



# Les collections et les listes d'associations

CSC 3101

Algorithmique et langage de programmation

Gaël Thomas



# Depuis le début du module

Vous avez mis en œuvre 4 grandes structures de données

- Des **tableaux dynamiques** avec l'armée de monstres (CI3)
- Des **listes chaînées** avec l'armée de monstres (CI3)
- Des **arbres binaires de recherche** avec la banque Lannister (CI4)
- Des **tables de hachage** pour votre voyage à Bilbao (CI7)

**Bravo !**

Vous avez donc gagné le droit d'utiliser  
la bibliothèque Java qui vous fournit  
ces structures de données 😊

# La bibliothèque `java.util`

- Fournit un ensemble d'interfaces abstrayant les structures de données les plus fréquemment utilisées
- Fournit un ensemble de classes mettant en œuvre ces structures de données
- Toutes ces classes et interfaces sont génériques

# Avant de commencer (1/2)

La bibliothèque de structures de données Java nécessite des méthodes utilitaires

- Pour effectuer une comparaison par valeurs entre objets  
(comme avec `CityId.equals` dans nos tables de hachage)
- Pour connaître le code hachage d'un objet  
(comme avec `CityId.hashCode` dans nos tables de hachage)
- Pour savoir si un objet est plus petit qu'un autre  
(comme avec la banque Lannister avec les comptes classés suivant un ordre lexicographique dans l'arbre binaire de recherche)

# Avant de commencer (2/2)

La bibliothèque de structures de données Java nécessite des méthodes utilitaires

- Pour effectuer une comparaison par valeurs entre objet  
Offert directement par `Object` via la méthode `equals`
- Pour connaître le code hachage d'un objet  
Offert directement par `Object` via la méthode `hashCode`
- Pour savoir si un objet est plus petit qu'un autre  
Offert par l'interface `Comparable` via la méthode `compareTo`

# Deux familles de structures de données

■ La **collection** stocke une collection d'éléments

- Tableaux extensibles (stocke des monstres)
- Listes chaînées (stocke des monstres)

■ La **liste d'association** associe des clés et des valeurs

- La table de hachage (associe des noms à des ville)
- L'arbre binaire de recherche (associe des noms à des comptes)

# Deux familles de structures de données

## La **collection** stocke une collection d'éléments

- Tableaux extensibles
- Listes chaînées

```
java.util.ArrayList<E>
```

```
java.util.LinkedList<E>
```

## La **liste d'association** associe des clés et des valeurs

- La table de hachage
- L'arbre binaire de recherche

```
java.util.HashMap<K, V>
```

```
java.util.TreeMap<K, V>
```

# La collection d'éléments

L'interface `Collection<E>` représente une collection

- `boolean add(E e)` : ajoute l'élément `e`
  - `boolean remove(Object o)` : supprime l'élément `o`
  - `boolean contains(Object o)` : vrai si collection contient `o`
- 
- Pourquoi `E` ou `Object`  $\Rightarrow$  souvent pour raisons historiques, les génériques ne sont apparus que dans la version 5 de Java



# Exemple d'utilisation des collections

```
/* ArrayList ⇒ tableau extensible (cas ici) */  
/* LinkedList ⇒ liste chaînée */  
  
Collection<Monster> col = new ArrayList<Monster>();  
Monster m = new Monster("Pikachu");  
  
col.add(m);  
if (col.contains(m)) {  
    System.out.println("Les collections, c'est facile !");  
}
```

# La classe `Iterator`

Un `Iterator` permet de parcourir une collection

- `E next()` : renvoie l'élément suivant et avance dans la collection
- `boolean hasNext()` : renvoi vrai si il existe un suivant
- `void remove()` : supprime l'élément courant

Exemple d'utilisation

```
Collection<Monster> col = new LinkedList<Monster>();  
... /* ajoute des monstres à la collection ici */  
Iterator<Monster> it = col.iterator();  
while(it.hasNext()) {  
    Monster cur = it.next();  
    System.out.println("Monstre: " + cur.name);  
}
```

