ENSAH

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

Les collections en Java

Langage Java

2ème année Génie Informatique

Ecole Nationale des Sciences Appliquées – Al Hoceima

Prof A. Bahri
abahri@uae.ac.ma

A.U 2020/2021

Collection de données

❑ **Structures de données**

C'est l'organisation efficace d'un ensemble de données, sous la forme de tableaux, de listes, de piles etc.

❑ Les tableaux ne peuvent pas répondre à tous les besoins de stockage d’ensemble d’objets

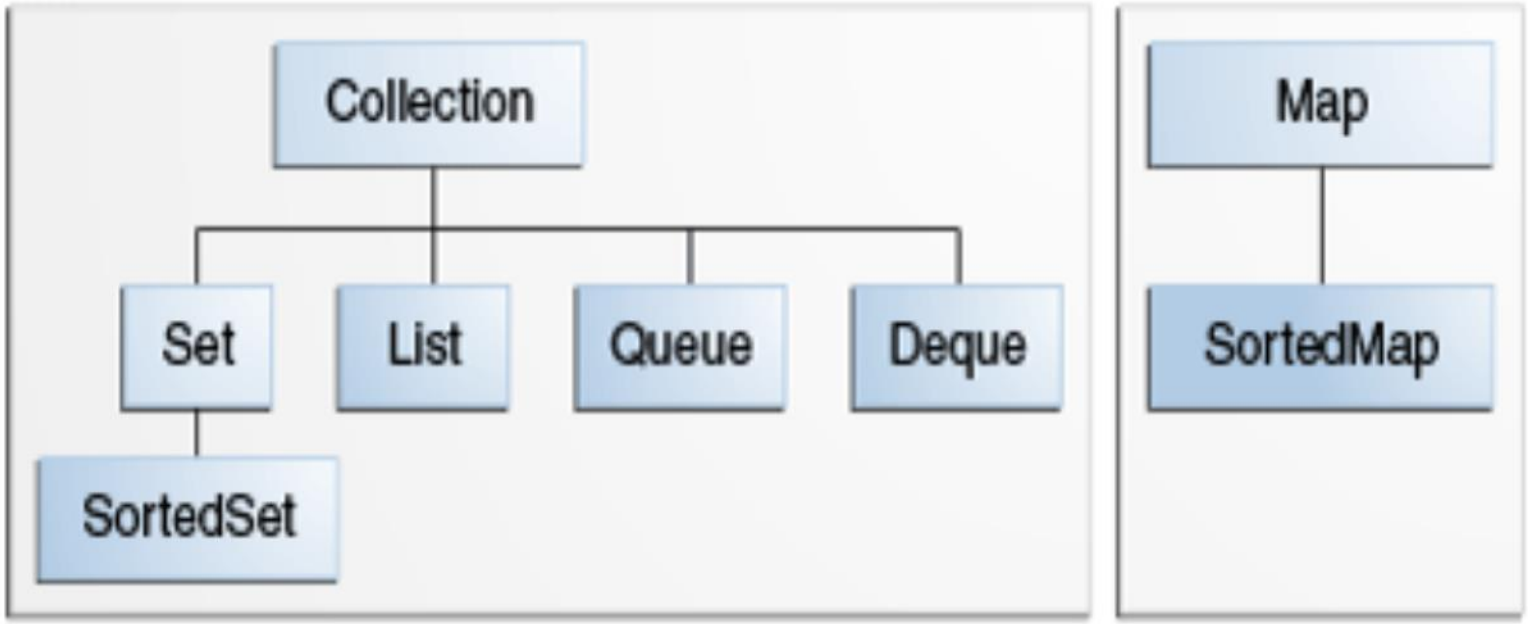
- Manque de fonctionnalités
- Éléments non structurés
- Taille fixe
- …

Les collections en Java

- Principales structures de données :
  - Vecteurs dynamiques
  - Ensembles
  - Listes chaînées
  - Queues
  - Tables associatives (dictionnaires)
  - Files d'attente
  - …
- But: gérer des ensembles d'objets
- Les collections!!

Collection de données

- Définition: « Une collection de données est un conteneur d'éléments de même type qui possède un protocole particulier pour l'ajout, le retrait et la recherche d'éléments »
- Les classes collection sont définies dans le package *java.util*
- Deux grandes familles de collections (Collection et Map) chacune définie par une interface de base
  - qui implémentent elles-mêmes l'interface Collection
  - qu'elles complètent par leurs fonctionnalités propres



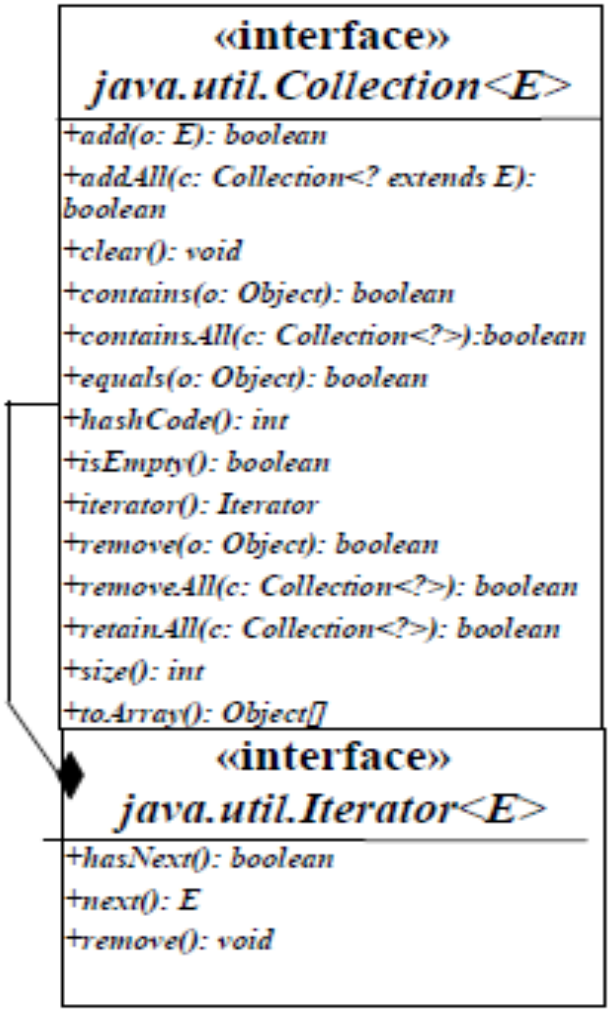
API des collections

- Ensemble d'interfaces et de classes permettant de stocker de multiples objets

- Le Framework Java collections prend en charge quatre types de collections:
  - Listes: **List**: éléments ordonnés qui accepte les doublons
  - Ensembles: **Set**: éléments non ordonnés par défaut qui n'accepte pas les doublons
  - Dictionnaire: **Map**: association de paire clé/valeur
  - **Queue** et **Deque**: éléments stockés dans un certain ordre avant d'être extraits pour traitement

L'interface Collection

L'interface Collection est l'interface racine pour manipuler une collection d'objets.



Ajoute un nouveau joint d'éléments à cette collection.
Ajoute tous les éléments de la collection c à cette collection.
Supprime tous les éléments de cette collection.
Retourne true si cette collection contient l'élément o.
Retourne true si cette collection contient tous les éléments c.
Retourne true si cette collection est égale à une autre collection o.
Retourne le code de hachage pour cette collection.
Retourne true si cette collection contient aucun élément.
Retourne un itérateur pour les éléments de cette collection.
Supprime l'élément o de cette collection.
Supprime tous les éléments de c de cette collection.
Conserve les éléments qui sont à la fois en C et dans cette collection.
Retourne le nombre d'éléments dans cette collection.
Retourne un tableau d'objets pour les éléments de cette collection.

