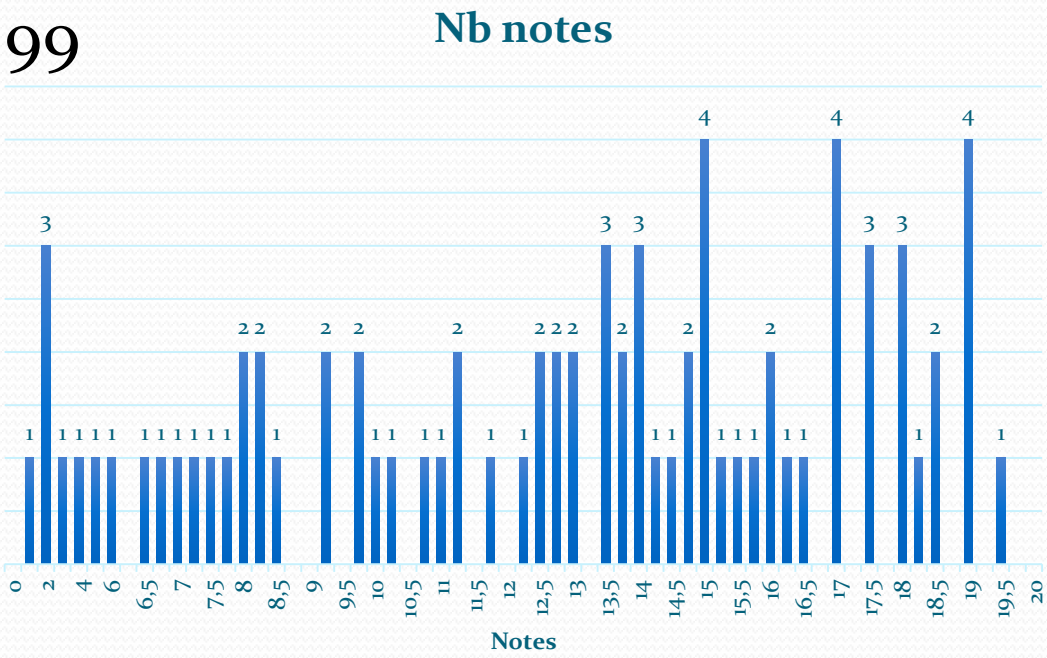
Collections et Programmation générique

Christelle CAILLOUET
(christelle.caillouet@unice.fr)

Un mot sur l'IE du 16 mars

- Moyenne générale = 11,99
- Note min = 0,5
- Note max = 19,5
- Note médiane = 13



- Notes dans intracursus cet après-midi
- Copies consultables le mercredi

Les collections

Le package *java.util*

- Contient de nombreuses classes « utilitaires »
  - Autour des tableaux, des ensembles, …
- Contient également la plupart des classes, classes abstraites, et interfaces sur les structures de données permettant de stocker des éléments

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

Concepts généraux des collections

- 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 sont des conteneurs qui regroupent les objets en une seule entité

