



COURS DE PROGRAMMATION ORIENTÉE OBJET

L2 Info

L.KAHLEM - laure.kahlem@univ-orleans.fr

LES TYPES DE DONNÉES ABSTRAITS (TDA)

- Les TDA sont nés des problèmes liés au développement d'applications:
 - maîtriser la complexité -> **modularité**
 - réutiliser du code existant -> fournir des solutions pour des familles de problèmes (**bibliothèques à usage générale**).
 - traiter des problèmes de haut niveau en s'affranchissant des problèmes de niveau inférieur déjà résolus -> **abstraction des données**.



LES TYPES DE DONNÉES ABSTRAITS (TDA)

- Idée directrice
 - **Parallèle avec les types primitifs**
 - Le type int : représente un entier,
 - est fourni avec une notation et des opérations : + - / * %.
 - Il n'est pas nécessaire de connaître la représentation interne ou les algorithmes de ces opérations pour les utiliser.



LES TYPES DE DONNÉES ABSTRAITS (TDA)

- Idée directrice
 - **Faire de même pour des types plus complexes indépendamment d'un langage de programmation**
 - Créer un type, dont la représentation interne est cachée.
 - Offrir les opérations de haut niveau nécessaires.

LES TYPES DE DONNÉES ABSTRAITS (TDA)

○ Définition

Un **TDA** est

- un type de données
- et l'ensemble des opérations permettant de gérer ces données, les détails de l'implantation restant cachés.



LES TYPES DE DONNÉES ABSTRAITS (TDA)

○ Exemple : le TDA liste

Type LISTE

Utilise Element, Boolean, Place

Les opérations :

- **Constructeurs**

- **Creer**: \Rightarrow Liste (crée une liste vide)
- **Ajouter**: $\text{Element} \times \text{Liste} \Rightarrow \text{Liste}$
- **AjouterPos**: $\text{Element} \times \text{Liste} \times \text{Place} \Rightarrow \text{Liste}$

LES TYPES DE DONNÉES ABSTRAITS (TDA)

○ Exemple : le TDA liste

• Selecteurs

- **tete** : Liste => Element
- **queue** : Liste=>Liste
- **longueur** : Liste=> Element
- **estVide** : Liste => Boolean

• Modificateurs

- **Enlever** : Liste × Place => Liste
- **Modifier** : Liste × Place × Element => Liste



LES TYPES DE DONNÉES ABSTRAITS (TDA)

○ **Spécification d'un TDA en Java**

● **Interface**

- Degré d'abstraction plus élevé.
- Définition formelle d'un TDA.
- Déclaration des méthodes de manière abstraite.

● **Généricité**

- Par paramètres de type

● **Exceptions**

- Permettent une gestion de erreurs pouvant survenir lors de l'utilisation d'un TDA.



LES TYPES DE DONNÉES ABSTRAITS (TDA)

○ **Réalisation d'un TDA**

Définition : Une **structure de données** est une construction du langage de programmation permettant de représenter un TDA.

- Exemple : On peut utiliser un tableau comme structure de données pour implanter une liste.
- On pourra construire **plusieurs structures de données pour un même TDA.**



LES TYPES DE DONNÉES ABSTRAITS (TDA)

- **Réalisation d'un TDA en Java**
 - **Classes implémentant l'interface définissant le TDA**

Encapsulation de la structure interne du TDA :

- Visibilité privée pour les attributs de la classe décrivant la structure de donnée choisie.
- Visibilité publique pour les méthodes.



LES TYPES DE DONNÉES ABSTRAITS (TDA)

- Nous allons nous intéresser aux types de données abstraits décrivant des **conteneurs d'objets : les collections**.



LES TYPES DE DONNÉES ABSTRAITS (TDA)

Exemples:

1. Gestion d'un annuaire :

- Insertion de personnes
- Suppression de personnes
- Recherche de personnes suivant leur nom

2. Gestion d'une file d'attente:

- Ajouter des personnes en fin de file
- Traiter la personne en début de file



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

Les objectifs :

- Découvrir la bibliothèque des Collections Java qui fournit les interfaces et les classes correspondant aux TDA classiques définissant des conteneurs d'objets.
- Comprendre les différentes implémentations proposées.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

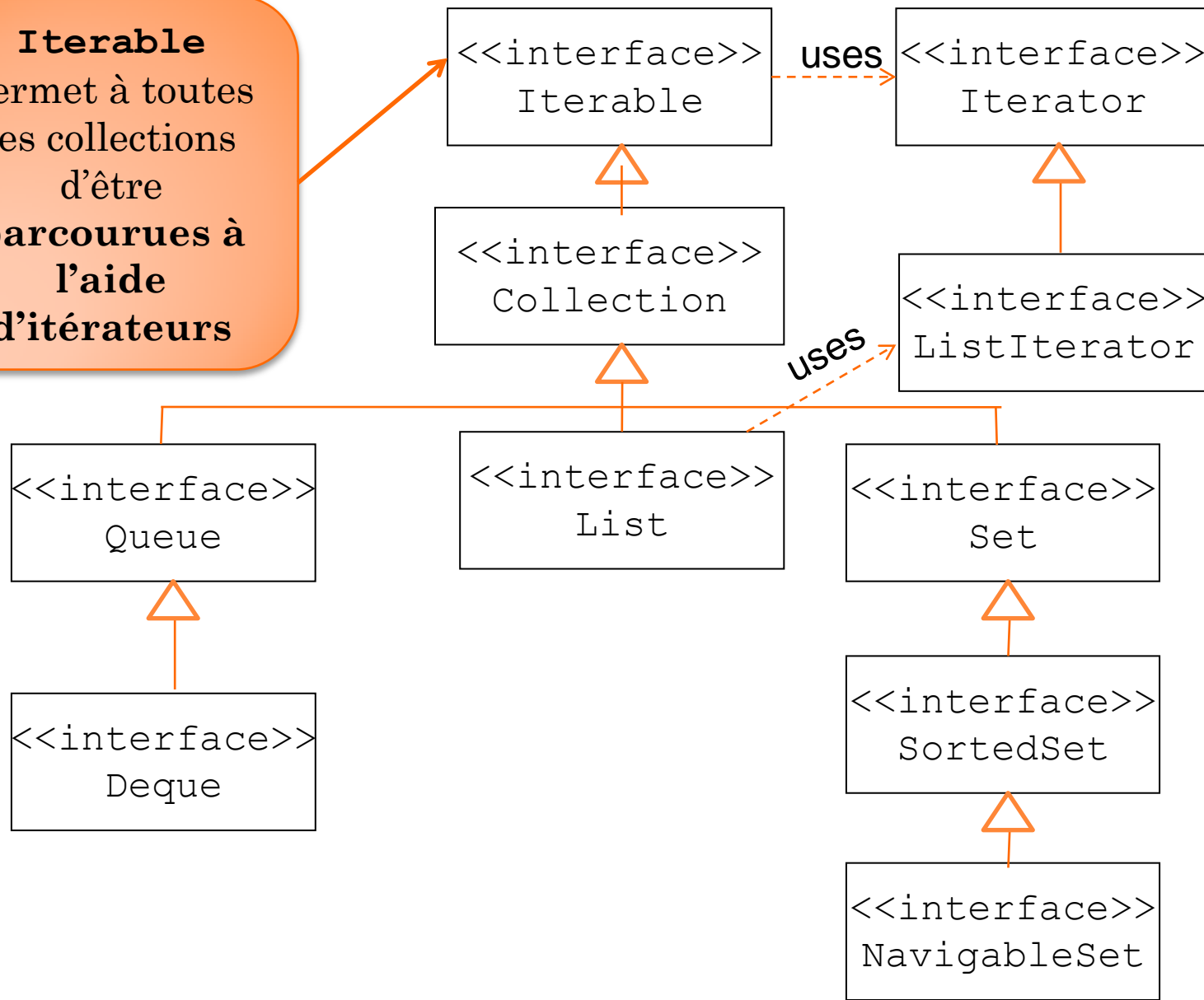
○ Les interfaces

- Cette bibliothèque est composée d'**interfaces** définies dans le paquetage **java.util**.
- Il existe deux interfaces fondamentales pour les conteneurs:
 - **Collection**
 - **Map**



