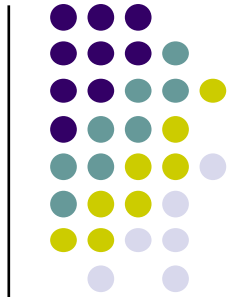
Les collections – partie 2



Licence informatique

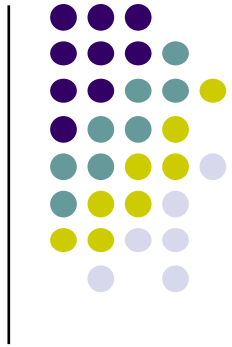
POA - Prog. Objet Avancée

Partie 2

Françoise Greffier

(+ Pierre-Alain Masson)

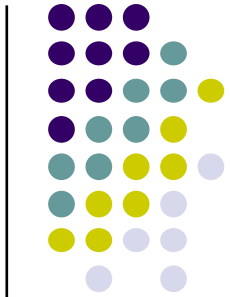
Deuxième catégorie de collection



Nous avons vu les listes.

Examinons
l'autre catégorie de collection
appelée « ensemble »

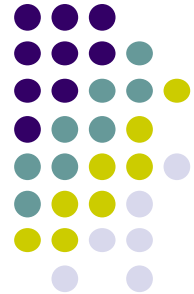
Conception des ensembles



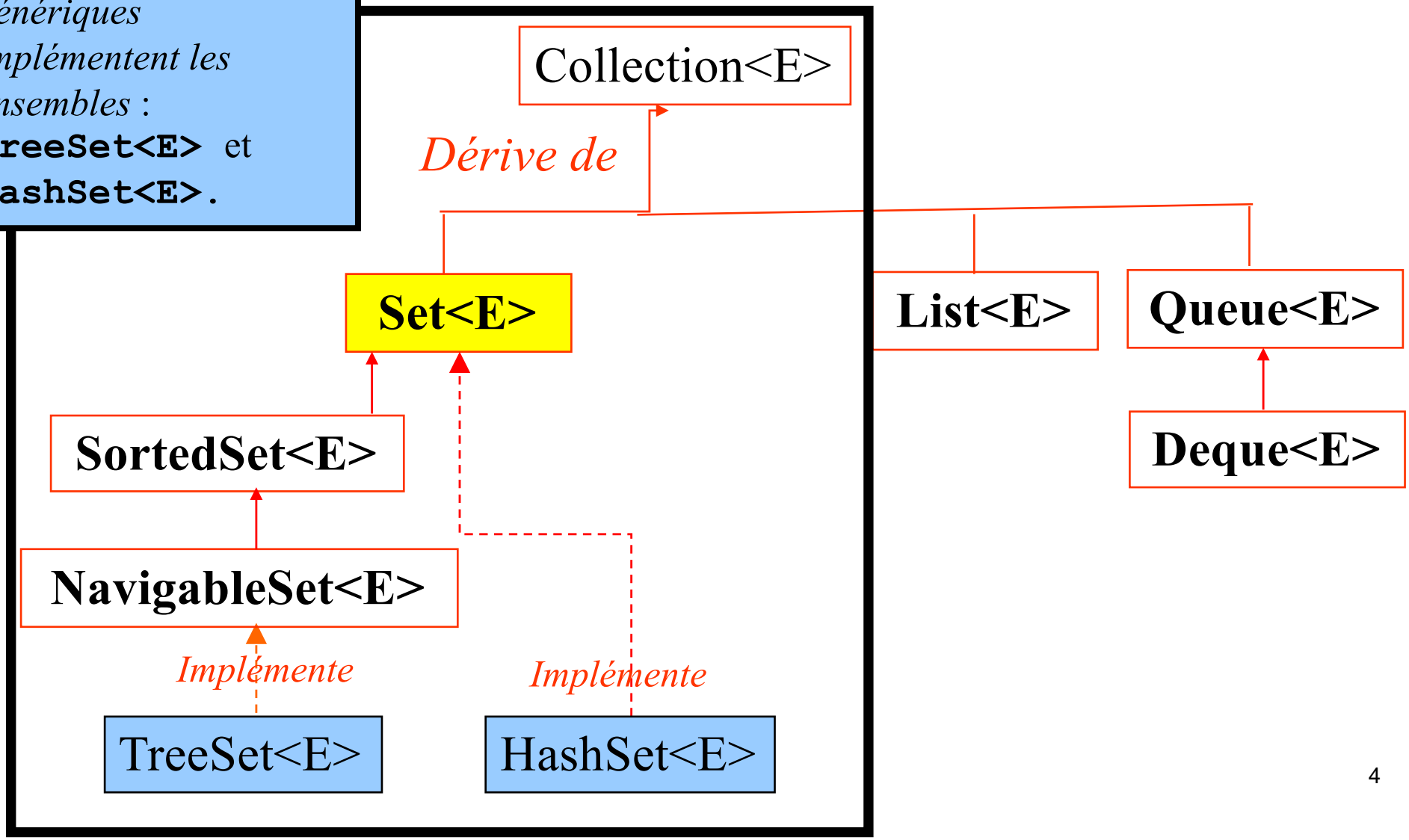
La caractéristique principale des ensembles est que ce sont des collections qui ne renferment **pas de doublons**.
Cette propriété est maintenue toute la durée de vie de la collection.
Exemple : si on tente d'opérer un ajout (`add(…)`) d'un élément déjà «présent» dans un ensemble alors cet ajout échoue. La méthode `add` retourne `false` en cas d'échec.

Un ensemble est vu comme une collection dans laquelle la **recherche d'un élément par sa valeur ou par une partie de sa valeur est efficace.**
En effet, cette opération a une complexité < O(N)

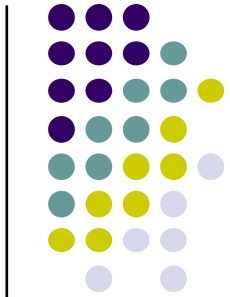
Les ensembles : interface Set<E>



Deux classes génériques implémentent les ensembles : **`TreeSet<E>`** et **`HashSet<E>`.**



Recherche d'un élément dans un ensemble



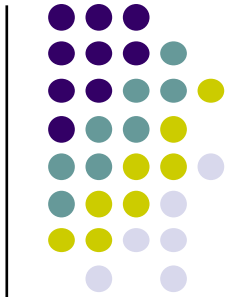
En théorie un ensemble ne devrait pas être ordonné.
Pour diminuer la complexité de l'opération de recherche d'un élément (test d'appartenance) dans un ensemble, deux solutions sont proposées :