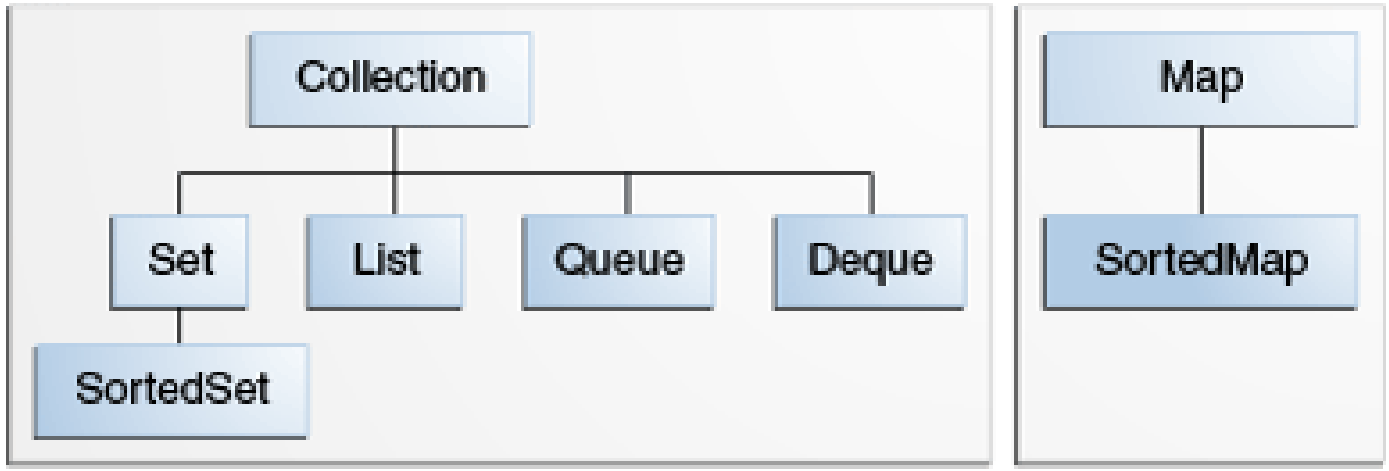
API des collections

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

- Quatre types de structure de données :
  - **List** : éléments ordonnés qui accepte les doublons
  - **Set** : éléments non ordonnés par défaut qui n'accepte pas les doublons
  - **Map** : association de paire clé/valeur
  - **Queue** et **Deque** : éléments stockés dans un certain ordre avant d'être extraits pour traitement

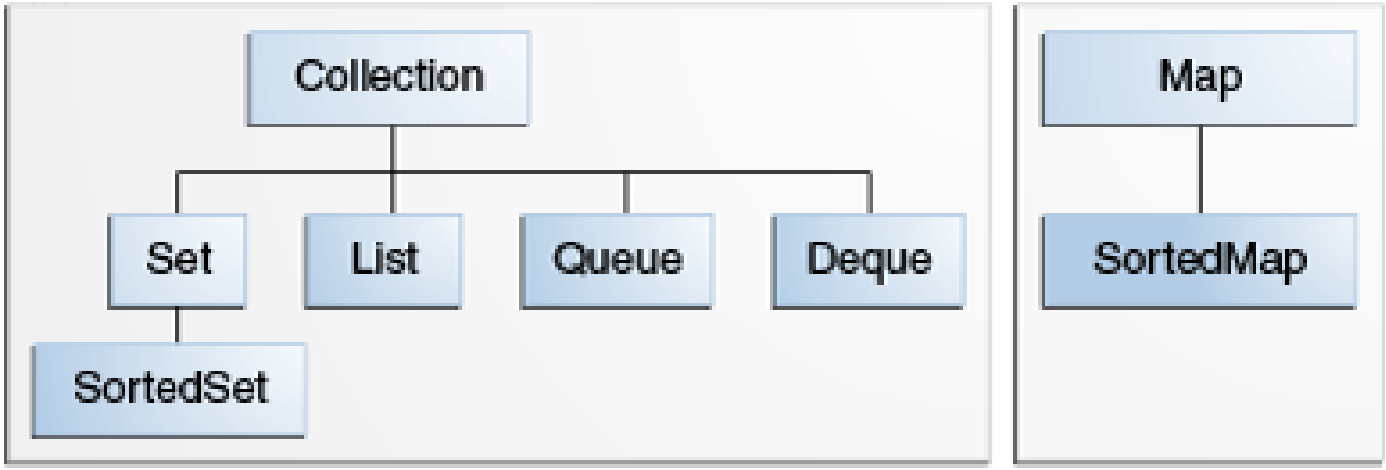
API des collections

- Deux grandes familles de collections chacune définie par une interface de base



Objets implémentant les collections

- Les fonctionnalités des collections sont définies dans les interfaces de l'API :
  - qui implémentent elles-mêmes l'interface `Collection`
  - qu'elles complètent par leurs fonctionnalités propres



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

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

https://www.jmdoudoux.fr/java/dej/chap-collections.htm

La programmation générique

Qu'est-ce que la généricité ?