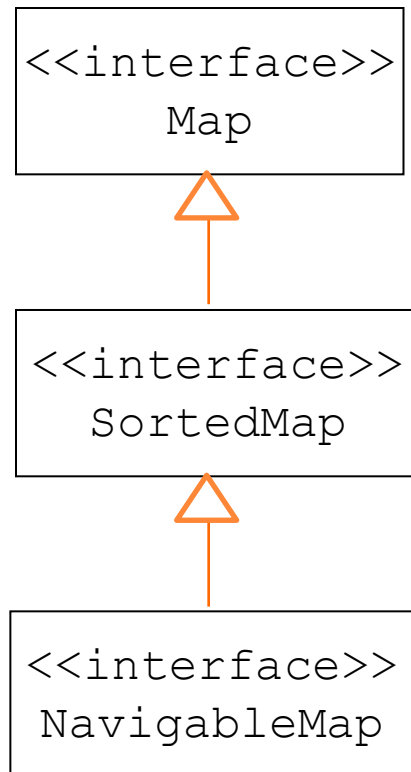
LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

Iterable
permet à toutes
les collections
d'être
parcourues à
l'aide
d'itérateurs



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface Collection

```
public interface Collection<E> extends  
    Iterable<E>{  
    //consultation des éléments  
    int size();  
    boolean isEmpty();  
    boolean contains (Object o);  
    boolean containsAll (Collection<?> c);  
    Object[] toArray ();  
    <T> T[] toArray (T[] a);
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface Collection

//mise à jour des éléments

```
boolean add (E e);  
boolean addAll(Collection<? extends E> c);  
void clear();  
boolean remove(Object object);  
boolean removeAll(Collection<?> c);  
boolean retainAll (Collection<?> c);  
default boolean removeIf(Predicate<? super  
E> filter)
```

//héritée de l'interface Iterable

```
public Iterator<E> iterator();  
}
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface **Collection**

Rappel : Types Joker

- `Collection<? extends E>` remplace toute collection générique dont le type est un sous-type de `E`.
- **Exemple** : `Collection<? extends CompteBancaire>` remplace `Collection<CompteCourant>` ou `Collection<CompteEpargne>`.
- **Utilité** : On veut pouvoir ajouter une collection d'objets de types `CompteCourant` à une collection d'objets de type `CompteBancaire`.
Mais `Collection <CompteCourant>` n'étend pas `Collection <CompteBancaire>`.

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

- **Les interfaces**
 - **Interface Collection**

Les méthodes **contains**, **containsAll**, **remove**, **removeAll** et **retainAll** s'appuient sur la méthode **equals** qui doit être correctement redéfinie.

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface Collection**

- La méthode **removeIf** prend en paramètre un **prédicat**. Elle permet de supprimer les éléments de la collection pour lesquels la méthode **test** renvoie **true**. Elle renvoie `true` si la collection a été modifiée.

- Exemple :

```
Collection<Integer> c = /* création d'une  
collection vide*/
```

```
c.add (4); c.add(15); c.add(22); c.add(5);
```

```
//suppression des entiers>10
```

```
c.removeIf (i -> i>10); //c contient 4 et 5
```

```
//suppression des entiers pairs
```

```
c.removeIf (i -> i %2 ==0); //c contient 5 et  
15
```

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface **List**

- Spécification du **TDA liste**.
- Permet de manipuler des **collections organisées sous forme d'une séquence d'éléments d'un type donné**.
- Chaque élément possède **un rang**.
- Un objet pourra être inséré à une position adéquate indiquée soit
 - par un indice entier
 - par un itérateur de liste



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface **List**

- `public interface List<E> extends Collection<E>`

- Les méthodes supplémentaires

//consultation et mise à jour

- `E get (int index);`

- `E set (int index, E element);`

- `E remove (int index);`

- `void add(int i, E element)`

- `boolean addAll(int i, Collection<? extends E> c)`

- `List<E> subList (int start, int stop);`



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface `List`**

- Les méthodes supplémentaires

//récupération d'indices

- `int indexOf (Object object);`

- `int lastIndexOf (Object object);`

//Mise à jour par lambda expressions

- `default void replaceAll (UnaryOperator<E> op)`
 `//maj`

- `default void sort (Comparator<? super E> c)`
 `//tri`



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface **List**

- Les méthodes supplémentaires

//parcours par itérateur spécialisé

- `ListIterator<E> listIterator ();`

- `ListIterator<E> listIterator (int index);`



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface **Set**

- Spécification du **TDA ensemble**.
- Permet de manipuler une collection d'éléments d'un type donné **qui ne comporte pas de doublons**.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface Set

- `public interface Set<E> extends Collection<E>`
- Aucune méthode n'est ajoutée en plus.
- Le comportement des méthodes y est défini de manière plus précise:
 - La méthode **add** rejette les valeurs doubles.
 - La méthode **equals** teste si deux ensembles ont les mêmes éléments mais pas nécessairement dans le même ordre.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface SortedSet

- `public interface SortedSet<E> extends Set<E>`
- Représente les ensembles **dont les éléments sont ordonnables selon une relation d'ordre.**



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface SortedSet

○ Méthodes supplémenaires :

//comparateur utilisé

```
public Comparator<? super E> comparator();
```

//les éléments de [x .. y[

```
SortedSet<E> subSet(E x, E y);
```

//les éléments plus petits que e

```
SortedSet<E> headSet(E e);
```

//les éléments plus grands que e

```
SortedSet<E> tailSet(E e);
```

E first() //le plus petit élément

E last() //le plus grand élément



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Protocole d'itération - Interface **Iterable<T>**

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
    default void forEach(Consumer<? Super T>  
        action) { /*...*/ }  
}
```

Les classes qui implémentent l'interface `Iterable` devront implémenter la méthode `iterator` mais pas la méthode `forEach` qui est une méthode par défaut.

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Protocole d'itération - Interface `Iterable<T>`**
- La méthode **`forEach`** prend en paramètre un **consommateur**. Elle applique l'action spécifiée en paramètre à tous les éléments de l'`Iterable`. L'interface **`Consumer`** définit une méthode `void accept (T t).`

- Exemple :

```
Collection<Integer> c = /* création d'une  
collection vide*/
```

```
c.add (4); c.add(15); c.add(22); c.add(5);
```

```
//affichage de tous les éléments
```

```
c.forEach ( i -> System.out.println(i) );
```

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface **Iterator<E>**

- Un **itérateur** est un objet associé à une collection qui permet de
 - **Parcourir séquentiellement** tous les éléments de la collection.
 - **Faire des mises à jour** de la collection.
- La méthode **public Iterator<E> iterator()** définie dans l'interface **Iterable** permet d'associer un itérateur à une collection.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface `Iterator<E>`

```
public interface Iterator<E>{  
    public boolean hasNext() ; //renvoie true s'il  
    //existe un prochain élément  
    public E next() ;  
    public void remove() ;  
}
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

