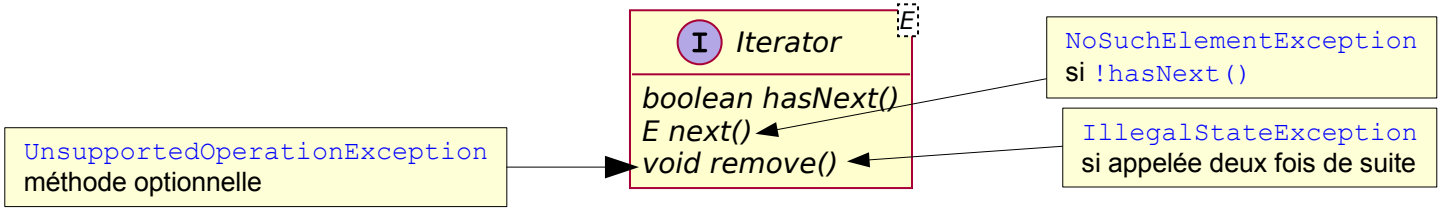
Itérateur : objet étroitement lié à une collection donnée, qui permet d'en énumérer tous les éléments indépendamment de la structure interne de cette collection.

Contrats : (pas de préconditions)
- `next()@post` : retourne l'élément courant **puis** passe au suivant
- `hasNext()@post` : `result` ⇔ il y a un prochain élément à retourner
- `remove()@post` : supprime de la collection le dernier élément retourné par `next()`



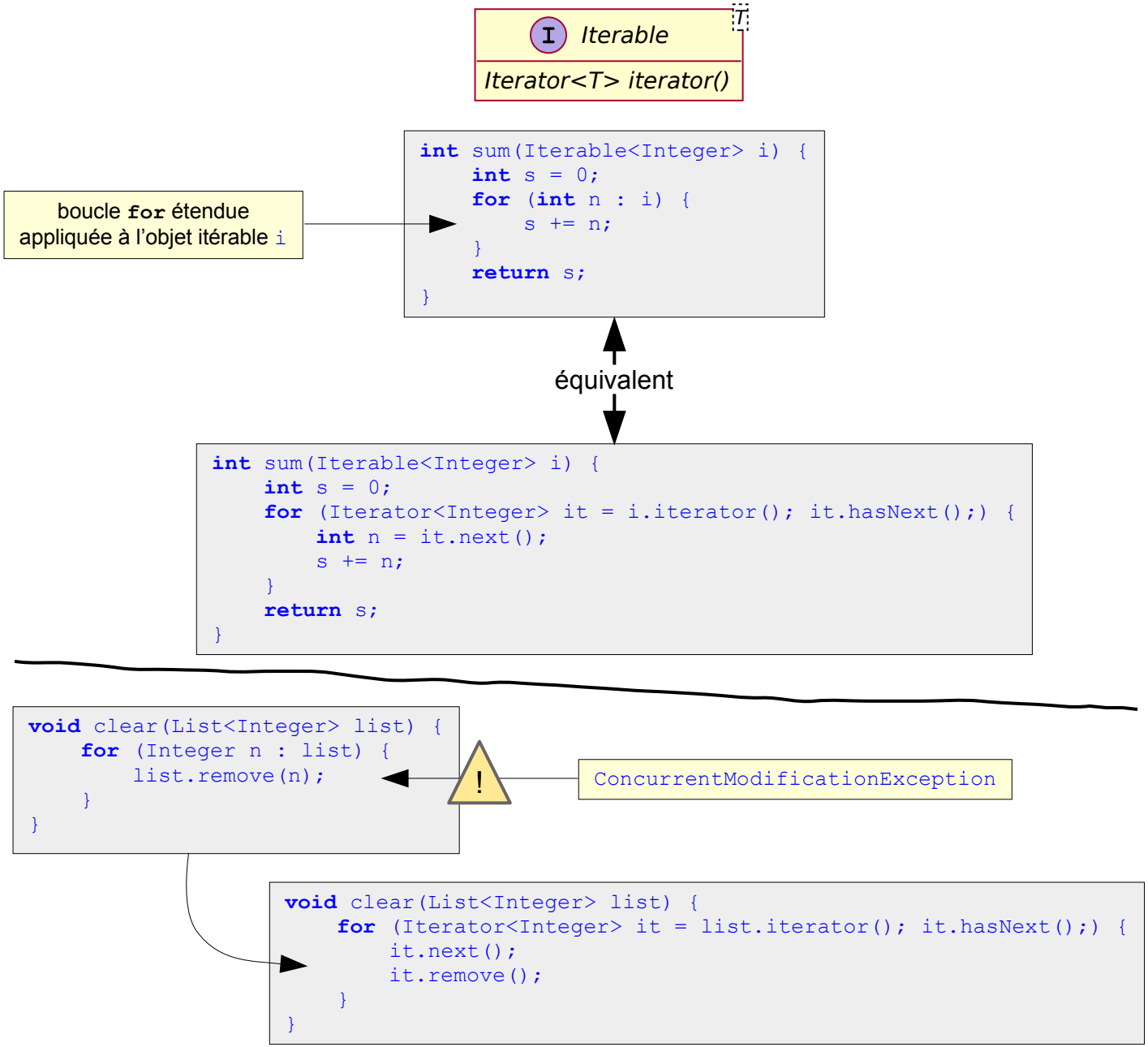

```
Collection<E> c = ...;
Iterator<E> it = c.iterator();
it.next();
c.add(...);
it.next();
...
```