Retourne vrai si cet itérateur a plus d'éléments à parcourir.
Retourne l'élément suivant de cette itération.
Supprime le dernier élément obtenu en utilisant la méthode suivante.

Objets implémentant les collections

- Chaque collection est fournie sous forme d'une «classe» dédiée qui implémente les interfaces nécessaires :
  - Les listes chaînées: classe LinkedList
  - Les vecteurs dynamiques : classes ArrayList et Vector
  - Les ensembles: classes HashSet et TreeSet
  - Les queues avec priorité : classe PriorityQueue
  - Les queues à double entrée : classe ArrayDeque

- Et les tables associatives (qui n'implémentent pas Collection à la base mais Map) : classes HashMap et TreeMap

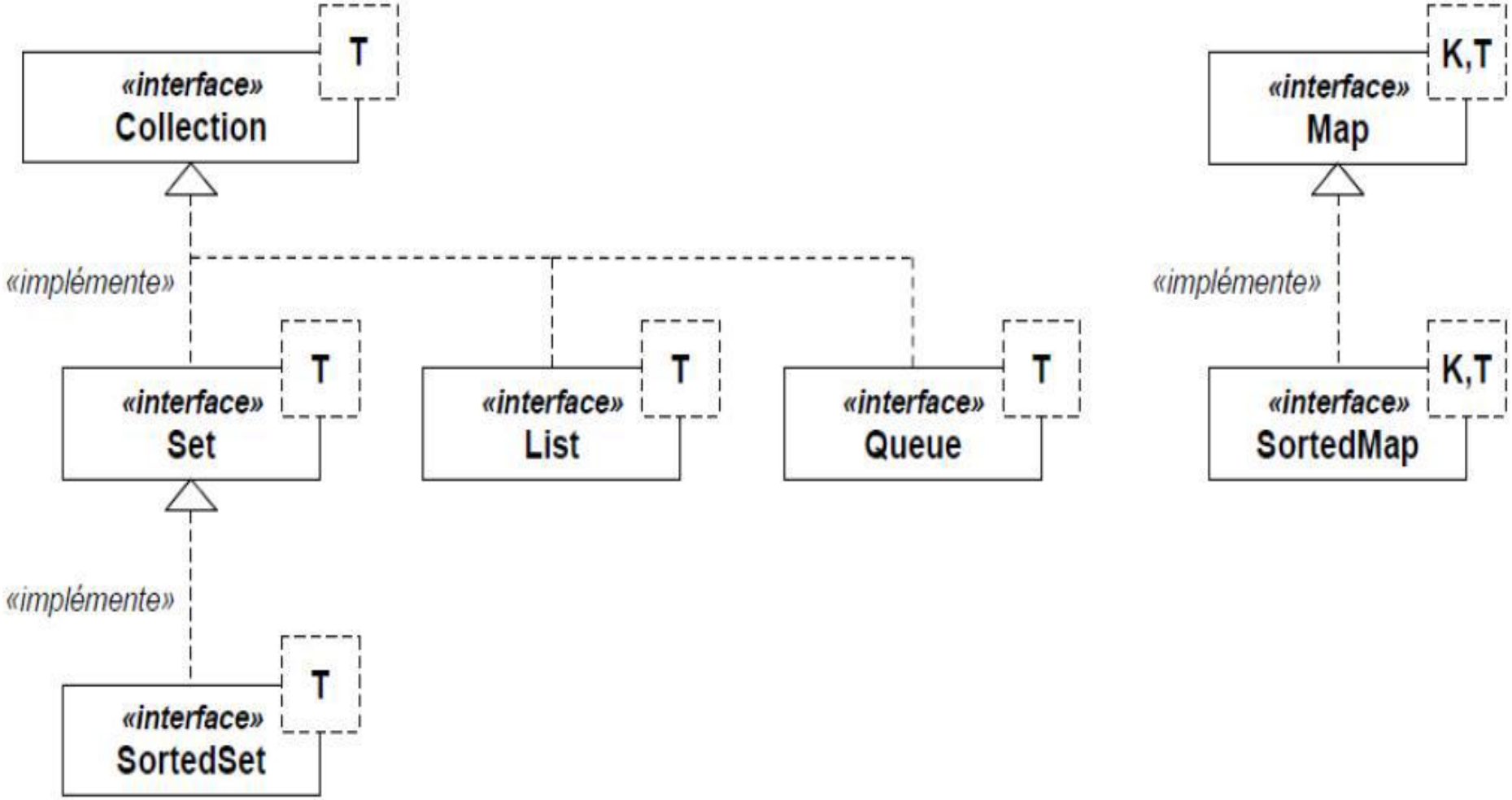
Opérations de base sur une collection

- Constructeurs (instancier une collection)
- Redéfinition des méthodes de base de la classe Object
- Ajouter un nouvel élément
- Contrôler (éventuellement) l'unicité
- Modifier un élément quelconque
- Supprimer un élément quelconque
- Calculer le cardinal de la collection
- Obtenir ses éléments
- Parcourir ses éléments
- Appliquer un algorithme sur tous les éléments

Opérations de base sur une collection

boolean add(E e)	Ajouter un élément à la collection (optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle)
void clear()	Supprimer tous les éléments de la collection (optionnelle)
boolean contains(Object o)	Retourner un booléen qui précise si l'élément est présent dans la collection
boolean containsAll(Collection<?> c)	Retourner un booléen qui précise si tous les éléments fournis en paramètres sont présents dans la collection
boolean equals(Object o)	Vérifier l'égalité avec la collection fournie en paramètre
int hashCode()	Retourner la valeur de hachage de la collection
boolean isEmpty()	Retourner un booléen qui précise si la collection est vide
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments de la collection
boolean remove(Object o)	Supprimer un élément de la collection s'il est présent (optionnelle)
boolean removeAll(Collection<?> c)	Supprimer tous les éléments fournis en paramètres de la collection s'ils sont présents (optionnelle)
boolean retainAll(Collection<?> c)	Ne laisser dans la collection que les éléments fournis en paramètres : les autres éléments sont supprimés (optionnelle). Elle renvoie un booléen qui précise si le contenu de la collection a été modifié
int size()	Retourner le nombre d'éléments contenus dans la collection
Object[] toArray()	Retourner un tableau contenant tous les éléments de la collection
<T> T[] toArray(T[] a)	Retourner un tableau typé de tous les éléments de la collection

Les interfaces des collections génériques



Généricité et collections

SANS la généricité

- Collection d'objets de type Object
  - Tout objet de type Object(pouvant référencer un objet de type dérivé) peut être ajouté dans la collection
  - Aucun contrôle préalable

➡ **Contenu hétérogène**

AVEC la généricité

- Collection générique de type T
  - T à définir
  - Seuls les objets de type T peuvent être ajoutés dans la collection
  - Contrôle préalable automatique

➡ **Contenu homogène**

Généricité et collections

❑ SANS typage générique

- Aucune précision du type des éléments contenus dans la collection

```
LinkedList notes =newLinkedList();
ArrayList personnes =newArrayList();
```

- On peut ajouter aux collections n'importe quel élément qui hérite de Object

❑ AVEC typage générique

- Définition du type précis des éléments contenus dans la collection :

```
LinkedList<Float>notes = new LinkedList<Float>();
ArrayList<Personne>personnes = new ArrayList<Personne>();
TreeSet<String>s =new TreeSet<String>();
```