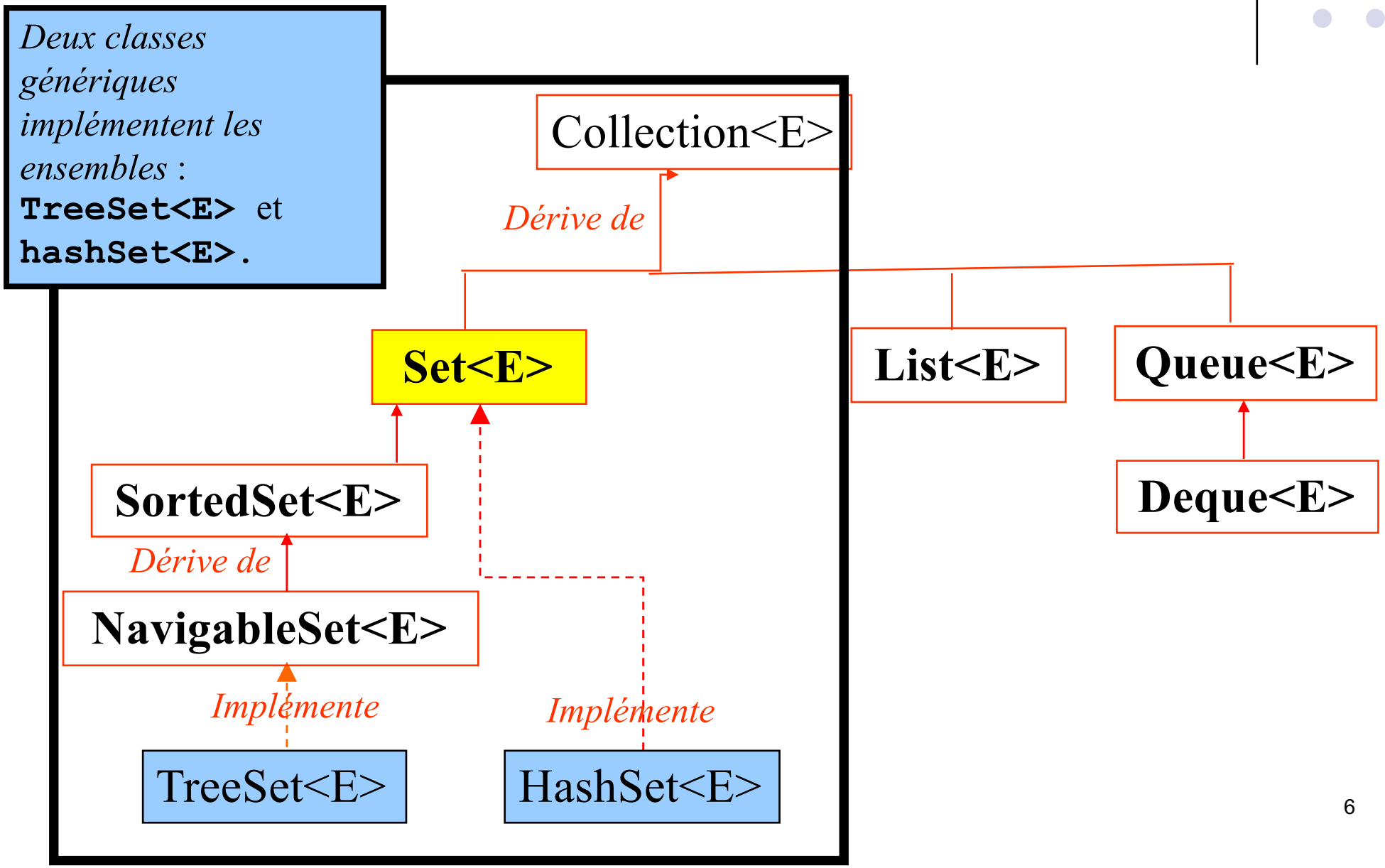
`HashSet<E>` : recours à une technique de hachage,
le test d'appartenance a une complexité de O(1)

TreeSet<E> : on utilise un arbre binaire de recherche pour ordonner complètement les éléments de la collection.
La relation d'ordre est donnée par `compareTo(..)` de la classe E ou par un objet comparateur passé à la construction du TreeSet.

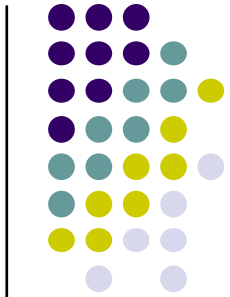
Autrement dit, les emplacements des éléments d'un TreeSet sont structurés selon une relation d'ordre.

Les ensembles : interface Set<E>





Interface Set<E>

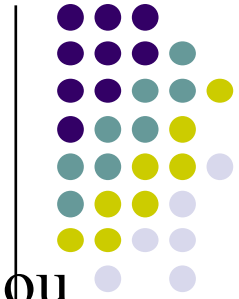


- L'interface Set définit les fonctionnalités d'une collection qui ne peut pas contenir de doublons.

API de l'interface Set<E>

- Pour garantir l'absence de doublons : la classe modélisant les éléments ajoutés dans une collection de type Set doit implémenter ces deux méthodes : **equals() et hashCode().** Ces méthodes sont utilisées lors de l'ajout d'un élément pour déterminer si un élément est déjà présent dans la collection.

Les fonctionnalités dans un ensemble



Toute opération par rang (indice), i.e. insertion ou suppression ou modification, n'a plus de sens car dans un ensemble les emplacements des éléments ne sont pas structurés par rang.
Ils sont structurés selon la valeur des éléments.
Par conséquent, les opérations par rang sont impossibles.

Quelques méthodes spécifiques pour les ensembles (Set<E>) :

```
boolean add (E e)
//ajout garanti sans doublons

boolean contains (object o)
//recherche dichotomique si TresSet

boolean remove(object o)
```

Ajout d'un élément dans un ensemble

Surcharge de la méthode `add` dans les classes qui implémentent Set<E> :

```
public boolean add(E e)
// L'élément e est ajouté à la collection this si celle-ci ne le contient pas déjà
// Est retourné :  vrai si la collection a été modifiée ; faux sinon
```

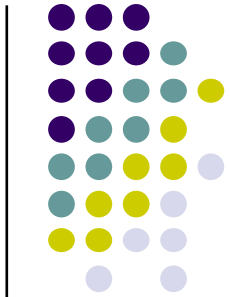
Ajout :