- Dans une fonction « classique », les paramètres sont des valeurs

➔ **Dans un générique, les paramètres sont des types**

- Un générique est **un modèle** :
  - Instanciation = création d'un élément à partir d'un modèle

  ➔ **Instancier un générique = fixer le type de ses paramètres**

- Composants pouvant être génériques en Java (depuis Java 5.0)
  - Classes, interfaces, méthodes

Programmation générique

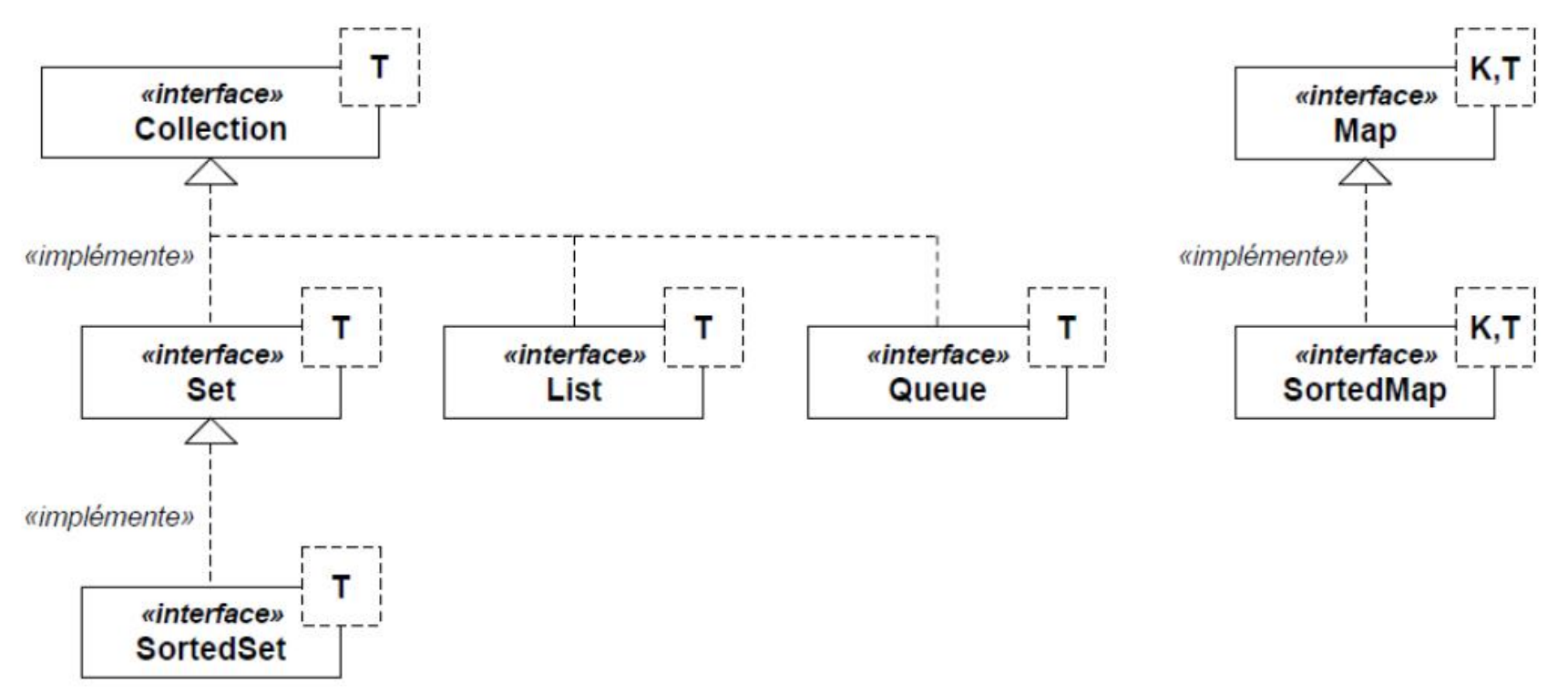
- Définition :
  - Code source unique utilisable avec des objets ou des variables de type quelconque
- Intérêt :
  - Indépendance du code vis-à-vis du type des données manipulées
- Exemple :
  - méthodes de tris applicable à des objets de type quelconque (`Point`, `Double`, `String`, ...)