- Unicité du type des éléments contenus

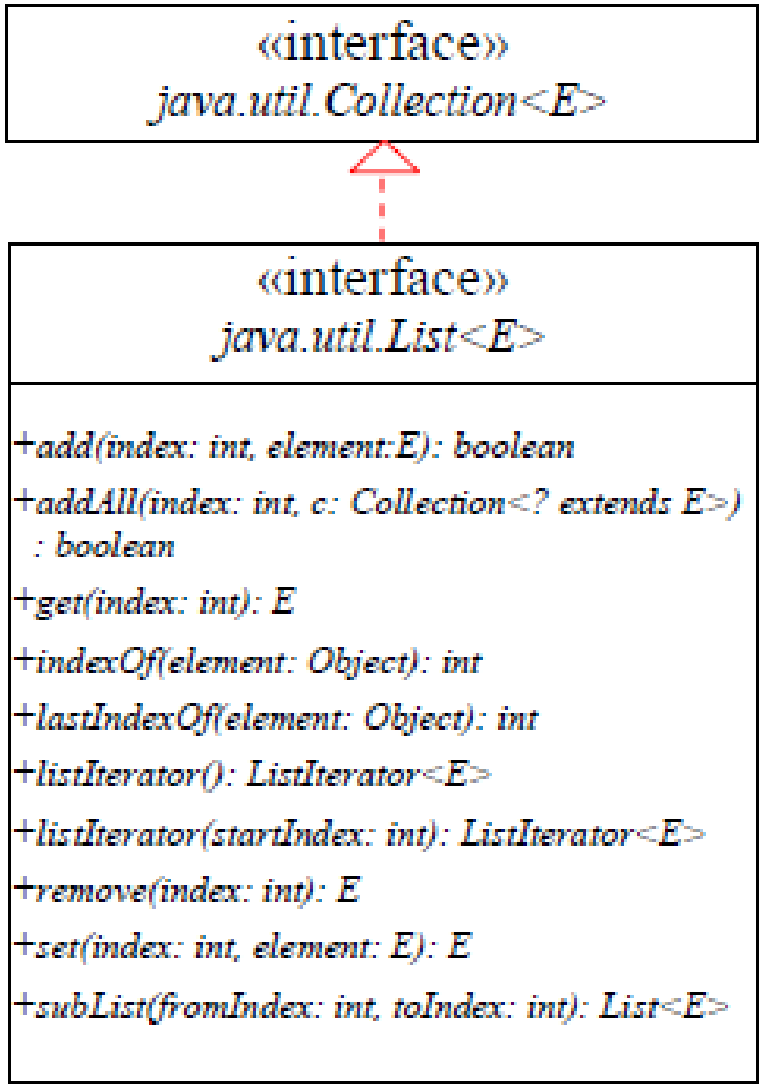
Intérêts de la généricité

- **Sans généricité :**
  - En pratique, l'usage des collections hétérogènes est très peu employée
  - *Cast* obligatoire pour l'accès à un élément
  - Risque d'erreur de conversion de type (détectée à l'exécution…)

- **Avec généricité :**
  - Vérification des types à la compilation
  - Moins de contrôle à l'exécution
  - Transtypage inutile (transtypage implicite, héritage)

Les collections de type « **List** »

L'interface List



Ajoute un nouvel élément à l'index spécifié.
Ajoute tous les éléments de c à cette liste à l'index spécifié.
Retourne l'élément dans cette liste à l'index spécifié.
Retourne l'index du premier élément correspondant.
Retourne l'index du dernier élément correspondant.
Retourne la liste iterator pour les éléments dans cette liste.
Retourne l'itérateur pour les éléments de startIndex.
Supprime l'élément à l'index spécifié.
Définit l'élément à l'index spécifié.
Retourne une sous-liste de fromIndex à toIndex.

Principaux services

- Taille courante d'une liste (size)
- Accesseurs de consultation (get, getFirst, …)
- Mutateurs (set)
- Accesseurs de position (indexOf)
- Ajout d'un élément (add)
- Suppression/retrait d'un élément (remove)
- Vider une liste (clear)
- Contrôle d'appartenance (contains)
- Transférer dans un tableau (toArray)

Les vecteurs dynamiques : « **ArrayList** »

- Tableaux de taille dynamique
- Implémentation la plus simple de l'interface List

java.util

Class ArrayList<E>

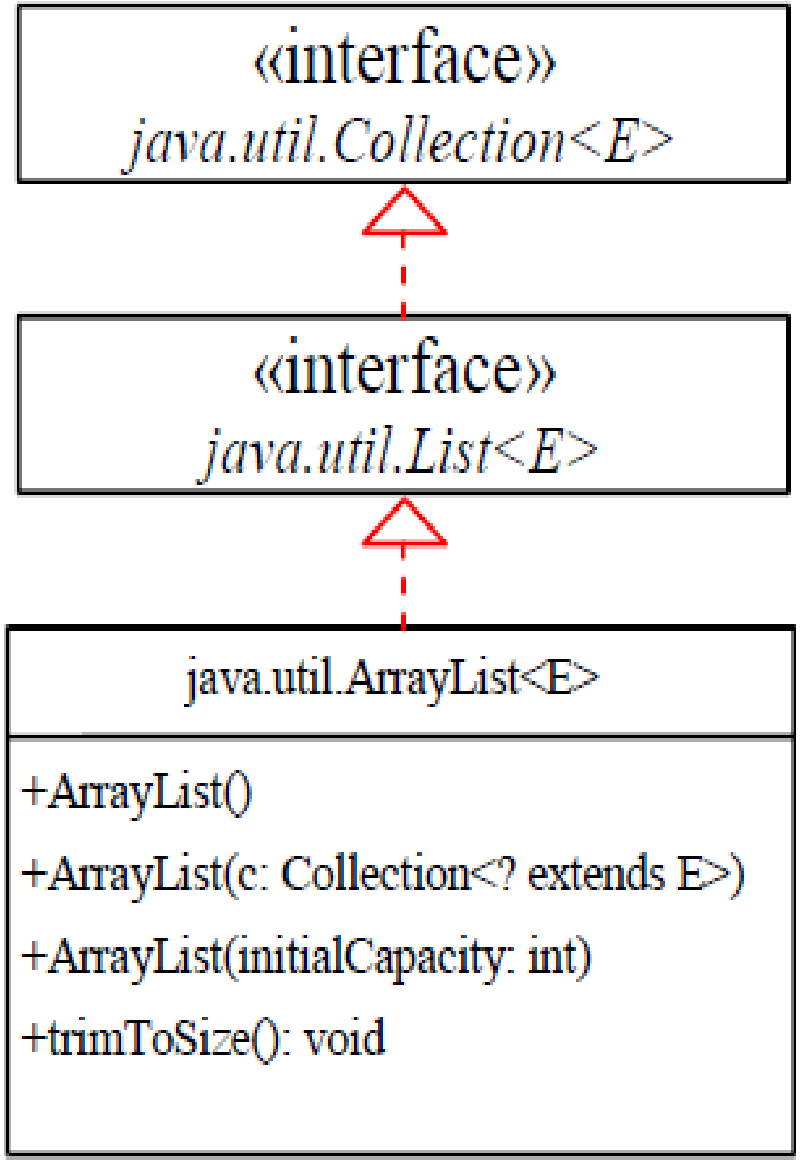
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

- Elle (ArrayList) présente plusieurs caractéristiques :
  - Utilise un tableau pour stocker ses éléments (le premier élément de la collection possède l'index 0)
  - L'accès à un élément se fait grâce à son index
  - Implémente toutes les méthodes de l'interface List
  - Autorise l'ajout d'élément null

Les vecteurs dynamiques : « **ArrayList** »



Crée une liste vide avec la capacité initiale par défaut.
Crée une liste de tableau à partir d'une collection existante.
Crée une liste vide avec la capacité initiale spécifiée.
Garnitures la capacité de cette instance ArrayList soient la taille actuelle de la liste.

