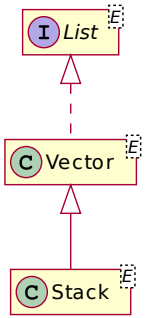
Généricité



Transtypage vers un type paramétré : l'insertion d'une instruction `checkcast` se fait sur le type brut.

```
Stack<Integer> iv = new Stack<Integer>();
List<Integer> il = [(List<Integer>)] iv;
```

compilation → 1/ CT(`iv`) compatible ou PE avec CT(`il`) ?
OK : `Stack<Integer>` compatible avec `List<Integer>`

exécution → OK : RT(`iv`) = `Stack`

```
List<Integer> il = new Stack<Integer>();
Stack<Integer> is = (Stack<Integer>) il;
```

compilation → 1/ CT(`il`) compatible ou PE avec CT(`is`) ?
OK : PE de `List<Integer>` vers `Stack<Integer>`
2/ insertion de `checkcast Stack`

exécution → OK : RT(`il`) = `Stack`

```
List<Integer> il = new Vector<Integer>();
Stack<Integer> is = (Stack<Integer>) il;
```

compilation → 1/ CT(`il`) compatible ou PE avec CT(`is`) ?
OK : PE de `List<Integer>` vers `Stack<Integer>`
2/ insertion de `checkcast Stack`

exécution → erreur : `ClassCastException`
`Vector cannot be cast to Stack`
→ RT(`il`)

```
List<Integer> il = new Stack<Integer>();
Vector<String> sv = (Vector<String>) il;
```

compilation → 1/ CT(`il`) compatible ou PE avec CT(`sv`) ?
erreur : `cannot cast from List<Integer> to Vector<String>`

Transtypage vers un type paramétré

quand le transtypage n'est pas sûr,
le compilateur émet un avertissement `unchecked`.

```
Object o = new Stack<Integer>();
Stack<Integer> is = (Stack<Integer>) o;
```

compilation :
1/ CT(`o`) compatible ou PE avec CT(`is`) ?
OK : PE de `Object` vers `Stack<Integer>`
MAIS : *unchecked warning* (`Integer` non garanti)
2/ insertion de `checkcast Stack`

exécution :
OK : RT(`o`) = `Stack`

```
Object o = new Stack<String>();
Stack<Integer> is = (Stack<Integer>) o;
```

compilation :
1/ CT(`o`) compatible ou PE avec CT(`is`) ?
OK : PE de `Object` vers `Stack<Integer>`
MAIS : *unchecked warning* (`Integer` non garanti)
2/ insertion de `checkcast Stack`

exécution :
OK sur le moment : RT(`o`) = `Stack`
... MAIS CCE
quand on lira des entiers dans une pile de chaînes

```
Stack s = new Stack<String>();
Stack<Integer> is = (Stack<Integer>) s;
```

compilation :
1/ CT(`s`) compatible ou PE avec CT(`is`) ?
OK : PE de `Stack` vers `Stack<Integer>`
MAIS : *unchecked warning* (`Integer` non garanti)

exécution :
OK sur le moment : RT(`s`) = `Stack`
... MAIS CCE
quand on lira des entiers dans une pile de chaînes

Généricité

Méthode pont : lorsqu'une classe concrétise une méthode abstraite, le compilateur doit parfois synthétiser un corps pour la méthode abstraite héritée afin de la connecter à sa concrétisation.

```
class Human implements Comparable<Human> {
    public int compareTo(Human x) {
        return 0;
    }
}
```

effacement

```
class Human implements Comparable {
    public int compareTo(Human x) {
        return 0;
    }
}
```

compilation

```
class Human implements java.lang.Comparable {
  public int compareTo(Human);
    flags: (0x0001) ACC_PUBLIC
    Code:
      0: iconst_0
      1: ireturn

  public int compareTo(java.lang.Object);
    flags: (0x1041) ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      0: aload_0
      1: aload_1
      2: checkcast     #1   // class Human
      5: invokevirtual #20  // Method compareTo:(LHuman;)I
      8: ireturn

  ...
}
```

```
interface Comparable<E> {
    int compareTo(E x);
}
```