Complexité : ~0(1) pour **HashSet<E>**

0(Log(N)) pour **TreeSet<E>**

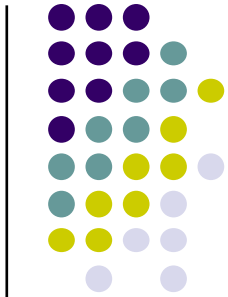
Un collection de type Set utilise la méthode equals() pour vérifier si un élément est déjà présent ou non dans la collection

suppression d'un élément dans un ensemble



Suppression :
On utilise la méthode `remove(...)` de `Collection <E>.`
Complexité : ~0(1) pour **HashSet<E>**
0(Log(N)) pour **TreeSet<E>**

Les opérations ensemblistes



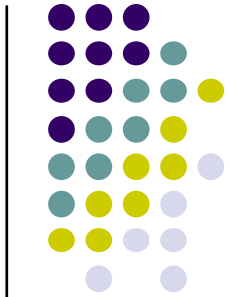
Les méthodes suivantes sont dans l'interface **Collection<E>** :
`removeAll( )` `addAll( )` et `retainAll( )`
Elles prennent tout leur sens lorsque les collections sont des ensembles (nommés ici : e1 et e2).

```
boolean addAll (Collection <? extends E> c2)
```
// e1.addAll(e2) => e1 = e1 ∪ e2 *Union ensembliste*

```
boolean retainAll (Collection <? extends E> c2)
```
// e1. retainAll(e2) => e1 = e1 ∩ e2 *Intersection ensembliste*

```
boolean removeAll (Collection <? extends E> c2)
```
// e1. removeAll(e2) => e1 = e1 ∖ e2 *Différence ensembliste*

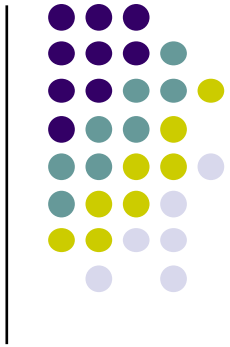
Les itérateurs dans un ensemble Set<E>



Les classes **`HashSet<E>`** et **TreeSet<E>** possèdent un itérateur monodirectionnel puisqu'elles héritent de l'interface **`Collection<E>.`**

Cet itérateur est utilisé pour parcourir la collection.
On peut utiliser le « sucre syntaxique » de la boucle *for each*

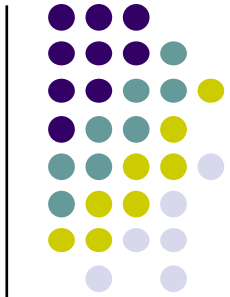
Les ensembles



Nous nous intéressons d'abord à la classe générique TreeSet<E>

Nous examinerons la classe générique HashSet<E> dans un deuxième temps.

TreeSet<E> : constructeurs et relation d'ordre.



Constructeurs de TreeSet<E> :

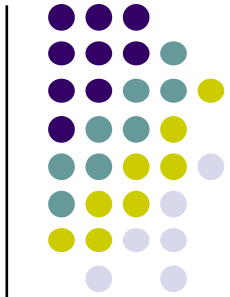
`TreeSet()` : crée un ensemble vide, la relation d'ordre associée étant celle donnée par la méthode **CompareTo()** de la classe E.
La classe E doit donc implémenter l'interface Comparable<E>

TestSetEtudiant

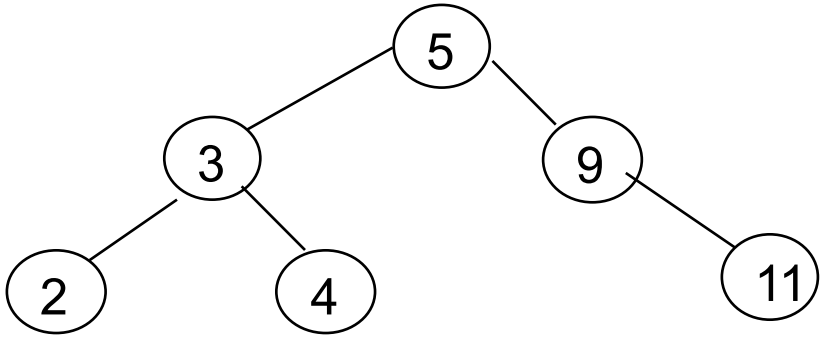
`TreeSet(Comparator<? super E> comp)` : crée un ensemble vide, la relation d'ordre associée étant celle donnée par le **comparateur** comp passé en paramètre.

TestSetCompAnnee

TreeSet<E> : relation d'ordre.

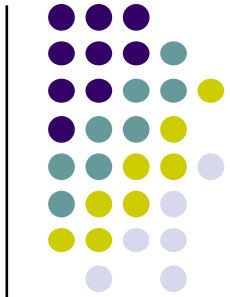


En interne, la classe TreeSet utilise un **arbre binaire** pour stocker ses éléments. Chaque élément est encapsulé dans un noeud. Chaque noeud peut faire référence à aucun, un ou deux autres noeuds.

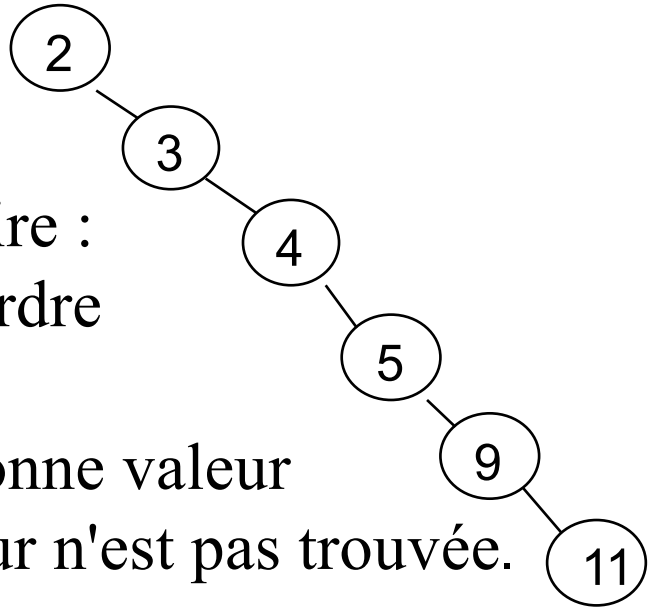


La **recherche d'un élément** dans un arbre binaire est rapide : elle nécessite généralement un temps proportionnel à **log(n)** où n est le nombre d'éléments dans la collection.

TreeSet<E> : relation d'ordre.



La classe TreeSet utilise un **arbre binaire** pour stocker ses éléments.

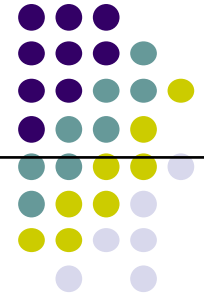


Ce mode de fonctionnement est efficace
si l'ordre d'insertion des éléments est aléatoire :
si tous les éléments sont ajoutés dans leur ordre
alors le parcours reviendra à parcourir tous
les éléments un par un jusqu'à trouver la bonne valeur
ou une valeur supérieure auquel cas la valeur n'est pas trouvée.

