Module de mise à niveau en Java
Cours 3

Laure KAHLEM
laure.kahlem@univ-orleans.fr

Références

- Kathy Sierra et Bert Bates. Java Tête la première. O'Reilly, 2007.
- Mickaël Kerboeuf. Fondements de la programmation orientée objet avec Java8. Ellipses, 2016
- David Flanagan. Java in a Nutshell. Manuel de référence. O'Reilly, 2002.
- Cyrille Herby. Apprenez à programmer en JAVA. Broché, 2012.
- José Paumard. « Java en ligne » [en ligne]. http://blog.paumard.org/cours/java/ (Consulté le 15/07/2015).

- Les types de données abstraits
- La bibliothèque des collections Java



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:`** `=>Liste` (crée une liste vide)
    - **`Ajouter:`** `Element × Liste => Liste`
    - **`AjouterPos:`** `Element × Liste × Place => 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:

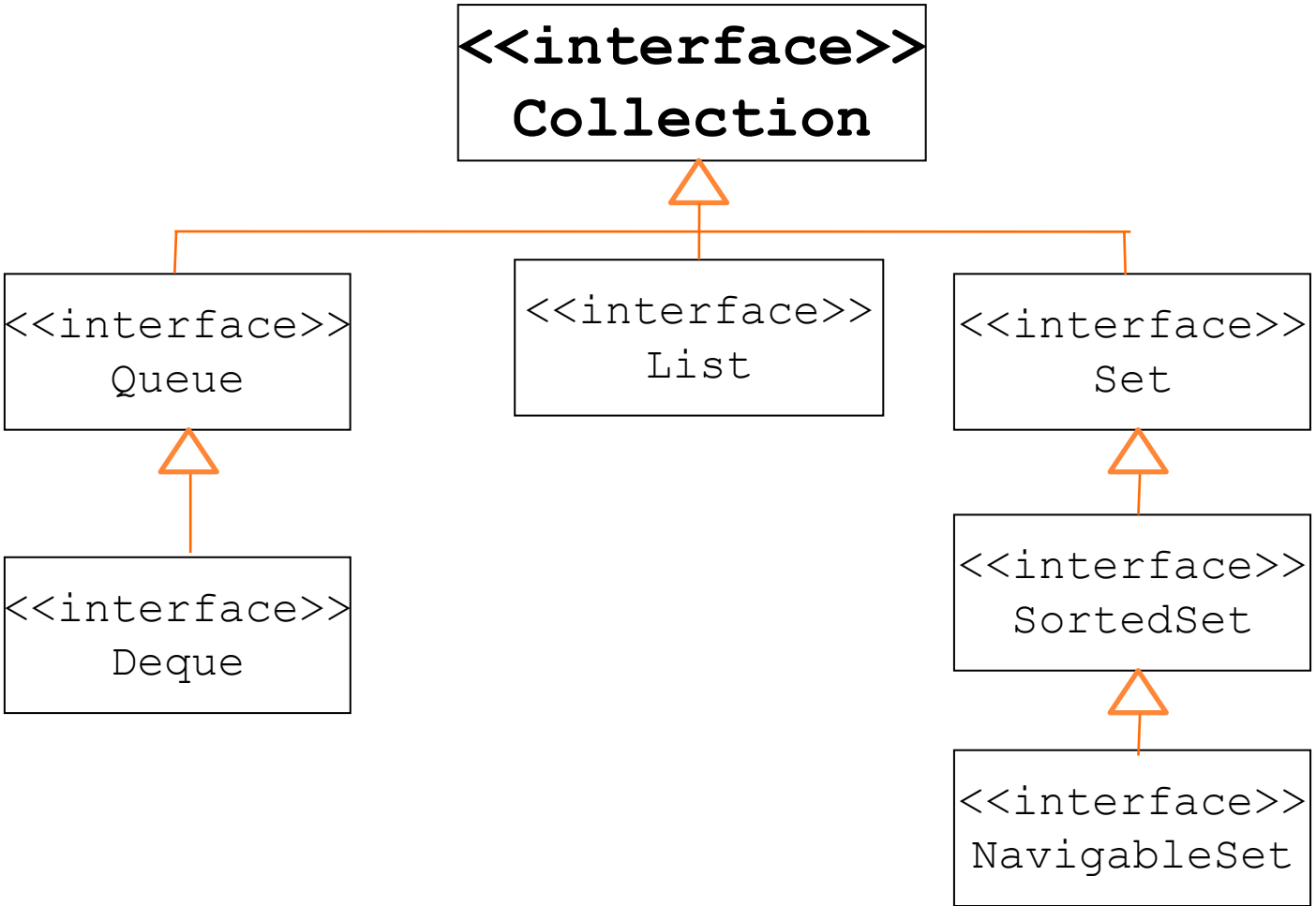
1. **Gestion d'un annuaire :**
   - Insertion de personnes
   - Suppression de personnes
   - Recherche de personnes suivant leur nom
2. **Gestion d'une file d'attente:**
   - Ajouter des personnes en fin de file
   - Traiter la personne en début de file