Domaines d'application

- Modules paramétrés
- Classes génériques (🡐🡒 template en C++)
  - Classe appliquée à plusieurs types de données différents
- **Toutes les collections** sont passées en version générique avec Java 5
  - Ensembles : `Set, SortedSet, HashSet, TreeSet`
  - Listes : `List, ArrayList, Vector, LinkedList`
  - Files d'attente : `Queue, PriorityQueue`
  - Associations : `Map, SortedMap, HashMap, TreeMap`
- Ancienne version compatible mais génère des *warnings* dans Eclipse

Les collections génériques

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 = new LinkedList();
ArrayList personnes = new ArrayList();
```

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

➔ Plus employé depuis Java 5

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

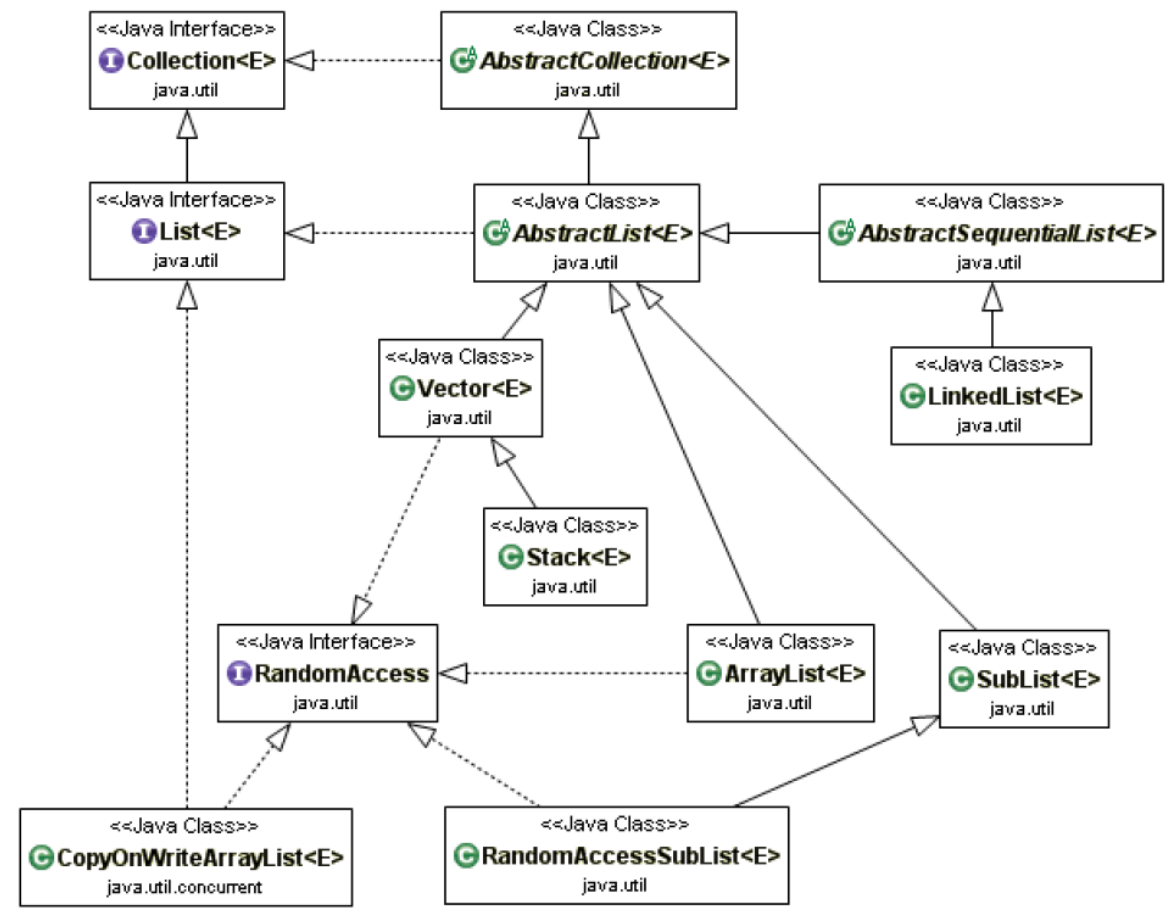
➔ Usage courant des collections

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`



