opérations manipulant des collections :
  - ⇒ boolean containsAll(Collection< ?> c), boolean addAll(Collection< ? extends E> c), boolean removeAll(Collection< ?> c), et boolean retainAll(Collection< ?> c)
- Il est aussi possible de convertir une collection en tableau :
  - ⇒ Object[] toArray() et <T> T[] toArray(T[] a)

Parcourir une `Collection`

- En utilisant une boucle `for-each`

```
for (Object o : collection) System.out.println(o);
```

- En utilisant un itérateur
  - un itérateur est un objet qui vous permet de parcourir et supprimer (si on le souhaite) les éléments d'une collection

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove(); //optionelle
}
```

  - un objet `Iterator` est obtenu à partir d'une collection en appelant la méthode `iterator`

```
void filtrer(Collection<?> c) {
  for (Iterator<?> it = c.iterator();
       it.hasNext(); )
  if (!condition(it.next())) it.remove();
}
```

  - L'utilisation d'un `Iterator` plutôt que la boucle `for-each` intervient lorsqu'on souhaite supprimer des éléments de la collection

L'interface Set

- L'interface `Set` contient uniquement les méthodes de l'interface `Collection` tout en ajoutant des contraintes concernant le fait qu'on ne puisse pas avoir deux fois le même élément
- Les éléments d'un `Set` sont gérés via les méthodes `equals` et `hashCode`
- Java propose trois implémentations de l'interface `Set` :
  - `HashSet` - sauvegarde les éléments dans une table de hachage. L'avantage est que les opérations sont réalisées rapidement. Cependant, il n'y a plus d'ordre concernant le parcours des éléments
  - `TreeSet` - sauvegarde les données dans un arbre rouge et noir. Cette méthode est un peu plus lente que `HashSet` et a le même inconvénient concernant l'ordre de parcours des éléments
  - `LinkedHashSet` - sauvegarde les éléments à la fois dans un liste chaînée et dans une table de hachage. Permet à peu près les même performances que `HashSet` et conserve l'ordre d'ajout des éléments
- Ici est un exemple simple d'utilisation d'un `Set` contenant les éléments d'une collection d'éléments fournit en paramètre

```
Collection<Type> noDups = new HashSet<Type>(c);
```

L'interface NavigableSet<E>

- Cette interface permet de fournir les méthodes afin de parcourir un objet de type Set. Les méthodes sont :
  - `ceiling(E e)` : retourne le plus petit élément de la collection >= `e`
  - `descendingIterator()` : retourne un itérateur parcourant la collection dans le sens inverse du sens naturel de tri
  - `descendingSet()` : retourne un `NavigableSet<E>` trié à l'inverse
  - `floor(E e)` : retourne le plus grand élément de cet ensemble inférieur ou égal à l'élément passé en paramètre, ou null en l'absence d'un tel élément
  - `headSet(E toElement)` : retourne un objet de type SortedSet<E> contenant tous les éléments plus petits que l'élément passé en paramètre
  - `higher(E e)` : retourne le plus petit élément strictement plus grand que le paramètre passé à la méthode
  - `lower(E e)` : retourne le plus grand élément strictement plus petit que le paramètre passé à la méthode
  - `pollFirst()` : retrouve et retire le plus petit élément de la collection
  - `pollLast()` : retrouve et retire le plus grand élément de la collection
  - `subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)` : retourne un objet NavigableSet<E> contenant la portion définie en paramètre avec les limites inclusives ou exclusives de bornes minimum et maximum

TreeSet<E>

- Cet objet implémente plusieurs interfaces permettant de trier ses éléments
- Il implémente l'interface `SortedSet<E>`
- Cette interface complémente l'interface `Set<E>` sur deux points :
  - elle stipule que l'itérateur généré devra parcourir la collection dans le sens ascendant
  - elle comporte certaines méthodes supplémentaires comme first(), last() etc.
- Les éléments seront triés selon leur ordre naturel
- Cela signifie que tous les éléments ajoutés devront implémenter l'interface `Comparable<E>`, sinon vous aurez une belle exception de type `ClassCastException`
- **Remarque :** Le fait que les éléments soient triés est très pratique mais ceci a un impact sur les performances de votre application !

L'interface SortedSet<E>

- Cette interface fournit des méthodes permettant de considérer une structure dont les éléments sont triés. Voici la liste des méthodes :
  - `comparator()` : retourne le comparateur utilisé pour ranger les éléments ou null s'il utilise l'ordre naturel des éléments
  - `first()` : retourne le premier élément de la collection, donc le plus petit
  - `headSet(E toElement)` : retourne un objet de type SortedSet<E> contenant tous les éléments plus petits que l'élément passé en paramètre
  - `last()` : retourne le dernier élément de la collection, donc le plus grand
  - `subSet(E fromElement, E toElement)` : retourne un objet SortedSet<E> contenant la portion définie en paramètre
  - `tailSet(E fromElement)` : retourne un objet SortedSet<E> contenant les éléments plus grands que les paramètres passés à la méthode

