

NFP121

Programmation avancée

Les Collections en Java

Les Collections

- Définition, Intérêts, Objectifs
- Les Interfaces de collections dans le JDK
- Les implémentations de collections dans le JDK
 - Partielles : Classes abstraites
 - Complètes : implémentations prêtes à l'emploi
- Les classes utilitaires pour manipuler les collections dans le JDK

Collections : intérêts et objectifs ⁽¹⁾

- Une collection est une structure de données qui permet de stocker de manière organisée des données pour y accéder et les traiter.

Liste, Table, Arbre, Pile, File, Sac, Ensemble etc.

- Les traitements informatiques manipulent des volumes énormes de données.

- La structure de données de base « Le tableau » n'est plus suffisante
- Nécessité d'organiser ces données dans des structures (collections) adaptées aux traitements à réaliser.
- Choix de la collection selon besoins et contraintes (dualité espace / temps)
 - Performances à l'insertion, la lecture
 - Occupation mémoire

Collections : intérêts et objectifs (2)

- Avant Java 2 (1.4.2) :

- **Peu de collections fournies :**

- Vector, Stack, Hashtable. Properties*

- **Parcourues en utilisant l'interface Enumeration**

- Les Collections à partir de Java 2 :

- **Grande variété de collections :**

- Collection, List, Set, Map (Les Interfaces)*

- AbstractList, AbstractSet ... (Les Classes abstraites)*

- ArrayList, TreeSet, HashMap ... (Les Classes concrètes prêtes à l'emploi)*

Apports des Collections en Java 2

- Réduction des efforts de programmation : réutilisation
- Implémentations « out of the box » performantes
- Interopérabilité entre collection (*obtenue comment ?*)
- Facilité d'apprentissage: même logique et méthodes pour la plupart
- Le JDK met à disposition des utilitaires: les classes *Collections* et *Arrays*.

La Généricité ⁽¹⁾

- **Avant Java 5 dans les collections tous les éléments sont Object :**
 - Mélange d'éléments de types différents
 - Nécessité de « caster » les objets en sortie des collections (lisibilité --)
 - Risque d'erreur

```
// Liste d'objets
List l = new ArrayList();

// Mélanges de types différents
l.add("Une String");
l.add(42);

// Le cast est obligatoire pour la compilation
// Je pense récupérer un entier mais...
Integer valeur = (Integer) l.get(0);
```

La Généricité depuis Java 5 ⁽²⁾

- Le type devient un « paramètre » de la classe collection et est « défini » à sa construction.
- Le compilateur vérifie (analyse statique) pour toutes les opérations sur la collection que le type utilisé est conforme (*que signifie conforme ?*)
- Utilisation d'une stratégie d'inférence de type au moment de la compilation
- Les « paramètres de type » ne sont pas conservés dans le bytecode produit: Tout est objet

```
// Liste de Nombres
List<Number> l= new ArrayList<Number>();
// Impossible de Mélanger: que des Number
l.add(42);
l.add(55.2);
// Ne compile pas
Integer valeur = l.get(0);
// Je ne peux récupérer qu'un Number
Number maValeur = l.get(0);
```

La Généricité depuis Java 5: exemple (3)

```
/**
 * Collection à 1 seul élément
 *
 * @param <E> type de l'élément
 */
public class CollectionSingleton<E> {
    private E element;
    /**
     * affectation de l'élément
     * @param element
     */
    public void setElement(E element) {
        this.element = element;
    }
    /**
     * lecture de l'élément
     * @return
     */
    public E getElement() {
        return element;
    }
}
```

```
// Collection singleton de String
CollectionSingleton<String> colSinglString = new CollectionSingleton<>();
colSinglString.setElement("Str element");
String valeurStr = colSinglString.getElement();

// Collection singleton de Integer
CollectionSingleton<Integer> colSinglInt = new CollectionSingleton<>();
colSinglInt.setElement(42);
Integer valeurInt = colSinglInt.getElement();
```

La Généricité: notation en Java (4)

```
// <?> : le joker => n'importe quelle classe (contraintes d'utilisations)
// quels est type de l.get(0), quel est le type autorisé à la compilation ?
List<String> lstr = new ArrayList<String>();
List<?> l = lstr;
```

<? extends E> : ? Est une sous-classe de E

<? super E> : ? Est une super classe de E, E hérite de ?

<? extends Comparable<E>> : ? Implémente Comparable de E

<? extends Comparable<? Super E>> : Implémente Comparable de E ou Comparable d'une super classe de E

<? extends Comparable<E> & Serializable> : ? Implémente Comparable de E et aussi Serializable

Public <T> T normaliser(T arg) : T est le paramètre de type de la méthode (le même type que l'argument)

Public <T> T getValeur(String key) : Quel est le type de T, à quoi ressemble le « return » de cette méthode?

Facile à Lire ?