La bibliothèque des collections Java

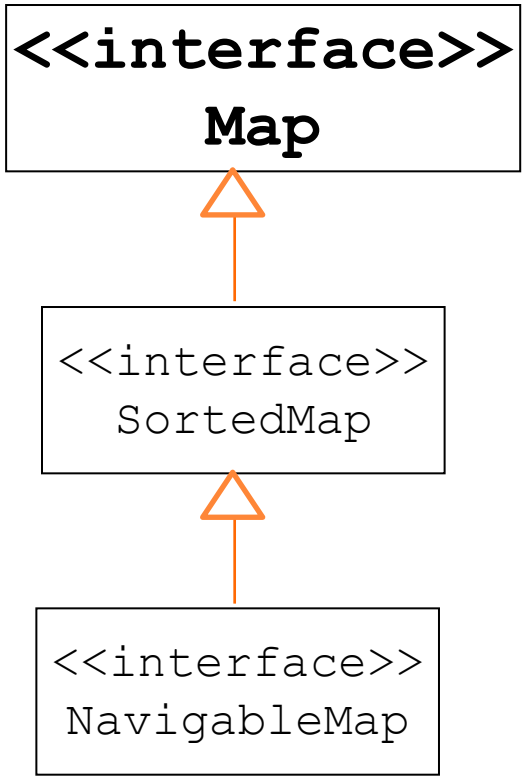
- **Les interfaces**
  - Cette bibliothèque est composée d'**interfaces** définies dans le paquetage **`java.util`**.
  - Il existe deux interfaces fondamentales pour les conteneurs:
    - **`Collection`**
    - **`Map`**

La bibliothèque des collections Java



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)
}
```



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 `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 `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

- **Le protocole d'itération**

Comment parcourir les éléments d'une collection sans révéler sa structure interne?

Solution : extraire de la collection les fonctions en charge des accès et parcours pour les placer dans un objet **itérateur**.

La bibliothèque des collections Java

Le protocole d'itération

Un **itérateur** permet de

- **Parcourir séquentiellemen**t tous les éléments de la collection.
- **Faire des mises à jour** de la collection.

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

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

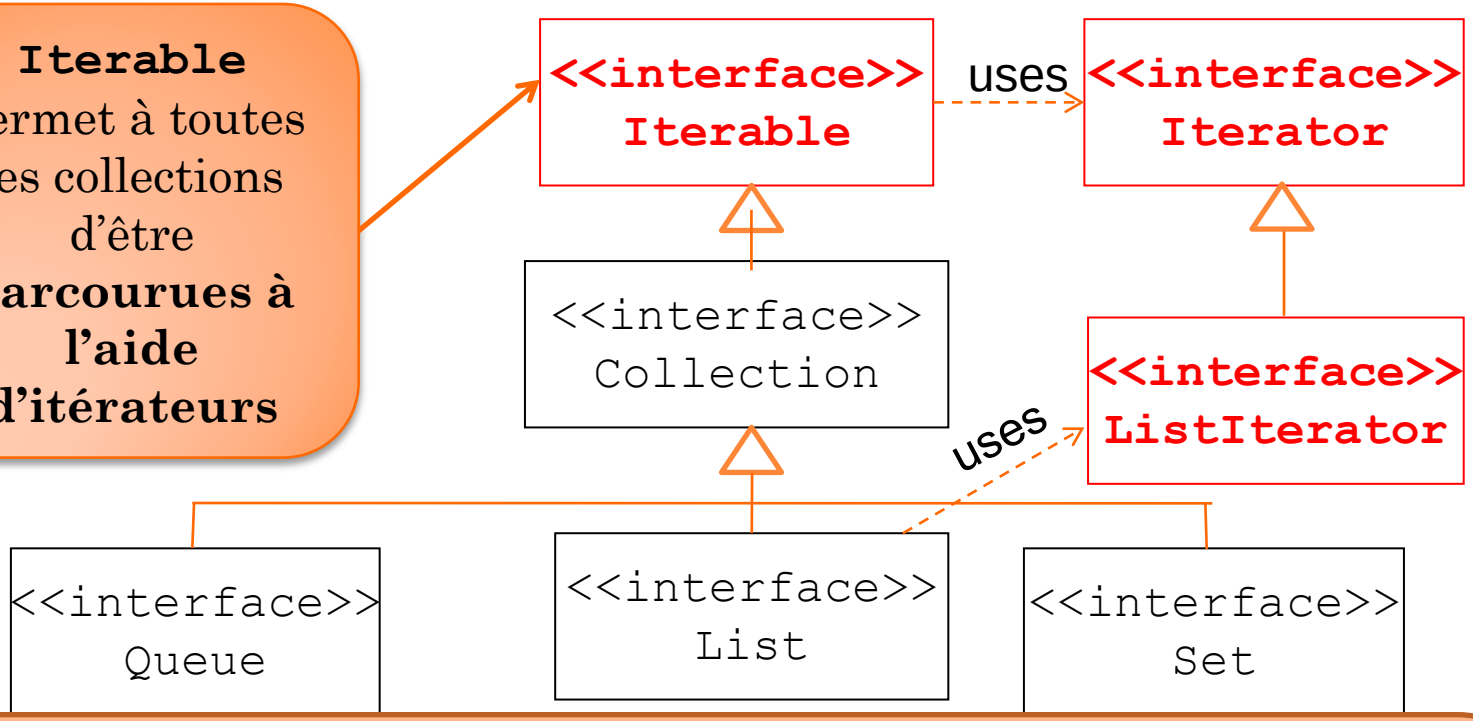
```
public interface Iterable<T> {
  Iterator<T> iterator();

