Les collections

- Les collections sont des objets qui permettent de gérer des ensembles d'objets.
- Ces ensembles de données peuvent être définis avec plusieurs caractéristiques :
- la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...
- Chaque objet contenu dans une collection est appelé un élément.

Présentation du framework collection

- Dans la version 1 du J.D.K., il n'existe qu'un nombre restreint de classes pour gérer des ensembles de données :
- · Vector
- · Stack
- · Hashtable
- · Bitset
- L'interface Enumeration permet de parcourir le contenu de ces objets.

Les interfaces à utiliser par des objets qui gèrent des collections sont :

- **Collection** : interface qui est implementée par la plupart des objets qui gèrent des collections
- **Map** : interface qui définit des méthodes pour des objets qui gèrent des collections sous la forme clé/valeur
- **Set** : interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble
- **List** : interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément
- **SortedSet** : interface qui étend l'interface Set et permet d'ordonner l'ensemble
- **SortedMap** : interface qui étend l'interface Map et permet d'ordonner l'ensemble

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés :

- **HashSet** : HashTable qui implémente l'interface Set
- **TreeSet** : arbre qui implémente l'interface SortedSet
- **ArrayList** : tableau dynamique qui implémente l'interface List
- **LinkedList** : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
- **HashMap** : HashTable qui implémente l'interface Map
- **TreeMap** : arbre qui implémente l'interface SortedMap

Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :

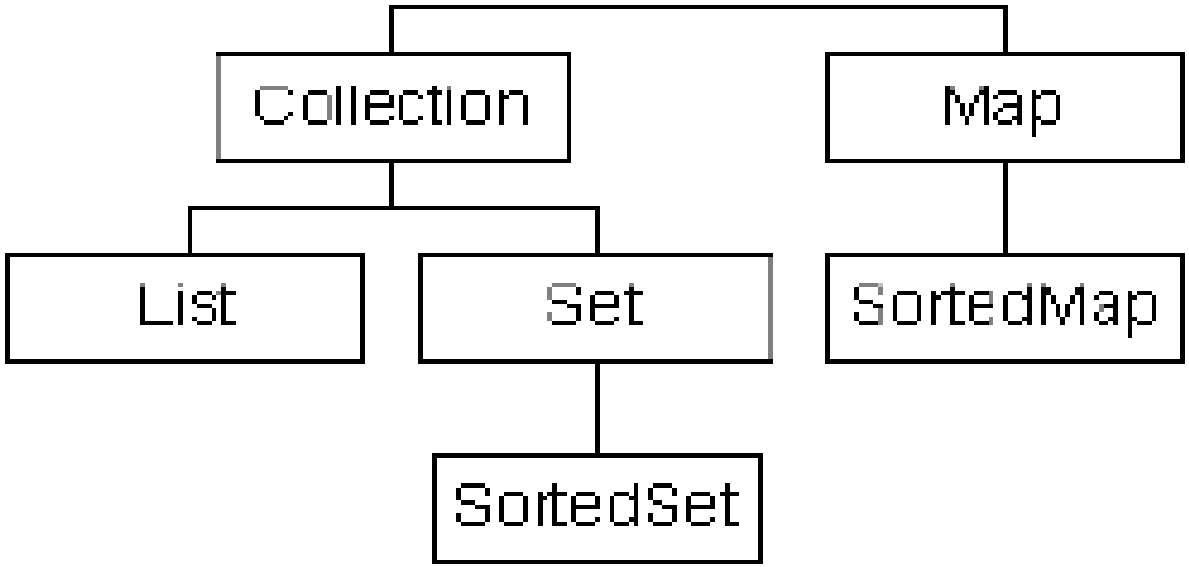
- **Iterator** : interface pour le parcours des collections
- **ListIterator** : interface pour le parcours des listes dans les deux sens et modifier les éléments lors de ce parcours
- **Comparable** : interface pour définir un ordre de tri naturel pour un objet
- **Comparator** : interface pour définir un ordre de tri quelconque