Les collections du package java.util

- Des interfaces génériques:

`Collection<E>, List<E>, Set<E>, SortedSet<E>, Map<K,V>`

Les interfaces Collections

`Iterable<E>, Iterator<E>, Comparator<E>, Comparable<E>`

Les interface complémentaires

- Des bases d'implémentation (implémentations partielles: classes abstraites) :

`AbstractCollection<E>, AbstractList<E>, AbstractSet<E>, AbstractMap<K,V>`

- Des implémentations complètes prêtes à l'emploi :

`ArrayList<E>, LinkedList<E>, PriorityQueue<E>,`

`TreeSet<E>, HashSet<E>,`

`HashMap<K,V>, WeakHashMap<K,V>` (voir Strong, Soft et Wheak références: <http://www.baeldung.com/java-weakhashmap>)

- Des Classes utilitaires pour manipuler les collections, et faire des conversions entre Array et Collection

`java.util.Collections, java.util.Arrays`

Deux Patterns derrière le framework Collection

- Le pattern « **Template Methode** » :

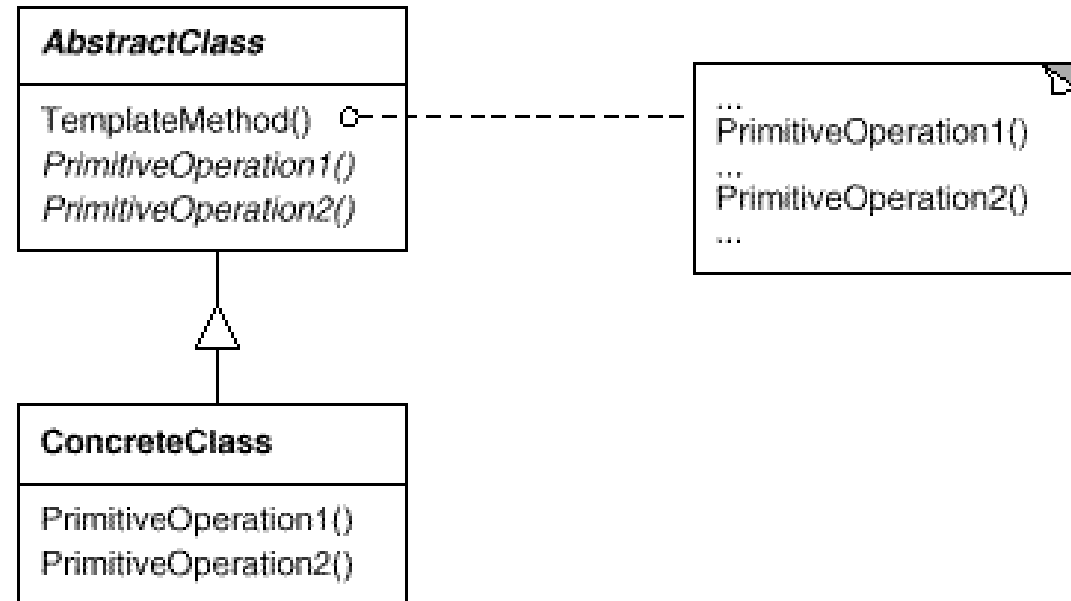
- Délégation de la réalisation à la sous-classe (c'est elle qui sait le mieux comment le faire)
- Utilisé dans les implémentations partielles :

`AbstractCollection<E>, AbstractList<E>, AbstractSet<E>, AbstractMap<K,V>`

- Le pattern « **Iterator** » :

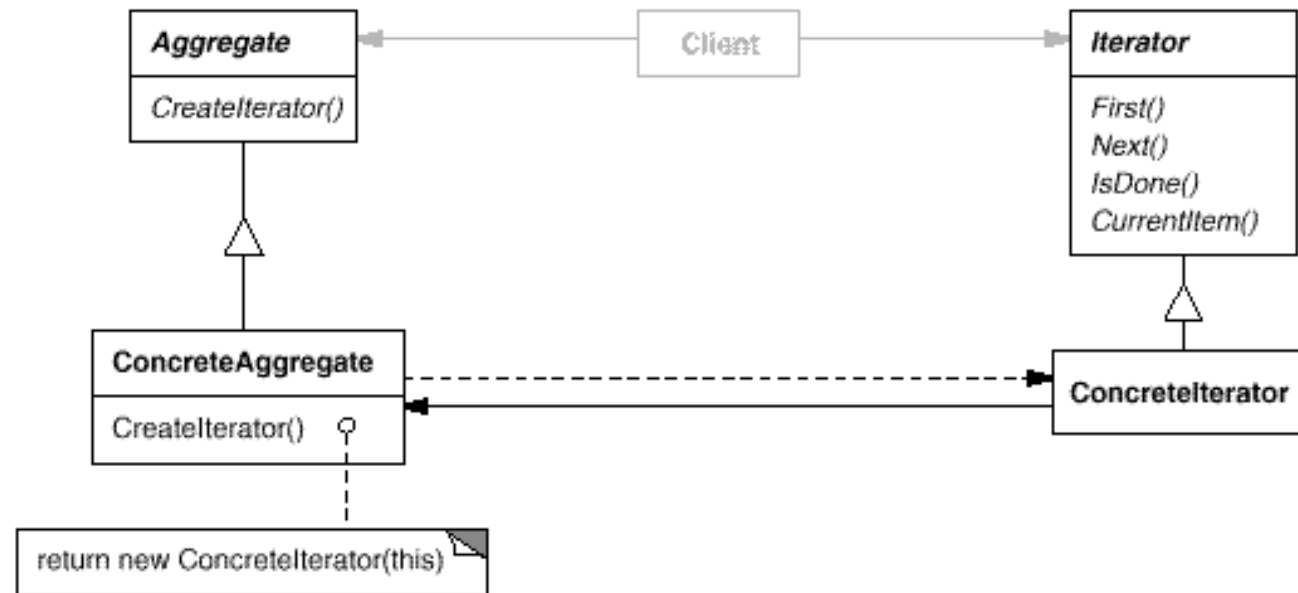
- Permet le parcours de l'ensemble des éléments d'une structure de données sans connaître son implémentation ni sa structure interne.
- Toutes les collections fournissent un itérateur permettant de parcourir leurs éléments.

