

Généricité

I. Types génériques

1. Généricité / Polymorphisme
2. Avantages de la généricité
3. Type générique / paramétré
4. Effacement de la généricité
5. Type brut
6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale
8. Transtypage
9. Méthode pont

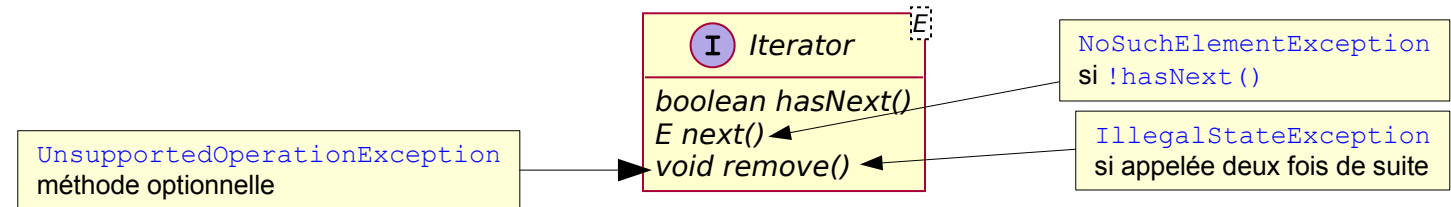
II. Collection génériques

1. Présentation
2. Implémentation
3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - c. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

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` \Leftrightarrow 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 <i>fail-fast</i>	échoue dès la première co-modification (<code>ConcurrentModificationException</code>)	<code>java.util</code> (sauf <code>EnumSet</code> , <code>EnumMap</code>)
à vue instantanée <i>snapshot</i>	ignore les co-modifications	<code>java.util.concurrent</code> (<code>CopyOnWriteArray*</code>)
faiblement cohérent <i>weakly consistent</i>	prend en compte certaines co-modifications et ignore les autres	<code>EnumSet</code> , <code>EnumMap</code> <code>java.util.concurrent</code>