```
import java.util.TreeSet;
public class Test {
   public static void main(String[] args) {
      TreeSet<Double> set = new TreeSet<Double>();
      set.add(12.52); set.add(-5); set.add(102.56);
      System.out.println(set);
      Double d = set.ceiling(12.52);
      System.out.println(d); //Retourne bien 12.52
      System.out.println(set.last()); //102.56
      //on extrait une sous-section de notre objet
      NavigableSet<Double> nSet = set.subSet(12.52, true, 102.56, false);
      System.out.println(nSet);
   }
}
```

Exercices

- En vous appuyant sur l'interface `Set`, implémentez une méthode pour chacune des opérations standards sur les ensembles (union, intersection, différence et différence symétrique). Ces méthodes prennent en paramètre des objets de type `Collection` et retourne une collection

Exercices

- En vous appuyant sur l'interface `Set`, implémentez une méthode pour chacune des opérations standards sur les ensembles (union, intersection, différence et différence symétrique). Ces méthodes prennent en paramètre des objets de type `Collection` et retourne une collection

```
<T> Collection<T> makeUnion(Collection<T> s1, Collection<T> s2){
  Set<T> union = new HashSet<T>(s1);
  union.addAll(s2);
  return union;
}

<T> Collection<T> makeIntersection(Collection<T> s1, Collection<T> s2){
  Set<Type> intersection = new HashSet<Type>(s1);
  intersection.retainAll(s2);
  return intersection;
}

<T> Collection<T> makeDifference(Collection<T> s1, Collection<T> s2){
  Set<Type> difference = new HashSet<Type>(s1);
  difference.removeAll(s2);
  return difference;
}

<T> Collection<T> makeDifferenceSym(Collection<T> s1, Collection<T> s2){
  Set<Type> symmetricDiff = new HashSet<Type>(s1);
  symmetricDiff.addAll(s2);
  Set<Type> tmp = new HashSet<Type>(s1);
  tmp.retainAll(s2);
  symmetricDiff.removeAll(tmp);
}
```

Table des matières

L'interface `List`

- L'interface `List` propose, en plus des méthodes implémentées dans l'interface `Collection`, les fonctionnalités suivantes :
  - **Accès via la position** - permet de manipuler les éléments en fonction de leur position dans la liste. Les méthodes sont : get, set, add, addAll, et remove
  - **Recherche** - recherche dans la liste un objet spécifique et retourne sa position. Les méthodes sont : indexOf et lastIndexOf
  - **Parcours** : surcharge la méthode `iterator` afin de prendre en compte la nature séquentiel de la liste. La méthode listIterator permet de retourner un `Iterator`
  - **Sous liste** - permet la manipulation de sous listes
- L'API Java fournit deux implémentations de liste :
  - `ArrayList` - gère la liste avec un tableau. Cette implémentation est très efficace
  - `LinkedList` - gère la liste avec une liste chaînée. Cette implémentation fonctionne bien sous certaines conditions

Accès via la position et recherche

- Les opérations de bases prenant en compte la position des éléments dans la liste sont :
  - `E get(int i)` : retourne l'élément en position `i`
  - `void set(int i, E e)` : modifie l'élément en position `i` en le remplaçant par `e`
  - `E remove(int i)` : supprime et retourne l'élément en position `i`
  - `void add(E e)` : ajoute l'élément en fin de liste
- Attention : `IndexOutOfBoundsException` si (i < 0 || i >= size())
- Les opérations `indexOf` et `lastIndexOf` retournent respectivement le premier et le dernier indice de l'élément passé en paramètre et -1 si l'élément n'existe pas

Exercice

- Proposez une méthode pour échanger deux éléments d'une liste

Exercice

- Proposez une méthode pour échanger deux éléments d'une liste