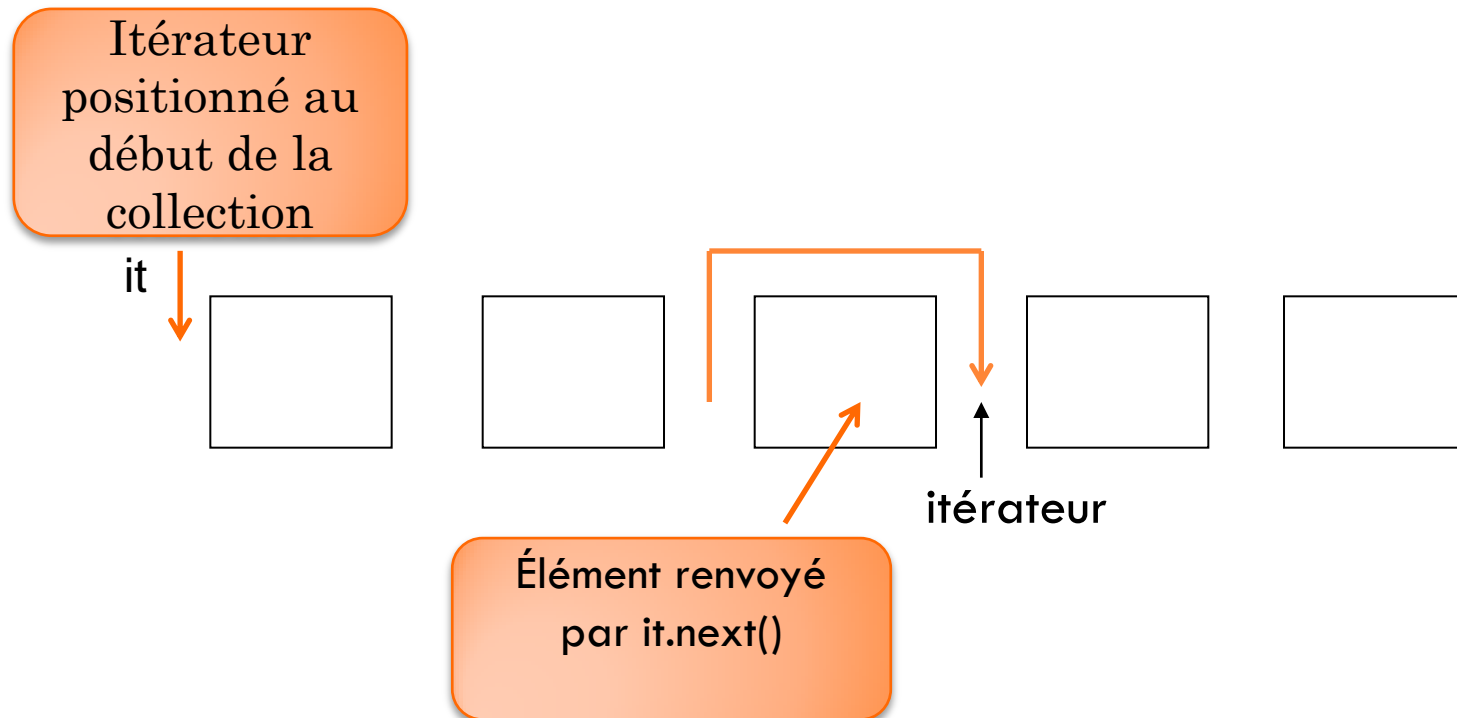
- Interface **Iterator<E>** : la méthode **next**
public E next() throws NoSuchElementException
 - En appelant plusieurs fois la méthode `next`, on peut parcourir tous les éléments de la collection .
 - Lorsque la fin de la collection est atteinte, une exception est levée : `NoSuchElementException` qui dérive de `RuntimeException`.
 - Il faut donc appeler la méthode `hasNext` avant la méthode `next`.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface `Iterator<E>`** : la méthode **`next`**
Progression d'un itérateur



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface `Iterator<E>`**: la méthode **`next`**

Parcours de tous les éléments de type `E` d'un objet conteneur de type `Collection<E>`.

//création d'un itérateur sur le conteneur

```
Iterator<E> it = conteneur.iterator();
```

```
while (it.hasNext()) {
```

```
    E element = it.next();
```

```
    //utilisation de l'object element
```

```
}
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface `Iterator`** : la méthode **`remove`**

`public void remove() throws IllegalStateException`

- Supprime l'élément renvoyé par le dernier appel à `next`.

- **Obligation d'appeler `next` avant `remove`**, si on souhaite effacer un élément en fonction de sa position.

```
Iterator<E> it = conteneur.iterator();
```

```
it.next(); //passer l'élément
```

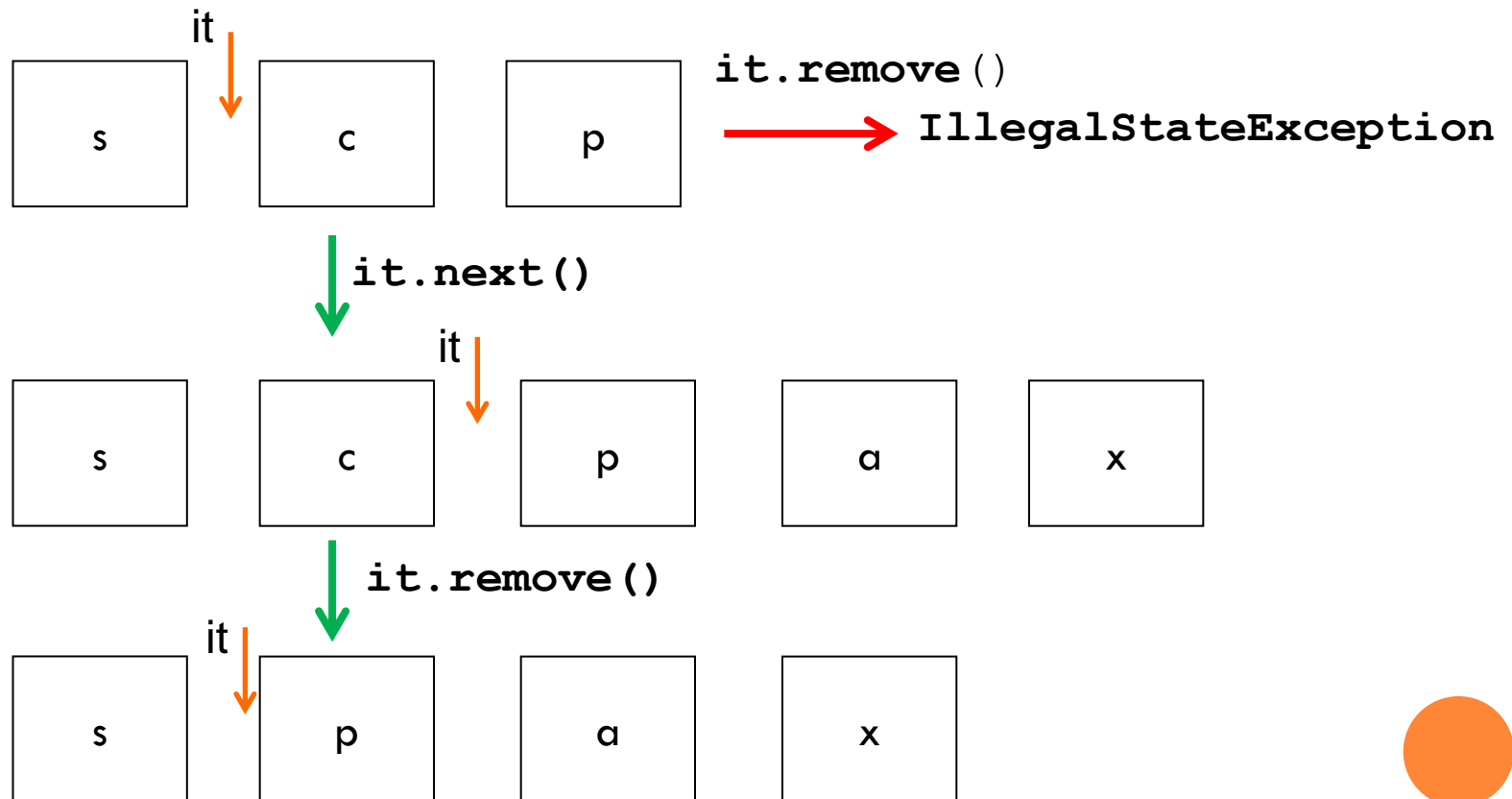
```
it.remove(); //supprimer l'élément
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface **Iterator** : la méthode **remove**