Pour pallier à cette problématique, la classe TreeSet met en œuvre un algorithme particulier qui va permettre d'équilibrer l'arbre. Un arbre est équilibré lorsque les feuilles de l'arbre sont à peu près à la même distance de la racine de l'arbre. La distance est le nombre de nœuds parent entre la feuille et le nœud racine.

TreeSet : exemple

TestSet.java



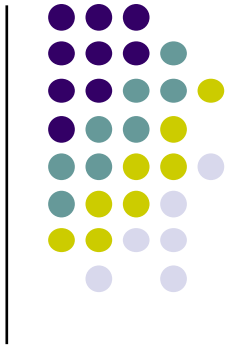
```
import java.util.Set;
import java.util.TreeSet;
public static void main(final String[] args) {
        Set<Integer> e = new TreeSet<Integer>();
        int k;
        for (int i = 1; i < 10; i++) {
                k=(int)(Math.random()*10); //chiffre au hasard
                e.add(new Integer(k));          // sans doublons
        }
        System.out.print("Ensemble : ");
        afficherSet(e);
        System.out.println();
        }
}
```

Exemple d'exécution (ensemble ordonné et sans doublons) :

```
Ensemble : [3, 5, 7, 8, 9]
```

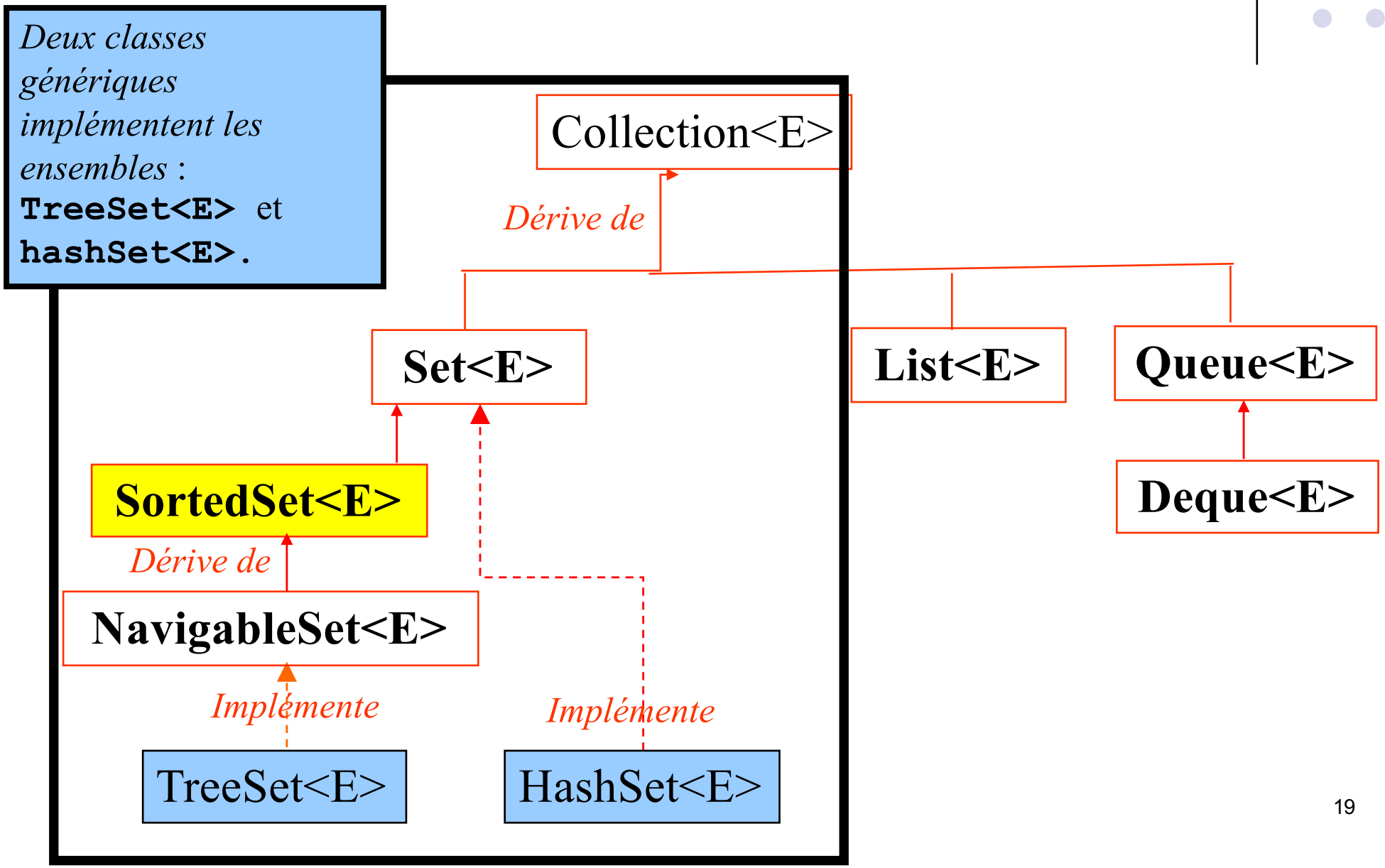
Exemple : itérateur

TestSet.java

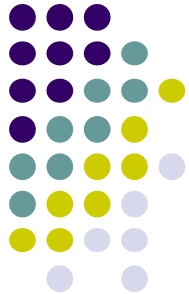


```
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class TestSet {
        public static void afficherSet(final Set<Integer> ens) {
        Iterator<Integer> iterator = ens.iterator();
        while (iterator.hasNext()) {
                Integer element = iterator.next();
                System.out.println(element);
        }
}
```

Les ensembles : interface SortedSet<E>



Interface SortedSet<E>

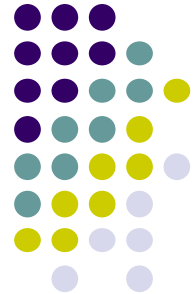


SortedSet<E> est une interface dérivée de Set<E> pour gérer les ensembles ordonnés.

API de l'interface SortedSet<E>

TreeSet<E> implémente l'interface SortedSet<E>.
Celle-ci comprend des méthodes exploitant l'ordre total sur les éléments d'un `TreeSet` (organisé en un arbre binaire).

Interface SortedSet<E>



TreeSet<E> implémente l'interface SortedSet<E>
Celle-ci comprend des méthodes exploitant l'ordre total sur les éléments d'un `TreeSet` (organisé en un arbre binaire).