```
public static <E> void swap(List<E> a, int i, int j)
{
  E tmp = a.get(i);
  a.set(i, a.get(j));
  a.set(j, tmp);
}
```

Exercice

- Proposez une méthode pour échanger deux éléments d'une liste

```
public static <E> void swap(List<E> a, int i, int j)
{
  E tmp = a.get(i);
  a.set(i, a.get(j));
  a.set(j, tmp);
}
```

- Proposez un algorithme pour *shuffle* une liste (vous avez en paramètre un objet de type `Random` et vous pouvez utiliser la méthode `nextInt(i)` pour avoir le $i^{ème}$ nombre aléatoire)

```
public static void shuffle(List<?> list, Random rnd)
{
  for (int i = list.size(); i > 1; i--)
      swap(list, i - 1, rnd.nextInt(i));
}
```

La méthode `ListIterator`

- L'itérateur de base retourné par la méthode `iterator` permet de parcourir les éléments dans l'ordre
- L'itérateur retourné par la méthode `ListIterator` permet en plus de se déplacer dans **toutes les directions**
- Les trois méthodes héritées (`hasNext`, `next` et `remove`) de la classe `Iterator` **font exactement la même chose pour les deux méthodes**
- Nous avons deux méthodes supplémentaires qui sont :
  - `hasPrevious` - retourne vrai si pas en début de liste
  - `previous` - retourne l'élément précédent la position actuel du curseur

```
ListIterator<Type> it = list.listIterator(list.size());
while (it.hasPrevious()) {
  Type t = it.previous();
  ...
}
```

- Deux implémentations de la méthode `listIterator` sont disponibles :
  - sans argument - pointe le début de la liste
  - avec une position - pointe la position dans la liste fournit en argument

Exercice

- En supposant que la liste est circulaire, c'est-à-dire que le précédent du premier élément est le dernier, implémentez une méthode qui prend en paramètre un élément et retourne un itérateur sur l'élément précédent ou `null` si l'élément n'est pas dans la liste

Exercice

- En supposant que la liste est circulaire, c'est-à-dire que le précédent du premier élément est le dernier, implémentez une méthode qui prend en paramètre un élément et retourne un itérateur sur l'élément précédent ou `null` si l'élément n'est pas dans la liste

```
<E> void searchPrevious(List<E> list, E val) {
  ListIterator<E> it = list.listIterator();
  if(!it.hasNext()) return null;
  if(val.equals(it.next()))
    return list.listIterator(list.size());

  while(it.hasNext()){
    if(val.equals(it.next())) return it.previous();
  }

  return null;
}
```

Les sous listes

- La méthode `subList(int fromIndex, int toIndex)`, retourne une liste vue comme la portion de la liste entre `fromIndex`, inclue, à `toIndex`, exclu
- Cette méthode retourne physiquement la sous liste : cela signifie que toute modification sur la sous liste a un impact sur la liste initial
- Il est donc possible de supprimer une sous liste de la manière suivante :