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface **Iterator<E>**

- Application :

- 1- Supprimer un mot sur deux dans une liste.

```
public static <E> void supprime(List<E> l) {  
    Iterator<E> it= l.iterator();  
    while (it.hasNext()) {  
        it.next();  
        if (it.hasNext()) {  
            it.next();  
            it.remove();  
        }  
    }  
}
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface `ListIterator<E>` extends `Iterator<E>`**
 - L'interface **`ListIterator`** représente les itérateurs spécialisés pour les listes.
 - La méthode `add` héritée de collection permet uniquement d'ajouter un élément à la fin d'une liste.
 - **`ListIterator`** fournit une méthode qui permet d'ajouter un élément à la position de l'itérateur.
 - **`public void add (E e)`**
//après l'appel de `add`, l'itérateur se trouve après
//l'élément inséré.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface `ListIterator<E>`

Fournit également:

- deux méthodes permettant de parcourir la liste à l'envers:
 - `boolean hasPrevious()`
 - `E previous()`
- une méthode permettant de remplacer le dernier élément renvoyé par `next` ou `previous` par un nouvel élément:
 - `void set (E e)`
- Deux méthodes permettant de récupérer l'indice d'un élément:
 - `int nextIndex()`
 - `int previousIndex()`



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface `ListIterator<E>`

- Applications :

- 1- Proposer une méthode pour fusionner les éléments d'une liste `l2` dans une liste `l1` .

Ex : à partir de `l1 = [a1, a2, a3]` et `l2 = [b1, b2, b3, b4]` , on obtient

`l1 = [a1, b1, a2, b2, a3, b3, b4]` et

`l2 = []` .

- 2 – Dans une liste d'entiers, supprimer les entiers négatifs et incrémenter les entiers positifs impairs.

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

```
public static <E> void fusionner(List<E> l1, List<E> l2)
{
    ListIterator<E> it1 = l1.listIterator();
    Iterator<E> it2 = l2.iterator();
    while (it1.hasNext() && it2.hasNext()) {
        it1.next();
        it1.add(it2.next());
        it2.remove();
    }
    if (it2.hasNext()) {
        l1.addAll(l2);
        l2.clear();
    }
}
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

```
public static void supprimeIncremente(List<Integer> l){  
    //on peut positionner l'itérateur à la fin de la liste  
    ListIterator<Integer> it= l.listIterator(l.size());  
    while(it.hasPrevious()) {  
        Integer n=it.previous();  
        if (n<0)  
            it.remove();  
        else if(n%2==1)  
            it.set(n+1);  
    }  
}
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

• Interface Map

- Une table est un conteneur qui permet un accès direct par type d'index.
- Une table est une séquence de lignes constituées de paires (clé, valeur):
 - la **clé sert d'index** dans la table.
 - la **valeur** du composant clé. Elle contient les informations recherchées.
- Exemple : le dictionnaire
 - La clé est le mot consulté
 - La valeur est la définition du mot



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Exemple de table

Paires (prénom, âge) où on suppose que le prénom peut être pris comme clé (toutes les personnes enregistrées ont des prénoms différents).

| Prénom | âge |
|---------|-----|
| Jean | 18 |
| Jacques | 21 |
| Vincent | 10 |
| Lucien | 10 |
| Pierre | 14 |



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface Map

```
public interface Map<K,V>
```

```
//Consultation des éléments
```

```
    int size();
```

```
    boolean isEmpty();
```

```
    boolean containsKey(Object key);
```

```
    boolean containsValue(Object value);
```

```
//Récupération de la valeur connaissant la clé
```

```
    V get(Object key);
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface Map

//Mise à jour

```
V put(K key, V value);
```

```
void putAll (Map<? extends K, ? extends V> m);
```

```
default V putIfAbsent(K key, V value);
```

```
default boolean replace(K key, V oldValue, V  
newValue);
```

```
default V replace(K key, V value);
```



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface Map

//suppression

```
void clear();
```

```
V remove(Object key);
```

//création de vues

```
Set<K> keySet();
```

```
Collection<V> values();
```

```
Set<Map.Entry<K, V>> entrySet();
```



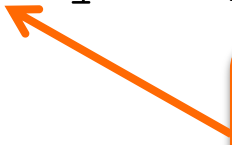
LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface Map

```
Set<Map.Entry<K, V>> entrySet();  
//retourne l'ensemble des lignes de la table  
//Chaque ligne retournée est un Map.Entry<K, V>
```

```
public interface Entry <K, V>{  
    K getKey();  
    V getValue();  
    V setValue(V value);  
    boolean equals(Object o);  
    int hashCode();  
}
```



Interface interne
à l'interface Map

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- **Interface Map – Les vues d'une table**
- Une **table** n'est pas une collection.
- Possibilité d'obtenir **une vue d'une table**, i.e. un objet qui implémente l'interface `Collection` ou une de ses sous-interface.



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

- Interface **Map** – Les vues d'une table

Il existe 3 vues différentes :

- L'ensemble des clés fournie par la méthode

public Set<K> keySet();

- L'ensemble des valeurs fournie par la méthode

public Collection<V> values();

- L'ensemble des paires clé/valeur fournie par la méthode

public Set<Map.Entry<K, V>> entrySet();

LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les interfaces

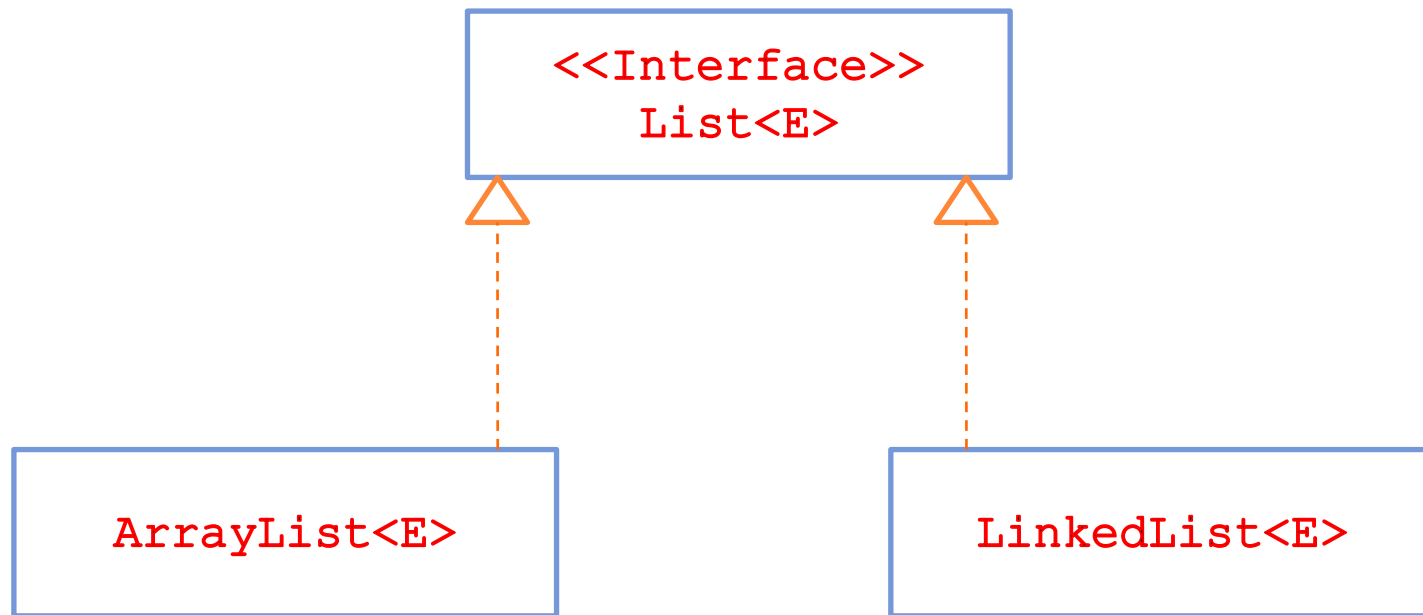
- Interfaces **SortedMap** et **NavigableMap**