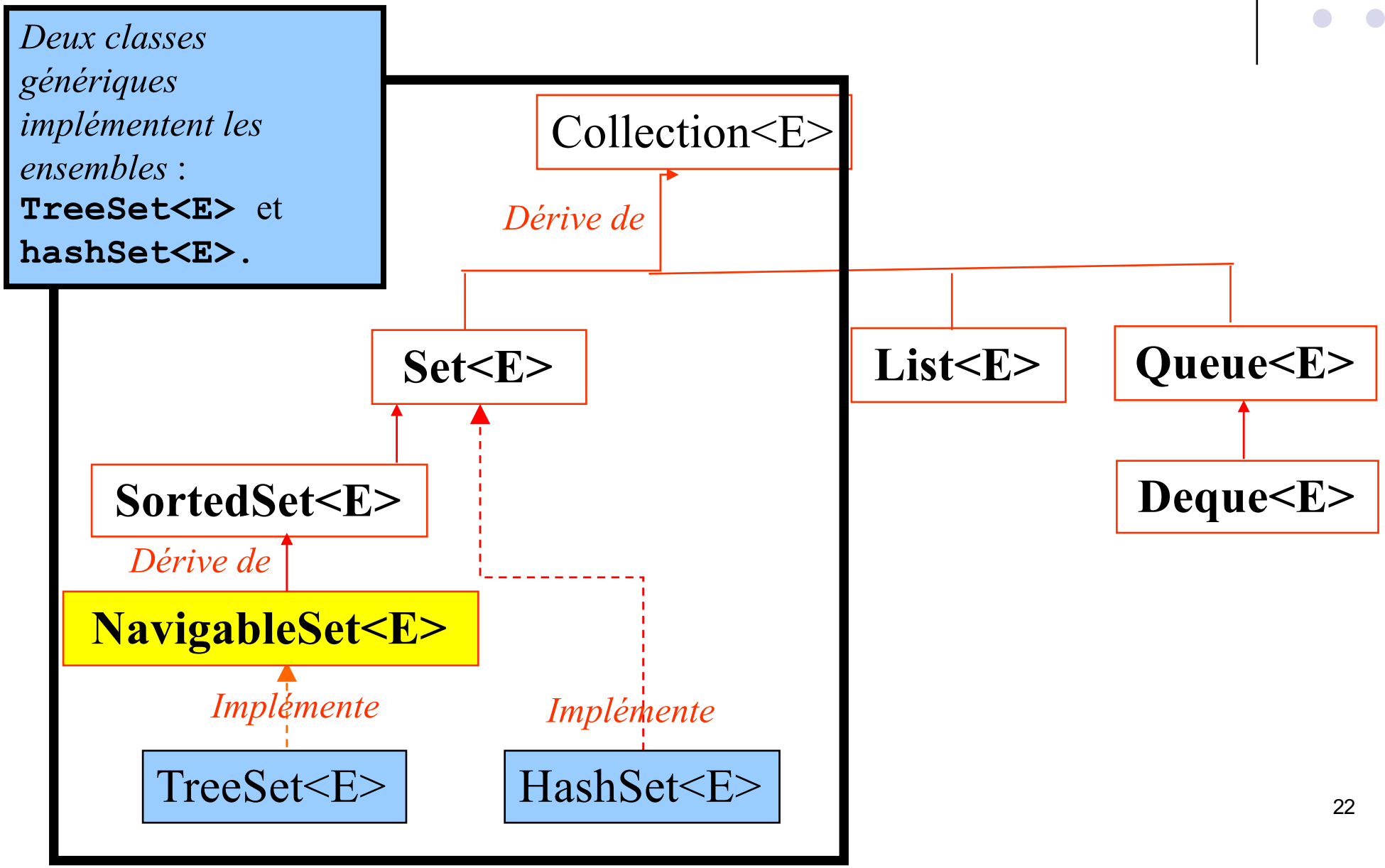
Exemples :

- retourner le comparateur utilisé à la construction de l'ensemble.
- retourner le premier (`first( )`) ou le dernier (`last()`) élément de la collection.
- Obtenir une « vue » d'une partie de l'ensemble, formée par les éléments de valeur supérieure à une valeur donnée (ou de valeur inférieure à une valeur donnée).

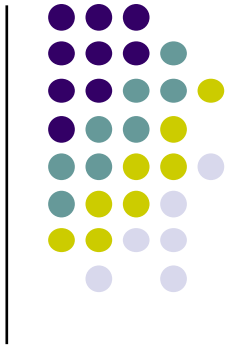
Ex : `SortedSet<E> subSet(E debut, E fin)`

```
    // debut inclus, fin exclu
```

Les ensembles : interface NavigableSet<E>



Interface NavigableSet<E>



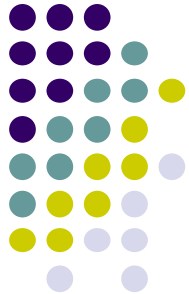
`NavigableSet<E>` est une interface dérivée de `SortedSet<E>`.

API de l'interface NavigableSet<E>

L'interface `NavigableSet` qui hérite de l'interface `SortedSet` définit des fonctionnalités qui permettent :

- le parcours de la collection dans l'ordre ascendant ou descendant
- d'obtenir des éléments proches d'un autre élément.

Interface NavigableSet<E>



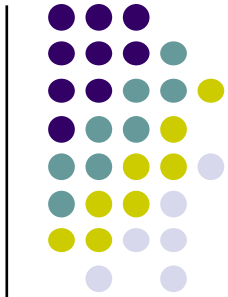
TreeSet<E> implémente l'interface `NavigableSet<E>`
Celle-ci comprend des méthodes exploitant l'ordre total sur les éléments d'un TreeSet (organisé en un arbre binaire).