Le pattern « Template Method »



- D'après le GOF: « **implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.** »

Le pattern « Iterator »



- D'après le GOF: « to access an aggregate object's contents without exposing its internal representation. »

Iterator de java: java.util.Iterator<E>

```
// java.util.Iterator
public interface Iterator<E> {
    /** Returns {@code true} if the iteration has more elements. */
    boolean hasNext();
    /** Returns the next element in the iteration. */
    E next();
    /** Removes from the underlying collection the last element
    returned by this iterator */
    void remove();
}
```

■ 3 méthodes pour parcourir (« itérer sur ») les éléments d'une collection

hasNext() : Existe-t-il encore un élément à parcourir ?

next() : Passer à l'élément suivant et le retourner

remove() : Supprimer l'élément courant (le dernier obtenu par next)

Interface Iterable: java.util.Iterable<T>

- Une seule méthode retournant un itérateur pour parcourir des éléments

```
// java.util.Iterable
public interface Iterable<T> {
    /** Returns an iterator over elements of type T */
    Iterator<T> iterator();
}
```

- L'interface Collection hérite de l'interface Iterable: (qu'en déduire ?)

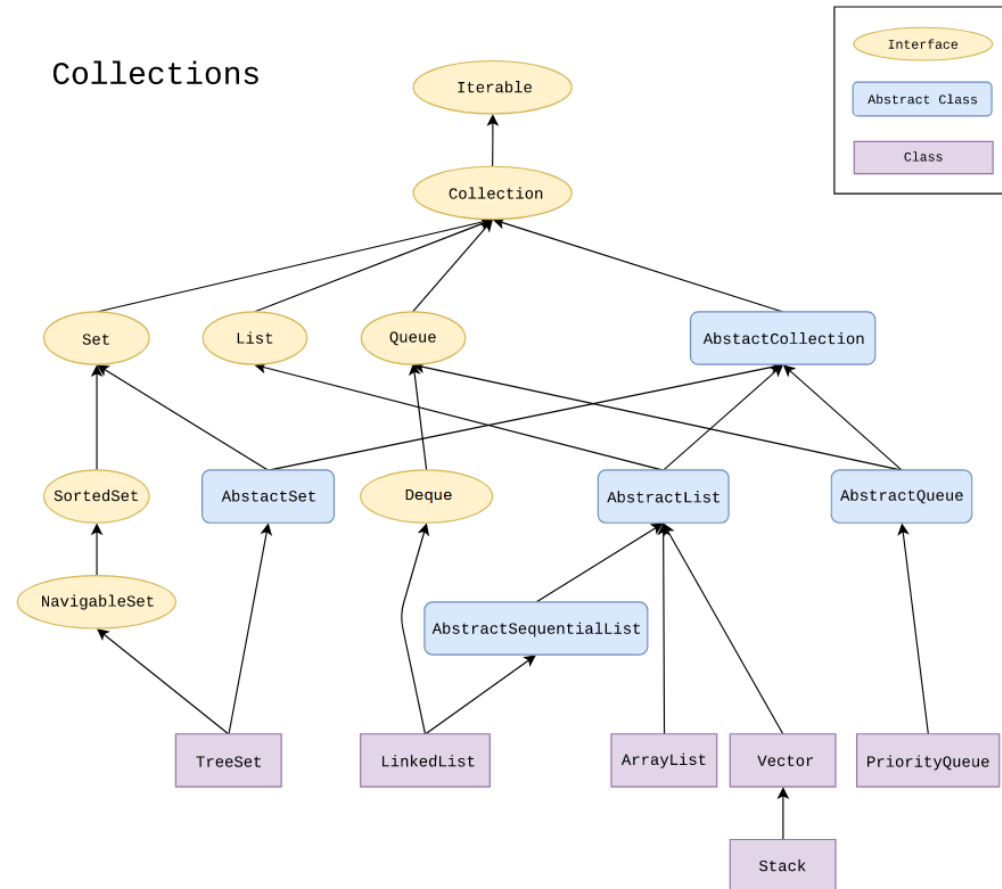
```
// java.util.Collection
public interface Collection<E> extends Iterable<E> {
    ...
}
```

- Les boucles automatiques peuvent être réalisées pour tous les Iterables:

```
for(T element : collectionElements) {
    . . .
}
```