Deux classes existantes dans les précédentes versions du JDK ont été modifiées pour implémenter certaines interfaces du framework :

- **Vector** : tableau à taille variable qui implémente maintenant l'interface List
- **HashTable** : table de hashage qui implémente maintenant l'interface Map

Les interfaces des collections

- Le framework de java 2 définit 6 interfaces en relation directe avec les collections qui sont regroupées dans deux arborescences :



- Le JDK ne fourni pas de classes qui implémentent directement l'interface Collection.
- Le tableau ci dessous présente les différentes classes qui implémentent les interfaces de bases Set, List et Map :

	Set	List	Map
Tableau redimensionnable		ArrayList,Vector (JDK 1.1)	
Arbre	TreeSet		TreeMap
Liste chaînée		LinkedList	
Collection sous la forme de paire clé/valeur	HashSet		HashMap, HashTable (JDK 1.1)
Classes du JDK 1.1		Stack	

L'interface Collection

- Cette interface définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale.
- Elle est la super interface de plusieurs interfaces du framework.
- Plusieurs classes qui gèrent une collection implémentent une interface qui hérite de l'interface Collection.
- Cette interface est une des deux racines de l'arborescence des collections.
- Cette interface définit plusieurs méthodes :

Méthode	Rôle
boolean add(Object)	ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
boolean addAll(Collection)	ajoute à la collection tous les éléments de la collection fournie en paramètre
void clear()	supprime tous les éléments de la collection
boolean contains(Object)	indique si la collection contient au moins un élément identique à celui fourni en paramètre
boolean containsAll(Collection)	indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean isEmpty()	indique si la collection est vide
Iterator iterator()	renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection
boolean remove(Object)	supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
boolean removeAll(Collection)	supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
int size()	renvoie le nombre d'éléments contenu dans la collection
Object[] toArray()	renvoie d'un tableau d'objets qui contient tous les éléments de la collection

L'interface Iterator

- Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection.

Méthode	Rôle
boolean hasNext()	indique si il reste au moins à parcourir dans la collection
Object next()	renvoie la prochain élément dans la collection
void remove()	supprime le dernier élément parcouru

Exemples

```
Iterator iterator = collection.Iterator();
while (iterator.hasNext()) {
System.out.println(“objet = ”+iterator.next());
}
Iterator iterator = collection.Iterator();
if (iterator.hasNext()) {
iterator.next();
itérator.remove();
}
```

Si aucun appel à la méthode next() ne correspond à celui de la méthode remove(), une exception de type IllegalStateException est levée

Les listes

- Une liste est une collection ordonnée d'éléments qui autorise d'avoir des doublons.
- Etant ordonnée, un élément d'une liste peut être accédé à partir de son index.

L'interface List

- Cette interface étend l'interface Collection.
- Les collections qui implémentent cette interface autorisent les doublons dans les éléments de la liste.
- Ils autorisent aussi l'insertion d'éléments null.
- L'interface List propose plusieurs méthodes pour un accès à partir d'un index aux éléments de la liste. La gestion de cet index commence à zéro.
- Pour les listes, une interface particulière est définie pour assurer le parcours dans les deux sens de la liste et assurer des mises à jour : l'interface ListIterator

Méthode	Rôle
ListIterator listIterator()	renvoie un objet capable de parcourir la liste
Object set (int, Object)	remplace l'élément contenu à la position précisée par l'objet fourni en paramètre
void add(int, Object)	ajouter l'élément fourni en paramètre à la position précisée
Object get(int)	renvoie l'élément à la position précisée
int indexOf(Object)	renvoie l'index du premier élément fourni en paramètre dans la liste ou −1 si l'élément n'est pas dans la liste
ListIterator listIterator()	renvoie un objet pour parcourir la liste et la mettre à jour
List subList(int,int)	renvoie un extrait de la liste contenant les éléments entre les deux index fournis (le premier index est inclus et le second est exclus). Les éléments contenus dans la liste de retour sont des références sur la liste originale. Des mises à jour de ces éléments impactent la liste originale.
int lastIndexOf(Object)	renvoie l'index du dernier élément fourni en paramètre dans la liste ou −1 si l'élément n'est pas dans la liste
Object set(int, Object)	remplace l'élément à la position indiquée avec l'objet fourni