```
list.subList(fromIndex, toIndex).clear();
```

- Il faut faire très attention lorsque des opérations sont réalisées sur la liste initial tandis que des modifications sont aussi réalisées sur la sous liste : **comportement indéterminé ! ! !**

Quelques algorithmes disponibles pour les listes

- `sort` : tri les éléments de la liste
- `shuffle` : permute aléatoirement les éléments d'une liste
- `reverse` : inverse les éléments d'une liste
- `rotate` : fait une rotation d'un certain nombre de case des éléments
- `swap` : échange deux éléments en fonction de leur position
- `replaceAll` : remplace toutes les occurrence d'un élements par un autre
- `copy` : copie la liste source dans la liste destination
- `binarySearch` : recherche un élément dans un liste triée dichotomiquement
- `indexOfSubList` : retourne l'indice du premier élément de la première sous liste d'une liste qui est égal à une autre
- `lastIndexOfSubList` : retourne l'indice du premier élément de la dernière sous liste d'une liste qui est égal à une autre

Table des matières

L'interface Queue

- Un objet de type `Queue` est une structure de données permettant la manipulation des données suivant une certaine priorité

```
public interface Queue<E> extends Collection<E> {
  E element();
  boolean offer(E e);
  E peek();
  E poll();
  E remove();
}
```

- Il existe deux comportements pour ces méthodes :

Opération	Lance une exception	Retourne une valeur spéciale
Insertion	`add(e)`	`offer(e)`
Suppression	`remove()`	`poll()`
Examination	`element()`	`peek()`

- L'instanciation d'une `Queue` peut se faire via une `Collection` :

```
Queue<Integer> queue = new LinkedList<Integer>();
```

- Typiquement, mais pas nécessairement, les objets de type `Queue` ordonne les éléments en FIFO. Parmi les exceptions, les files de priorité considère un ordre

Comportement des méthodes

- Il est possible de restreindre la capacité d'un objet de type `Queue`
- Quelques implémentations de `java.util.concurrent` sont bornées, mais les implémentations de `java.util` ne sont pas bornées

- **add** : insère un élément tant que la capacité de la `Queue` n'est pas atteinte (sinon `IllegalStateException`)
- **offer** : diffère uniquement de la méthode `add` par le fait qu'elle retourne `false` si la capacité de la `Queue` est atteinte
- **remove** : supprime et retourne l'élément en tête. Si la file est vide alors `NoSuchElementException` est lancée
- **poll** : comme pour `remove`. Si la file est vide alors `null` est retournée
- **element** : retourne l'élément en tête mais ne le supprime pas. Si la file est vide alors `NoSuchElementException` est lancée
- **peek** : comme pour `element`. Si la file est vide alors `null` est retournée

Exercice

- Remplissez une file avec des entiers et réalisez un compte à rebours en défilant les éléments de cette dernière. Vous pouvez utiliser `Thread.sleep(1000)` en ce qui concerne le timer.

Exercice

- Remplissez une file avec des entiers et réalisez un compte à rebours en défilant les éléments de cette dernière. Vous pouvez utiliser `Thread.sleep(1000)` en ce qui concerne le timer.

```
import java.util.*;

public class Compteur {
  public static void main(String[] args) throws
      InterruptedException {
    int time = Integer.parseInt(args[0]);
    Queue<Integer> queue = new LinkedList<Integer>();

    for (int i = time; i >= 0; i--) queue.add(i);