co-modification structurelle externe au parcours

Type d'itérateur	Comportement lors des co-modifications	Exemples types
à échec rapide *fail-fast*	échoue dès la première co-modification (`ConcurentModificationException`)	`java.util` (sauf `EnumSet`, `EnumMap`)
à vue instantanée *snapshot*	ignore les co-modifications	`java.util.concurrent` (`CopyOnWriteArray*`)
faiblement cohérent *weakly consistent*	prend en compte certaines co-modifications et ignore les autres	`EnumSet`, `EnumMap` `java.util.concurrent`

Généricité

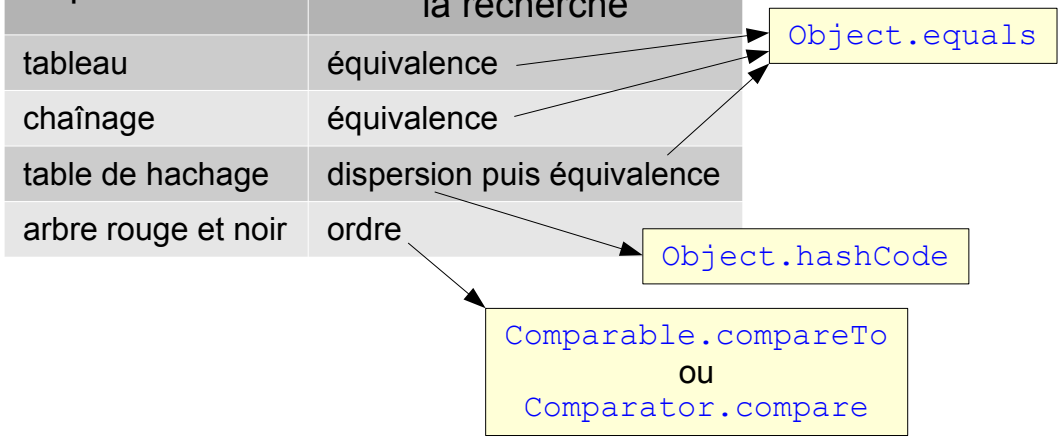
Itérable : objet capable de fournir des itérateurs
et donc utilisable comme source d'une boucle **`for`** étendue.



Généricité

Recherches dans une collection : elles se basent sur
- une relation d'équivalence dans les structures non ordonnées
- une relation d'ordre dans les structures ordonnées.