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`

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

- Tableaux de **taille dynamique**
- Implémentation la + 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`

`ArrayList`

- Elle 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`

Exemple

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

// On ajoute 10 objets de type Integer
for (int i=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));

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

Les listes chaînées

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

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()`)

`LinkedList`

- Implémente toutes les méthodes de l'interface `List` (même optionnelles)
- 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`

Exemple

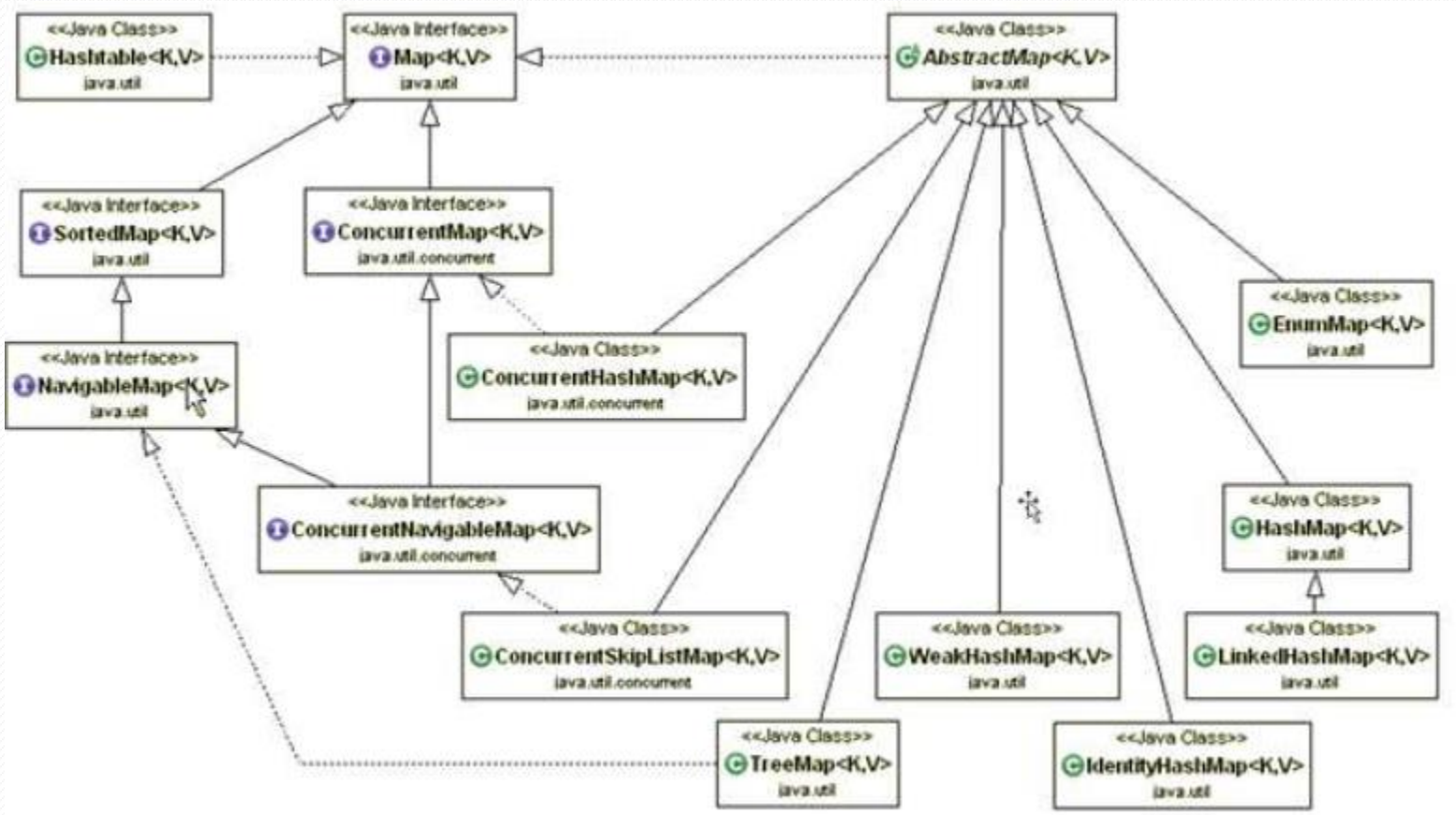
```
LinkedList<Float> notes = new LinkedList<Float>();