    while (!queue.isEmpty()) {
      System.out.println(queue.remove());
      Thread.sleep(1000);
    }
  }
}
```

Table des matières

L'interface Deque

- L'interface `Deque` permet d'ajouter/supprimer des éléments en tête ou en queue
- Ainsi, il est possible d'implémenter simplement une pile ou une file
- Les méthodes offertes par cette interface sont :

Opération	En tête de l'instance `Deque`	En fin de l'instance `Deque`
Insertion	`addFirst(e)` `offerFirst()`	`addLast(e)` `offerLast()`
Suppression	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Examination	`elementFirst()` `peekFirst()`	`elementLast()` `peekLast()`

- Concernant le retour des méthodes, le comportement est identique à la class `Queue` (exception ou retourne un élément spécial)

Exercice

- Écrivez une classe ma pile (avec les méthodes `empiler`, `depiler` et `sommet`) générique en utilisant un objet de type `Deque`

Exercice

- Écrivez une classe ma pile (avec les méthodes `empiler`, `depiler` et `sommet`) générique en utilisant un objet de type `Deque`

```
import java.util.*;

public class Mapile<T> {
  private Deque<T> tmp;

  public Mapile(){
    tmp = new  new LinkedList<T>();
  }

  public void empiler(T e){
    tmp.addLast(e);
  }

  public T depiler() throws NoSuchElementException{
    return tmp.removeLast();
  }

  public T sommet() throws NoSuchElementException{
    return tmp.elementLast();
  }
}
```

Table des matières

L'interface Map

- Une Map est un objet permettant **d'associer des éléments avec des clefs**
- Il est important de noter qu'une Map ne peut pas contenir deux fois la même clef !
- L'interface Map inclut les opérations suivantes :
  - `containsKey(Object)` : retourne true si l'objet passé en paramètre correspond à une clé de la collection
  - `containsValue(Object)` : comme ci-dessus mais sur une valeur
  - `entrySet()` : donne le contenu de la map (`Set<Map.Entry<K,V> >`)
  - `get(Object key)` : retourne la valeur associée à la clé passée en paramètre ou `null` si cette clé n'existe pas
  - `keySet()` : retourne la liste des clés (`Set<K>`)
  - `put(K key, V value)` : ajoute la clé et la valeur dans la collection en retournant la valeur insérée. Si la clé existe déjà alors sa valeur est écrasée
  - `putAll(Map<? extends K,? extends V> m)` : ajoute le contenu de la collection en paramètre dans la collection appelante
  - `remove(Object key)` : supprime le couple clé-valeur associé à la clé passée en paramètre et retourne la valeur supprimée.
  - `values()` : retourne une objet de type `Collection<V>` contenant toutes les valeurs de la collection

Implémentations de l'interface Map

- Il existe trois principales implémentations de l'interface Map :
  - `HashMap` : implémentation utilisant une table de hachage pour stocker ses éléments, mais cet objet n'est pas thread-safe
  - `TreeMap` : implémentation qui stocke les éléments triés, de façon naturelle par défaut, mais utilisable avec un comparateur
  - `LinkedHashMap` : implémentation qui combine table de hachage et liens chaînés pour stocker ses éléments, ce qui facilite leur insertion et leur suppression

- Cependant, il en existe d'autres un peu moins utilisées mais qui ont une importance dans des situations bien particulières :
  - `WeakHashMap` : permet une suppression des clefs non référencées
  - `IdentityHashMap` : les clefs sont comparées avec `==` et pas `equals()`
  - `EnumMap` : toutes les clefs doivent provenir d'énumérations

HashMap<K,V> et LinkedHashMap<K,V>

- Ce sont les implémentations les plus utilisées lorsqu'on souhaite utiliser un Map
- La différence entre ces deux implémentations réside dans le fait que `HashMap<K,V>` ne conserve pas l'ordre d'ajout des éléments tandis que `LinkedHashMap<K,V>` le conserve

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

public class Test {
 public static void main(String[] args) {
   Map<Integer, String> hm = new HashMap<>();
   hm.put(10, "1");
   hm.put(20, "2");
   hm.put(30, "3");

   System.out.println("Parcours de l'objet HashMap : ");
   Iterator<Entry<Integer,String>>it=(hm.entrySet()).iterator();
   while(it.hasNext()){
     Entry<Integer, String> e = it.next();
     System.out.println(e.getKey() + " : " + e.getValue());
   }
 }
}
```