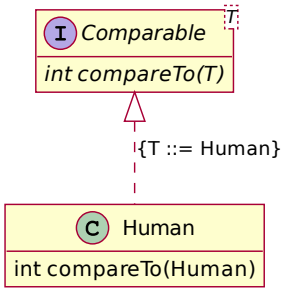
Implémentation	Relation de base pour la recherche
tableau	équivalence
chaînage	équivalence
table de hachage	dispersion puis équivalence
arbre rouge et noir	ordre



Type de collection	Méthodes impliquant une recherche
`Collection`	`contains`, `containsAll` `remove(Object)`, `removeAll`, `retainsAll`
`Queue`	*idem* `Collection`
`Deque`	*idem* `Queue` `remove{First,Last}Occurrence`
`List`	*idem* `Collection` `indexOf`, `lastIndexOf`
`Set`	*idem* `Collection`
`Map`	`containsKey`, `containsValue`, `get` `put`, `putAll`

Généricité

Ordre naturel sur C : relation d'ordre sur `C`, définie *dans* `C`, par implémentation de l'interface `Comparable<C>`.

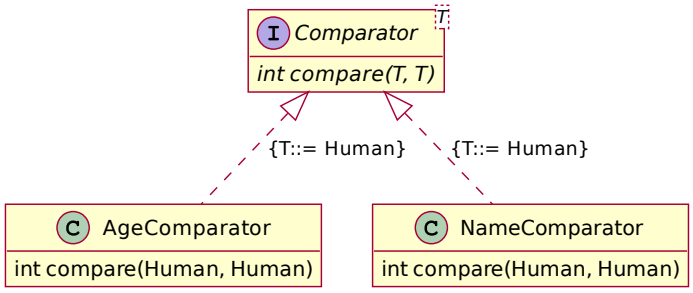



```
new TreeSet<Human>()
```

h1 ≤ h2 ⇔ `h1.compareTo(h2) <= 0`

```
class Human implements Comparable<Human> {
    public int age() { ... }
    public String name() { ... }
    ...
    public int compareTo(Human other) {
        if (other == null) {
            throw new NullPointerException();
        }
        int result = this.name().compareTo(other.name());
        if (result == 0) {
            result = this.age() < other.age()
                ? -1
                : this.age() == other.age() ? 0 : +1;
        }
        return result;
    }
}
```

Ordre (quelconque) sur C : relation d'ordre sur `C`, définie *dans une classe* `X`, par implémentation de l'interface `Comparator<C>`.




```
new TreeSet<Human>(Comparator<Human>)
```

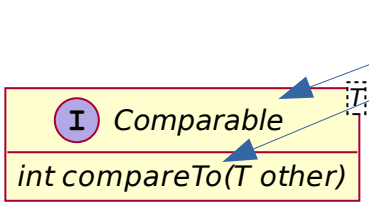
h1 ≤$_q$ h2 ⇔ `q.compare(h1, h2) <= 0`

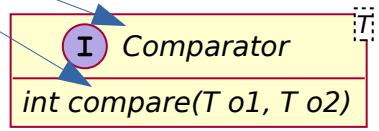
```
class AgeComparator implements Comparator<Human> {
    public int compare(Human h1, Human h2) {
        if ((h1 == null) || (h2 == null)) {
            throw new NullPointerException();
        }
        return h1.age() < h2.age()
            ? -1
            : h1.age() == h2.age() ? 0 : +1;
    }
}
```

```
class NameComparator implements Comparator<Human> {
    public int compare(Human h1, Human h2) {
        if ((h1 == null) || (h2 == null)) {
            throw new NullPointerException();
        }
        return h1.name().compareTo(h2.name());
    }
}
```

Généricité

attention aux noms !