Les vecteurs dynamiques : « **ArrayList** »

Exemple:

```
ArrayList<Integer> v= new ArrayList<Integer> ();

// On ajoute 10 objets de type Integer
for(inti=0; i<10; i++)
        v.add(new Integer(i));

// Suppression des éléments de position donnée
v.remove(3);
v.remove(5);

// Ajout d'éléments à une position donnée
v.add(2, new Integer(100));

// Modification élément de rang 2
v.set(2, new Integer(1000));
```

Les listes chaînées: « LinkedList »

java.util

Class LinkedList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.AbstractSequentialList<E>
                java.util.LinkedList<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

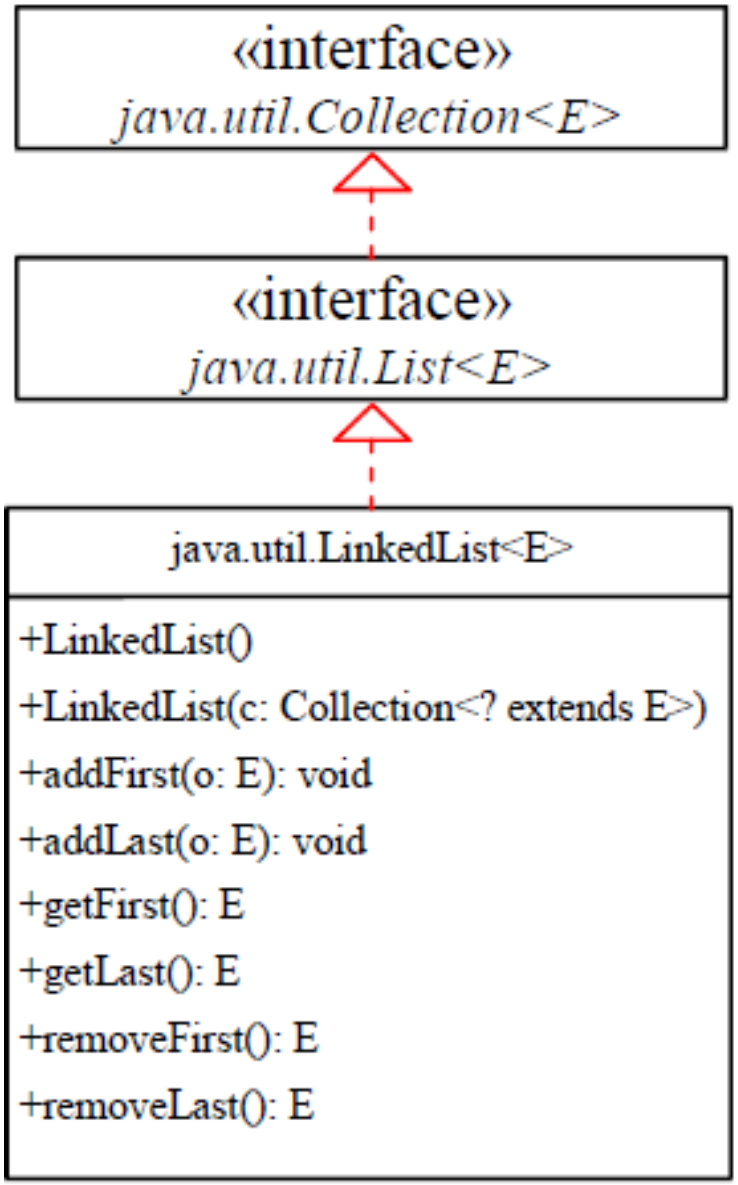
`Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>`

- Agrégat ordonné d'objets quelconques
- Listes doublement chaînées : chaînage avant et arrière
- Accès direct à la tête et à la queue (`get/add/removeFirst()`, `get/add/removeLast()`)

Les listes chaînées: « LinkedList »

- Implémente toutes les méthodes de l'interface `List`
- Implémente l'interface `Deque` depuis Java 6
- Elle n'a pas besoin d'être redimensionnée quelque soit le nombre d'éléments qu'elle contient
- Permet l'ajout d'élément `null`

Les listes chaînées: « LinkedList »



Utiliser souvent pour implanté des files d'attentes (Queue)

Crée une liste chaînée vide par défaut.
Crée une liste chaînée à partir d'une collection existante.
Ajoute l'objet à la tête de cette liste.
Ajoute l'objet à la queue de la liste.
Retourne le premier élément de cette liste.
Retourne le dernier élément de cette liste.
Retourne et supprime le premier élément de cette liste.
Retourne et supprime le dernier élément de cette liste.

Les listes chaînées: « LinkedList »

Exemple

```
LinkedList<Float> notes= newLinkedList<Float>();

notes.add(12.5f);
notes.add(8.f);
notes.add(10.0f);
notes.add(14.f);

System.out.println("Liste de notes = "+ notes);

notes.addFirst(1.0f);
notes.addLast(20.0f);
notes.add(8.5f);

System.out.println("Liste de notes = "+ notes);

doublef= notes.removeLast();

System.out.println("Liste de notes = "+ notes);
```

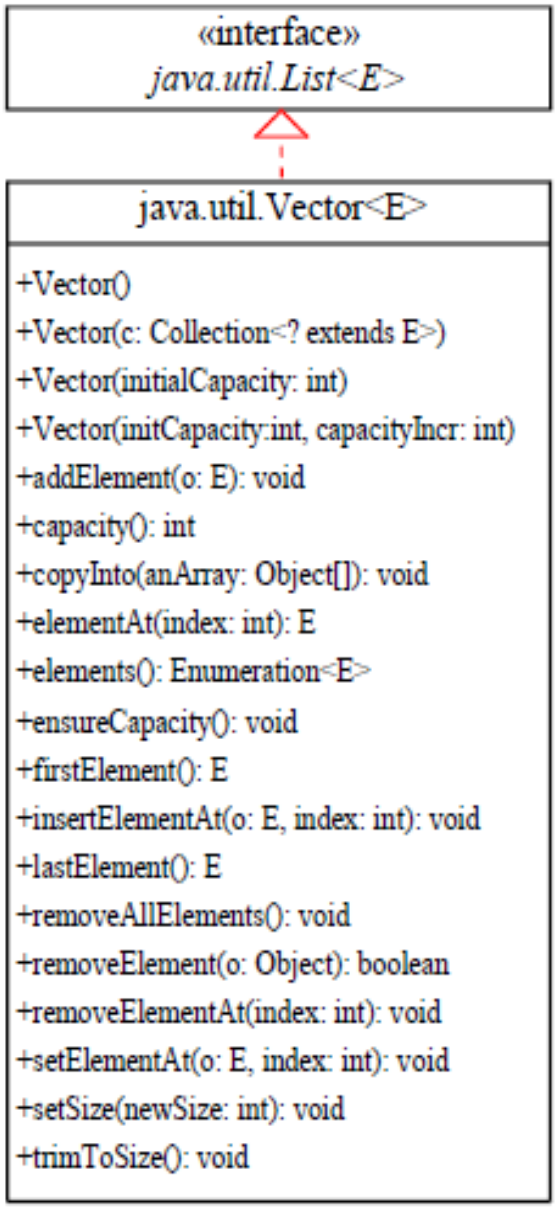
Les listes chaînées: « LinkedList »

Choix de l'objet de type liste

- `ArrayList`: Tableau dynamique
  - Ajout à la fin en 0(1), ajout au début en O(n), accès indexé en O(1)
- `LinkedList`: Liste doublement chaînée
  - Ajout à la fin en 0(1), ajout au début en O(1), accès indexé en O(n)

	get	add	contains	next	Remove(0)
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)