WeakHashMap<K,V>

- Cet objet embarque un mode de fonctionnement un peu particulier car les références contenues seront automatiquement supprimées dès lors que les clés ne sont plus accessibles
- **Ce nettoyage est réalisé par le `Garbage Collector`**

```
import java.util.WeakHashMap;

public class Test {
  public static void main(String[] args) {
    WeakHashMap<Integer, String> whm = new WeakHashMap<>();
    Integer k1 = new Integer(1), k2 = new Integer(1), k3 = new
        Integer(1);
    whm.put(k1, "toto");
    whm.put(k2, "titi");
    whm.put(k3, "tutu");

    System.out.println(whm);
    k1 = null;
    System.gc();

    System.out.println("\nAprès l'appel GC : ");
    System.out.println(whm);
  }
}
```

IdentityHashMap<K,V>

- Cet objet viole la contrainte de base des objets de types Map<K,V> car il n'utilise pas un système d'égalité (**`equals()`**) par objet mais par référence (**==**)

```
import java.util.HashMap;
import java.util.IdentityHashMap;

public class Main3 {
  public static void main(String[] args) {
    Integer i1 = new Integer(1);
    Integer i2 = new Integer(1);

    IdentityHashMap<Integer, String> ihm = new
        IdentityHashMap<>();
    ihm.put(i1, "toto");
    ihm.put(i2, "titi");
    //Ici, vu que i1 == i2 => false
    //nous aurons deux entrées dans la collection
    System.out.println(ihm);

    //Ces trois instructions seront différentes
    System.out.println(ihm.get(i1));
    System.out.println(ihm.get(i2));
    System.out.println(ihm.get(1));
  }
}
```

EnumMap<K,V>

- Les clés doivent provenir de l'énumération définie à l'instanciation de l'objet
- La valeur null n'est pas autorisée comme clé !

```
import java.util.EnumMap;

public class Test {
  public static void main(String[] args) {
    EnumMap<Days, String> em = new EnumMap<>(Days.class);
    em.put(Days.LUNDI, "au soleil");
    em.put(Days.MARDI, "c'est permis");
    em.put(Days.MERCREDI, "club dorothée");
    em.put(Days.JEUDI, "noir");
    em.put(Days.VENDREDI, "essais");
    em.put(Days.SAMEDI, "qualification");
    em.put(Days.DIMANCHE, "formule 1");

    System.out.println(em);
    for(Days d : Days.values())
       System.out.println(em.get(d));
  }
}
```

Interfaces SortedMap<K,V> et NavigableMap<K,V>

- Il est parfois nécessaire de conserver l'ordre dans lequel les éléments ont été ajoutés à une Map
- Pour réaliser cela l'API Java propose l'utilisation de deux interfaces :
  - `SortedMap<K,V>` : permet d'avoir une collection dont les éléments possèdent une relation d'ordre
  - `NavigableMap<K,V>` : permet de parcourir une collection
- Ces deux interfaces ajoutent des comportements à l'interface de base
- Elles n'ont qu'une implémentation directe, l'objet `TreeMap<K,V>`

L'interfaces SortedMap<K,V>

- Cette interface permet d'avoir une collection dont les éléments sont triés selon leur ordre naturel et ce tri sera visible lorsque vous parcourrez votre collection
- Il est important de noter que le tri s'effectue sur les clés
- Ce seront donc les objets utilisés comme clé qui devront implémenter l'interface `Comparable<T>`. Les méthodes de l'interface :
  - `comparator()` : retourne le comparateur utilisé pour ranger les clé de la collection, ou null si celle-ci utilise l'ordre naturel des clés
  - `entrySet()` : retourne un Set<Map.Entry<K,V» représentant une vue des couple clé-valeur de la collection
  - `firstKey()` : retourne la première clé de la collection.
  - `headMap(K toKey)` : retourne une partie de la collection allant du début jusqu'au paramètre, exclu
  - `lastKey()` : retourne la plus grande de la collection
  - `subMap(K fromKey, K toKey)` : retourne une sous-collection des clés comprises entre le premier paramètre, inclus, et le deuxième paramètre, exclu
  - `tailMap(K fromKey)` : retourne une collection où les clés sont supérieures ou égales au paramètre.
  - `values()` : retourne une collection contenant la liste des valeurs