- Les tableaux peuvent aussi être utilisés par les boucles automatiques

Hiérarchie java.util.Collection



https://en.wikipedia.org/wiki/Java_collections_framework

Implémentation de notre Collection ⁽¹⁾

■ 3 Niveaux comme dans le `java.util.Collection`

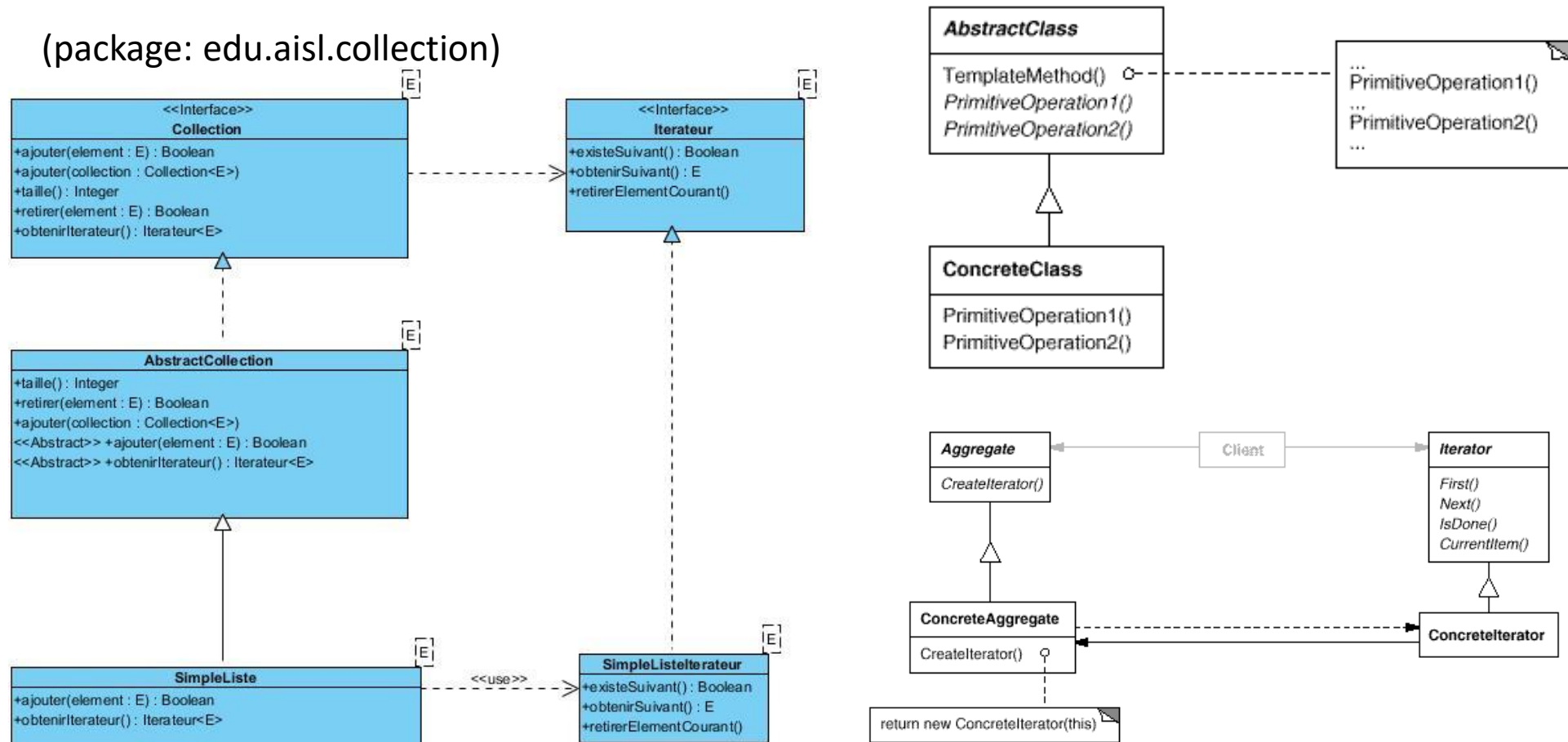
Interface: La spécification de notre collection (ajouter, taille, retirer, obtenir l'itérateur)

Classes abstraites: Collection de base, comportant des méthodes concrètes et abstraites.

Classes concrètes: Implémentation de Collections

Implémentation de notre Collection (2)

(package: edu.aisl.collection)



Implémentation de notre Collection ⁽³⁾

■ Implémentation basée sur « Template Method » en utilisant:

- `obtenirIterateur() : Iterateur<E>`

- `ajouter(element : E)`

- `taille()` est obtenue en comptant le nombre des éléments parcourus en utilisant l'itérateur.

- `retirer()` est implémentée en recherchant l'élément et en utilisant la méthode `remove()` de l'itérateur.

- `ajoute(collection Collection<E>)` est implémentée en utilisant les méthodes `Iterateur()` et `ajouter()`.

etc.