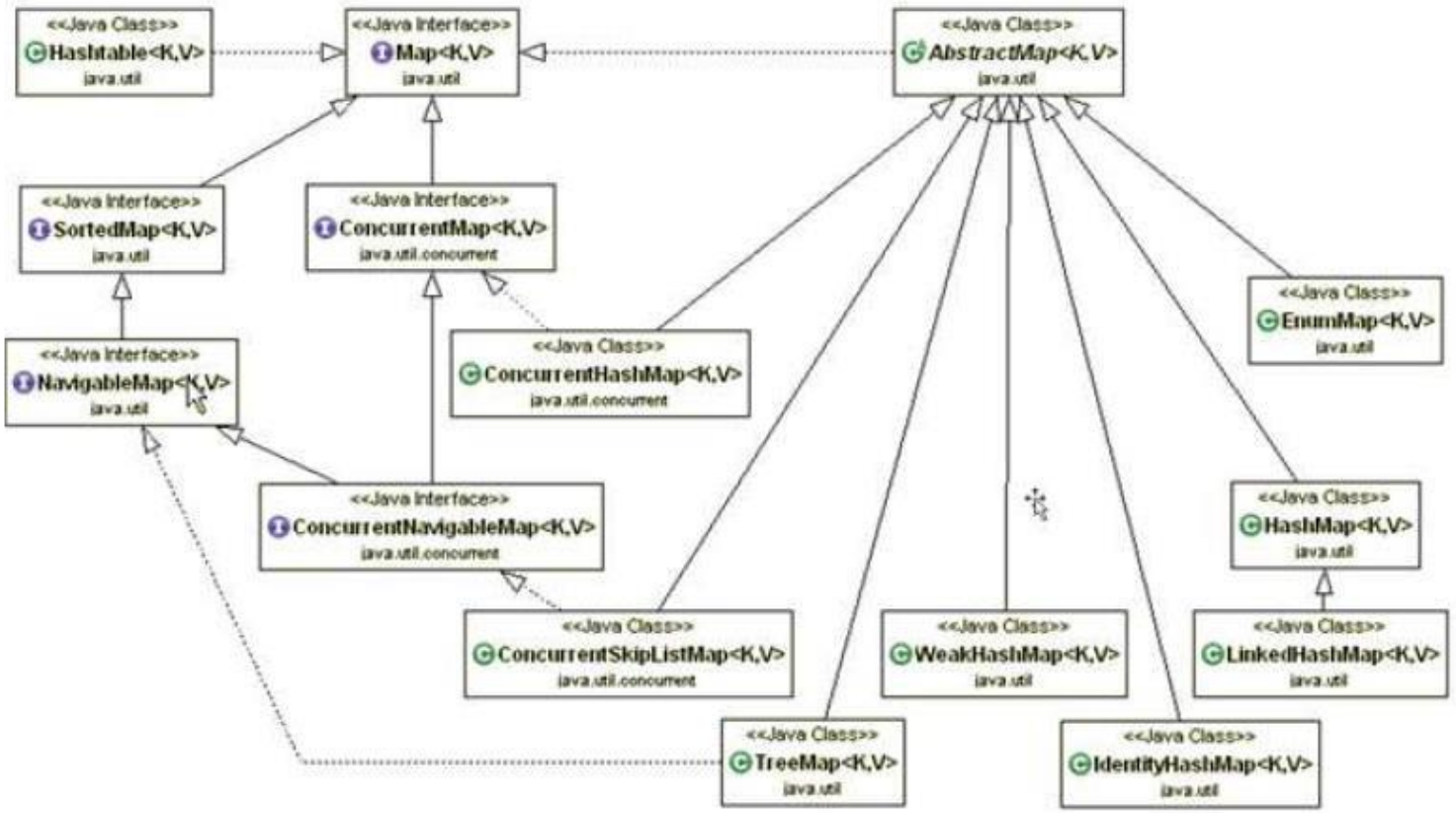
La classe « Vector »



La classe Vector est la même que ArrayList

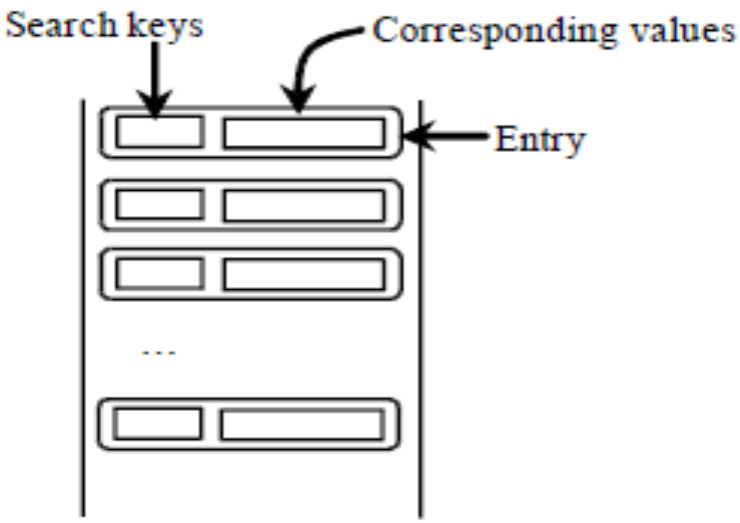
Crée un vecteur vide par défaut avec une capacité initiale de 10.
Crée un vecteur à partir d'une collection existante.
Crée un vecteur avec la capacité initiale spécifiée.
Crée un vecteur avec la capacité initiale spécifiée et l'incrément.
Ajoute l'élément à l'extrémité de ce vecteur.
Retourne la capacité actuelle de ce vecteur.
Copie les éléments de ce vecteur dans le tableau.
Renvoie l'objet à l'index spécifié.
Retourne une énumération de ce vecteur.
Augmente la capacité de ce vecteur.
Retourne le premier élément dans ce vecteur.
Inserts o à ce vecteur à l'index spécifié.
Retourne le dernier élément dans ce vecteur.
Supprime tous les éléments de ce vecteur.
Supprime le premier élément correspondant dans ce vecteur.
Supprime l'élément à l'index spécifié.
Définit un nouvel élément à l'index spécifié.
Définit une nouvelle taille dans ce vecteur.
Versions de la capacité de ce vecteur à sa taille.

Les collections de type « Map »



L’interface «Map»

- L'interface **Map** permet de mapper clés aux éléments.
  - Les clés sont comme des indices.
  - Dans List, les indices sont des nombres entiers.
  - Dans Map, les clés peuvent être des objets.



L'interface «Map»

