L3 SRI - COO

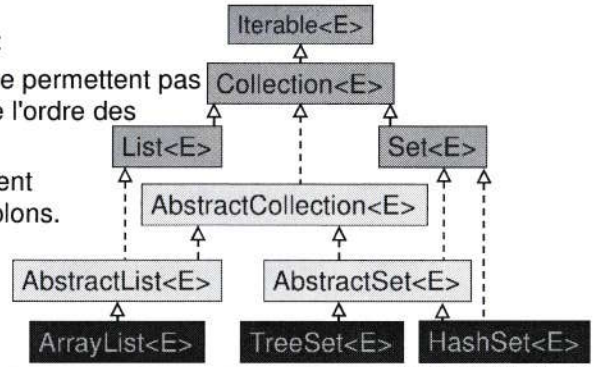
Cours 5 : Comparaisons d'objets / Les ensembles / vue sur des collections

Comparaison d'Objet
(ordre naturel, ordre imposé)
Collection TreeSet

Auteur : CHAUDET Christelle

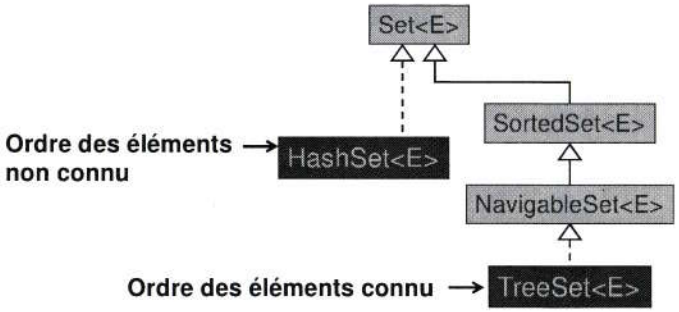
Introduction

- Les listes et les tableaux :
  - permettent de spécifier l'ordre des éléments,
  - peuvent contenir des doublons.
- Les ensembles :
  - par défaut ne permettent pas de connaitre l'ordre des éléments,
  - ne contiennent pas de doublons.



Les ensembles

- L'interface Set hérite de l'interface Collection, les méthodes add et addAll tiennent compte de cette caractéristique (absence de doublons).
- Implémentation concrète de Set<E>



Method Summary — Set<E>

boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specif
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

java.util
Interface Set<E>

Type Parameters:
E - the type of elements maintained by this set

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
NavigableSet<E>, SortedSet<E>

All Known Implementing Classes:
AbstractSet, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

: Opération Optimisée

Introduction

- La comparaison d'objets:
  - Méthode equals vue dans le cours précédent :
    - **méthode de la classe Object** donc héritée par l'ensemble des classes,
    - retourne vrai si deux objets possèdent la même référence mémoire et sont donc le même objet,
    - destinée à être redéfinie dans la classe à comparer afin de comparer deux objets sur leur état et non sur leur référence.
  - Dans ce cours nous voulons ordonner des objets. Pour cela nous verrons deux techniques :
    - L'ordre naturel (défini dans la classe à comparer)
    - L'ordre imposé (défini en dehors de la classe)

Ordre naturel : interface Comparable<T> (1/4)

Method Summary

int	**compareTo**(T o) Compares this object with the specified object for order.

- a.compareTo(b) retourne :
  - 0 si a = b,
  - un nombre négatif si a<b,

```
assert ((new Integer(1)).compareTo(new Integer(2)))<0;
```

  - un nombre positif si a>b.

  Le nombre retourné n'a pas d'importance, seul son signe compte.
- Les enveloppeurs (Integer, Float ...) & la classe String implémentent cette interface.
- **Attention** il n'existe aucune implémentation par défaut dans la classe Object => Pour vos propres objets il faut définir un ordre en implémentant l'interface Comparable.