Utilisation de java.util.Iterator

■ Utilisation des itérateurs:

- Tout appel à **remove ()** doit être précédé d'un appel à **next ()**

```
Collection<String> colStr = new ArrayList<>();  
// ajout éléments ...  
Iterator<String> itColStr = colStr.iterator();  
itColStr.remove(); // => java.lang.IllegalStateException
```

- Modifier (ajouter / supprimer) une collection impacte les processus qui sont en train de la parcourir.

```
Collection<String> colStr = new ArrayList<>();  
// ajout éléments ...  
colStr.add("A");  
colStr.add("B");  
Iterator<String> itColStr1 = colStr.iterator();  
Iterator<String> itColStr2 = colStr.iterator();  
itColStr1.next();  
itColStr1.remove();  
itColStr2.next(); // java.util.ConcurrentModificationException
```

Java.util.AbstractCollection<E> : étude du code

- `iterator`
- `add`
- `contains`
- `addAll`
- `remove`
- `removeAll`
- `retainAll`

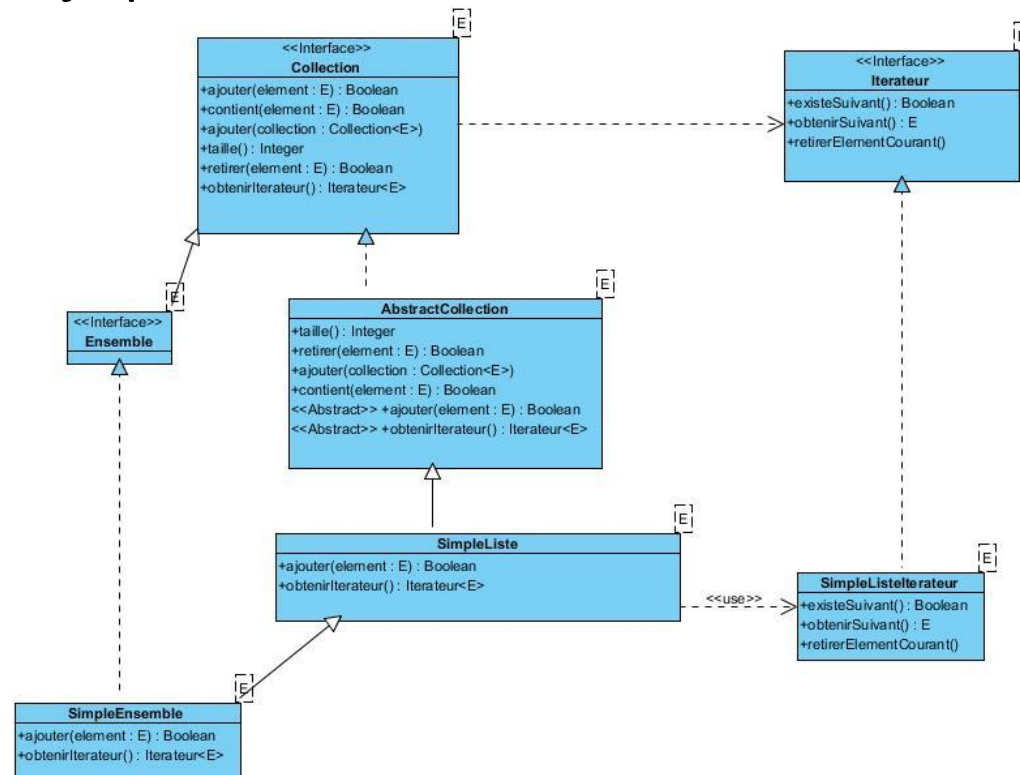
Implémentation d'une structure « Ensemble »

- **Structure de données dans laquelle il ne peut exister qu'une seule occurrence pour un élément.**

Vos Propositions pour ajouter cette implémentation à edu.aisl.collection?

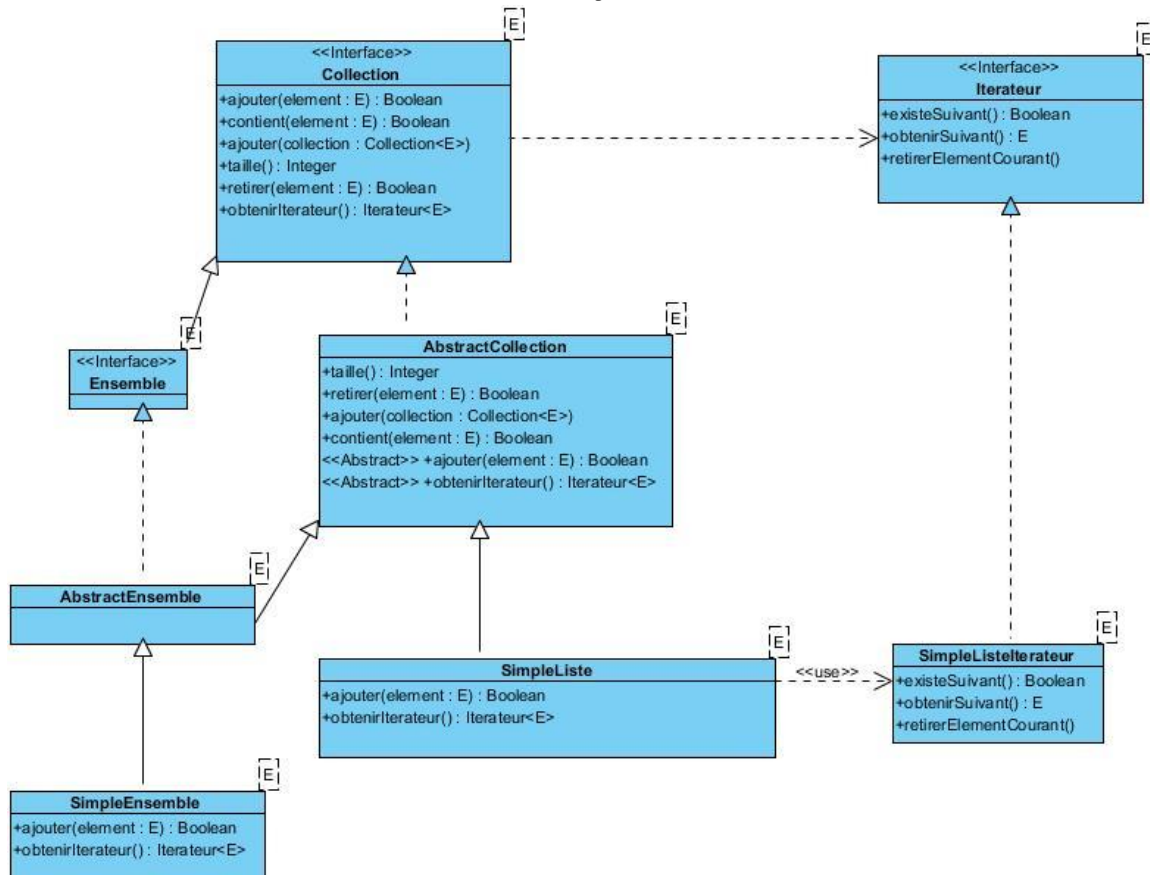
Implémentation d'une structure « Ensemble »