L'interfaces NavigableMap<K,V> I

- Cette interface rajoute toute une batterie de méthodes qui permettent la navigation dans votre collection
- Elle permet aussi et surtout de pouvoir naviguer dans votre collection dans les deux sens (des valeurs vers les clefs). Les méthodes de cette interface sont :
  - `ceilingEntry(K key)` : retourne un objet Map.Entry<K,V> correspondant à la plus petite clé supérieure ou égale au paramètre, ou null s'il n'y a pas de clé
  - `ceilingKey(K key)` : retourne la plus petite clé supérieure ou égale au paramètre, ou null s'il n'y a pas de clé
  - `descendingKeySet()` : retourne un objet NavigableSet<K> correspondant aux clés de la collection, mais dans l'ordre inverse de celui de la collection
  - `descendingMap()` : retourne un objet NavigableMap<K,V> contenant notre collection dans l'ordre inverse
  - `firstEntry()` : retourne un objet Map.Entry<K,V> représentant le couple clé-valeur de la plus petite clé de la collection, ou null s'il n'y a pas de clé

L'interfaces NavigableMap<K,V> II

- `floorEntry(K key)` : idem que ci-dessus, mais sur la clé la plus grande inférieure ou égale au paramètre
- `floorKey(K key)` : retourne la plus grande clé inférieure ou égale au paramètre, ou null s'il n'y a pas de clé
- `headMap(K toKey, boolean inclusive)` : retourne un objet NavigableMap<K,V> qui contiendra tous les couples clé-valeur dont la clé est inférieure au paramètre (ou égale si le deuxième paramètre est à true)
- `higherEntry(K key)` : retourne un objet Map.Entry<K,V> représentant le couple clé-valeur de la plus petite clé de la collection strictement supérieure au paramètre, ou null s'il n'y a pas de clé
- `higherKey(K key)` : idem que ci-dessus mais retourne uniquement la clé
- `lastEntry()` : retourne un objet Map.Entry<K,V> représentant le couple clé-valeur de la plus grande clé de la collection, ou null s'il n'y a pas de clé
- `lowerEntry(K key)` : idem que ci-dessus mais pour la plus petite clé de la collection
- `lowerKey(K key)` : retourne la plus grande clé strictement plus petite que le paramètre

L'interfaces NavigableMap<K,V> III

- `navigableKeySet()` : retourne un objet NavigableSet<K> représentant une vue des clés de la collection.
- `pollFirstEntry()` : retourne un objet Map.Entry<K,V> correspondant à la plus petite clé, tout en supprimant cette entrée de la collection. Renvoie null si la clé n'existe pas
- `pollLastEntry()` : idem que ci-dessus mais pour la plus grande clé de la collection
- `subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)` : retourne une portion de la collection délimitée par les clés passées en paramètre et où les booléens servent à déterminer si les limites sont inclusives ou exclusives
- `tailMap(K fromKey, boolean inclusive)` : retourne un objet NavigableMap<K,V> qui contiendra tous les couples clé-valeur dont la clé est strictement supérieure au paramètre (ou supérieure ou égale si le deuxième paramètre est à true)