Généricité

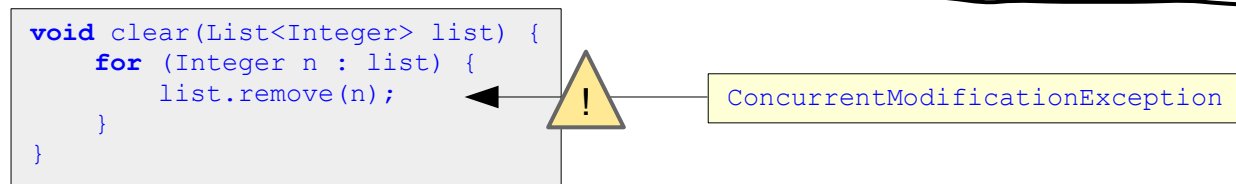
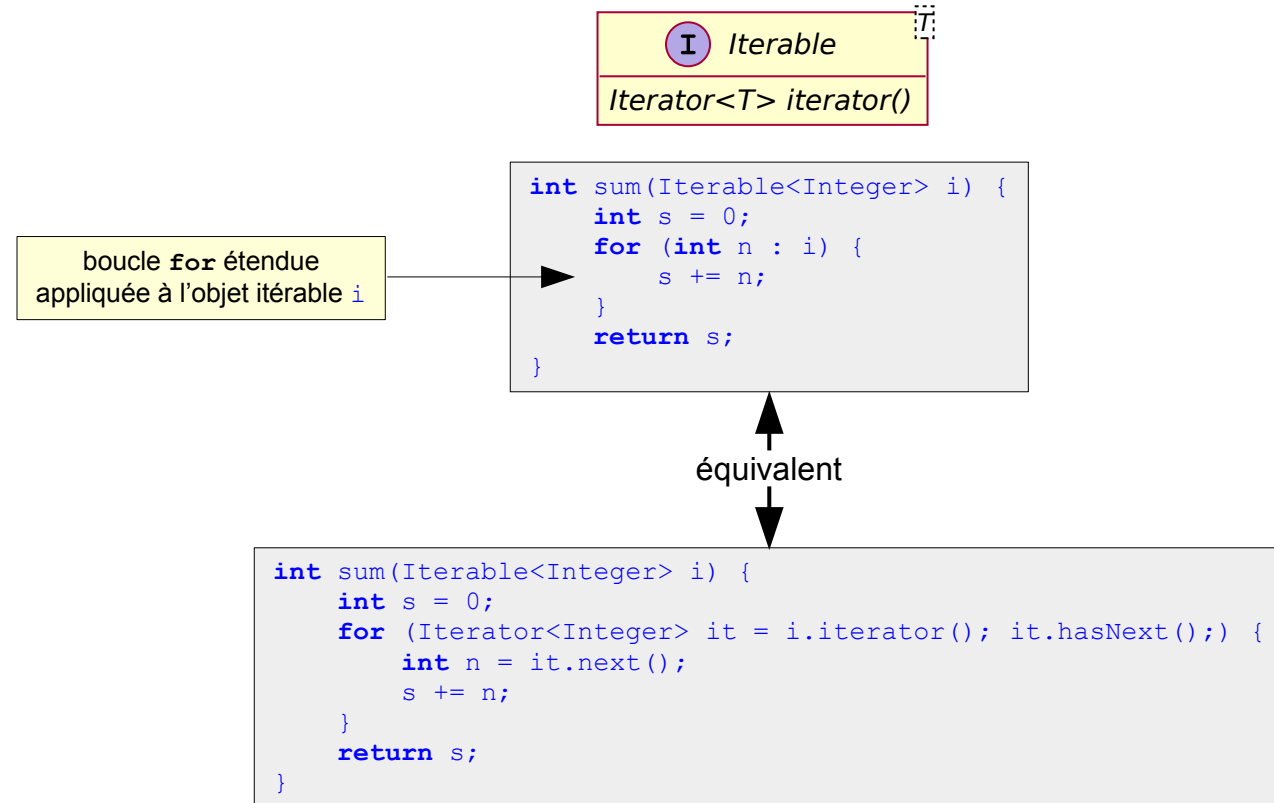
I. Types génériques

1. Généricité / Polymorphisme
2. Avantages de la généricité
3. Type générique / paramétré
4. Effacement de la généricité
5. Type brut
6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale
8. Transtypage
9. Méthode pont

II. Collection génériques

1. Présentation
2. Implémentation
3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - iii. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

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



The diagram shows a corrected version of the `clear` method using an explicit iterator:

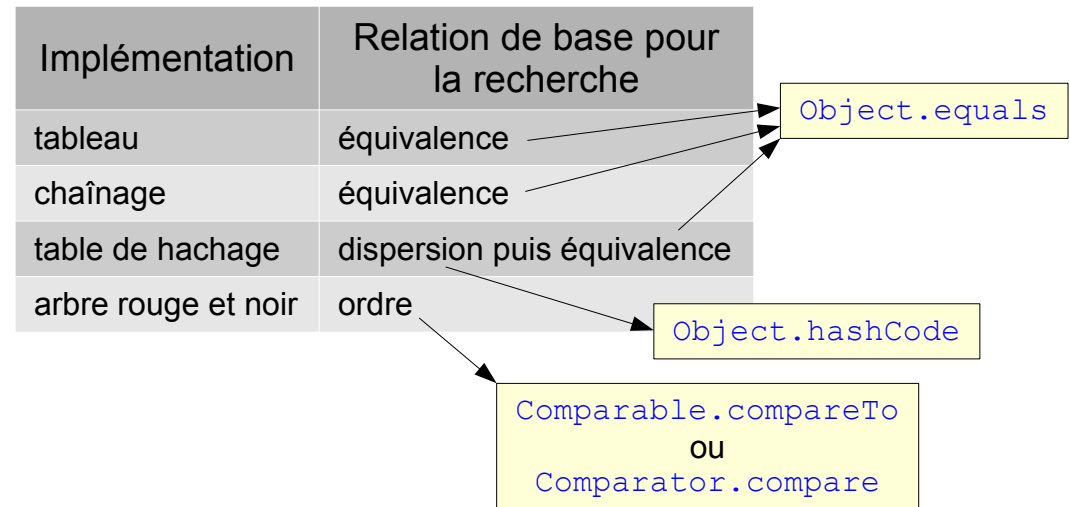
```
void clear(List<Integer> list) {
    for (Iterator<Integer> it = list.iterator(); it.hasNext();) {
        it.next();
        it.remove();
    }
}
```