Représentent les tables dont les clés sont munies d'une relation d'ordre total.



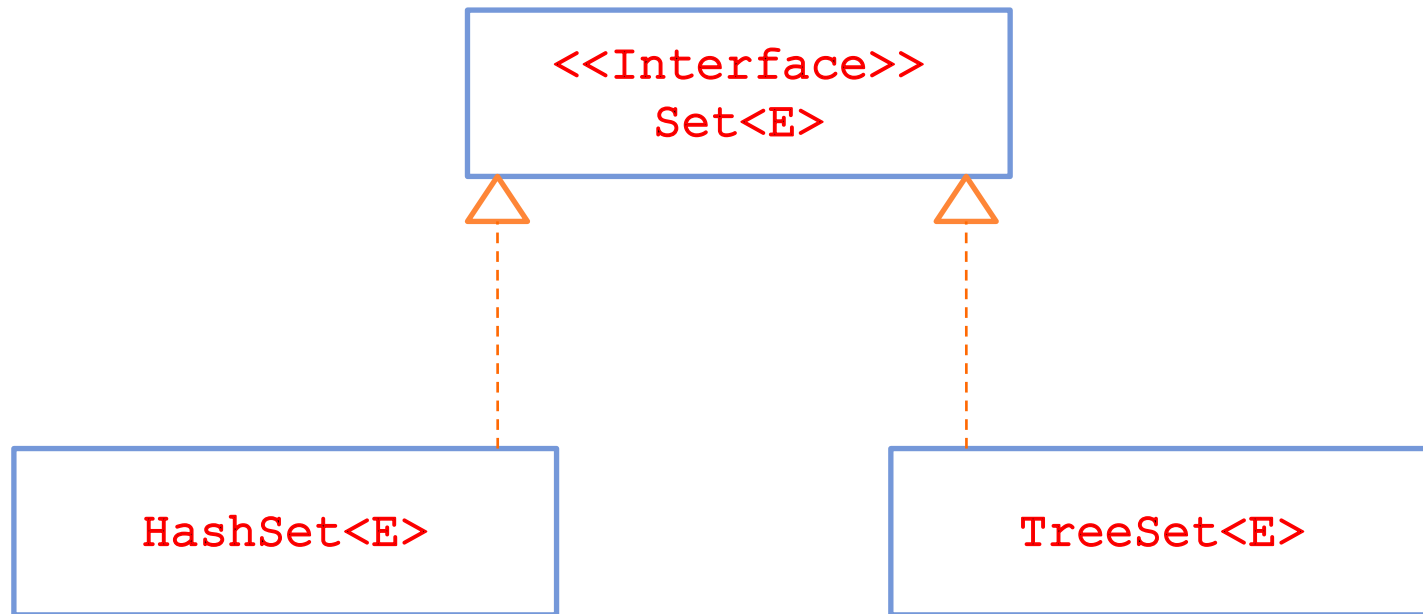
LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les classes concrètes



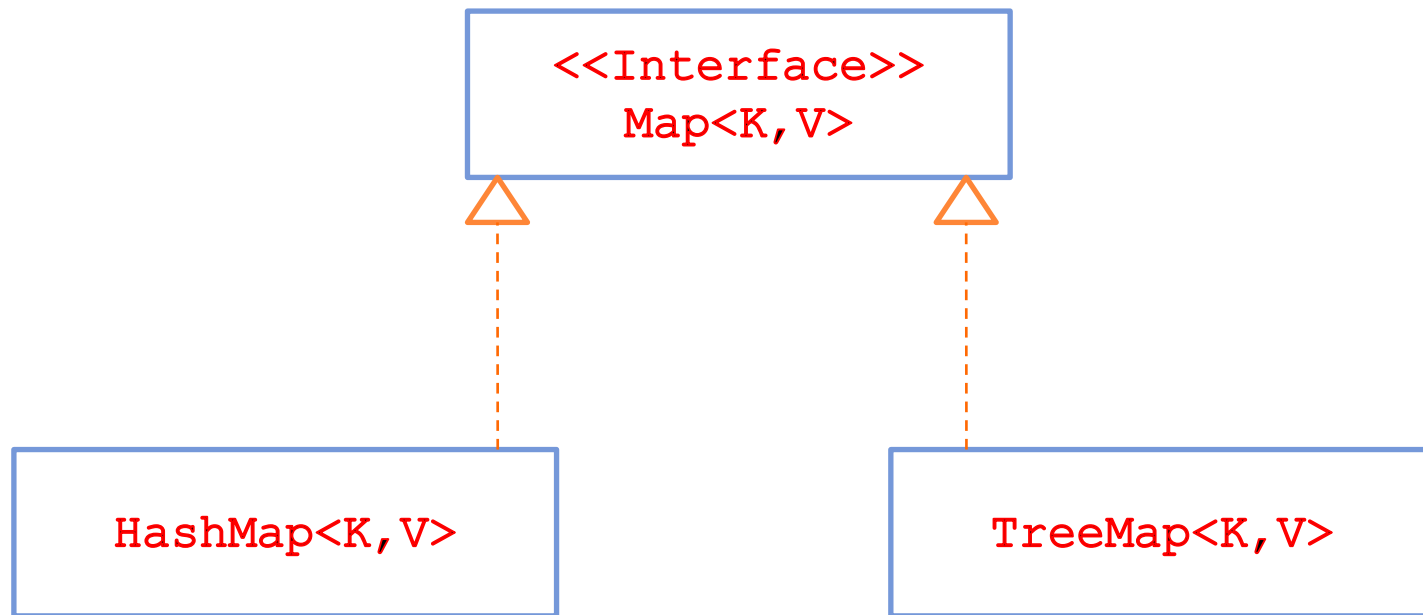
LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les classes concrètes



LA BIBLIOTHÈQUE DES COLLECTIONS JAVA

○ Les classes concrètes



IMPLANTATION DE L'INTERFACE LIST

- **Les classes concrètes implémentant `List<E>`:**
 - **`ArrayList<E>`**
 - utilise un tableau dynamique relogable.
 - **`LinkedList<E>`**
 - utilise une liste chaînée.