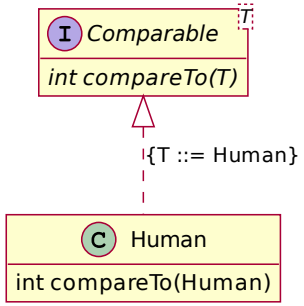


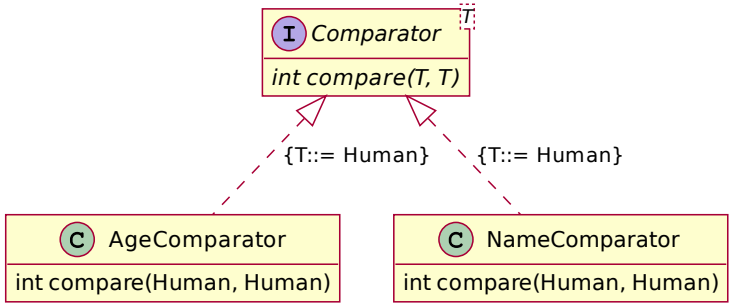

```
@pre -
@post
    this plus petit que other ==> result  < 0
    this équivalent à other   ==> result == 0
    this plus grand que other ==> result  > 0
@Throws NPE si other == null
```

```
@pre -
@post
    o1 plus petit que o2 ==> result  < 0
    o1 équivalent à o2   ==> result == 0
    o1 plus grand que o2 ==> result  > 0
@Throws NPE si o1 == null || o2 == null
```

```
@inv forall x, y, z in T :
  sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
  x.compareTo(y) < 0 && y.compareTo(z) < 0
      ==> x.compareTo(z) < 0
  x.compareTo(y) == 0
      ==> sgn(x.compareTo(z)) == sgn(y.compareTo(z))
```

```
@inv forall x, y, z in T :
  sgn(compare(x, y)) == -sgn(compare(y, x))
  compare(x, y) < 0 && compare(y, z) < 0
      ==> compare(x, z) < 0
  compare(x, y) == 0
      ==> sgn(compare(x, z)) == sgn(compare(y, z))
```





Généricité

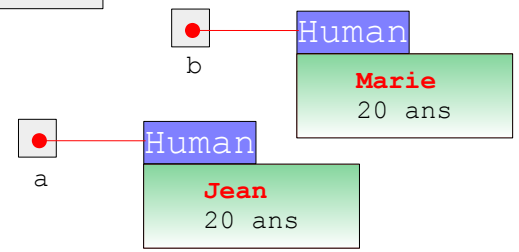
`Comparable.compareTo` ***doit être cohérente*** avec `equals` :
`a.compareTo(b) == 0` ⇔ `a.equals(b)`

```
class Human implements Comparable<Human> {
    public int age() { ... }
    public String name() { ... }
    ...
    public int compareTo(Human other) {
        if (other == null) {
            throw new NullPointerException();
        }
        return this.age() < other.age()
                ? -1
                : this.age() == other.age() ?
    }
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Human) {
            Human that = (Human) other;
            result = that.canEquals(this)
                && this.name().equals(that.name())
                && this.age() == that.age();
        }
        return result;
    }
    public boolean canEquals(Object other) {
        return other instanceof Human;
    }
}
```

```
public int compareTo(Human other) {
    if (other == null) {
        throw new NullPointerException();
    }
    int result = this.name().compareTo(other.name());
    if (result == 0) {
        result = this.age() < other.age()
            ? -1
            : this.age() == other.age() ? 0 : +1;
    }
    return result;
}
```

```
Set<Human> x = new TreeSet<Human>();
System.out.println(x.add(a));
System.out.println(x.add(b));
```

→ `true`
→ `?`



	Contrat de Set `a.equals(b) == false`	Contrat de TreeSet `a.compareTo(b) == 0`	Contrat de TreeSet `a.compareTo(b) == -3`
`x.add(a)`	**true**	**true**	**true**
`x.add(b)`	**true**	**false**	**true**

Généricité

`Comparator.compare` ***peut ne pas être cohérente*** avec `equals`.

```
/**
 * Pour ce comparateur on a seulement :
 *     p1.equals(p2) ==> compare(p1, p2) == 0
 */
class NameComparator implements Comparator<Human> {
    public int compare(Human p1, Human p2) {
        if ((p1 == null) || (p2 == null)) {
            throw new NullPointerException();
        }
        return p1.name().compareTo(p2.name());
    }
}
```

préciser obligatoirement l'incohérence avec `equals`

```
/**
 * Pour ce comparateur on a seulement :
 *     p1.equals(p2) ==> compare(p1, p2) == 0
 */
class AgeComparator implements Comparator<Human> {
    public int compare(Human p1, Human p2) {
        if ((p1 == null) || (p2 == null)) {
            throw new NullPointerException();
        }
        return p1.age() < p2.age()
            ? -1
            : p1.age() == p2.age() ? 0 : +1;
    }
}
```

... et les utiliser avec précaution !

~~`Set<Human> x = new TreeSet<Human>(new NameComparator());`~~

```
TreeSet<Human> x = new TreeSet<Human>(new NameComparator());
```

Contrainte générique : notation placée dans la déclaration d'un paramètre de type, restreignant les substitutions possibles.