Généricité

- I. Types génériques
 1. Généricité / Polymorphisme
 2. Avantages de la généricité
 3. Type générique / paramétré
 4. Effacement de la généricité
 5. Type brut
 6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
 7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale
 8. Transtypage
 9. Méthode pont
- II. Collection génériques
 1. Présentation
 2. Implémentation
 3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
 4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
 5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - c. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

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.



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

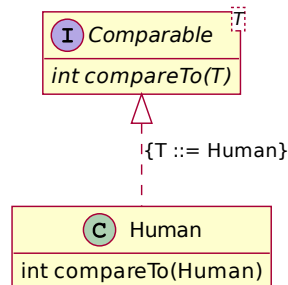
I. Types génériques

1. Généricité / Polymorphisme
2. Avantages de la généricité
3. Type générique / paramétré
4. Effacement de la généricité
5. Type brut
6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale
8. Transtypage
9. Méthode pont

II. Collection génériques

1. Présentation
2. Implémentation
3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - c. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

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



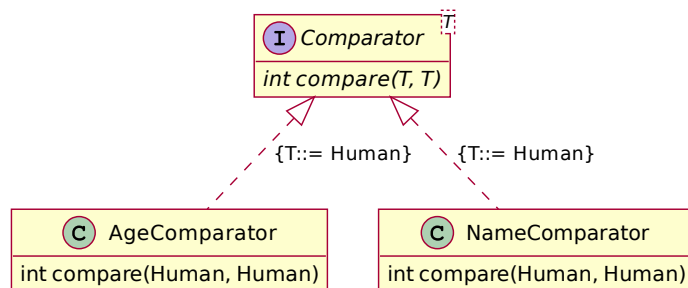
```
new TreeSet<Human>()
```

```
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

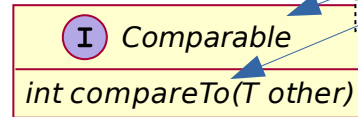
I. Types génériques

1. Généricité / Polymorphisme
2. Avantages de la généricité
3. Type générique / paramétré
4. Effacement de la généricité
5. Type brut
6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale
8. Transtypage
9. Méthode pont

II. Collection génériques

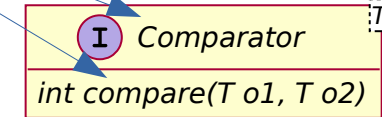
1. Présentation
2. Implémentation
3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - c. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

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
    
```



```

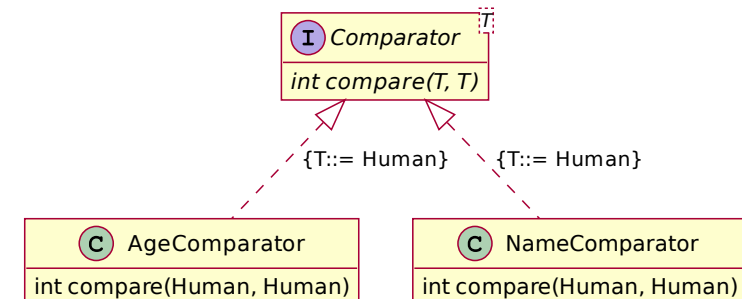
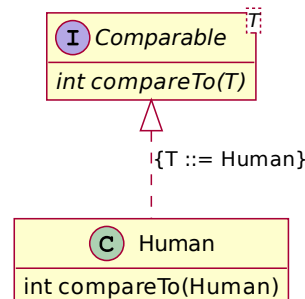
@pre -
@post
    o1 plus petit que o2 ==> result < 0
    o1 équivalent à o2 ==> result == 0
    o1 plus grand que o2 ==> result > 0
@Throws NPE si o1 == null || o2 == null
    
```

```

@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

I. Types génériques

1. Généricité / Polymorphisme
2. Avantages de la généricité
3. Type générique / paramétré
4. Effacement de la généricité
5. Type brut
6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale

II. Collection génériques

1. Présentation
2. Implémentation
3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - c. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

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

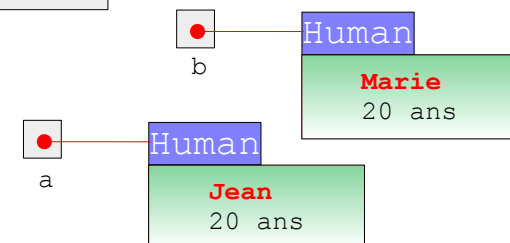
```
class Human implements Comparable<Human> {
    public int age() { ... }
    public String name() { ... }
    ...
    public int compareTo(Human other) {
        if (other == null) {
            throw new NullPointerException();
        }
        return this.age() < other.age()
            ? -1
            : 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 <code>a.equals(b) == false</code>	Contrat de TreeSet <code>a.compareTo(b) == 0</code>	Contrat de TreeSet <code>a.compareTo(b) == -3</code>
<code>x.add(a)</code>	true	true	true
<code>x.add(b)</code>	true	false	true