IMPLÉMENTATION DE L'INTERFACE LIST

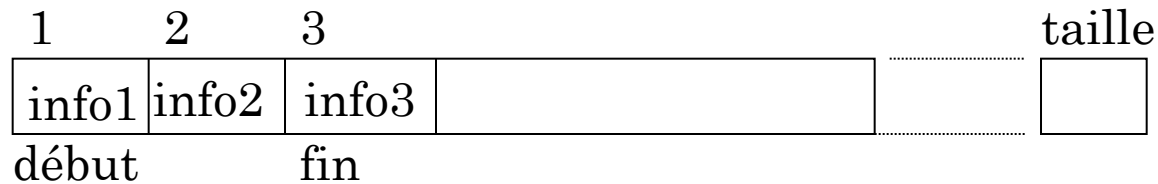
- **Comparaison des deux structures de données pour implémenter une liste**
 1. Implémentation statique par un tableau
 2. Implémentation dynamique par une liste chaînée



IMPLÉMENTATION DE L'INTERFACE LIST

○ Implémentation par un tableau

- Représentation



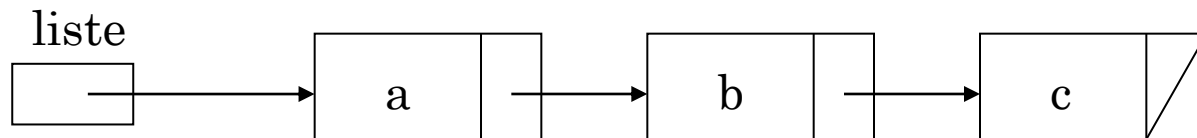
- Entraîne des temps de calculs importants pour réaliser les insertions et les suppressions.
- Nécessité de redimensionner le tableau quand on atteint la capacité initiale.



IMPLÉMENTATION DE L'INTERFACE LIST

○ **Implantation par une liste chaînée**

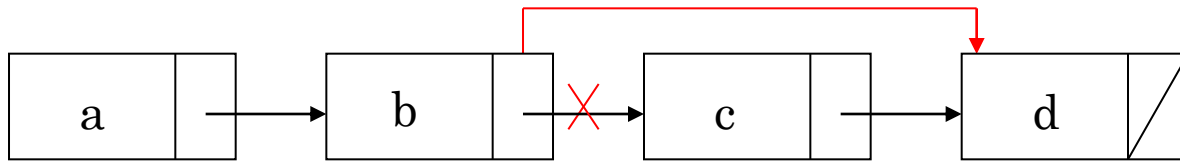
- Une liste chaînée est constituée d'un ensemble de cellules chaînées entre elles. Chaque cellule contient une information et l'adresse de la cellule suivante.
- C'est l'adresse de la première cellule qui détermine la liste



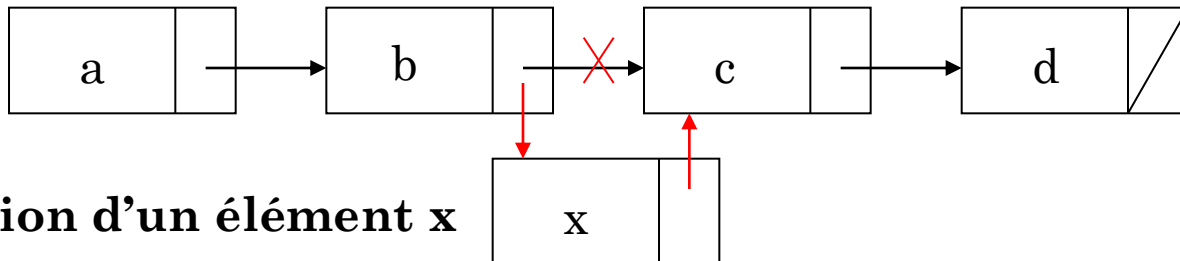
IMPLÉMENTATION DE L'INTERFACE LIST

○ Implantation par une liste chaînée

- Allocation dynamique à la demande. Les éléments sont dispersés en mémoire centrale.
- Pas de déplacement des éléments en cas d'insertion ou de suppression.



Suppression de l'élément c



Insertion d'un élément x



IMPLÉMENTATION DE L'INTERFACE LIST

- **Tableaux**
 - Accès aléatoire immédiat : $O(1)$.
 - Insertions et suppressions nécessitant un certain nombre de transferts de données : $O(n)$.
- **Listes chaînées**
 - Insertions et suppressions en $O(1)$.
 - Accès aléatoire en $O(n)$ car pas d'utilisation d'index.



IMPLÉMENTATION DE L'INTERFACE SET

- **Les classes concrètes implémentant Set<E>:**
 - **HashSet<E>**
 - Utilise une table de hachage définie avec une capacité initiale de 101.
 - **TreeSet<E>**
 - Utilise un arbre binaire de recherche équilibré (arbre rouge et noir).



IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- La structure de données table de hachage est **un tableau de taille N fixée contenant les éléments de l'ensemble.**
- La répartition des éléments de l'ensemble dans le tableau se fait en utilisant une fonction appelée **fonction de hachage.**



IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- Une fonction de hachage est une fonction h qui, pour chaque élément x de l'ensemble, calcule une valeur $h(x)$ comprise entre 0 et $N-1$ correspondant à un indice du tableau.

$$h : Ensemble \rightarrow [0..N-1]$$

$$x \quad \quad \quad 2$$

Table de hachage

| | |
|-----|---|
| 0 | |
| 1 | |
| 2 | X |
| | |
| | |
| N-1 | |

IMPLÉMENTATION DE L'INTERFACE SET

2- Les tables de hachage

- En utilisant la même fonction de hachage pour l'insertion et la recherche, l'exécution de ces opérations demande **se fait en temps constant**.
- En pratique, ce cas idéal n'existe pas du fait de **collisions**.

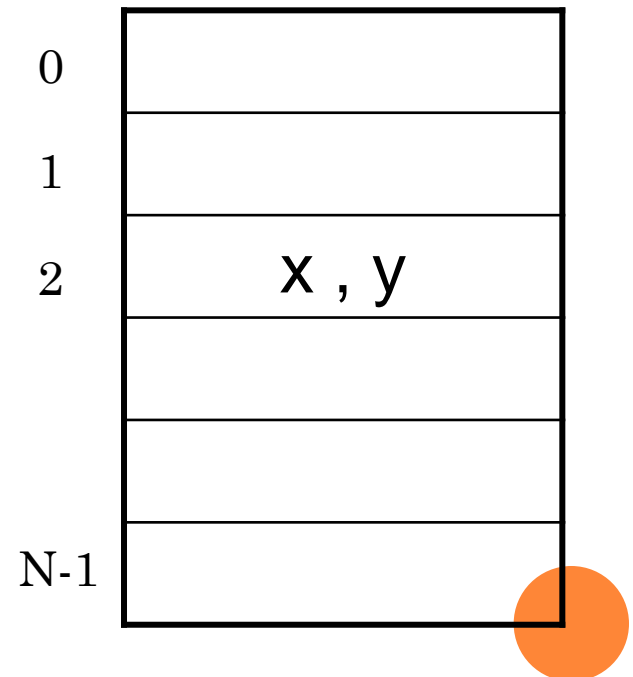


IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- **Collision** : pour deux éléments différents de l'ensemble, la fonction de hachage produit la même valeur.
- Exemple

$$h(x) = h(y) = 2 \quad \text{et} \quad x \neq y$$



IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- Une **bonne fonction de hachage** doit calculer rapidement une valeur mais aussi **diminuer le nombre de collisions**.
- **Exemple de fonction de hachage pour un ensemble dont les éléments sont des chaînes de caractères :**

$$h(\text{chaîne}) = ((\sum \text{codes de caractères}) \bmod N)$$

↑
taille du tableau

IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- Application à l'ensemble { **Jean**, **Jacques**, **Vincent**, **Lucien**, **Pierre**}

et une table de taille 10.