Exemples :

• `E higher(E elt)` : retourne le plus petit élément strictement supérieur à l'élément fourni en paramètre
Renvoie `null` si aucun élément n'est trouvé

• Exemple : `E lower(E e)` : retourne le plus grand élément strictement inférieur à celui fourni en paramètre.
Renvoie `null` si aucun élément n'est trouvé

Interface NavigableSet<E>

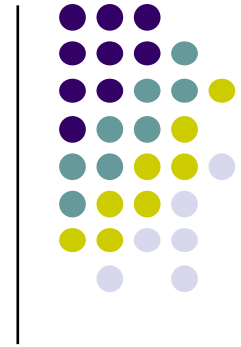


Exemples :

- `E ceiling(E e)` : Retourne le plus petit élément qui soit plus grand ou égal à celui fourni en paramètre.
  Renvoie `null` si aucun élément n'est trouvé
- `E floor(E e)` : retourne le plus grand élément qui soit plus petit ou égal à celui fourni en paramètre.
  Renvoie `null` si aucun élément n'est trouvé

- `Iterator<E> descendingIterator()` : retourne un `Iterator` qui permet le parcours dans un ordre descendant des éléments de la collection
- `NavigableSet<E> descendingSet()` : retourne une *vue* de l'ensemble parcourable dans le sens inverse de l'ordre de la collection actuelle

Interface NavigableSet<E>



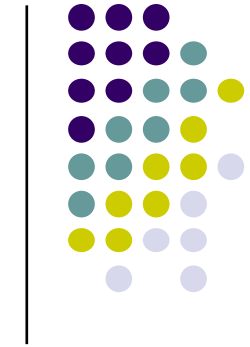
Exemples :

`subSet` (de SortedSet) est surchargée pour pouvoir inclure les bornes ou non.

```
NavigableSet<E> subSet (E fromElement,boolean fromInclusive,
        E toElement, boolean toInclusive)
//Returns a view of the portion of this set whose elements
//range from fromElement to toElement.
```

Exemple

TestNavigableSet.java



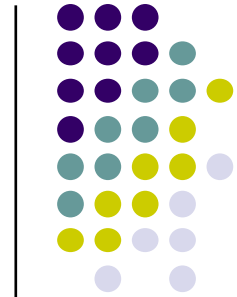
```
import java.util.NavigableSet;
import java.util.TreeSet;

public class TestNavigableSet {
      public static void main(final String[] args) {
NavigableSet<Integer> e = new TreeSet<Integer>();
for (int i = 1; i < 10; i++)
      e.add(i);
System.out.println(e);
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Exemple… suite

TestNavigableSet.java

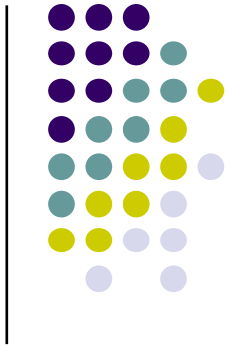
```
public static void main(final String[] args) {
NavigableSet<Integer> e = new TreeSet<Integer>();
for (int i = 1; i < 10; i++)
      e.add(i);
System.out.println(e);
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
System.out.println("ceiling(5)=" + e.ceiling(5));
// ceiling(5)=5
System.out.println("floor(5)=" + e.floor(5));
// floor(5)=5
System.out.println("higher(5)=" + e.higher(5));
// higher(5)=6
System.out.println("lower(5)=" + e.lower(5));
//lower(5)=4
```

Un exemple… suite



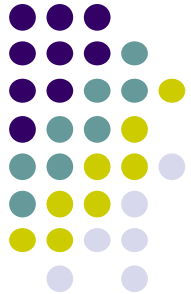
```
System.out.print("Ordre descendant=");
afficherSet(e.descendingSet());
// Ordre descendant=9, 8, 7, 6, 5, 4, 3, 2, 1

System.out.println("Sous ensembles : ");
System.out.print("headSet(5)=");
afficherSet(e.headSet(5));
// headSet(5)=1, 2, 3, 4