Ordre naturel : interface Comparable<T> (2/4)

```
public class Gaulois implements Comparable<Gaulois> {
  private String nom; private int age;

  public Gaulois(String nom, int age) {
     this.nom = nom; this.age = age;
  }
  public boolean equals(Object obj) {
    if(obj instanceof Gaulois) {
      Gaulois gaulois = (Gaulois) obj;
      return nom.equals(gaulois.nom);
    } return false;
  }
  public int compareTo(Gaulois gauloisToCompare) {
    return this.nom.compareTo(gauloisToCompare.nom);
  }
}
```

Classement selon l'ordre alphabétique des noms des gaulois

Ordre naturel : interface Comparable<T> (3/4)

- Testons la méthode compareTo :

```
Gaulois asterix = new Gaulois("Astérix", 35);
Gaulois obelix = new Gaulois ("Obélix", 30);
System.out.println(asterix.compareTo(obelix));
```

- Nous obtenons -14 c'est-à-dire un nombre négatif,
  asterix.compareTo(obelix) < 0
  asterix < obelix
  ⟹ suivant l'ordre alphabétique de l'attribut nom l'objet asterix est positionné avant l'objet obelix.

Ordre naturel : interface Comparable<T> (4/4)

- Attention les méthodes equals & compareTo doivent être cohérente !

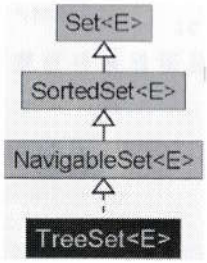
```
Gaulois asterix1 = new Gaulois("Astérix", 35);
Gaulois asterix2 = new Gaulois("Astérix", 35);
System.out.println(asterix1.equals(asterix2));
System.out.println(asterix1.compareTo(asterix2));
```

affichage

```
true
0
```

- Faites attention :
  - La méthode equals retourne true si et seulement si votre méthode compareTo retourne 0.
  - Baser l'égalité et votre comparaison d'objets sur les mêmes attributs.

Les arbres



- TreeSet : implémentation concrète des ensembles comme une structure de donnée en arbre, triée soit dans un ordre naturel des éléments soit dans un ordre imposé. (arbre binaire de recherche)

```
NavigableSet<String> ensemble = new TreeSet<>();
ensemble.add("Odralfabétix");
ensemble.add("Astérix");
ensemble.add("Obélix");
System.out.println(ensemble);
// [Astérix, Obélix, Ordralfabétix]
```

⟹ Les chaînes sont affichées dans leur ordre naturel c'est-à-dire dans l'ordre alphabétique

Limite de l'interface Comparable

- L'interface ne peut être implémentée qu'une seule fois.
- Problèmes
  - l'objet doit répondre à différents critères dans des collections différentes.

    Exemple : La classe Gaulois ne peut implémenter Comparable qu'une seule fois => impossibilité de trier les objets de cette classe selon
    - son âge en ordre croissant,
    - son nom par ordre alphabétique puis son âge.
  - le concepteur de la classe n'a pas pris le soin d'implémenter l'interface comparable.
- Solution : l'ordre imposé qui est une autre méthode de comparaison d'objet.

Ordre imposé : interface Comparator<T>

- Vous pouvez imposer un ordre à des objets que vous n'avez pas créé ou dont l'ordre naturel ne vous convient pas.
- L'interface Comparator<T> possède 2 méthodes.

Method Summary

int	**compare**(T o1, T o2) Compares its two arguments for order.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this comparator.

Méthode compare (1/3)

```
public class GauloisComparator
                          implements Comparator<Gaulois> {
    public int compare( Gaulois gaulois1,
                        Gaulois gaulois2) {
        Integer ageGaulois1 = gaulois1.getAge();
        Integer ageGaulois2 = gaulois2.getAge();
        return ageGaulois1.compareTo(ageGaulois2);
    }
}
```

```
Gaulois asterix = new Gaulois("Astérix",35);
Gaulois obelix = new Gaulois("Obélix",30);
GauloisComparator comparator = new GauloisComparator();
System.out.println(comparator.compare(asterix, obelix));
```

affichage

1 > 0 ⟹ **asterix > obelix**

Méthode compare (2/3)

- Les classes de type **Comparator** peuvent être utilisées dans les collections qui utilisent un tri imposé.
- Exemple : La classe **GauloisComparator** peut être utilisée dans un **TreeSet (implemente NavigableSet)**