  default void forEach(Consumer<? Super T>
    action){/*…*/}
}
```

Permet d'**associer un itérateur à une collection**.

La bibliothèque des collections Java

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



- **Toutes les collections fournissent la méthode**
  `public Iterator<E> iterator();`
- **Les listes fournissent les méthodes**
  - `ListIterator<E> listIterator ();`
  - `ListIterator<E> listIterator (int index);`

La bibliothèque des collections Java

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

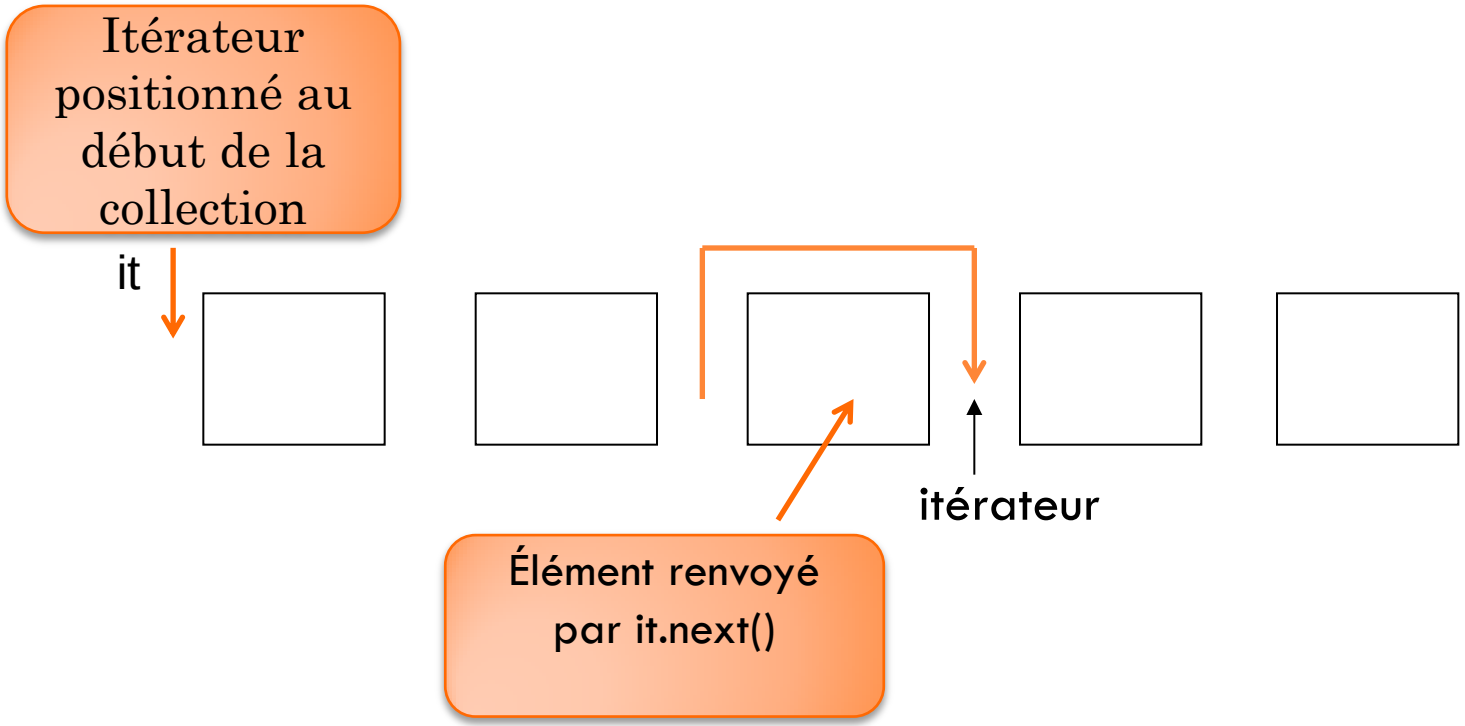
```
public E next() throws NoSuchElementException