System.out.print("headSet(5,true)=");
afficherSet(e.headSet(5, true));
// headSet(5,true)=1, 2, 3, 4, 5
```

Les ensembles



HashSet<E>

La fonction de hachage (HashSet<E>)

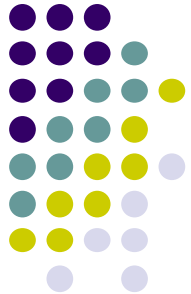


Une **fonction de hachage** est une fonction qui retourne un entier. Ce dernier est appelé « **code de hachage** ». Il permet de retrouver rapidement la valeur d'un élément.
Fonction de hachage : `int hashCode ( )`

La classe `Object` comprend une fonction de hachage nommée `hashCode()` qui utilise la référence des objets.
La classe `String` et les *classes enveloppes (ou wrappers en anglais)* `Integer`, `Double`... redéfinissent la méthode `hashCode()`. Celle-ci utilise la valeur effective des objets.

Par exemple : la fonction de hachage sur une chaîne de caractères (classe `String`) retourne un calcul fait à partir des codes ASCII des caractères de la chaîne.

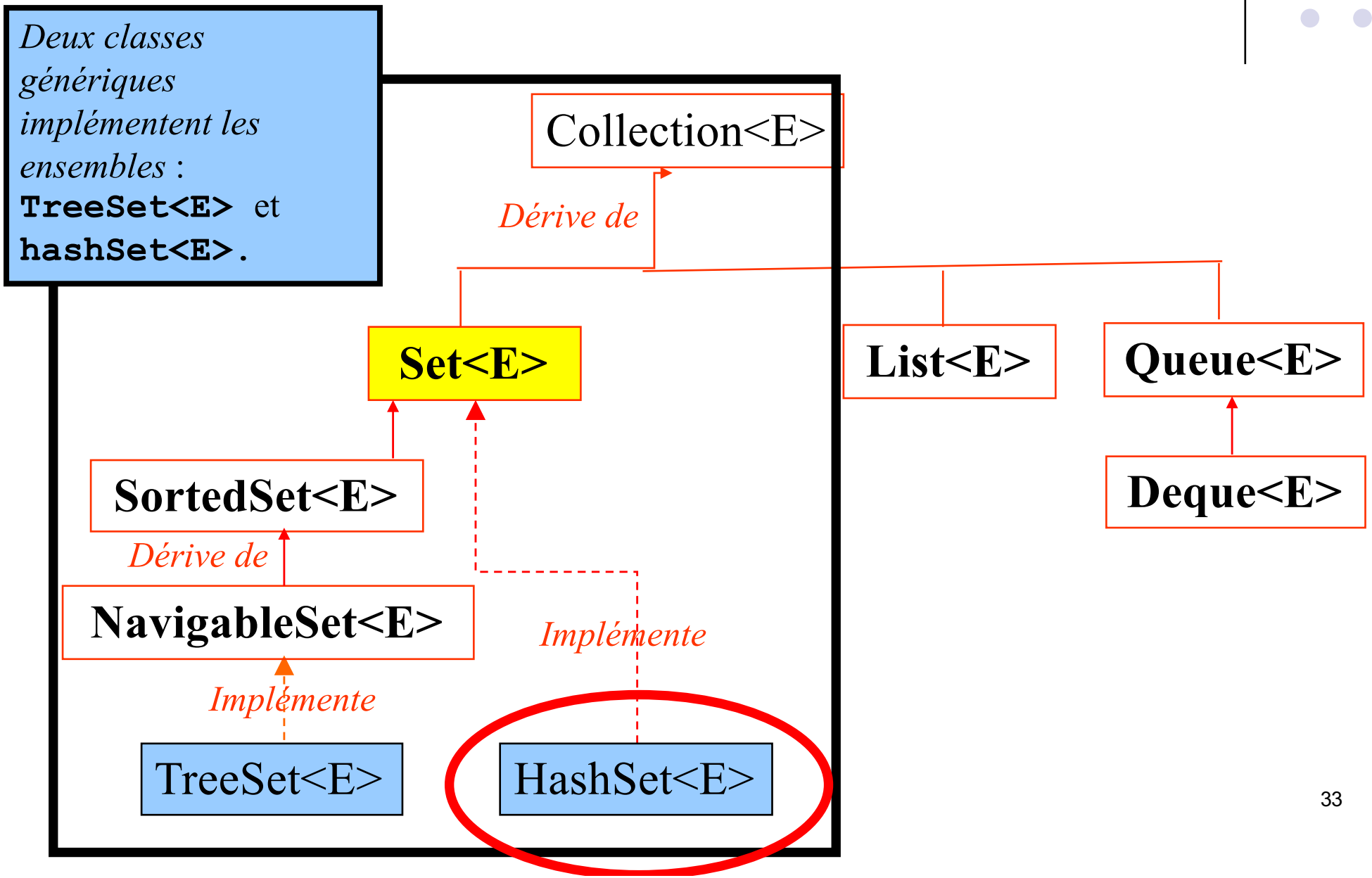
La fonction de hachage (HashSet<E>)



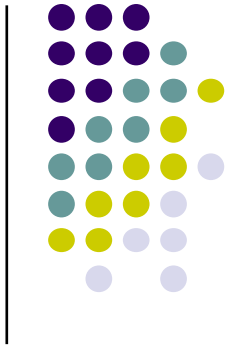
Fonction de hachage : `int hashCode ( )`

Règle : un code de hachage peut correspondre à un ou plusieurs éléments de la collection.
Deux éléments de même valeur (au sens d'`equals`) doivent avoir un code de hachage identique.
Par conséquent lorsqu'une classe redéfinit la méthode `equals` alors elle doit aussi redéfinir la méthode : `int hashcode( )` en cohérence avec `equals`.

Les ensembles : interface NavigableSet<E>



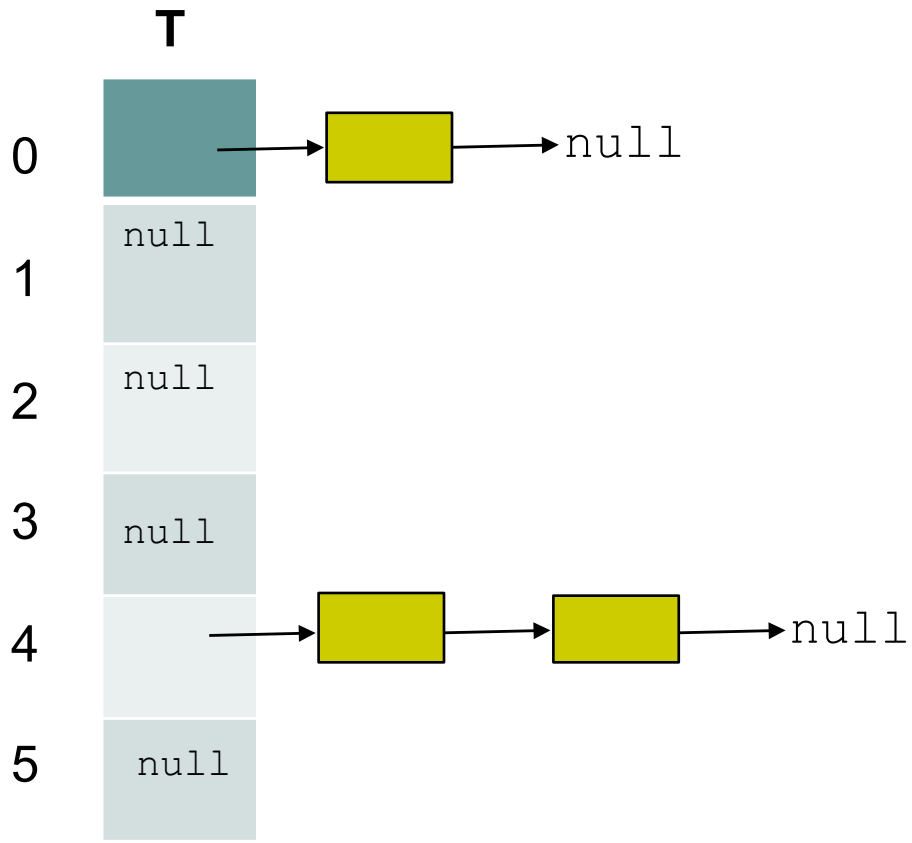
Classe HashSet<E>



Cette classe fonctionne comme une table de hachage.
Elle organise les positions des éléments d'un ensemble en fonction du résultat retourné par la « **fonction de hachage** » associée à la classe E.
Pour organiser les éléments, on constitue un tableau (dynamique) **T** de K listes chaînées. Chaque indice du tableau correspond à un code de hachage (appelé aussi valeur de hachage).
Les éléments de l'ensemble qui ont un code ***c*** sont alors rangés dans la liste chaînée située à l'indice correspondant au code ***c*** du tableau **T**.

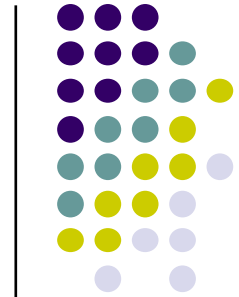
Vocabulaire :
les listes chaînées sont appelées « **seaux** » (buckets en anglais)

Table de hachage



6 seaux

Facteur de hachage



Si la table de hachage comprend **N** éléments (size),
le **facteur de hachage** (load factor) est égal à :

nombre de seaux / N.

Plus ce facteur est grand, moins les seaux comprennent d'éléments. On recherchera un facteur proche de 1. On estime qu'il est bon pour maintenir la complexité de la recherche d'un élément à 0(1), s'il est supérieur ou égal à 0,75.

Pour rechercher un élément dans un ensemble ou pour savoir s'il est présent, on commence par calculer son code de hachage ***c*.**
Ensuite l'indice du tableau est donné par ***: c* % N.** Puis on teste si la valeur de l'élément se trouve dans le seau (à l'aide de la méthode `equals` pour comparer l'égalité de deux éléments).

`equals` et `hashCode` dans la classe E

- La classe représentant les éléments ajoutés dans une collection de type HashSet doit implémenter ces deux méthodes :

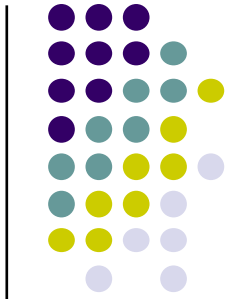
  `equals(...)` et `hashCode()`.

- Ces méthodes sont utilisées lors de l'ajout d'un élément pour déterminer s'il est déjà présent dans la collection.

La valeur retournée par `hashCode()` est recherchée dans la collection : si aucun objet de la collection n'a la même valeur de hachage alors l'objet n'est pas encore dans la collection et peut être ajouté.

si un ou plusieurs objets de la collection ont la même valeur de hachage alors la méthode `equals(...)` de l'objet à ajouter est invoquée sur chacun des objets pour déterminer si l'objet est déjà présent ou non dans la collection

`equals` et `hashCode` dans la classe E



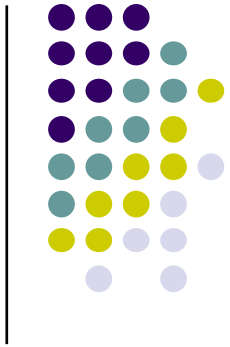
- La classe représentant les éléments ajoutés dans une collection de type HashSet doit donc implémenter ces deux méthodes :

  `equals(…)` et `hashCode()`

- Leurs implantation doivent être cohérentes.

- En d'autres termes , **si deux objets sont égaux alors ils doivent avoir la même valeur de hachage .**

Les ensembles



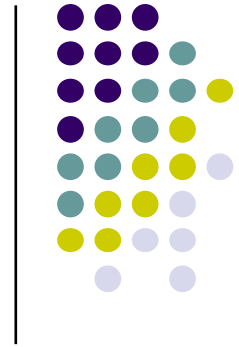
Exemples : TreeSet<Etudiant> et HashSet<Etudiant>

La classe Etudiant