TreeSet(Comparator<? super E> c)
Constructs a new, empty set, sorted according to the specified comparator.

```
NavigableSet<Gaulois> gauloisOrdreImpose = new TreeSet<>(new GauloisComparator());
gauloisOrdreImpose.addAll(ensemble);
System.out.println("Tri par ordre croissant des âges");
for (Iterator<Gaulois> it = gauloisOrdreImpose.iterator(); it.hasNext();) {
    Gaulois gaulois = it.next();
    System.out.println(gaulois);
}
```

```
Tri par ordre croissant des âges
Obélix à 30 ans
Astérix à 35 ans
Bonemine à 36 ans
Abraracourcix à 40 ans
Panoramix à 90 ans
```

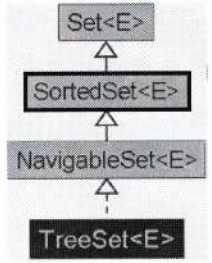

Méthode compare (3/3)

- La classe **GauloisComparator** n'a pas forcement lieu d'exister (si elle n'est utilisée que dans une seule collection). Elle peut être créée de manière **anonyme** (**classe interne anonyme**) dans les collections qui utilisent un tri imposé.
- Exemple :

TreeSet(Comparator<? super E> c)
Constructs a new, empty set, sorted according to the specified comparator.

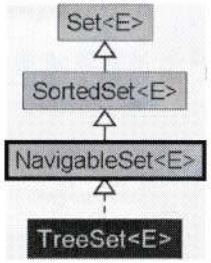
```
NavigableSet<Gaulois> gauloisOrdreImpose =
  new TreeSet<>(
    new Comparator<Gaulois>() {
      public int compare( Gaulois gaulois1, Gaulois gaulois2) {
        Integer ageGaulois1 = gaulois1.getAge();
        Integer ageGaulois2 = gaulois2.getAge();
        return ageGaulois1.compareTo(ageGaulois2);
      }
    }
  );
```

SortedSet<E>



- SortedSet<E>
  - Ajoute au contrat de Set une garantie que son itérateur visitera les éléments dans l'ordre croissant.
  - Utilisation de l'ordre naturel (implémentation de l'interface Comparable) ou de l'ordre imposé (implémentation de l'interface Comparator)
- Méthodes définies par SortedSet
  - Obtenir le premier / le dernier éléments : E first() / E last()
  - Obtenir le comparateur : Comparator<? super E> comparator()
- Obtenir les vues bornées
  - SortedSet<E> subSet(E fromElement, E toElement)
  - SortedSet<E> headSet(E toElement)
  - SortedSet<E> tailSet(E from Element)

NavigableSet<E> (1/3)



- NavigableSet<E> (depuis Java 6) doit être préféré à SortedSet<E>
- Méthodes :
  - Obtenir et retirer le premier et le dernier élément : ■ E pollFirst() ■ E pollLast()
  - Obtenir des vues bornées (améliore les méthodes de SortedSet en incluant ou non les bornes)
    - NavigableSet<E> subSet(
      E fromElement, boolean fromInclusive,
      E toElement, boolean toInclusive)
    - SortedSet<E> headSet(E toElement, boolean inclusive)
    - SortedSet<E> tailSet(E fromElement , boolean inclusive)

NavigableSet<E> (2/3)

- Méthodes (suite) :
  - Obtenir les correspondances les plus proches
    - E ceiling(E e)
      //retourne le plus petit élément de cet ensemble qui
      //soit supérieur ou égal à e, ou null s'il n'existe pas
    - E floor(E e) //le plus grand élément
      //inférieur ou égal à e
    - E higher(E e) //strictement supérieur à e
    - E lower(E e) //strictement inférieur à e

```
NavigableSet<Integer> ensemble = new TreeSet<>();
Collections.addAll(ensemble, 1,2,4,5,6,7);
System.out.println(ensemble.ceiling(3));   //4
System.out.println(ensemble.ceiling(5));   //5
System.out.println(ensemble.floor(3));     //2
System.out.println(ensemble.floor(5));     //5
System.out.println(ensemble.higher(5));    //6
System.out.println(ensemble.lower(5));     //4
```