- Utilisation de la méthode **equals()** des éléments pour rechercher si l'élément est déjà présent



Implémentation d'une structure « Ensemble »

Autre choix: comme dans java.util



Le SortedSet<E> (Set avec relation d'ordre)

```
public interface SortedSet<E> extends Set<E> {
    /** Returns the comparator used to order the elements in this set */
    Comparator<? super E> comparator();

    /** Returns a view of the portion of this set whose elements range */
    SortedSet<E> subSet(E fromElement, E toElement);

    /** Returns a view of the portion of this set whose elements are
     * strictly less than <tt>toElement</tt>. */
    SortedSet<E> headSet(E toElement);

    /** Returns a view of the portion of this set whose elements are
     * greater than or equal to <tt>fromElement</tt>.
     */
    SortedSet<E> tailSet(E fromElement);

    /** Returns the first (lowest) element currently in this set. */
    E first();

    /** Returns the last (highest) element currently in this set. */
    E last();
}
```

Relation d'ordre

■ Le comparator : utilisé pour comparer 2 instances de type T

```
public interface Comparator<T> {  
    /**  
     * Compares its two arguments for order. Returns a negative integer,  
     * zero, or a positive integer as the first argument is less than, equal  
     * to, or greater than the second.<p> */  
    int compare(T o1, T o2);  
}
```

■ Comparable : les instances d'un type qui réalise cette interfaces sont en mesure de se comparer avec d'autres instances de leur type.

```
public interface Comparable<T> {  
    /**  
     * Compares this object with the specified object for order. Returns a  
     * negative integer, zero, or a positive integer as this object is less  
     * than, equal to, or greater than the specified object.  
     */  
    int compareTo(T o1);  
}
```

Intéropérabilité entre Collections

■ Liste<T> ↔ Set<T> :

- Les éléments d'une Liste de T peuvent être ajoutés à un Set<T>. (Pourquoi, comment ?)

```
List<Integer> colInt = new ArrayList<>();  
colInt.add(3);  
colInt.add(2);  
colInt.add(1);  
System.out.println("colInt:" + colInt); // « colInt:[3, 2, 1] »  
TreeSet<Integer> orderedSet = new TreeSet<Integer>();  
orderedSet.addAll(colInt);  
System.out.println("orderedSet:" + orderedSet); // «orderedSet:[1, 2, 3]»
```

- Par quel procédé les éléments ont été triés dans le TreeSet ?

Comparable<T>: un exemple

```
/**
 * Un Bidule comparable
 */
public class Bidule implements Comparable<Bidule>{

    protected int taille;
    protected int poids;

    public Bidule(int taille, int poids) {
        super();
        this.taille = taille;
        this.poids = poids;
    }
    @Override
    public int compareTo(Bidule autre) {
        // relation d'ordre basée sur le produit
        // de la taille et du poids
        int valComp = taille * poids;
        int autreValComp = autre.taille * autre.poids;
        if(valComp == autreValComp) {
            return 0;
        } else if(valComp < autreValComp) {
            return -1;
        } else {
            return 1;
        }
    }
}
```

```
// Test Bidules dans un TreeSet
List<Bidule> mesBidules = new Vector<>();
mesBidules.add(new Bidule(5, 10));
mesBidules.add(new Bidule(11, 2));
mesBidules.add(new Bidule(1, 40));
System.out.println("colBidule:" + mesBidules);
// colBidule:[Val:50, Val:22, Val:40]
Set<Bidule> mesBidulesOrdonnes = new
TreeSet<>(mesBidules);
System.out.println("mesBidulesOrdonnes:" +
mesBidulesOrdonnes );
// mesBidulesOrdonnes:[Val:22, Val:40, Val:50]
```