```

    - 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 `Map`**
    - Une table est un conteneur qui permet un accès direct par type d'index.
    - Un 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 un ensemble de définitions 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

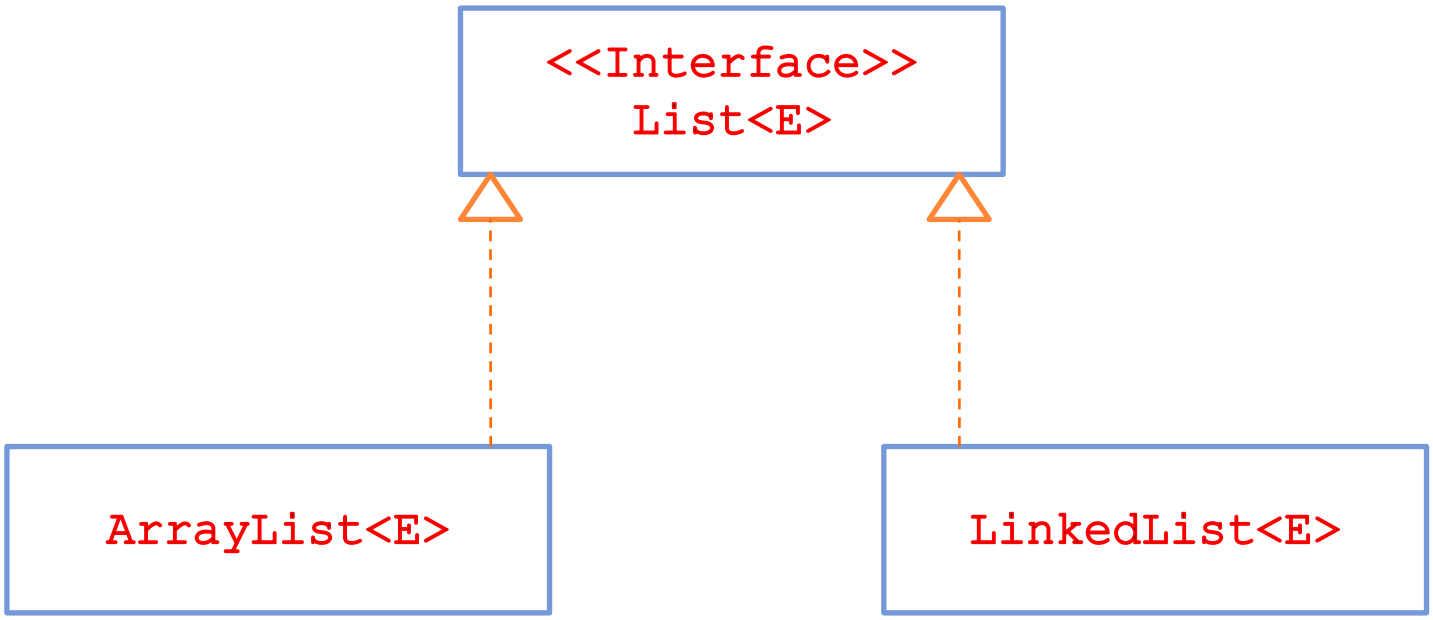
- **Les interfaces**
  - **Interfaces `SortedMap` et `NavigableMap`**

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



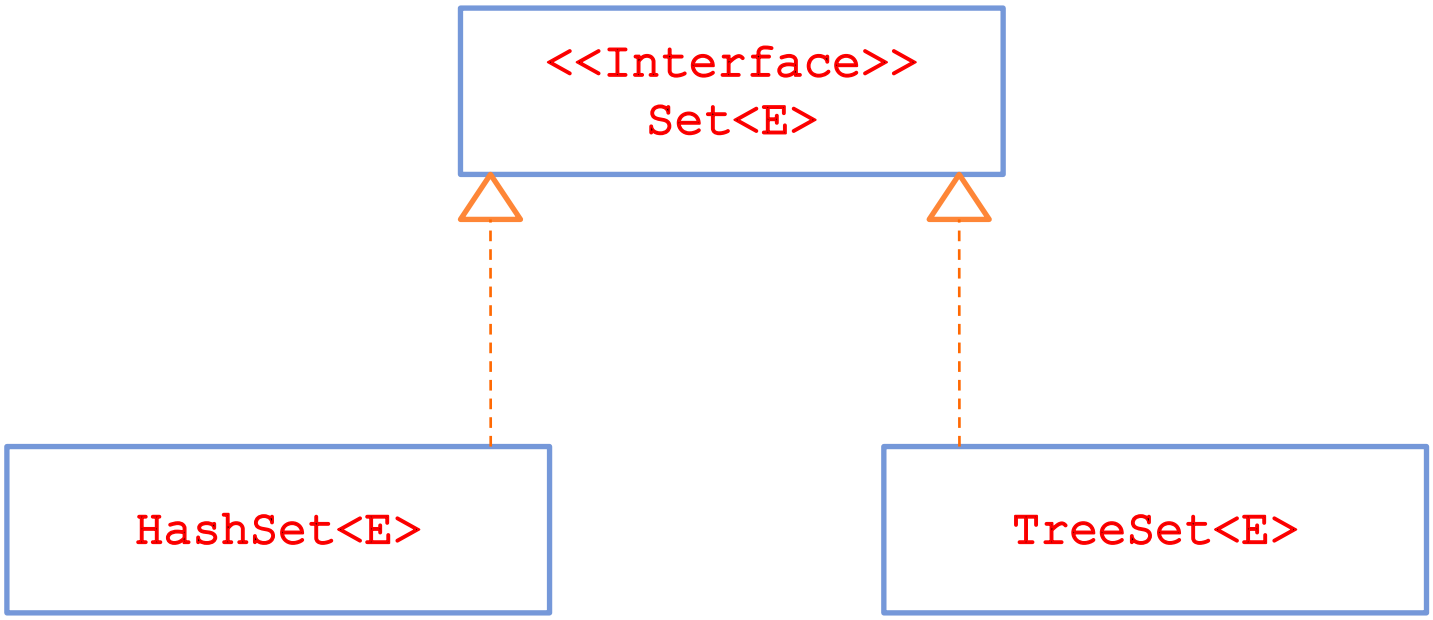
La bibliothèque des collections Java

- **Les classes concrètes**



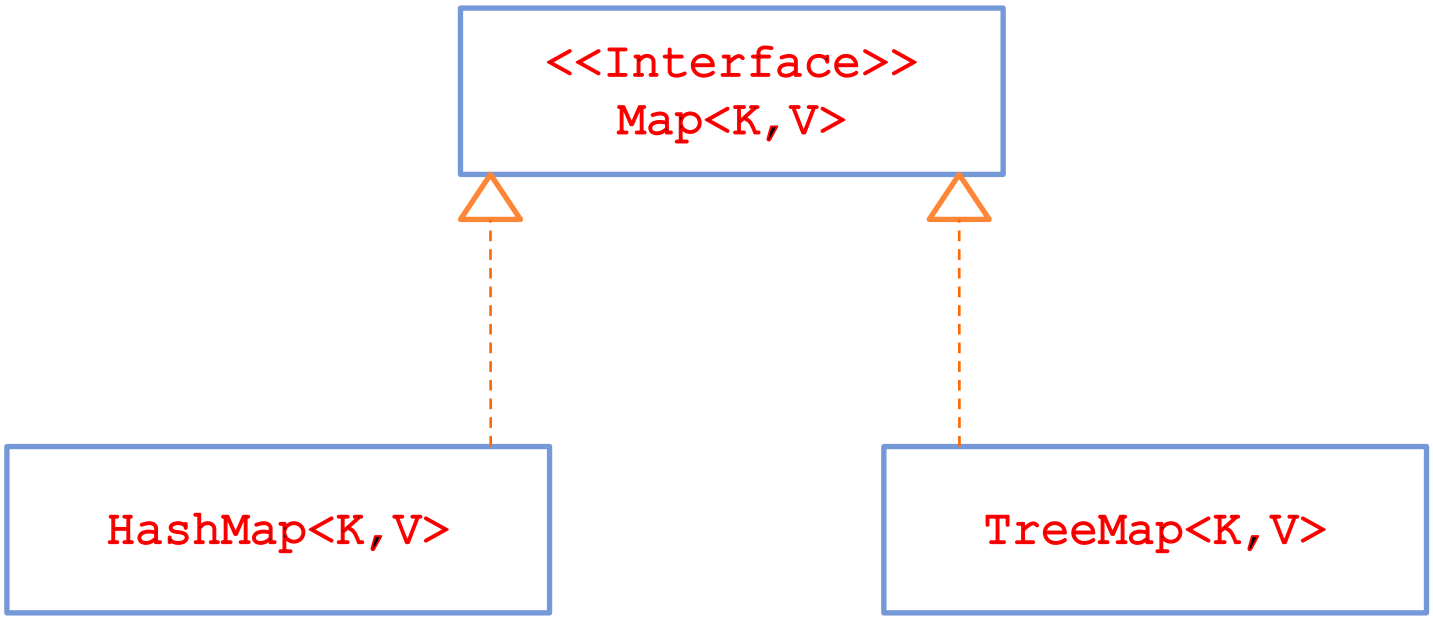