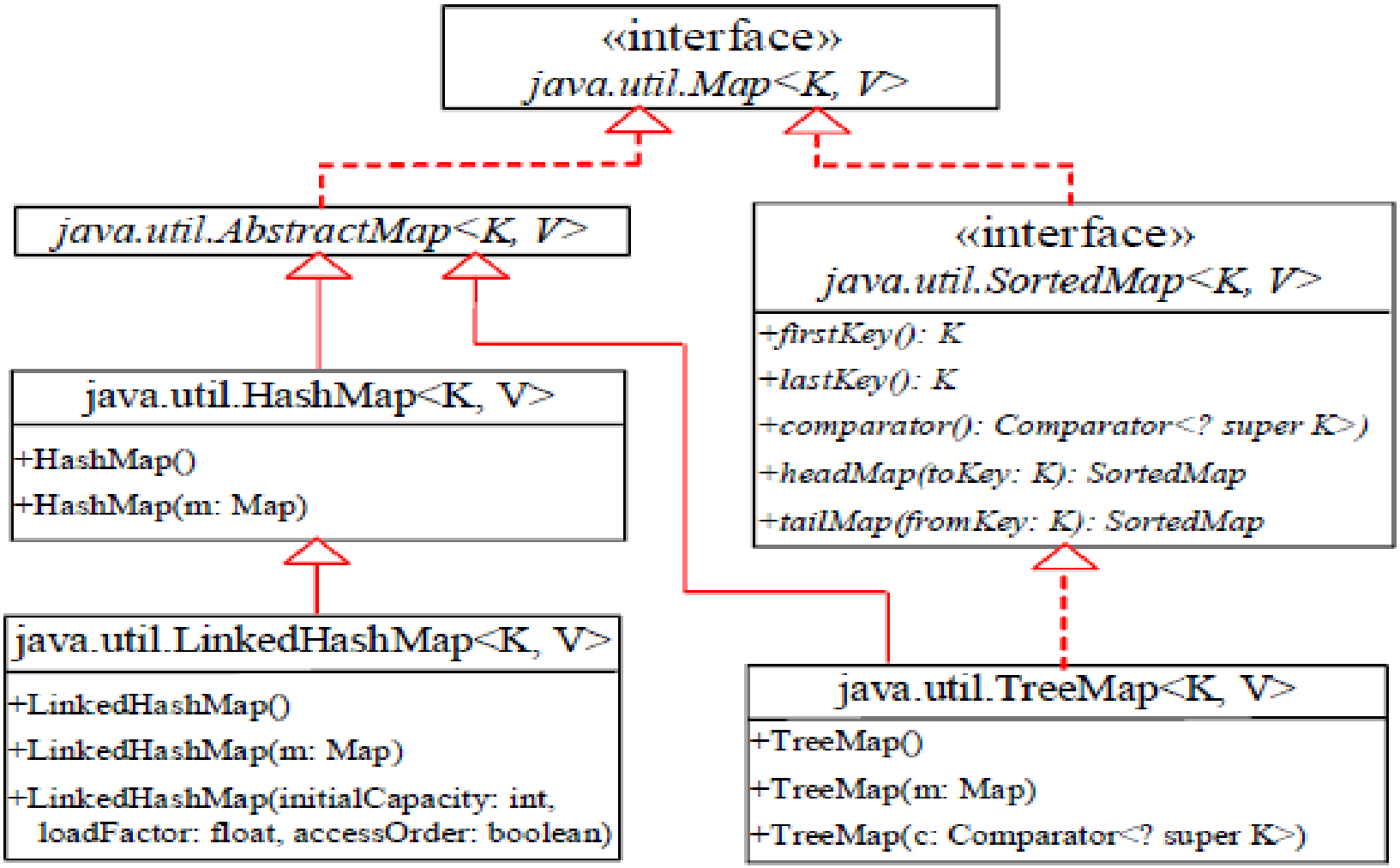
- Les collections de type « Map » ou *tableau associatif* en Java, sont définies à partir de la racine Interface *Map <K, V>* (et non *Collection <E>*)
- La raison est qu'une telle collection est un ensemble de **paires** d'objets, chaque paire associant un objet de l'ensemble de départ *K* à un objet de l'ensemble d'arrivée *V* ; on parle de **paires** (clé, valeur)
- Application: chaque fois qu'il faut retrouver une valeur en fonction d'une clé,
- Exemple:
  - dans un dictionnaire: mot -> définition du mot;
  - dans un annuaire: nom de personne -> adresse et n° de tél;
  - localisation: avion -> aéroport,
  - etc.

L'interface «Map»

java.util.Map<K, V>	
+clear(): void	Supprime tous les mappages de cette map.
+containsKey(key: Object): boolean	Retourne true si cette map contient une cartographie de la clé spécifiée.
+containsValue(value: Object): boolean	Renvoie true si cette map maps une ou plusieurs touches à la valeur spécifiée.
+entrySet(): Set	Retourne un ensemble constitué par les entrées de cette map.
+get(key: Object): V	Retourne la valeur de la clé spécifiée dans cette map.
+isEmpty(): boolean	Retourne true si cette map ne contient pas de mappages.
+keySet(): Set<K>	Retourne un ensemble constitué par les touches dans cette map.
+put(key: K, value: V): V	Met un mappage dans cette map.
+putAll(m: Map): void	Ajoute tous les mappages de M à cette map.
+remove(key: Object): V	Supprime le mappage pour la clé spécifiée.
+size(): int	Retourne le nombre de correspondances dans cette map.
+values(): Collection<V>	Retourne une collection comprenant les valeurs de cette map.

L'interface «Map»

Les classes concrète de « Map »



HashMap etTreeMap

- 2 implémentations concrètes de l'interface Map: HashMap et TreeMap:
  - Classe HashMap : efficace pour localiser une valeur, l'insertion d'une mapping, et la suppression d'une mapping.
  - Classe TreeMap : implemente SortedMap, elle est efficace pour traverser les clés dans un ordre trié.

La classe HashMap

java.util

Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:

LinkedHashMap, PrinterStateReasons

La classe HashMap

- **Déclaration / construction**
  - HashMap <K, V> m = new HashMap <K, V> (); // map vide
  - Ex: HashMap <String,Integer> m = new HashMap <String,Integer>();

- **put -** ajout d'une paire (ou mise à jour de la valeur si la clé existe)
  - m.**put**(cle, val); // où cle objet de K, val objet de V
  - Cas particulier: *val* peut être de type primitif int, float, char,…(≠ objet)

  exemple: m.**put**("occident", **1**);

- **get –** accès à la valeur associée à une clé.
- Exemple:

```
String cle = "occident";
Integer val = m.get(cle);
if (val == null) System.out.println("cette clé n'existe pas ! ")
```

La classe HashMap

- **remove** - suppression d'une paire en fonction de la clé.
- Exemple:

```
m.remove(cle); // supprime la paire ("occident" , val )
```

- **D'autres méthodes:**
  - Taille courant (**size**)
  - Accesseurs de consultation (**get**, **keySet**, **entrySet**, …)
  - Vider un dictionnaire (**clear**)
  - Contrôle d'appartenance (**containsKey**, **containsValue**)
  - etc.

La classe HashMap

Exemple

```
importjava.util.*;
Public class TestDictionnaire{
    Public static void main(String[] args) {
       // Construire un dictionnaire de test
    HashMap<String, String> annuaire= new HashMap<String,String>();
       annuaire.put("Durand", "04.93.77.18.00");
       annuaire.put("Dupuy", "04.93.66.38.76");
       annuaire.put("Leroy", "04.92.94.20.00");

       System.out.println("Annuaire= "+ annuaire);
       annuaire.put("toto", "111111");
       annuaire.put("toto", "2222222");
       System.out.println("Annuaire= "+ annuaire);
    }
}
Annuaire= {Dupuy=04.93.66.38.76, Durand=04.93.77.18.00,
Leroy=04.92.94.20.00}
Annuaire= {toto=2222222, Dupuy=04.93.66.38.76,
Durand=04.93.77.18.00, Leroy=04.92.94.20.00}
```