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);

double f = notes.removeLast();

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

Choix de l'objet de type liste

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

Les collections de type `Map`



Les dictionnaires (tables associatives)

- Ensemble non ordonné d'**associations**

➔ **Association = couple (clé, valeur)**

- Initialisation par insertion d'associations

- Accès par clé de chaque association

La classe `HashMap`

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

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`

Principaux services

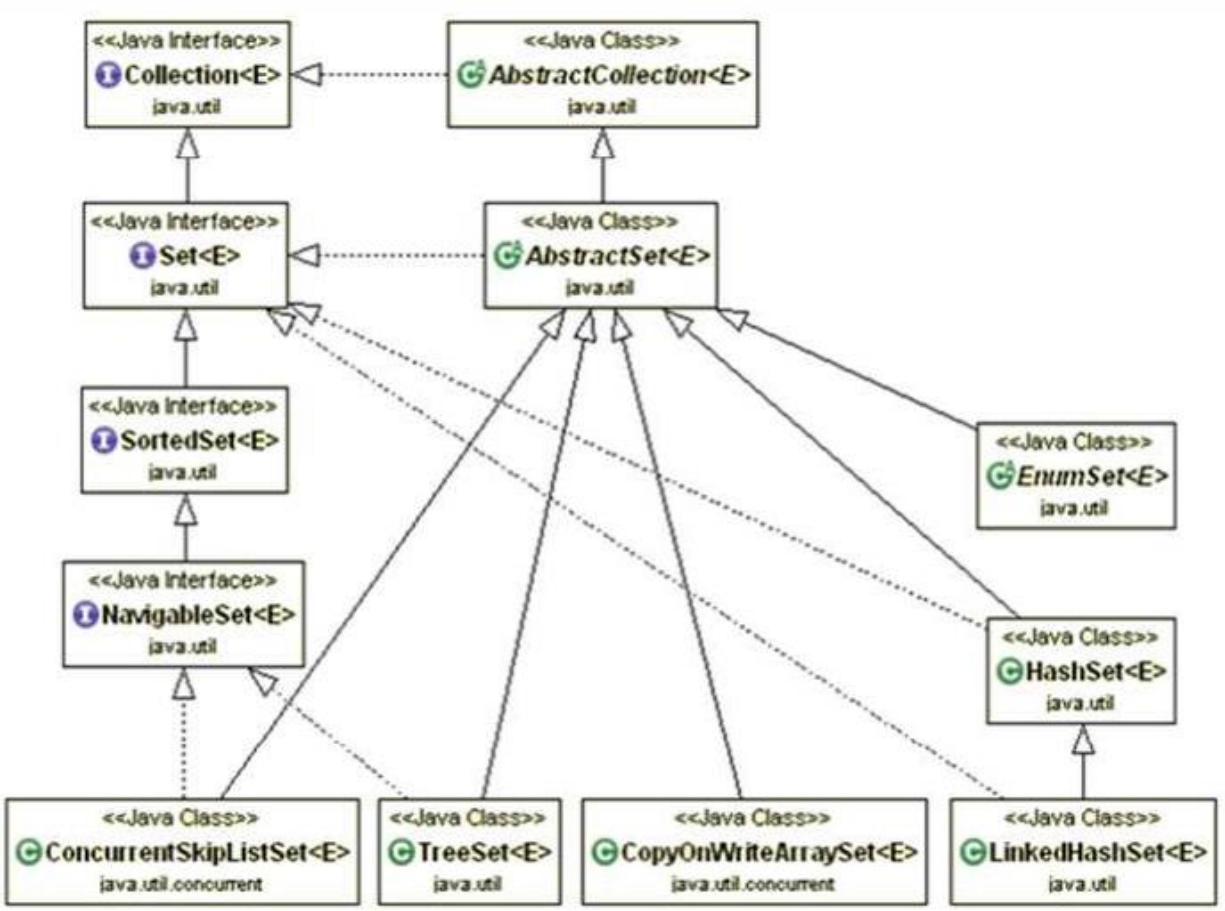
- Taille courant d'un dictionnaire (**size**)
- Accesseurs de consultation (**get**, **keySet**, **entrySet**, …)
- Mutateurs (**put**)
- Ajout d'un nouvel élément (**put**)
- Suppression/retrait d'un élément (**remove**)
- Vider un dictionnaire (**clear**)
- Contrôle d'appartenance (**containsKey**, **containsValue**)
- …

Exemple

```
import java.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}
```