effacement

```
interface Comparable {
    int compareTo(Object x);
}
```

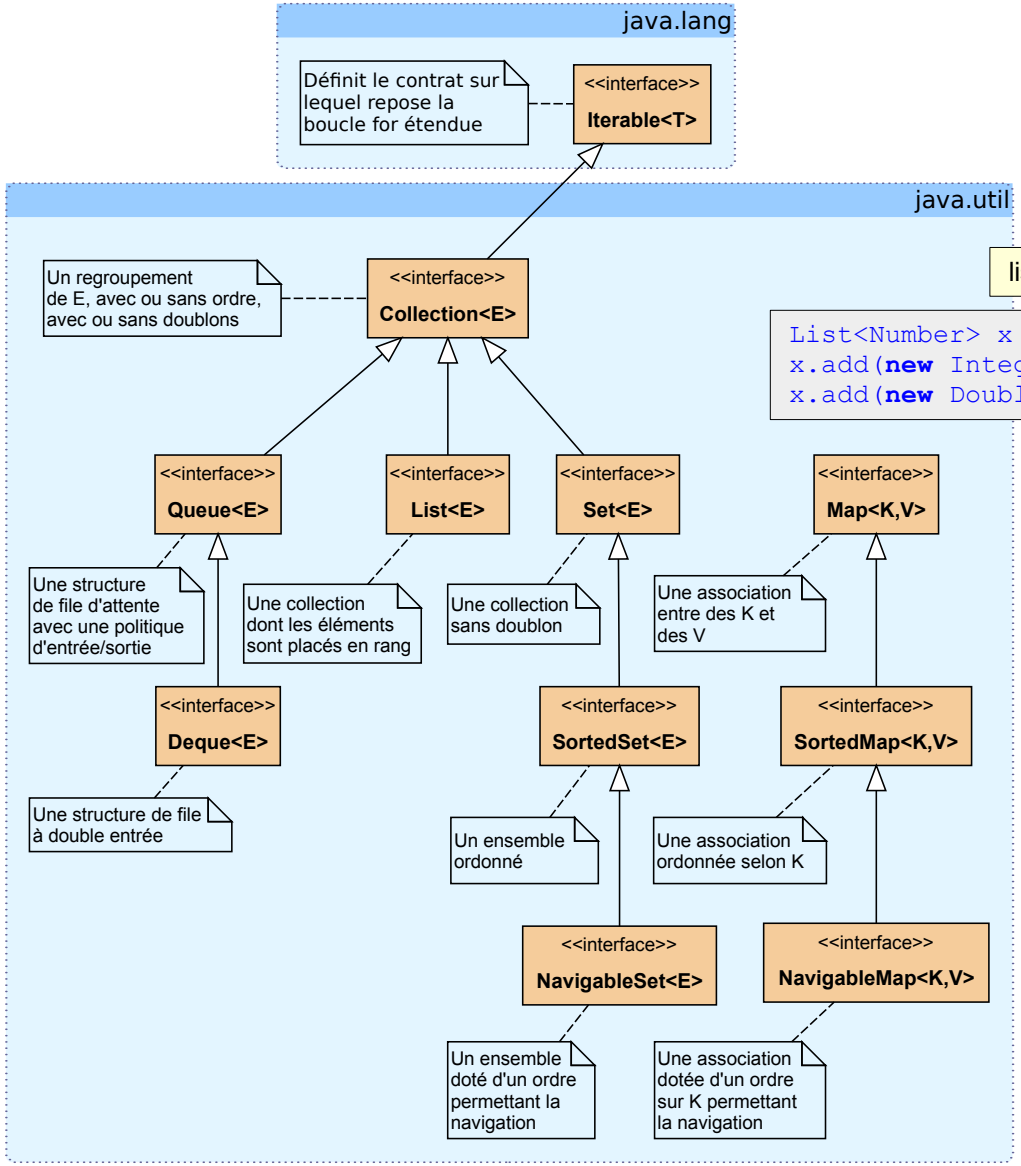
intérêt → *legacy code*

méthode pont implémentée par le compilateur

```
public int compareTo(Object x) {
    return compareTo((Human) x);
}
```