La classe HashMap

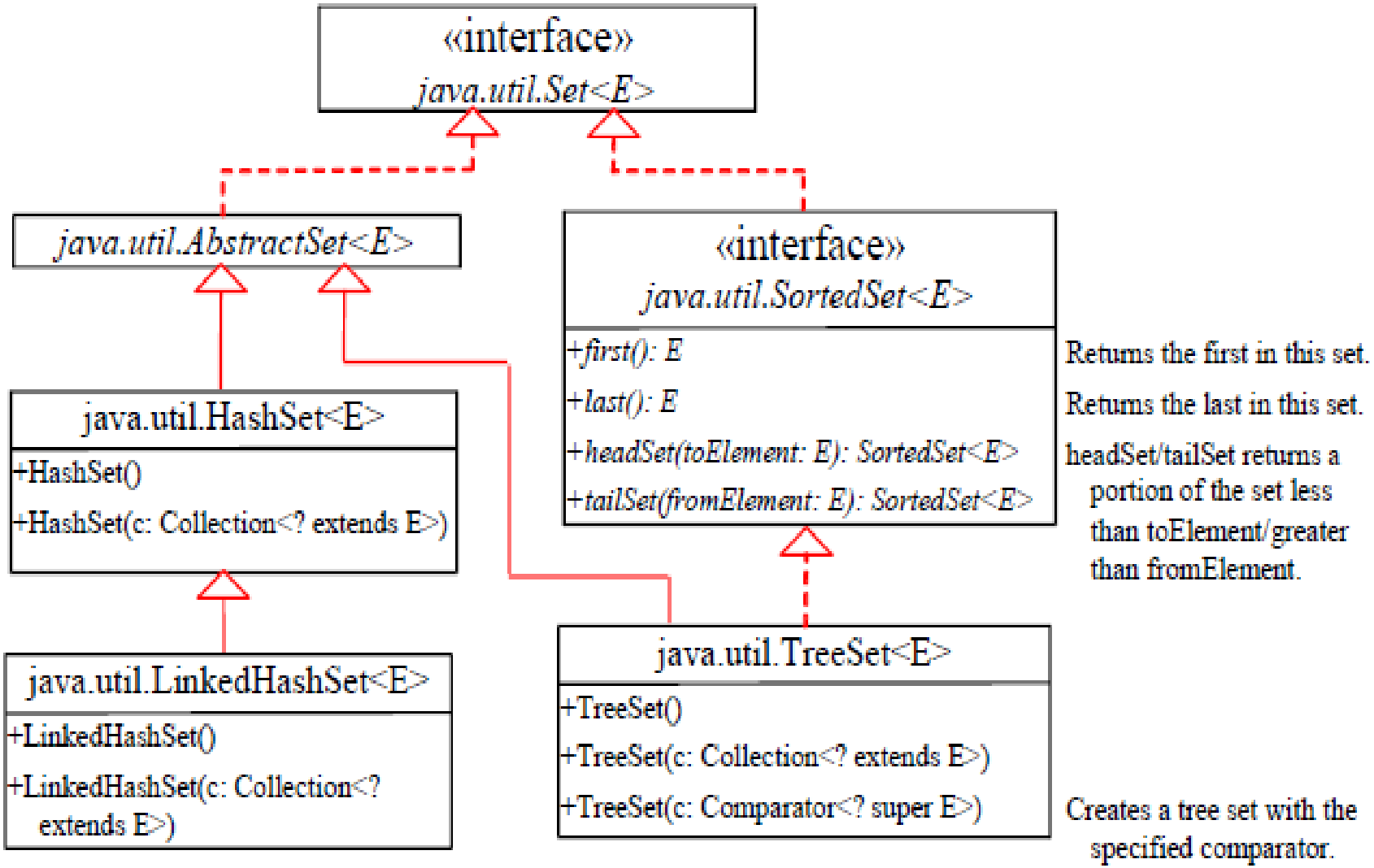
Exercice:

Ecrire un programme qui compte les occurrences de mots dans un texte et affiche les mots et leurs occurrences dans l'ordre croissant des mots.

-Utilise une table de hachage (HashTable) map pour stocker <mot, nombre>

- Pour chaque mot, vérifiez si elle est déjà une clé dans la carte. Sinon, ajoutez la clé et la valeur 1 à la carte.
- Sinon, augmentez la valeur du mot (clé) par 1 dans le map.
- Pour trier le map, le convertir en une TreeMap

Les collections de type Set



Les collections de type Set

L'interface Set

- l'interface Set étend l'interface Collection pour représenter un ensemble d'objet.
  - Une instance Set ne doit pas contenir d'éléments dupliqués.
  - Les classes concrètes qui implémentent Set doivent veiller à ce qu'aucun des éléments dupliqués peuvent être ajoutés à l'ensemble.
  - Pas de deux éléments E1 et E2 peuvent être dans l'ensemble (Set) tels que e1.equals (e2) est vrai.

Les collections de type Set

Les ensembles

- La classe HashSet:
  - Implémentation simple de l'interface Set qui utilise HashMap
  - Aucune garantie sur l'ordre de parcours des éléments lors de l'itération
  - Ne permet pas d'ajouter des doublons mais permet l'ajout d'un élément null

- Elle utilise en interne une HashMap dont la clé est l'élément et dont la valeur est une instance d'Object identique pour tous les éléments

Les collections de type Set

Les ensembles (suite)

- La classe TreeSet:
  - Stocke les éléments de manière ordonnée en les comparant entre eux
  - Permet d'insérer des éléments dans n'importe quel ordre et de les restituer dans un ordre précis lors du parcours
  - Ne peut pas contenir des doublons
  - Implémente l'interface NavigableSet depuis Java 6

- La classe LinkedHashSet: Comme HashSet mais éléments accessibles en ordre d'insertion (L'ordre des entrées préservée),

Les collections de type Set

Exemple:

Résultat:

```
import java.util.*;
public class SetExample {
    public static void main(String args[]) {
        Set<String> set = new HashSet<String>(); // Une table de Hachage
        set.add("Bernadine");
        set.add("Elizabeth");
        set.add("Gene");
        set.add("Elizabeth");
        set.add("Clara");
        System.out.println(set);  // [Gene, Clara, Bernadine, Elizabeth]
        Set<String> setTrie = new TreeSet<String>(set); // Un Set trié
        System.out.println(setTrie); //[Bernadine, Clara, Elizabeth, Gene]

    }
}
```