# La boucle « pour chaque »

Sucre syntaxique introduit dans Java 5

```
for(Monster cur: col) {  
    /* fait quelque chose avec cur */  
}
```



```
Iterator<Monster> it = col.iterator();  
while(it.hasNext()) {  
    Monster cur = it.next();  
    System.out.println("Monstre: " + cur.name);  
}
```

# La boucle « pour chaque »

Sucre syntaxique

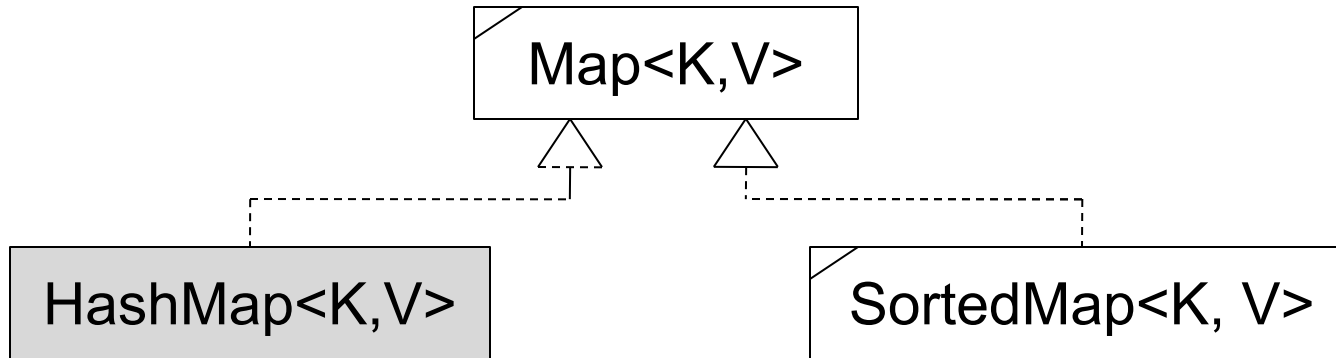
Bon à savoir

on peut aussi utiliser une boucle « pour chaque »  
pour parcourir un tableau

```
String[] tab = { "a", "b" };  
  
for (String s: tab) {  
    /* ... */  
}
```

... ;

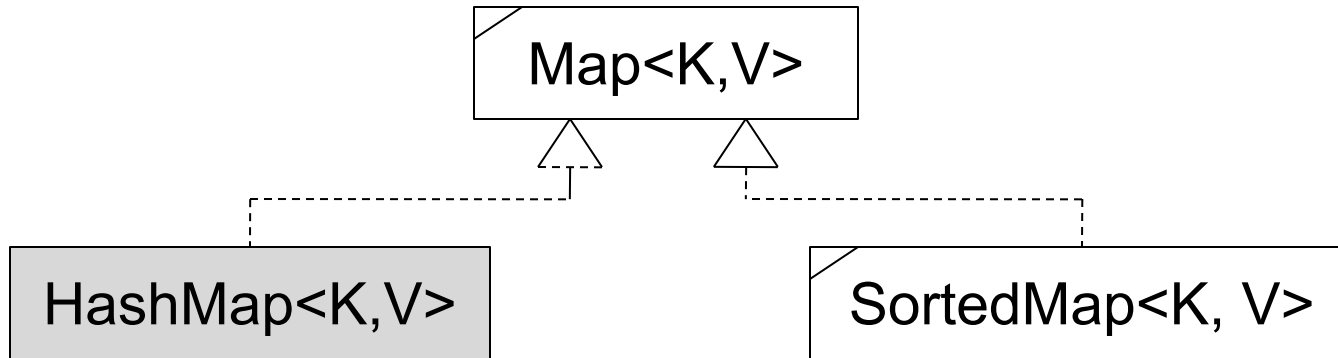
# Les tables d'association en Java (1/2)



`Map` définit principalement quatre méthodes

- `V put (K k, V v)` : associe `k` à `v` (renvoie ancienne valeur ou `null`)
- `V get (K k)` : renvoie la valeur associée à `v` ou `null`
- `boolean remove (K k)` : supprime l'association
- `Collection<V> values ()` : renvoie la collection des valeurs