Les listes chaînées : la classe LinkedList

- Cette classe hérite de la classe AbstractSequentialList et implémente donc l'interface List.
- Elle représente une liste doublement chaînée.
- Cette classe possède un constructeur sans paramètre et un qui demande une collection. Dans ce dernier cas, la liste sera initialisée avec les éléments de la collection fournie en paramètre..

```
import java.util.*;
public class ListExemple {
    public static void main(String args[]) {
        List list = new ArrayList();
        list.add("ALI");
        list.add("MOHAMED");
        list.add("SARA");
        list.add("ILHAM");
        list.add("NAWAL");
        System.out.println(list);
        System.out.println("2: " + list.get(2));
        System.out.println("0: " + list.get(0));
```

```
LinkedList queue = new LinkedList();
    queue.addFirst("Bernadine");
    queue.addFirst("Elisabeth");
    queue.addFirst("Gene");
    queue.addFirst("Elisabeth");
    queue.addFirst("Clara");
    System.out.println(queue);
    queue.removeLast();
    queue.removeLast();
    System.out.println(queue);
}}
```

Une liste chaînée gère une collection de façon ordonnée : l'ajout d'un élément peut se faire à la fin de la collection ou

après n'importe quel élément. Dans ce cas, l'ajout est lié à la position courante lors d'un parcours.

Pour répondre à ce besoin, l'interface qui permet le parcours de la collection est une sous classe de l'interface Iterator :

l'interface ListIterator.

- Comme les iterator sont utilisés pour faire des mises à jour dans la liste, une exception de type CurrentModificationException levé si un iterator parcours la liste alors qu'un autre fait des mises à jour (ajout ou suppression d'un élément dans la liste).
- Pour gérer facilement cette situation, il est préférable si l'on sait qu'il y ait des mises à jour à faire de n'avoir qu'un seul iterator qui soit utilisé.
- Plusieurs méthodes pour ajouter, supprimer ou obtenir le premier ou le dernier élément de la liste permettent d'utiliser cette classe pour gérer une pile :

Méthode	Rôle
void addFirst(Object)	insère l'objet en début de la liste
void addLast(Object)	insère l'objet en fin de la liste
Object getFirst()	renvoie le premier élément de la liste
Object getLast()	renvoie le dernier élément de la liste
Object removeFirst()	supprime le premier élément de la liste et renvoie le premier élément
Object removeLast()	supprime le dernier élément de la liste et renvoie le premier élément

L'interface ListIterator

- Cette interface définit des méthodes pour parcourir la liste dans les deux sens et effectuer des mises à jour qui agissent par rapport à l'élément courant dans le parcours.
- En plus des méthodes définies dans l'interface Iterator dont elle hérite, elle définit les méthodes suivantes :

Méthode	Roles
void add(Object)	ajoute un élément dans la liste en tenant de la position dans le parcours
boolean hasPrevious()	indique si il reste au moins un élément à parcourir dans la liste dans son sens inverse
Object previous()	renvoi l'élément précédent dans la liste
void set(Object)	remplace l'élément courante par celui fourni en paramètre