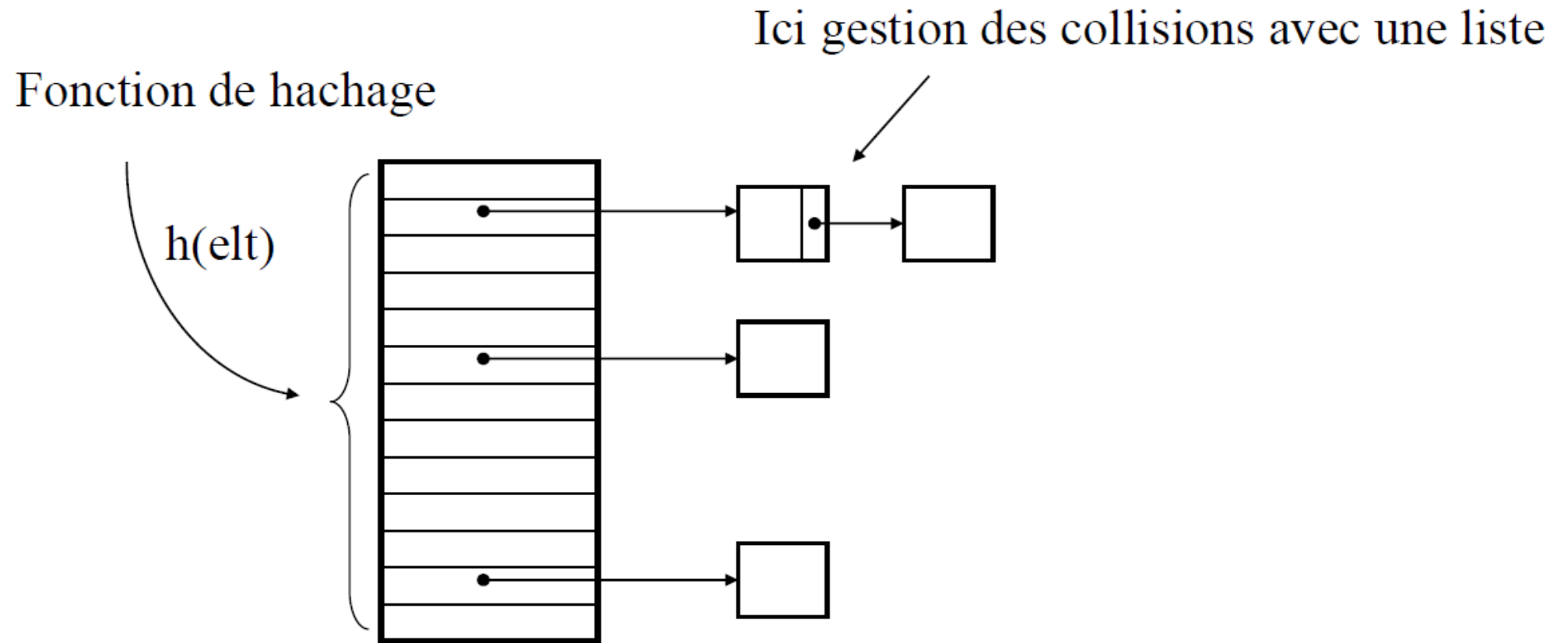
ATTENTION: si compareTo retourne « Egal » alors equals doit retourner vrai aussi. Sinon des effets de bord importants sont à prévoir !!!