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

https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

Les ensembles

- 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

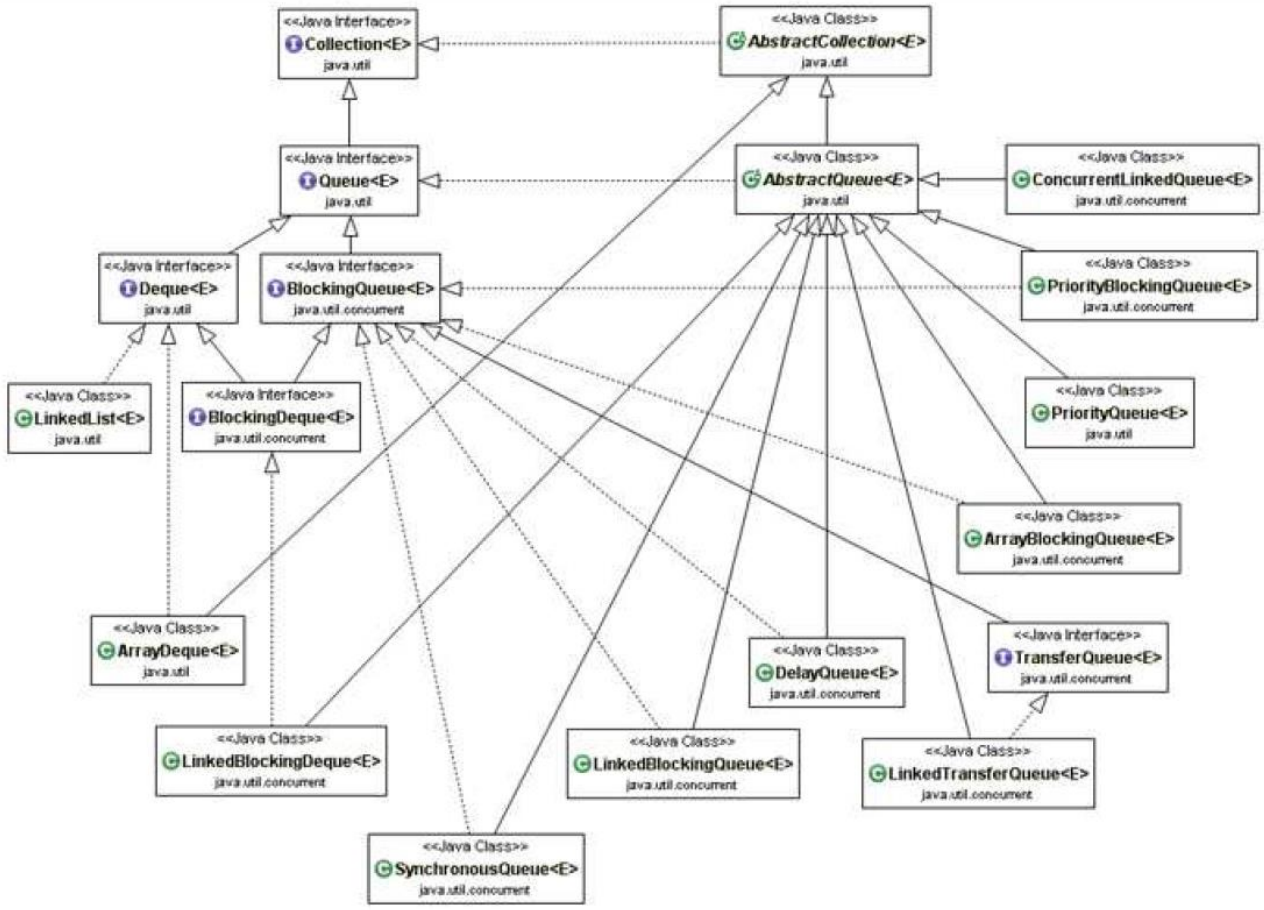
https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