La bibliothèque des collections Java

- **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 relogeable .**
  - **`LinkedList<E>`**
    - utilise **une liste chaînée.**



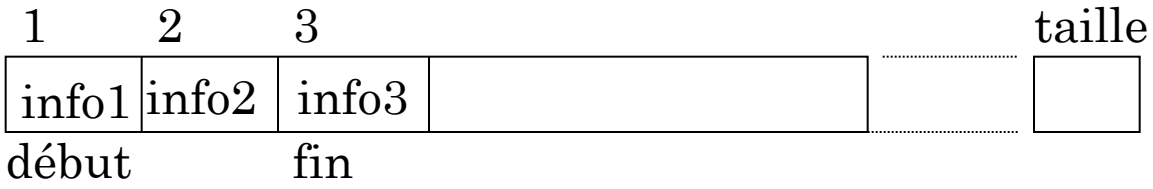
Implémentation de l'interface List

- **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

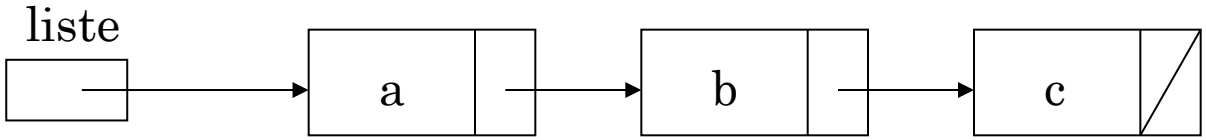
- **Implémentation par un tableau**
  - Représentation



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

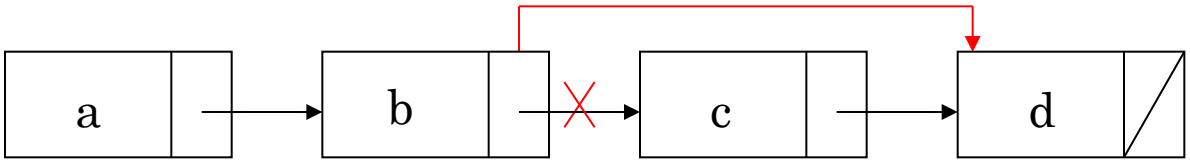
Implémentation de l'interface List

- **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