L'objet TreeMap<K,V>

- Cet objet est donc trié selon l'ordre naturel des clés le constituant
- Vous avez aussi la possibilité d'utiliser un comparateur si cet ordre ne vous convient pas
- Par contre, cet objet n'est pas synchronisé !

```
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class Test {
  public static void main(String[] args) {
    TreeMap<Integer, String> tm = new TreeMap<>();
    tm.put(10, "10"); tm.put(50, "50"); tm.put(20, "20");
    System.out.println(tm); // affiche trié

    System.out.println(tm.headMap(22)); // {10=10, 20=20}
    System.out.println(tm.tailMap(22)); // {30=30,40=40,50=50}
    System.out.println(tm.values()); // [10, 20, 30, 40, 50]
    System.out.println(tm.ceilingEntry(32)); // 40=40

    Iterator<Map.Entry<Integer, String>> it =
        (tm.entrySet()).iterator();
    while(it.hasNext()) System.out.println(it.next());
  }
}
```

Exercice

- Proposez une approche pour que l'affichage de votre Map soit réalisé dans le sens inverse de celui obtenu via la méthode `compare` de l'interface `Comparator`

Exercice

- Proposez une approche pour que l'affichage de votre Map soit réalisé dans le sens inverse de celui obtenu via la méthode `compare` de l'interface `Comparator`

```
import java.util.Comparator;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class Test {
  public static void main(String[] args) {
    TreeMap<Integer, String> tm = new TreeMap<>(new
        Comparator<Integer>(){
      public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
      }
    });

    tm.put(10, "10"); tm.put(50, "50"); tm.put(20, "20");
    System.out.println(tm);
  }
}
```

Les interfaces ConcurrentMap<K,V> et ConcurrentNavigableMap<K,V>

- De nombreuses applications sont `multi-thread`
- Il est donc nécessaires de s'assurer de l'intégrité des données en proposant des structures permettant un accès concurrentiel aux données
- Pour cela, l'API Java propose deux interfaces :
  - `ConcurrentMap<K,V>`
  - `ConcurrentNavigableMap<K,V>`
- Chacune de ces deux interfaces possède une implémentation héritant aussi de la classe `AbstractMap<K,V>`

L'interface ConcurrentMap<K,V>

- Cette interfaces ajoute quelques méthodes à `Map<K,V>` :
  - `putIfAbsent(K key, V value)` : si la clef `key` n'est pas déjà présente dans la collection, alors insérer le couple dans la collection. Cette méthode retourne la valeur correspondant à la clé passée, donc si elle existe déjà alors la valeur présente dans la collection sera retournée
  - `remove(K key, V value)` : supprime le couple clé/valeur passée en paramètre. Si ce couple n'est pas présent, aucune suppression n'est faite. Retourne true si la suppression est effective
  - `replace(K key, V value)` : fonctionne globalement comme la méthode ci-dessus mais ne supprime pas le couple clé/valeur. La valeur de la clé est remplacée et valeur modifiée est retournée ou null si la clé n'existe pas
  - `replace(K key, V oldValue, V newValue)` : comme la méthode ci-dessus mais contrôle l'existence du couple clé/valeur. Retourne un booléen représentant le résultat.
- L'objet `ConcurrentHashMap<K,V>` implémente cette interface
- Cette implémentation autorise autant de lectures que vous le voulez en environnement multi-thread
- Donc pas de `ConcurrentModificationException`

L'interface ConcurrentNavigableMap<K,V>

- Cette interface permet de lier l'interface `ConcurrentNavigableMap<K,V>` et l'interface `NavigableMap<K,V>`
- Ceci permet donc d'avoir une collection qui a les propriétés de ces deux interfaces : l'utilisation en milieu multit-hread et toute une panoplie de méthodes qui permettent de naviguer dans votre collection
- Voir les méthodes de l'interface `NavigableMap<K,V>`
- Une implémentation de cette interface est disponible : l'objet `ConcurrentSkipListMap<K,V>`
- C'est une collection triée selon l'ordre naturel de ses clés (sauf si vous utilisez un comparateur), et n'accepte pas de valeur null