Les tableaux redimensionnables la classe ArrayList

- Cette classe représente un tableau d'objets dont la taille est dynamique.
- Elle hérite de la classe AbstractList donc elle implémente l'interface List.
- Le fonctionnement de cette classe est identique à celui de la classe Vector.
- La différence avec la classe Vector est que cette dernière est multi thread (toutes ces méthodes sont synchronisées).
- Pour une utilisation dans un thread unique, la synchronisation des méthodes est inutile et coûteuse. Il est alors préférable d'utiliser un objet de la classe ArrayList.
- Elle définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
boolean add(Object)	ajoute un élément à la fin du tableau
boolean addAll(Collection)	ajoute tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	ajoute tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	supprime tous les éléments du tableau
void ensureCapacity(int)	permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	renvoie l'élément du tableau dont la position est précisée
int indexOf(Object)	renvoie la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	indique si le tableau est vide
int lastIndexOf(Object)	renvoie la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	supprime dans le tableau l'élément fourni en paramètre
void removeRange(int,int)	supprime tous les éléments du tableau de la première position fourni incluse jusqu'à la dernière position fournie exclue
Object set(int, Object)	remplace l'élément à la position indiquée par celui fourni en paramètre
int size()	renvoie le nombre d'élément du tableau
void trimToSize()	ajuste la capacité du tableau sur sa taille actuelle

Les ensembles

- Un ensemble (Set) est une collection qui n'autorise pas l'insertion de doublons.

L'interface Set

- Cette classe définit les méthodes d'une collection qui n'accepte pas de doublons dans ces éléments. Elle hérite de
- l'interface Collection mais elle ne définie pas de nouvelle méthode.
- Pour déterminer si un élément est déjà inséré dans la collection, la méthode equals() est utilisée.
- Le framework propose deux classes qui implémentent l'interface Set : TreeSet et HashSet
- Le choix entre ces deux objets est liés à la nécessité de trié les éléments :
- · les éléments d'un objet HashSet ne sont pas triés : l'insertion d'un nouvel élément est rapide
- · les éléments d'un objet TreeSet sont triés : l'insertion d'un nouvel éléments est plus long

L'interface SortedSet

- Cette interface définit une collection de type ensemble triée. Elle hérite de l'interface Set.
- Le tri de l'ensemble peut être assuré par deux façons :
- · les éléments contenus dans l'ensemble implémentent l'interface Comparable pour définir leur ordre naturel
- · il faut fournir au constructeur de l'ensemble un objet Comparator qui définit l'ordre de tri à utiliser
- Elle définit plusieurs méthodes pour tirer parti de cette ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier l'ensemble
Object first()	renvoie le premier élément de l'ensemble
SortedSet headSet(Object)	renvoie un sous ensemble contenant tous les éléments inférieurs à celui fourni en paramètre
SortedSet subSet(Object, Object)	renvoie un sous ensemble contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedSet tailSet(Object)	renvoie un sous ensemble contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre

La classe HashSet

- Cette classe est un ensemble sans ordre de tri particulier.
- Les éléments sont stockés dans une table de hashage : cette table possède une capacité.

Exemples

```
import java.util.*;
public class TestHashSet {
public static void main(String args[]) {
Set set = new HashSet();
set.add("CCCCC");
set.add("BBBBB");
set.add("DDDDD");
set.add("BBBBB");
set.add("AAAAA");
Iterator iterator = set.iterator();
while (iterator.hasNext())
   {System.out.println(iterator.next());}}}
```

La classe TreeSet

- Cette classe est un arbre qui représente un ensemble trié d'éléments.
- Cette classe permet d'insérer des éléments dans n'importe quel ordre et de restituer ces éléments dans un ordre précis lors de son parcours.
- L'implémentation de cette classe insère un nouvel élément dans l'arbre à la position correspondant à celle déterminée par l'ordre de tri.
- L'insertion d'un nouvel élément dans un objet de la classe TreeSet est donc plus lent mais le tri est directement effectué.
- L'ordre utilisé est celui indiqué par les objets insérés si ils implémentent l'interface Comparable pour un ordre de tri
- naturel ou fournir un objet de type Comparator au constructeur de l'objet TreeSet pour définir l'ordre de tri.

Exemple

```
import java.util.*;
public class TestTreeSet {
public static void main(String args[]) {
Set set = new TreeSet();
set.add("CCCCC");
set.add("BBBBB");
set.add("DDDDD");
set.add("BBBBB");
set.add("AAAAA");
Iterator iterator = set.iterator();
while (iterator.hasNext())
   {System.out.println(iterator.next());}}}
```

Les collections gérées sous la forme clé/valeur

- Ce type de collection gère les éléments avec deux entités : une clé et une valeur associée.
- La clé doit être unique donc il ne peut y avoir de doublons. En revanche la même valeur peut être associées à plusieurs clés différentes.
- Avant l'apparition du framework collections, la classe dédiée à cette gestion était la classe Hashtable.