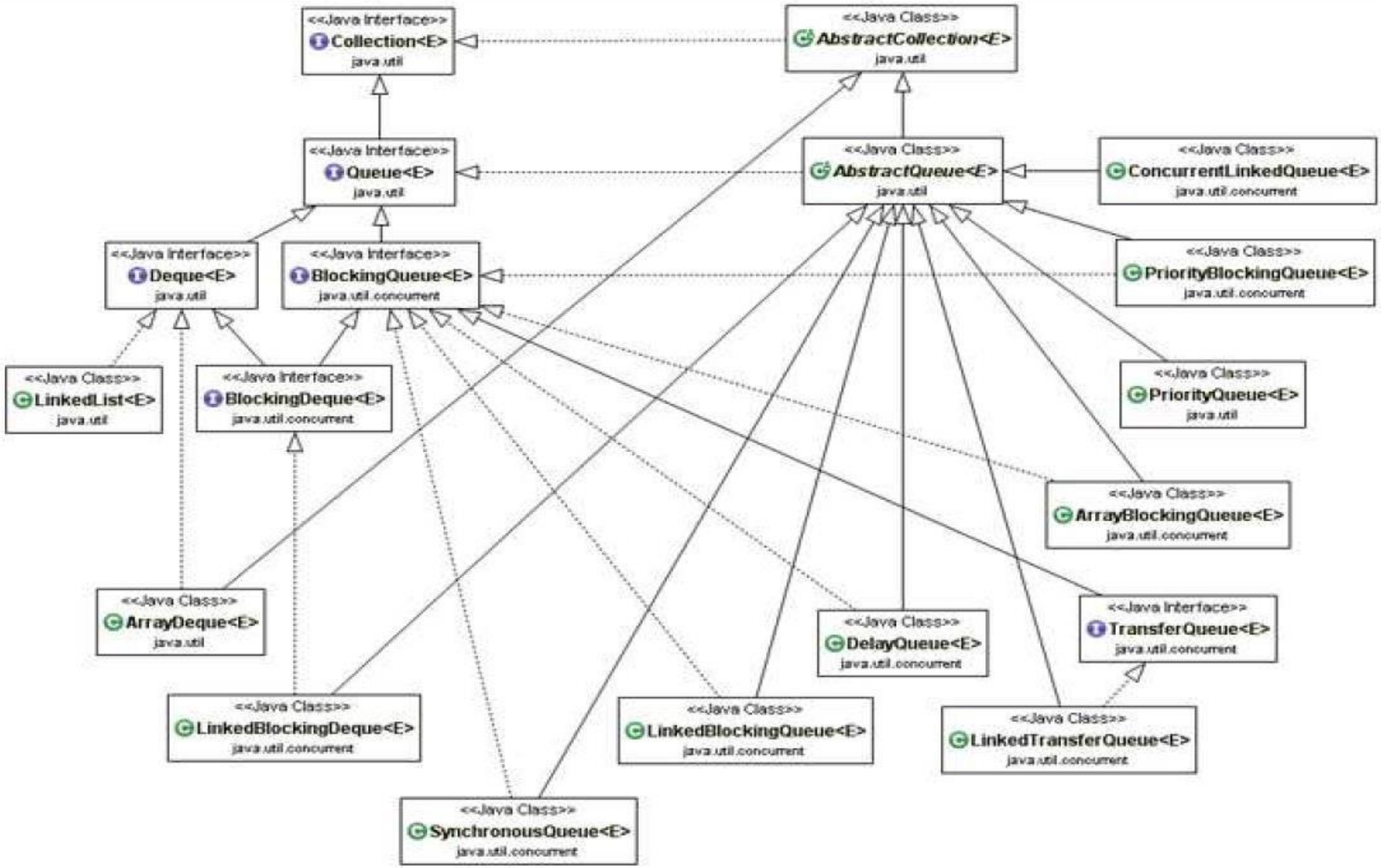
Les collections de type Set

Exemple:

```
System.out.println("Ensemble trié de String ");
TreeSet<String> s= new TreeSet<String>();
s.add("Marie");
s.add("Jean");
s.add("Paul");
System.out.println(s);
```

```
Ensemble trié de String
[Jean, Marie, Paul]
```

Les collections de type Queue



Les collections de type Queue

Parcourir une collection

- Nouvelle syntaxe **for** ... **each**:

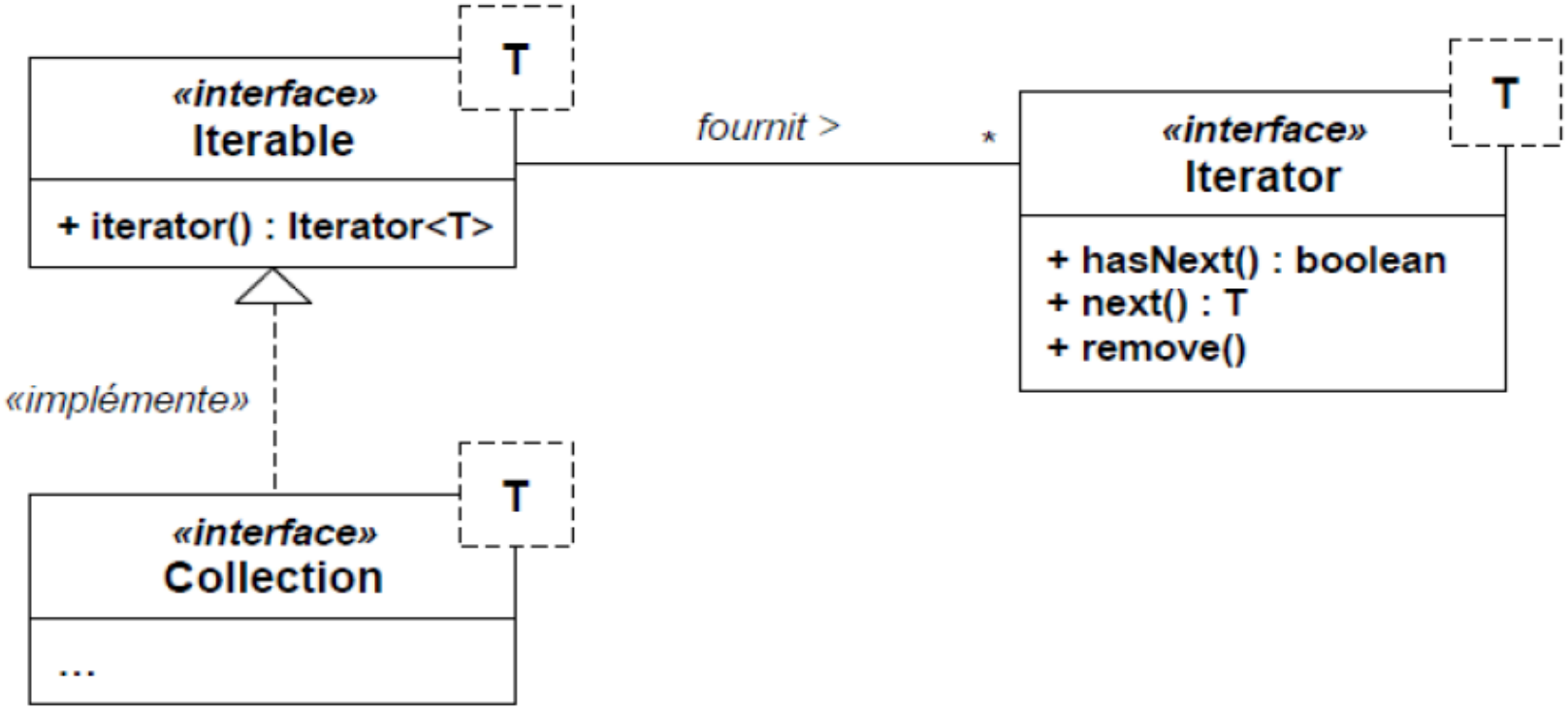
```
for (type variable : collection){
      instructions
}
```

```
Public static float moyenne (LinkedList<Float>notes){
      float somme=0.0f;
      for(float val :notes)
            somme +=val;
      return somme/notes.size();
}
```

Les collections de type Queue

Autres types de parcours

- Classe `Iterator` et interface `Iterator<T>`
  - Permet de parcourir les collections (équivalent boucle for)
  - Mais est utile si des éléments successifs des collections doivent être manipulés (comparés, remplacés, …)



Les collections de type Queue

Utilisation de l'itérateur

```
Publics tatic float moyenne (LinkedList<Float>notes){
        float somme=0.0f;
        Iterator<Float>i=notes.iterator();
        while(i.hasNext())
                somme +=i.next();
        return somme/notes.size();
}
```

- L'interface Iterator prévoit une fonction *remove* qui permet de supprimer le dernier élément renvoyé par *next* de la collection
- Attention à ne pas modifier la collection pendant que l'on utilise un itérateur !

Les collections de type Queue

Exemple

```
Public class TestCollection{
   Public static void main(String[] args) {
      ArrayList<Integer> v= newArrayList<Integer> ();
      System.out.println("En A : taille de v = "+v.size());
// On ajoute 10 objets de type Integer
      for(inti=0; i<10; i++)
          v.add(newInteger(i));
      System.out.println("En B : taille de v = "+v.size());
// Affichage du contenu par accès direct en utilisant
//for ... each
      System.out.println("En B : contenu de v = ");
      for(Integer e: v)
          System.out.print(e+" ");
      System.out.println();
// Suppression des éléments de position donnée
      v.remove(3);
      v.remove(5);
      v.remove(5);
      System.out.println("En C : contenu de v = "+v);
// Ajout d'éléments à une position donnée
      v.add(2, newInteger(100));
      v.add(2, newInteger(200));
      System.out.println("En D : contenu de v = "+v);

// Modification d'éléments de position donnée
      v.set(2, newInteger(1000)); // Modification élément de rang 2
      v.set(5, newInteger(2000)); // Modification élément de rang 5
      System.out.println("En E : contenu de v = "+v);
  }
}
```

Résultats

```
En A : taille de v = 0
En B : taille de v = 10
En B : contenu de v =
0 1 2 3 4 5 6 7 8 9
En C : contenu de v = [0, 1, 2, 4, 5, 8, 9]
En D : contenu de v = [0, 1, 200, 100, 2, 4, 5, 8, 9]
En E : contenu de v = [0, 1, 1000, 100, 2, 2000, 5, 8, 9]
```