Généricité

Collection :
objet spécialisé dans le stockage d'objets-éléments
(ajout, suppression, recherche et parcours),
dont le type des éléments est représenté par un paramètre de type.



liste d'objets (~ `List<Object>`)

```
List x = new ArrayList();
x.add(new Integer(5));
x.add("bonjour");
```
Java 1.4

liste de nombres (entiers ou flottants)

```
List<Number> x = new ArrayList<Number>();
x.add(new Integer(5));
x.add(new Double(3.14));
```
Java 6

Généricité

Implémentation	Efficacité	Classes
tableau	+ accès par rang - insertion, suppression	`ArrayList` `ArrayDeque`
chaînage	+ insertion, suppression - accès par rang	`LinkedList`

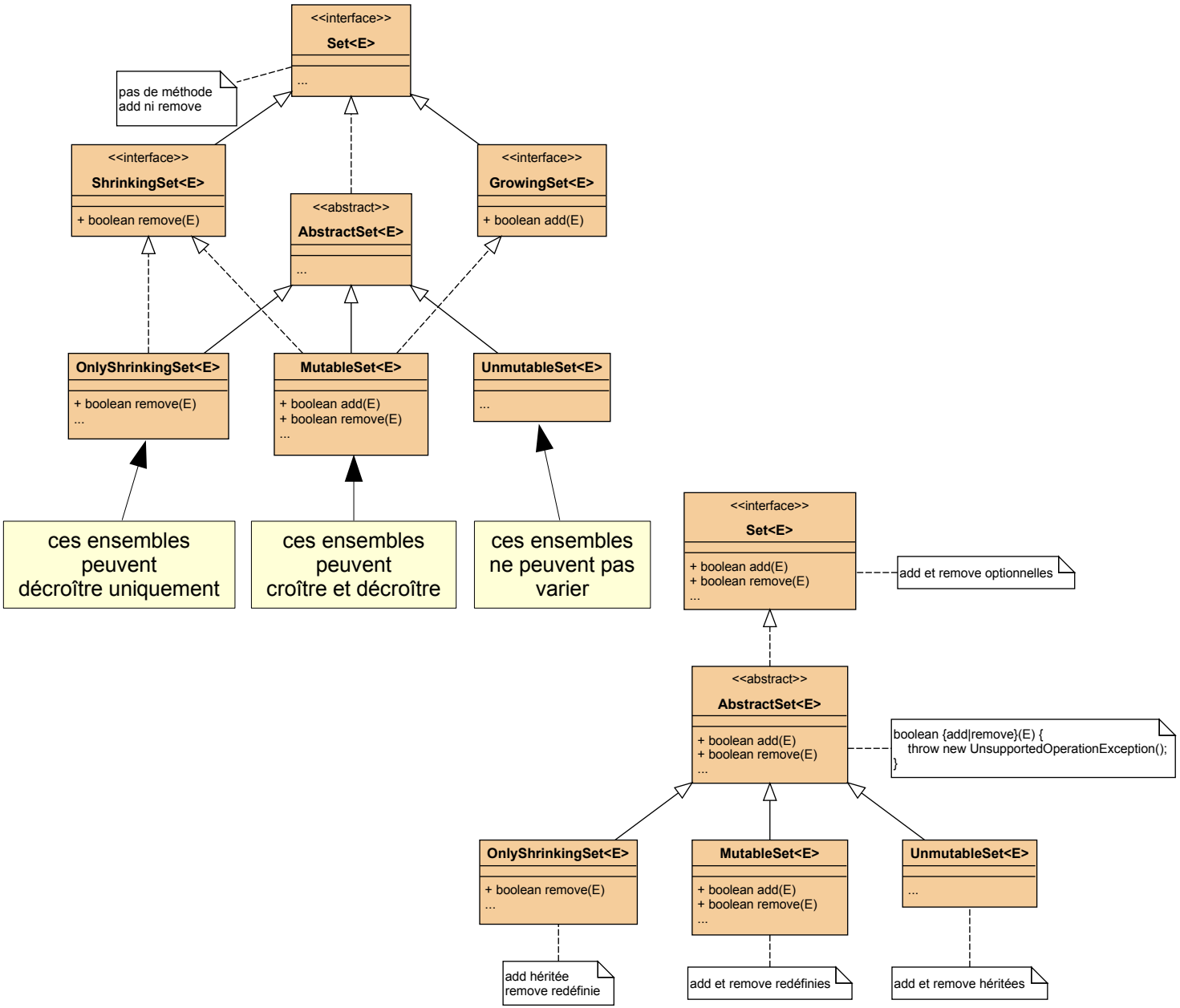
table de hachage (pas d'ordre)	+ recherche, insertion, suppression	`HashSet`
		`HashMap`
		`IdentityHashMap`