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`



Parcourir une collection

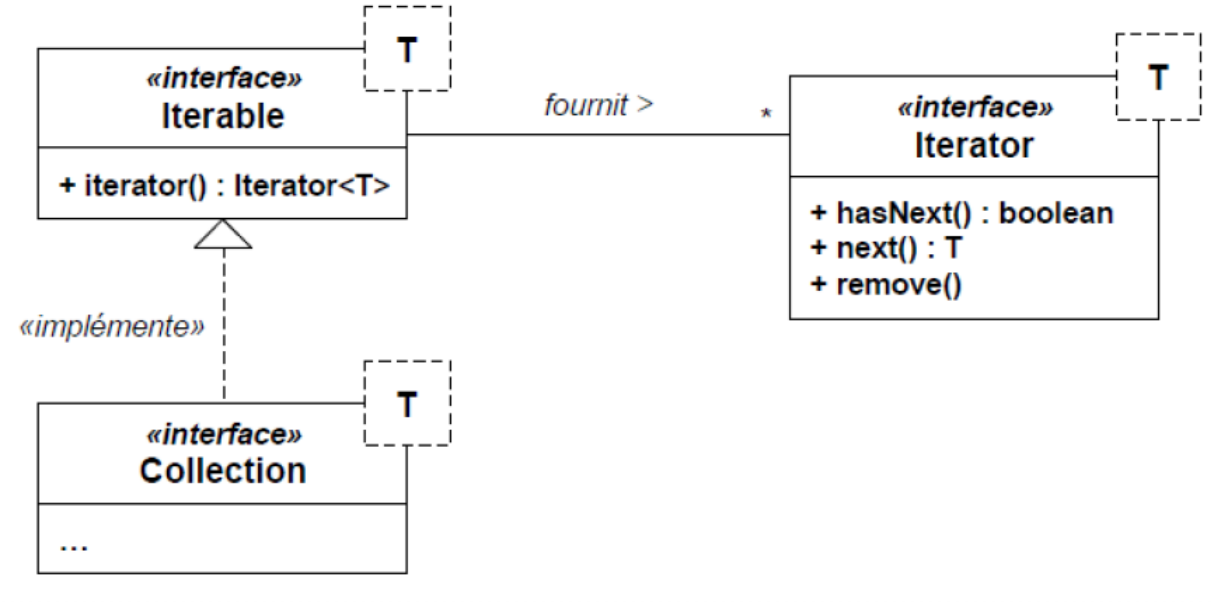
- Nouvelle syntaxe **`for ... each`** :

```
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();
}
```

Autre type de parcours

- Classe `Iterator` et interface `Iterator<T>`
  http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
  - Permet de parcourir les collections (équivalent boucle *for*)
  - Mais est utile si des éléments successifs des collections doivent être manipulés (comparés, remplacés, …)



Utilisation de l'itérateur

```
public static 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 !

Résultats :

```
public class TestCollection {
public static void main(String[] args) {
    ArrayList<Integer> v = new ArrayList<Integer> ();
    System.out.println("En A : taille de v = "+v.size());

    // On ajoute 10 objets de type Integer
    for (int i=0; i<10; i++) v.add(new Integer(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, new Integer(100));
    v.add(2,  new Integer(200));
    System.out.println("En D : contenu de v = "+v);

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

```
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]
```

Concepts abordés directement en TD

- Les méthodes (statiques) de manipulation des tableaux sont regroupés dans `java.util.Arrays`
  - Tri (`Arrays.sort`), remplissage (`Arrays.fill`), copie (`Arrays.copyOf`), …

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

```
import java.util.Arrays;

int[]    T1={0, 6, 2, -4, 3, 8, -11, 0, 1};
String[] T2={"bleu","rouge","blanc","vert","mauve","indigo"};

Arrays.sort(T1);
for (int i=0; i<T1.length; i++) System.out.print(T1[i] +" ");
Arrays.sort(T2);
for (int i=0; i<T2.length; i++) System.out.print(T2[i] +" ");

➔ Exécution
-11 -4 0 0 1 2 3 6 8
blanc bleu indigo mauve rouge vert
```

Concepts abordés directement en TD

- **Duplication** de collections (ou d'objets en général)
  - Copie de surface/en profondeur
  - Méthode `clone()` de la classe `Object`
  - Implémentation de l'interface `Cloneable`

- **Collections triées** : pour trier les éléments, il faut pouvoir les comparer
  - Implémentation de l'interface `Comparable`
  - Objets de type `Comparator`

Concepts non abordés

- **Depuis Java 8**, nouvelles fonctionnalités permettant d'associer un ***stream*** à une collection
  - Langage fonctionnel
  - Calcul parallèle