Un autre type de structure de données: La Map

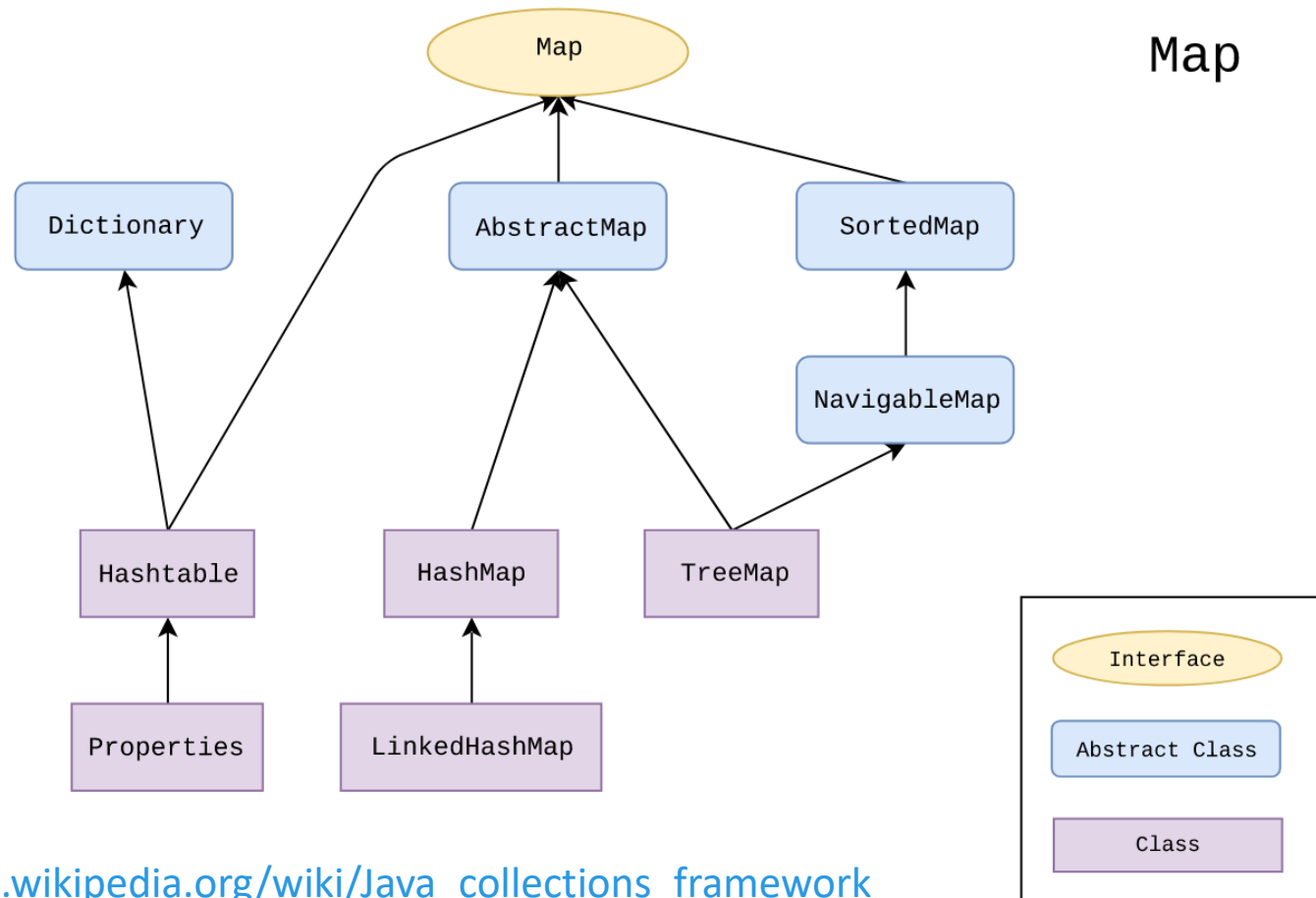
- Structure de données associant une valeur à une clé.
- Les Dictionnaires, Hashtable, HashMap etc. sont des Map.
- Implémentation de l'interface Map<K,V>
 - K est le type de la clé
 - V est le type de la valeur
- Les couples Clé,Valeur implémentent java.util.Map.Entry<K,V>
- Les implémentations utilisent une fonction de Hachage (la méthode hashCode()) pour alimenter une table

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + poids;  
    result = prime * result + taille;  
    return result;  
}
```

Une autre type de structure de données: La Map



Hiérarchie java.util.Map



https://en.wikipedia.org/wiki/Java_collections_framework

L'interface Map<K,V>

```
public interface Map<K,V> {  
    /** Returns the number of key-value mappings in this map. */  
    int size();  
    /** Returns <tt>true</tt> if this map contains no key-value mappings. */  
    boolean isEmpty();  
    /** Returns <tt>true</tt> if this map contains a mapping for the specified key. */  
    boolean containsKey(Object key);  
    /** Returns <tt>true</tt> if this map maps one or more keys to the specified value. */  
    boolean containsValue(Object value);  
    /** Returns the value to which the specified key is mapped */  
    V get(Object key);  
    /** Associates the specified value with the specified key in this map */  
    V put(K key, V value);  
    /** Removes the mapping for a key from this map if it is present */  
    V remove(Object key);  
    // ...  
}
```

Parcours d'une Map<K,V>

```
// Création de la Map
Map<String, Bidule> mapBidules = new HashMap<String, Bidule>();
// Ajout des éléments
mapBidules.put("BID01", new Bidule(5, 10));
mapBidules.put("BID02", new Bidule(11, 2));
mapBidules.put("BID03", new Bidule(1, 40));
Bidule b = mapBidules.get("BID01"); // Récupération du bidule déposé avec la clé « BID01 »
```

■ Accéder aux valeurs stockées dans la Map

```
for (Bidule bidule : mapBidules.values()) {
    //...
}
```

■ Accéder aux clés des valeurs stockées dans la Map

```
for (String cle : mapBidules.keySet()) {
    //...
}
```

■ Accéder aux couples Clé,Valeur. Ces couples sont stockés dans des structures qui implémentent java.util.Map.Entry<K,V>

```
for (Entry<String, Bidule> entry : mapBidules.entrySet()) {
    //...
}
```

Les Classes utilitaires: La classe java.util.**Collections**

■ **La classe Collections** : des fonctions et algorithmes utilitaires pour manipuler les Collections.

- **Tri de collections** (utilise Comparable ou Comparator)
- **Recherche de valeur** (BinarySearch)
- **Inversion ordre des éléments** (Reverse)
- **Mélange aléatoire** (Shuffle)
- **Swap, Fill, Copy, Min, Max etc.**

- **UnmodifiableCollection, UnmodifiableList** (Collections en lecture seule)
- **SynchronizedCollection** (accès ThreadSafe)

Les Classes utilitaires: La classe `java.util.Arrays`

■ **La classe `Arrays`** : des fonctions pour manipuler les tableaux

- **Tri de tableaux** (utilise `Comparable` ou `Comparator`, `sort`, `parallelSort`)
- **Recherche de valeur** (`BinarySearch`)
- **Conversion en List** (`asList(T..a)`)
- **Swap, Fill, Copy etc.**