| chaîne | h(chaîne) |
|---------|-----------|
| Jean | 7 |
| Jacques | 5 |
| Vincent | 6 |
| Lucien | 9 |
| Pierre | 6 |

| | |
|---|-----------------|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | Jacques |
| 6 | Vincent, Pierre |
| 7 | Jean |
| 8 | |
| 9 | Lucien |

IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- En cas de collision, deux approches sont utilisées:
 - Gestion par hachage ouvert
 - Gestion par hachage fermé

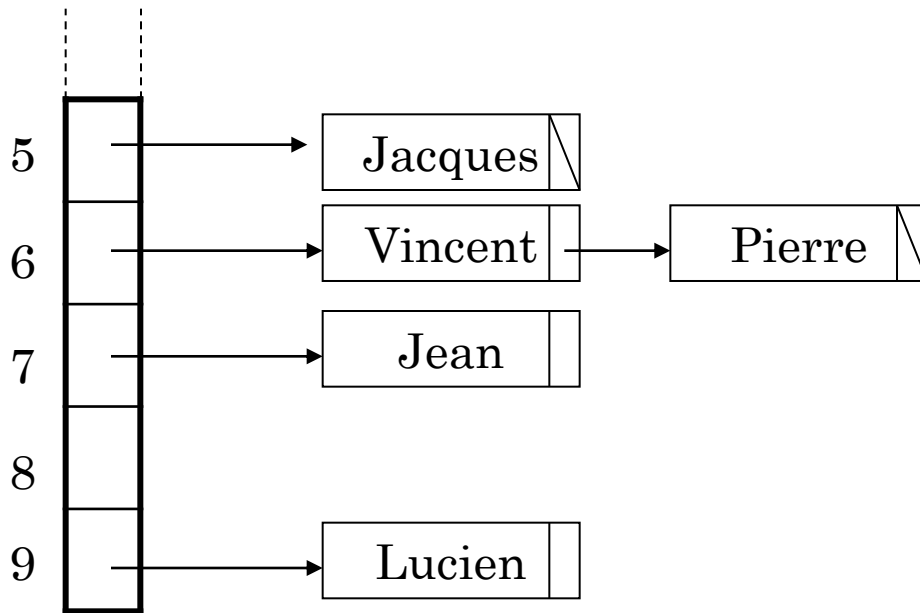


IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- **Gestion par hachage ouvert**

En cas de collision, le tableau est «ouvert» vers la droite.



Inconvénients :

Plus il y a de collisions

- Plus les listes sont longues

- Moins la recherche est efficace

IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- **Gestion par hachage fermé**
On cherche à remplacer l'élément en collision ailleurs dans le tableau en **appliquant une fonction de rehachage qui calcule un nouvel indice** et ceci jusqu'à ce que la nouvelle entrée soit libre.



IMPLÉMENTATION DE L'INTERFACE SET

Les tables de hachage

- **Gestion par hachage fermé**
 - Exemple de fonction de rehachage :
Fonction qui incrémente de 1 l'indice obtenu.
 - Ce choix conduit à une **dégradation des performances** dès que le taux de remplissage de la table dépasse 90%.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **HashSet<E>**

- Utilise les valeurs fournies par la méthode **hashCode** des objets pour les stocker dans la table de hachage.

Il faut redéfinir la méthode **hashCode** des éléments de type **E**.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **HashSet<E>**

- **Les collisions sont gérées par hachage ouvert.**
- Pour éviter au maximum les collisions, il faut que le nombre d'éléments de la table soit très inférieur à sa capacité.
- Le rapport entre ces deux paramètres est le **facteur de charge**.
- Une valeur plafond peut être définie au moment de la création de la table, par défaut : 75%.
- **La table est redimensionnée automatiquement** quand le facteur de charge atteint la valeur plafond spécifiée.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

- **TreeSet<E>** stocke ses objets **dans un arbre binaire de recherche équilibré** (arbre rouge et noir).
- **Moins efficace que les tables de hachage** pour les insertions, les suppressions et les recherches.
- **Plus efficace pour trier les données.** Nécessité de **définir un ordre** sur les éléments de l'ensemble.
- Implémente l'interface **NavigableSet<E>** et donc indirectement l'interface **SortedSet<E>**.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

- Comment définir une relation d'ordre total?
 - Définir une **relation d'ordre naturelle** sur E (natural ordering)
 - Définir une relation d'ordre totale sur E en implémentant l'interface **Comparable<T>**
 - Implémenter la méthode **int compareTo(T o)**
 - Redéfinir la méthode **equals** pour que la propriété d'antisymétrie de la relation d'ordre soit cohérente avec le critère d'égalité.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

- **Comment définir une relation d'ordre total?**
 - Définir une relation d'ordre **non liée à la classe E**
 - Définir une relation d'ordre en implémentant l'interface **Comparator<T>**
 - Implémenter la méthode

```
int compare (T o1, T o2)
```
 - Possibilité de définir plusieurs relations d'ordre sur une même classe.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

```
public class TreeSet<E> extends AbstractSet<E>  
implements NavigableSet<E> {
```

//constructeurs

```
public TreeSet ();  
public TreeSet (Comparator <? super E> comparator);
```



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

- **Exemple 1**

```
SortedSet<Integer> s = new TreeSet<>();  
//l'ordre utilisé est l'ordre naturel défini sur la  
classe Integer
```

```
s.add(1); s.add(5); s.add(2); s.add(3);  
Integer min = s.first(); //min contient 1  
Integer max = s.last(); //max contient 5  
System.out.println(s); //affichage [1, 2, 3, 5]
```



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

- **Exemple 2**

```
SortedSet<Integer> s = new TreeSet<>((i,j)->j-i);
```

**//l'ordre utilisé est l'ordre inverse de l'ordre naturel
défini sur la classe Integer**

```
s.add(1); s.add(5); s.add(2); s.add(3);
```

```
Integer min = s.first(); //min contient 5
```

```
Integer max = s.last(); //max contient 1
```

```
System.out.println(s); //affichage [5, 3, 2, 1]
```



IMPLÉMENTATION DE L'INTERFACE SET

- La classe **TreeSet<E>**

- Application à la gestion des comptes bancaires

On souhaite créer une classe **AgenceBancaire** permettant de gérer un **ensemble de comptes bancaires** : ajouter des comptes, rechercher des comptes, effectuer le traitement quotidien sur tous les comptes, afficher les comptes triés par leur numéro.

Rappel : la classe `CompteBancaire` implémente l'interface `Comparable<CompteBancaire>`

```
int compareTo (CompteBancaire c) {  
    return numero - c.numero;  
}
```

et redéfinit la méthode `equals`.