# Les tables d'association en Java (2/2)



## Deux principales mises en œuvre

- La table de hachage (`HashMap`)
  - `K` doit mettre en œuvre `equals` et `hashCode`
  - Table d'association + éléments non triés
- L'arbre binaire rouge-noir (`TreeMap` de type `SortedMap`)
  - `K` doit mettre en œuvre `Comparable`
  - Table d'association + éléments triés suivant l'ordre des clés

# Notions clés

`Collection` pour stocker des collections d'éléments

- Deux principales mises en œuvre
  - `ArrayList` : tableau extensible, ajout lent (extension du tableau), accès aléatoire rapide
  - `LinkedList` : liste chaînée, ajout rapide, accès aléatoire lent
- Parcours avec `Iterator` ou boucle « pour chaque »

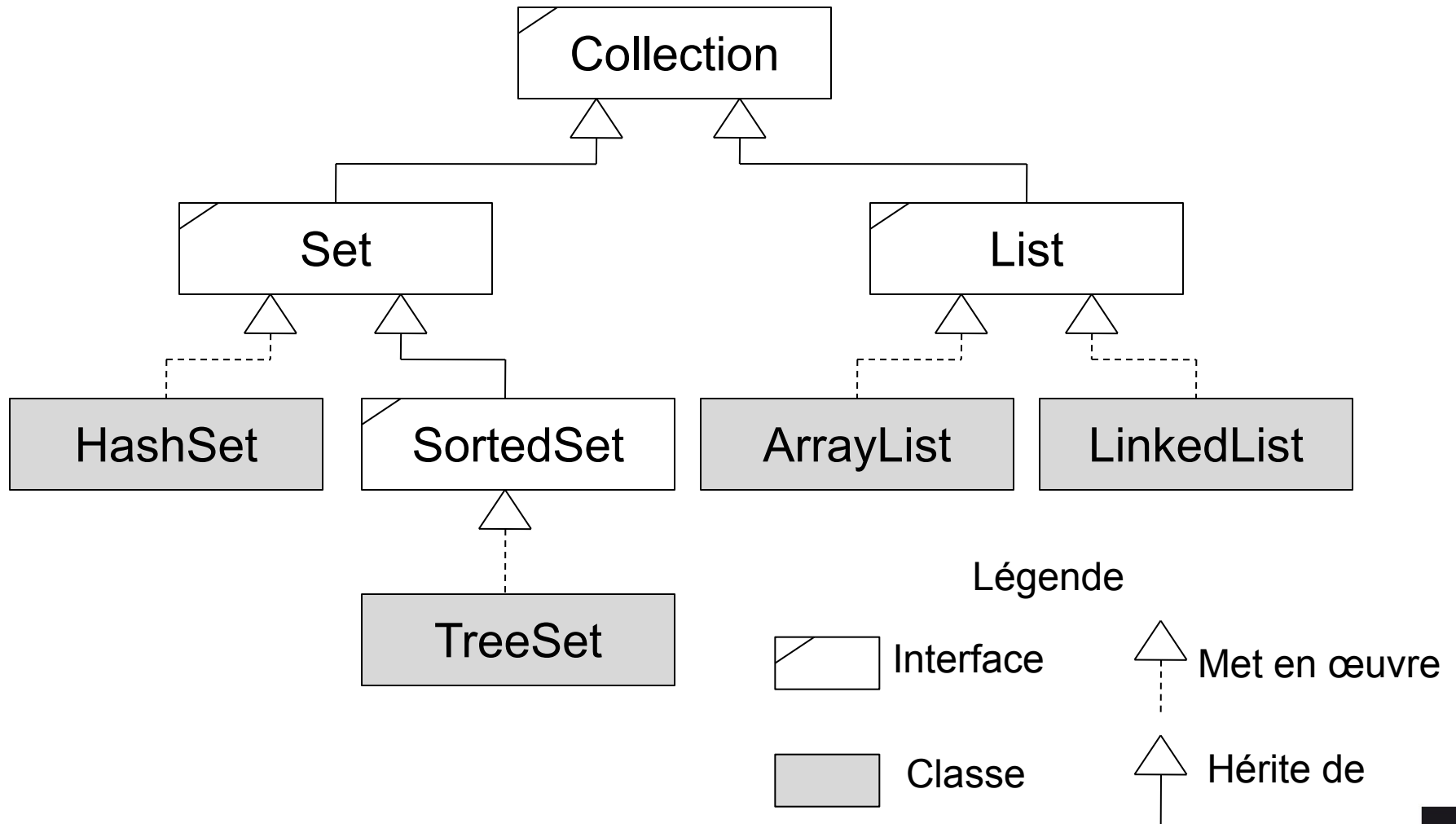
`Map` pour associer clé et valeur

- Deux principales mises en œuvre
  - `HashMap` : table de hachage, non triée
  - `TreeMap` : arbre binaire rouge-noir, trié suivant les clés

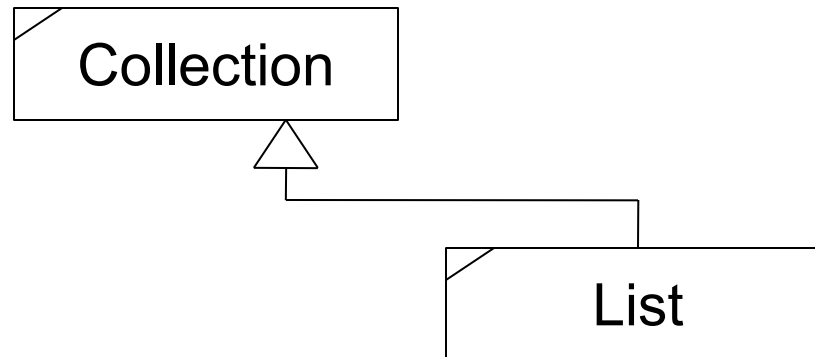
# Pour approfondir



# L'arbre d'héritage de **Collection**



# L'interface `List` (1/2)



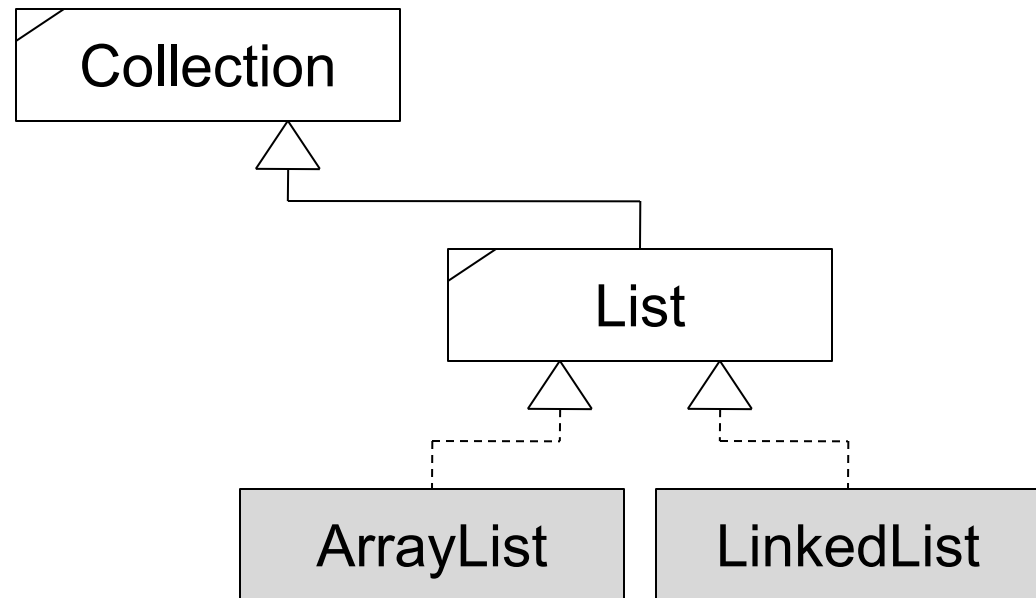
## Propriétés :