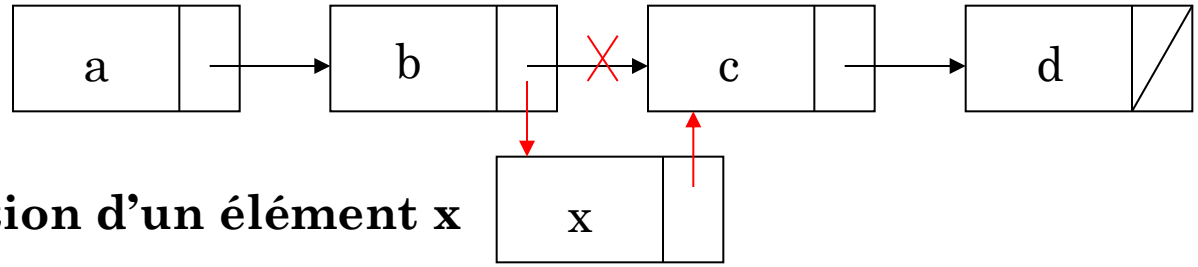


Implémentation de l'interface List

- **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>`**
    - Uilise **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**.



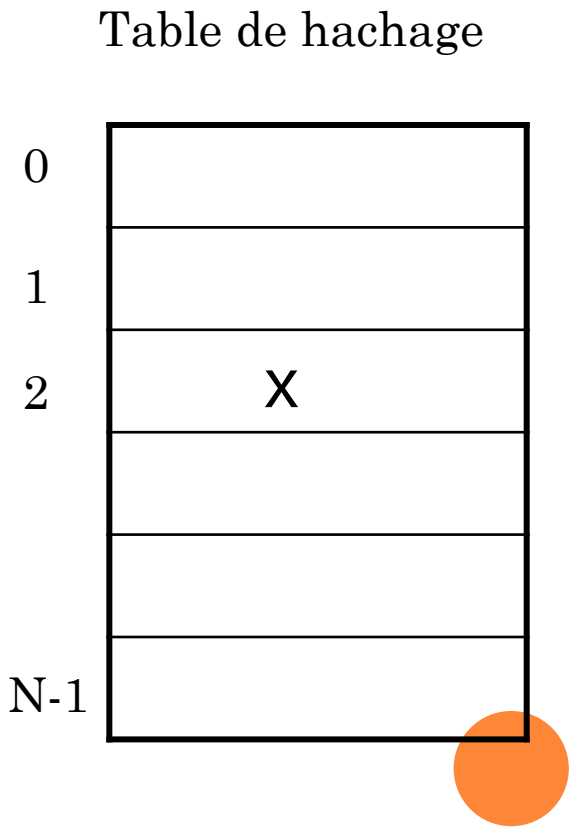
Implémentation de l'interface Set

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.

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

$$x \qquad\qquad 2$$



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**.

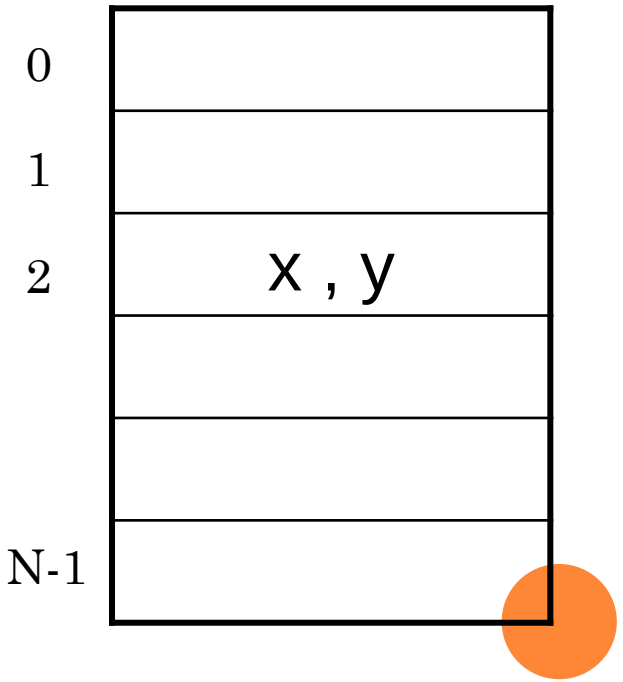


Implémentation de l'interface Set

Les tables de hachage

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

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



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 :**

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

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

- **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>`**
  - 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)
```

`Rappel` : **`Consumer`** est une interface fonctionnelle

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

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

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
  i, int j)
```