NavigableSet<E> (2/3)

- Méthodes (suite) :
  - Obtenir les correspondances les plus proches
    - E ceiling(E e)
      //retourne le plus petit élément de cet ensemble qui
      //soit supérieur ou égal à e, ou null s'il n'existe pas
    - E floor(E e) //le plus grand élément
      //inférieur ou égal à e
    - E higher(E e) //strictement supérieur à e
    - E lower(E e) //strictement inférieur à e
  - Parcourir l'ensemble en ordre inverse
    - NavigableSet<E> descendingSet()
      //retourne une vue inversée des éléments de l'ensemble
    - Iterator<E> descendingIterator()
      //retourne un itérateur en ordre inverse

Les objets « dégénérés »

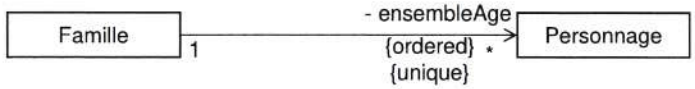
- On appelle des objets « dégénérés », des objets qui :
  - n'ont de sens d'exister que pour une durée très limitée,
  - dont seuls quelques attributs sont intéressants.

- L'interface NavigableSet permet de retourner un élément selon un objet de référence.
  - E ceiling(E e)
    //retourne le plus petit élément de cet ensemble qui
    //soit supérieur ou égal à e, ou null s'il n'existe pas

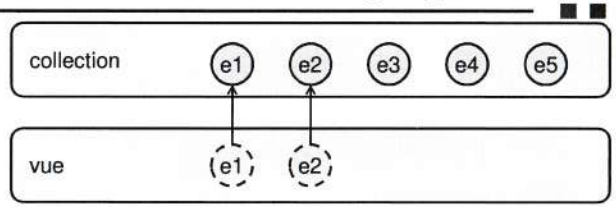
  Or tous les paramètres de l'objet ne sont pas forcément nécessaires pour obtenir l'élément désiré.
  On utilise donc un objet dégénéré. **Objet dégénéré**
  - Gaulois gaulois = ensemble.ceiling(new Gaulois("O", 1));
    System.*out*.println(gaulois); ⟹ `Obélix à 30 ans`

TreeSet dans UML

- Un TreeSet hérite de l'interface :
  - NavigableSet, ce qui signifie que la collection est ordonnée -> contrainte {ordered}
  - Set, ce qui signifie que la collection ne peut pas comporter de doublons -> contrainte {unique}
- Représentation UML



Les vues (1/3)



- Une vue sur une collection ne contient pas d'élément mais permet de retourner des éléments d'une collection selon un critère.
- Exemple méthode *headSet* de la classe **NavigableSet** :

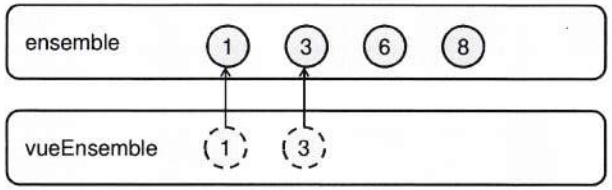
`NavigableSet<E>` `headSet(E toElement, boolean inclusive)`
Returns a view of the portion of this set whose elements are less than (or equal to, if `inclusive` is true) `toElement`.

Les vues (2/3)

- Exemple méthode *headSet* de la classe **NavigableSet** :

```
NavigableSet<Integer> ensemble = new TreeSet<>();
ensemble.add(1);
ensemble.add(3);
ensemble.add(6);
ensemble.add(8);

NavigableSet<Integer> vueEnsemble = ensemble.headSet(3, true);
System.out.println(vueEnsemble);
```



Les vues (3/3)

- Une vue ne contient pas d'élément, elle se met donc automatiquement à jour quand la collection est modifiée.

```
NavigableSet<Integer> ensemble = new TreeSet<>();
ensemble.add(1);
ensemble.add(3);
ensemble.add(6);
ensemble.add(8);

NavigableSet<Integer> vueEnsemble = ensemble.headSet(3, true);
System.out.println(vueEnsemble);
ensemble.add(2);
System.out.println(vueEnsemble);
```