```
public class Etudiant implements Comparable<Etudiant>
{
      private String nom;
      private String prenom;
      private int annee_naiss;
      private int no;
```

```
public int compareTo (Etudiant e) {
   int r = this.getNom().compareTo(e.getNom());
   if (r==0)
     r =this.getPrenom().compareTo(e.getPrenom());
   return r;
}
```

equals et hashCode dans la classe Etudiant



```
public class Etudiant {
…
```

```
   public boolean equals (Object o) {
// est retourné vrai si this et o ont un prénom et un nom
//identiques;faux sinon
   if (! (o instanceof Etudiant)) return false;
   Etudiant e=(Etudiant)o;
   return (this.getNom().equals(e.getNom()) &&
                  this.getPrenom().equals(e.getPrenom()))
   }
```

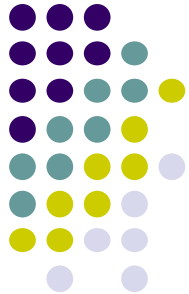
```
   public int hashCode() {
       return getNom().hashCode()+getPrenom().hashCode();
   }
```

```
}
```

Les trois méthodes : compareTo, equals et haschCode sont cohérentes

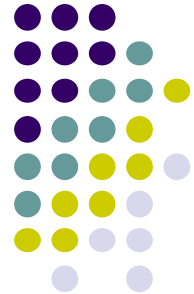
TreeSet <Etudiant>

TestTreeSetEtudiant.java



```
import java.util.Set;
import java.util.TreeSet;

public class TestTreeSetEtudiant {
      public static void main(final String[] args) {
      Set<Etudiant> e = new TreeSet<Etudiant>();


            . . .
      }
}
```

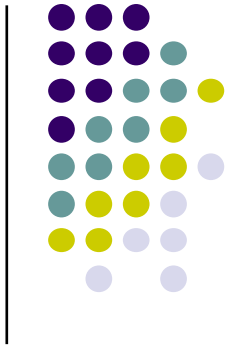
HashSet <Etudiant>



TestHashSetEtudiant.java

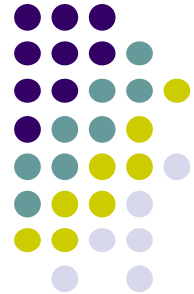
```
import java.util.Set;
import java.util.HashSet;

public class TestHashSetEtudiant {
      public static void main(final String[] args) {
      Set<Etudiant> e = new HashSet<Etudiant>();

            . . .
      }
}
```

Les ensembles



Performances de HashSet<E>

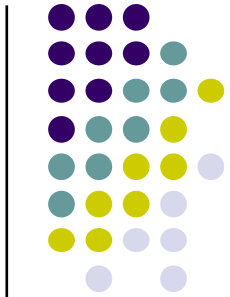
Performances d'un HashSet<E>



Cette implémentation offre des performances (quasi) constantes pour les opérations :
`add(E e), remove(Object o), contains(Object o)`
Le temps pris par ces opérations n'augmente donc pas
avec le cardinal de la collection de type HashSet.

L'itération sur les éléments d'un HashSet se fait avec un itérateur renvoyé par la méthode `iterator( )`
de la classe interface Collection<E>. Ou : `for each`
Notons que l'itération sur les éléments d'un HashSet se fait dans un ordre qui n'est pas prévisible.

HashSet<E> : caractéristiques



- Cette collection ne garantit pas un parcours ordonné lors de l'itération sur les éléments qu'elle contient.

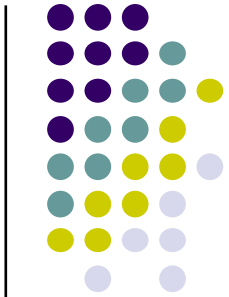
- Elle ne permet pas d'ajouter des doublons mais elle permet l'ajout d'un élément `null`.

- Elle offre des performances (quasi) constantes pour les opérations :
  `add(E e)`, `remove(Object o)`, `contains(Object o)`

MERCI

de votre attention

Constructeurs



HashSet()	Ce constructeur construit un HashSet par défaut.
HashSet(Collection c)	Ce constructeur initialise le hash set en utilisant les éléments de la collection c.
HashSet(int capacity)	Ce constructeur initialise la capacité du hachage défini sur la valeur entière donnée. La capacité augmente automatiquement à mesure que des éléments sont ajoutés au HashSet.
HashSet(int capacity, float fillRatio)	Ce constructeur initialise à la fois la capacité et le taux de remplissage (load capacity) de HashSet à partir de ses arguments. Ici, le taux de remplissage doit se situer entre 0,0 et 1,0, et il détermine la quantité de HashSet avant de le redimensionner vers le haut. Plus précisément, lorsque le nombre d'éléments est supérieur à la capacité du HashSet multipliée par son taux de remplissage, le HashSet est élargi.