L'interface Map

- Cette interface est une des deux racines de l'arborescence des collections.
- Les collections qui implémentent cette interface ne peuvent contenir des doublons. Les collections qui implémentent cette interface utilise une association entre une clé et une valeur.
- Elle définit plusieurs méthodes pour agir sur la collection :

Méthode	Rôle
void clear()	supprime tous les éléments de la collection
boolean containsKey(Object)	indique si la clé est contenue dans la collection
boolean containsValue(Object)	indique si la valeur est contenue dans la collection
Set entrySet()	renvoie un ensemble contenant les valeurs de la collection
Object get(Object)	renvoie la valeur associée à la clé fournie en paramètre
boolean isEmpty()	indique si la collection est vide
Set keySet()	renvoie un ensemble contenant les clés de la collection
Object put(Object, Object)	insère la clé et sa valeur associée fournies en paramètres
void putAll(Map)	insère toutes les clés/valeurs de l'objet fourni en paramètre
Collection values()	renvoie une collection qui contient toutes les éléments des éléments
Object remove(Object)	supprime l'élément dont la clé est fournie en paramètre
int size()	renvoie le nombre d'éléments de la collection

L'interface SortedMap

- Cette interface définit une collection de type Map triée sur la clé. Elle hérite de l'interface Map.
- Le tri peut être assuré par deux façons :
- · les clés contenues dans la collection implémentent l'interface Comparable pour définir leur ordre naturel
- · il faut fournir au constructeur de la collection un objet Comparator qui définit l'ordre de tri à utiliser
- Elle définit plusieurs méthodes pour tirer parti de cette ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier la collection
Object first()	renvoie le premier élément de la collection
SortedSet headMap(Object)	renvoie une sous collection contenant tous les éléments inférieurs à celui fourni en paramètre
Object last()	renvoie le dernier élément de la collection
SortedMap subMap(Object, Object)	renvoie une sous collection contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedMap tailMap(Object)	renvoie une sous collection contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre

La classe Hashtable

- Cette classe qui existe depuis le premier jdk implémente une table de hachage.
- La clé et la valeur de chaque élément de la collection peut être n'importe quel objet non nul.
- A partir de Java 1.2 cette classe implémente l'interface Map.
- Une des particularités de classe HashTable est quelle est synchronisée.

```
import java.util.*;
public class TestHashtable {
public static void main(String[] args) {
Hashtable htable = new Hashtable();
htable.put(new Integer(3), "données
   3");
htable.put(new Integer(1), "données
   1");
htable.put(new Integer(2), "données
   2");
System.out.println(htable.get(new
```

La classe TreeMap

- Cette classe gère une collection d'objets sous la forme clé/valeur stockés dans un arbre de type rouge?noir (Red?black tree).
- Elle implémente l'interface SortedMap. L'ordre des éléments de la collection est maintenu grace à un objet de type Comparable.
- Elle possède plusieurs constructeurs dont un qui permet de préciser l'objet Comparable pour définir l'ordre dans la collection.

Exemple

```
import java.util.*;
public class TestTreeMap {
public static void main(String[] args) {
TreeMap arbre = new TreeMap();
arbre.put(new Integer(3), "données 3");
arbre.put(new Integer(1), "données 1");
arbre.put(new Integer(2), "données 2");
Set cles = arbre.keySet();
Iterator iterator = cles.iterator();
while (iterator.hasNext()) {
System.out.println(arbre.get(iterator.next()));
}}}
```

La classe HashMap

- La classe HashMap est similaire à la classe Hashtable. Les trois grandes différences sont :
- · elle est apparue dans le JDK 1.2
- · elle n'est pas synchronisée
- · elle autorise les objets null comme clé ou valeur
- Cette classe n'étant pas synchonisée, pour assurer la gestion des accès concurents sur cet objet, il faut l'envelopper dans un objet Map en tulisant la méthode synchronizedMap de la classe Collection.