- Les éléments possèdent un indice  $\Rightarrow$  généralisation des tableaux
- Duplication d'éléments autorisée

## Ajoute principalement quatre nouvelles méthodes

- `void add(int i, E e)`
- `void set(int i, E e)`
- `E get(int i)`
- `E remove(int i)`

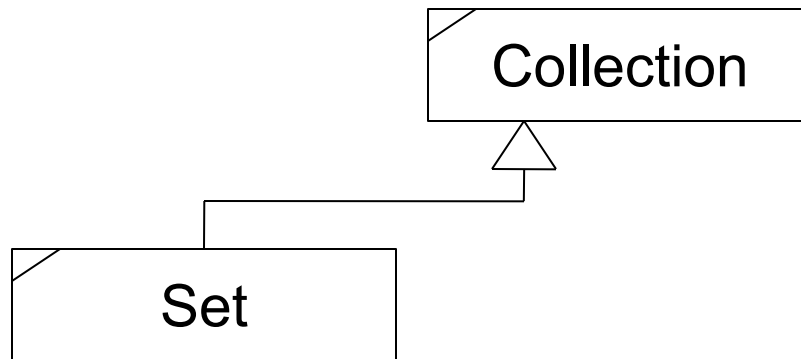
# L'interface `List` (2/2)