table de hachage + chaînage (ordre d'insertion)	+ recherche, insertion, suppression	`LinkedHashSet` `LinkedHashMap`
arbre rouge et noir (ordre configurable)	+ recherche, insertion, suppression	`TreeSet` `TreeMap`

Interface	Classe d'implémentation standard
`Queue` - pile - file simple	`ArrayDeque` (*via* `Collections.asLifoQueue`) `ArrayDeque`
`Deque`	`ArrayDeque`
`List`	- tableau dynamique : `ArrayList` - liste (doublement) chaînée : `LinkedList`
`Set`	- éléments quelconques : `HashSet` - éléments d'un type énuméré : `EnumSet`
`NavigableSet`	`TreeSet`
`Map`	- clés quelconques : `HashMap` - clés d'un type énuméré : `EnumMap`
`NavigableMap`	`TreeMap`

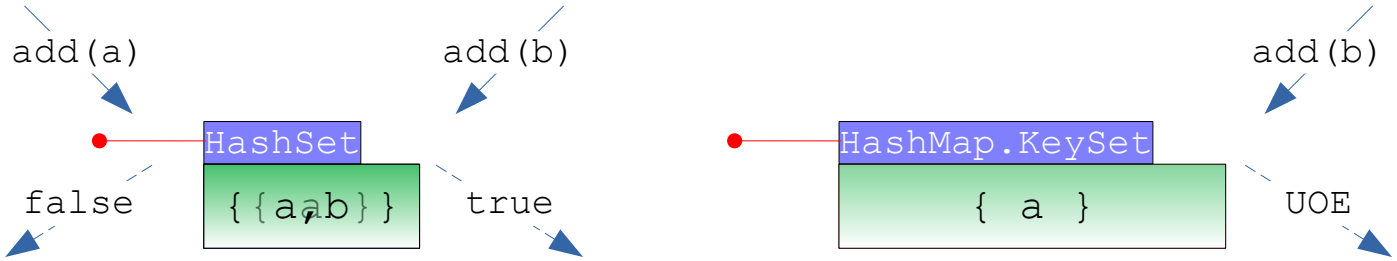
à travailler en autonomie
(lire poly + doc Java)

Généricité



Méthode optionnelle : non nécessairement supportée, elle peut
- réussir, ou
- échouer en levant une `UnsupportedOperationException`.

```
public interface Collection<E> {
    /**
     * Ajout d'un élément. Méthode optionnelle.
     * @post
     *      contains(e)
     *      result <==> la collection a changé d'état
     */
    boolean add(E e);
    ...
}
```



avantages	inconvénients
+ permet de diminuer le nombre de types	- contrats inutilisables - typage des variables avec les classes génératrices des objets (pas avec les interfaces)

pas de méthode optionnelle dans vos classes !