Le tri des collections

- L'ordre de tri est défini grace à deux interfaces :
- · Comparable
- · Comparator

L'interface Comparable

- Tous les objets qui doivent définir un ordre naturel utilisé par le tri d'une collection avec cet ordre doivent implémenter cette interface.
- Cette interface ne définit qu'une seule méthode : int compareTo(Object).
- Cette méthode doit renvoyer :
- · une valeur entière négative si l'objet courant est inférieur à l'objet fourni
- · une valeur entière positive si l'objet courant est supérieur à l'objet fourni
- · une valeur nulle si l'objet courant est égal à l'objet fourni
- Les classes wrappers, String et Date implémentent cette interface.

L'interface Comparator

- Cette interface représente un ordre de tri quelconque. Elle est utile pour permettre le tri d'objet qui n'implémente pas l'interface Comparable ou pour définir un ordre de tri différent de celui défini avec Comparable (l'interface Comparable représente un ordre naturel : il ne peut y en avoir qu'un)
- Cette interface ne définit qu'une seule méthode : int compare(Object, Object).
- Cette méthode compare les deux objets fournis en paramètre et renvoie :
- · une valeur entière négative si le premier objet est inférieur au second
- · une valeur entière positive si le premier objet est supérieur au second

- La classe Collections propose plusieurs méthodes statiques qui effectuer des opérations sur des collections.
- Ces traitements sont polymorphiques car ils demandent en paramètre un objet qui implémente une interface et retourne une collection.

Méthode	Rôle
void copy(List, List)	copie tous les éléments de la seconde liste dans la première
Enumaration enumeration(Collection)	renvoie un objet Enumeration pour parcourir la collection
Object max(Collection)	renvoie le plus grand élément de la collection selon l'ordre naturel des éléments
Object max(Collection, Comparator)	renvoie le plus grand élément de la collection selon l'ordre naturel précisé par l'objet Comparator
Object min(Collection)	renvoie le plus petit élément de la collection selon l'ordre naturel des éléments
Object min(Collection, Comparator)	renvoie le plus petit élément de la collection selon l'ordre précisé par l'objet Comparator
void reverse(List)	inverse l'ordre de la liste fournie en paramètre
void shuffle(List)	réordonne tous les éléments de la liste de façon aléatoire
void sort(List)	trie la liste dans un ordre ascendant selon l'ordre naturel des éléments
void sort(List, Comparator)	trie la liste dans un ordre ascendant selon l'ordre précisé par l'objet Comparator

Exemple

```
import java.util.*;
public class TestUnmodifiable{
public static void main(String args[])
{
List list = new LinkedList();
list.add("1");
list.add("2");
list = Collections.unmodifiableList(list);
list.add("3");
}}
```

- L'utilisation d'une méthode synchronizedXXX() renvoie une instance de l'objet qui supporte la synchronisation pour les opérations d'ajout et de suppression d'éléments. Pour le parcours de la collection avec un objet Iterator, il est nécessaire de synchroniser le bloc de code utilisé pour le parcours.
- Il est important d'inclure aussi dans ce bloc l'appel à la méthode pour obtenir l'objet de type Iterator utilisé pour le parcours.

```
import java.util.*;
public class TestSynchronized{
public static void main(String args[])
{
List maList = new LinkedList();
maList.add("1");
maList.add("2");
maList.add("3");
maList = Collections.synchronizedList(maList);
synchronized(maList) {
Iterator i = maList.iterator();
while (i.hasNext())
System.out.println(i.next());}}}
```

Les exceptions du framework

- L'exception de type UnsupportedOperationException est levée lorsque qu'une opération optionnelle n'est pas supportée par l'objet qui gère la collection.
- L'exception ConcurrentModificationException est levée lors du parcours d'une collection avec un objet Iterator et que cette collection subi une modification structurelle.