Généricité

I. Types génériques

1. Généricité / Polymorphisme
2. Avantages de la généricité
3. Type générique / paramétré
4. Effacement de la généricité
5. Type brut
6. Usage du paramètre
 - i. Contexte statique
 - ii. Contexte non statique
7. Héritage et sous-typage
 - i. Compatibilité verticale
 - ii. Incompatibilité horizontale
8. Transtypage
9. Méthode pont

II. Collection génériques

1. Présentation
2. Implémentation
3. Méthode optionnelle
 - i. Motivation
 - ii. Définition
4. Parcours d'une collection
 - i. Itérateurs
 - ii. Itérables
5. Recherche dans une collect°
 - i. Types de recherches
 - ii. Relations d'ordre
 - a. Ordres dans une collect°
 - b. Interfaces de comparaison
 - c. Cohérence ordre / équiv.
 1. Pour l'ordre naturel
 2. Pour un autre ordre

`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());
    }
}
```

```
/**
 * 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;
    }
}
```

préciser obligatoirement
l'**incohérence** avec `equals`

... et les utiliser
avec précaution !

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

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


Généricité

III. Généricité contrainte

1. Contrainte générique
2. Bornes multiples

IV. Joker

1. Définition
2. Bornes du joker
3. Types à joker borné
4. Sous-typage
5. Instanciation des types paramétrés

V. Méthodes génériques

1. Motivation
 2. Définition
 3. Inférence des paramètres de types
 - i. Algorithme d'inférence
 - ii. Exemple 1
 - iii. Exemple 2
 - iv. Exemple 3
 4. Capture du joker
 - i. Motivation et définition
 - ii. Exemples
 5. Règles méthodologiques
 - i. Méthode générique ou à joker ?
 - ii. Utilisation des jokers bornés
 - a. Règles 1 & 2
 - b. Règles 3 & 4
- ### VI. Compléments
1. Tableaux et généricité
 2. Exceptions et généricité
 3. Classe Class<E>
 4. Types énumérés
 - i. Classe Enum<E>
 - ii. Entête de la classe Enum
 - iii. Exemple élaboré

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