Deux principales mises en œuvre

- `ArrayList` : tableau extensible
- `LinkedList` : liste chaînée

# L'interface **Set** (1/2)

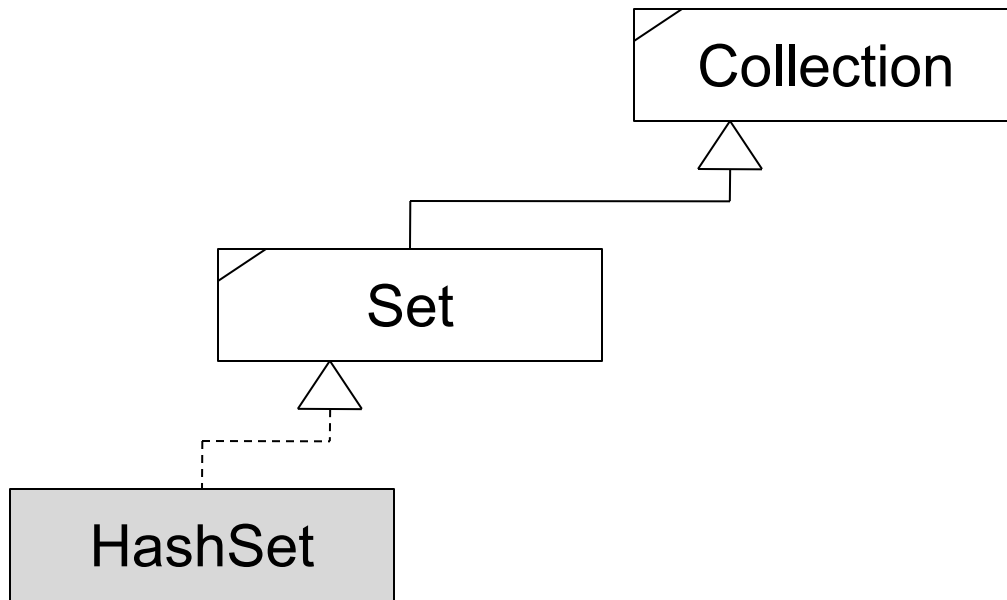


## Propriétés :

- Les éléments ne possèdent pas d'indice
- Duplication d'éléments interdite

Pas de nouvelle méthode ajoutée

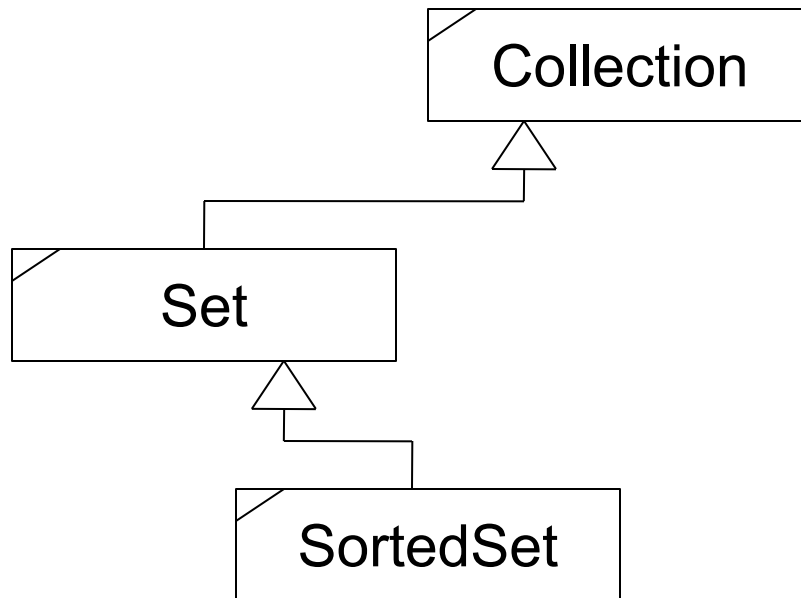
# L'interface **Set** (2/2)



**HashSet** = table de hachage dans laquelle clé == valeur

- Les éléments doivent mettre en œuvre `equals` et `hashCode`
- Les éléments sont non triés

# L'interface SortedSet (1/2)

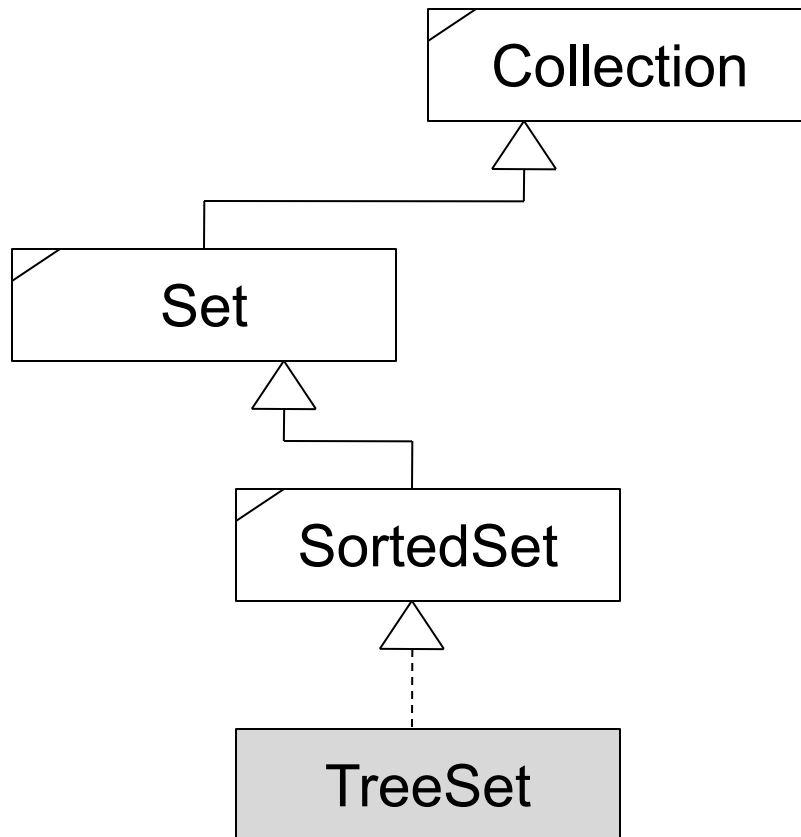


## Propriétés :

- Les éléments sont triés (mettent en œuvre `Comparable`)

Pas de nouvelle méthode ajoutée

# L'interface SortedSet (2/2)



■ TreeSet = arbre binaire rouge-noir dans lequel clé == valeur