IMPLÉMENTATION DE L'INTERFACE SET

○ La classe **TreeSet<E>**

```
import java.util.*;
```

```
public class AgenceBancaire {
```

```
    private SortedSet<CompteBancaire> lesComptes;
```

```
    public AgenceBancaire () {  
        lesComptes = new TreeSet<>();  
    }
```



IMPLÉMENTATION DE L'INTERFACE SET

```
public boolean ajouterCompte (CompteBancaire c)
{
    return lesComptes.add(c);
}

public void supprimerCompte (CompteBancaire c) {
    lesComptes.remove(c);
}

public String toString() {
    return lesComptes.toString();
}

public CompteBancaire premierCompte() {
    return lesComptes.first();
}

public CompteBancaire dernierCompte() {
    return lesComptes.last();
}
```



IMPLÉMENTATION DE L'INTERFACE SET

```
public Set<CompteBancaire> rechercherComptes( String
    nom) {
    Set<CompteBancaire> comptes=
                                new HashSet<CompteBancaire>();
    Iterator<CompteBancaire> it = lesComptes.iterator();
    while(it.hasNext()){
        CompteBancaire c=it.next();
        if (nom.equalsIgnoreCase(c.getNom()))
            comptes.add(c);
    }
    return comptes;
}
```



IMPLÉMENTATION DE L'INTERFACE SET

```
public void traitementQuotidien() {  
    Iterator<CompteBancaire> it = lesComptes.iterator();  
    while (it.hasNext()){  
        CompteBancaire c= it.next();  
        c.traitementQuotidien();  
    }  
}  
} //fin de la classe
```



IMPLÉMENTATION DE L'INTERFACE SET

- **Différentes manières de parcourir la collection de comptes pour effectuer le traitement quotidien:**
 - **Première méthode : Associer un itérateur sur la collection**

```
Iterator<CompteBancaire> it =lesComptes.iterator();  
while (it.hasNext()) {  
    CompteBancaire c= it.next();  
    c.traitementQuotidien();  
}
```



IMPLÉMENTATION DE L'INTERFACE SET

- **Différentes manières de parcourir la collection de comptes pour effectuer le traitement quotidien:**
 - **Deuxième méthode :** Parcourir la collection par une boucle de type `for each` comme les tableaux

```
for (CompteBancaire c : lesComptes)  
    c.traitementQuotidien();
```



IMPLÉMENTATION DE L'INTERFACE SET

- Différentes manières de parcourir la collection de comptes pour effectuer le traitement quotidien:
 - **Troisième méthode** : Utiliser la méthode définie par défaut dans l'interface `Iterable` **pour appliquer une action à tous les éléments de la collection** : **default void**
`forEach(Consumer<? Super T> action)`

lesComptes.forEach(c->c.traitementQuotidien());

- Utilisation d'une **lambda expression** dont le type cible est l'interface fonctionnelle **Consumer** définie dans le package `java.util.function` de l'API standard.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept (T t);
}
```



IMPLÉMENTATION DE L'INTERFACE MAP

- La classe abstraite **AbstractMap<K, V>** implémente partiellement l'interface `Map<K, V>`.
- Il existe deux classes concrètes principales:
 - **HashMap<K, V>**
 - **TreeMap<K, V>**
- Deux autres classes concrètes:
 - **WeakHashMap<K, V>** (gère les clés qui ne sont plus utilisées).
 - **LinkedHashMap<K, V>** (garde une trace de l'ordre d'insertion).



IMPLÉMENTATION DE L'INTERFACE MAP

- La classe **HashMap<K, V>**

Comme `HashSet<K, V>`, `HashMap<K, V>` implémente l'interface `Map<K, V>` à partir d'une **table de hachage**.



IMPLÉMENTATION DE L'INTERFACE MAP

○ La classe **TreeMap<K, V>**

- Implémente l'interface **NavigableMap<K, V>**.
- **TreeMap<K, V>** stocke ses objets **dans un arbre binaire de recherche équilibré** (arbre rouge et noir).



IMPLÉMENTATION DE L'INTERFACE MAP

- **La classe `TreeMap<K, V>`**

Application à la gestion des comptes bancaires:

Etant donné qu'un numéro identifie un compte, on souhaite gérer les couples (numéro, compte) de manière à pouvoir rechercher ou supprimer un compte à partir de son numéro.



IMPLÉMENTATION DE L'INTERFACE MAP

○ La classe **TreeMap**

```
import java.util.*;
```

```
public class AgenceBancaireBis {
```

```
    private SortedMap<Integer, CompteBancaire> lesComptes;
```

```
    public AgenceBancaireBis() {
```

```
        lesComptes = new TreeMap<Integer, CompteBancaire>();
```

```
    }
```




IMPLÉMENTATION DE L'INTERFACE MAP

```
public void ajouterCompte(CompteBancaire c) {  
    lesComptes.putIfAbsent(c.getNumero(), c);  
}  
    //rechercher un compte à partir du numero  
public CompteBancaire rechercherCompte( int numero) {  
    return lesComptes.get(numero);  
}  
public void supprimerCompte(int numero) {  
    lesComptes.remove(numero);  
}  
public void traitementQuotidien() {  
    Collection<CompteBancaire> comptes = lesComptes.values();  
    for(CompteBancaire c : comptes)  
        c.traitementQuotidien();  
}
```



IMPLÉMENTATION DE L'INTERFACE MAP

```
public String toString() {  
    String s= "";  
    Set<Map.Entry<K,V>> lesEntrees= lesComptes.entrySet() ;  
    Iterator<Map.Entry<K,V>> it=lesEntrees.iterator() ;  
    while (it.hasNext()) {  
        Map.Entry<K,V> uneEntree = it.next();  
        K key = uneEntree.getKey();  
        V value = uneEntree.getValue();  
        s = s+ key.toString() + " - " +value.toString()+  
            "\n";  
    }  
    return s;  
}  
}
```



CLASS UTILITAIRE COLLECTIONS

○ La classe Collections

- Propose des services qui s'appliquent à des collections et/ou qui retournent des collections
 - Création de collections constantes
 - Recherche d'éléments dans une collection
 - Ajout et remplacement d'éléments dans une collection
 - Modification de l'ordre des éléments dans une collection.



CLASS UTILITAIRE COLLECTIONS

○ Créations de collections constantes

○ Collections vides constantes

- Méthodes statiques qui retournent une collection vide non modifiable: `emptyList()`, `emptySet()`...

○ Exemple:

```
public static final <T> List <T> emptyList()
```

```
List<String> l = Collections.emptyList();
```



CLASS UTILITAIRE COLLECTIONS

○ Créations de collections constantes

○ Singletons constants

- Méthodes statiques qui renvoient une collection qui ne contient qu'un seul élément :
singleton,
singletonList, singletonMap...

○ Exemple:

```
public static <T> Set <T> singleton(T o)
```

```
Set<String> l =  
    Collections.singleton("bonjour");
```



CLASS UTILITAIRE COLLECTIONS

○ Créations de collections constantes

○ Copies constantes de collections

- Méthodes statiques qui permettent de récupérer une vue constante d'une collection fournie en paramètre:

`unmodifiableCollection,`
`unmodifiableList, unmodifiableSet...`

○ Exemple:

```
public static <T> List <T>  
    unmodifiableList(List<? extends T> list)
```



CLASS UTILITAIRE COLLECTIONS

○ Recherche d'éléments dans une collection

- `public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)`
- `public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)`
- `public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)`
- `public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)`



CLASS UTILITAIRE COLLECTIONS

○ Ajout d'éléments

- Ajout d'une suite d'éléments dans une collection passée en paramètre:

```
public static <T> boolean addAll(Collection<?  
    super T> c, T... elements)
```

Exemples :

```
Collection<Integer> c = new TreeSet<>();
```

```
Collections.addAll(c, 1, -5, 8, 1, 6, -5);
```

```
//élimination des doublons
```

```
//c contient dans l'ordre {-5, 1, 6, 8}
```



CLASS UTILITAIRE COLLECTIONS

○ Remplacement d'éléments

- Remplacement de toutes les occurrences d'un élément passé en paramètre par un nouvel élément:

```
public  
    static <T> boolean replaceAll(List<T> list,  
    T oldVal, T newVal)
```



CLASS UTILITAIRE COLLECTIONS

○ Modification de l'ordre des éléments

- Exemples:

```
public static void reverse(List<?> list)
```

```
public static void shuffle(List<?> list)
```

```
public static void sort(List<?> list)
```

```
public static void swap(List<?> list, int  
    j, int j)
```

