

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

Rappel : *capture de joker*

lors d'un appel de méthode générique, pendant l'inférence de types, si E est de la forme $H<?>$ et si F est de la forme $G<T>$ alors le joker est remplacé par un nouveau nom de type de la forme $\text{capture}\#n$ et $T ::= \text{capture}\#n$.



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

Il faut *utiliser un paramètre de type* quand on veut exprimer une dépendance entre le type des arguments et celui de la valeur retournée.



```
<T> Set<T> convertToSet(List<T> lst)
```

retourne un ensemble d'éléments
du même type que ceux de `lst`



```
Set<?> convertToSet(List<?> lst)
```

ne garantit pas que les éléments de l'ensemble
soient du même type que ceux de `lst`

Il faut *préférer un type à joker* quand on veut exprimer la flexibilité du type des arguments.



```
public interface Collection<E> extends Iterable<E> {  
    boolean containsAll(Collection<?> c);  
    ...  
}
```

indépendance du type des arguments
avec celui de la valeur retournée

variabilité du type des arguments



```
public interface Collection<E> extends Iterable<E> {  
    <T> boolean containsAll(Collection<T> c)  
    ...  
}
```

possible aussi, mais
plus verbeux et donc moins clair

Généricité

III. Généricité contrainte

1. Contrainte générique
2. Bornes multiples

IV. Joker

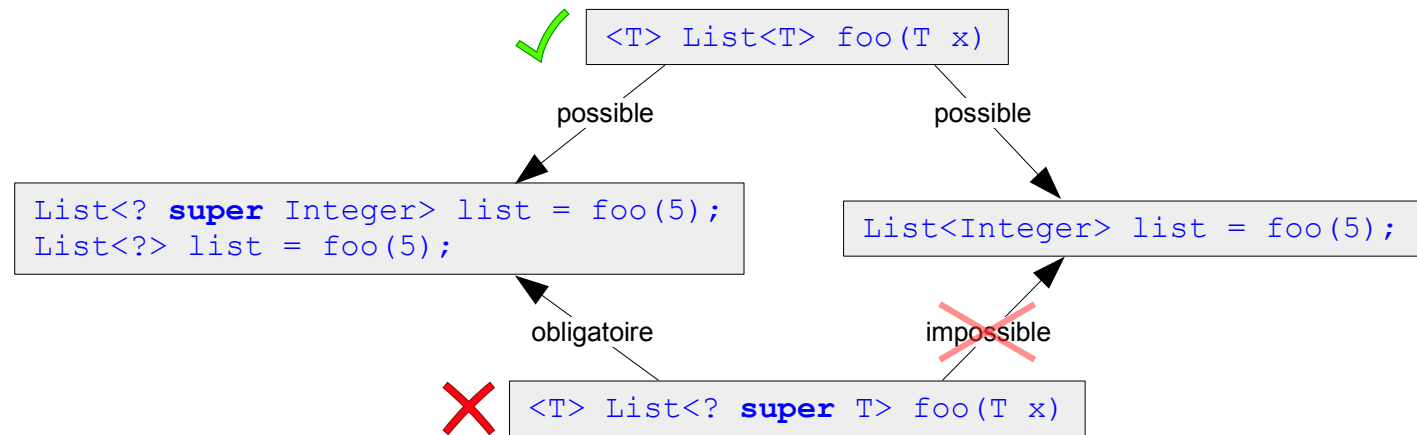
1. Définition
2. Bornes du joker
3. Types à joker borné
4. Sous-typage
5. Instanciation des types paramétrés

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

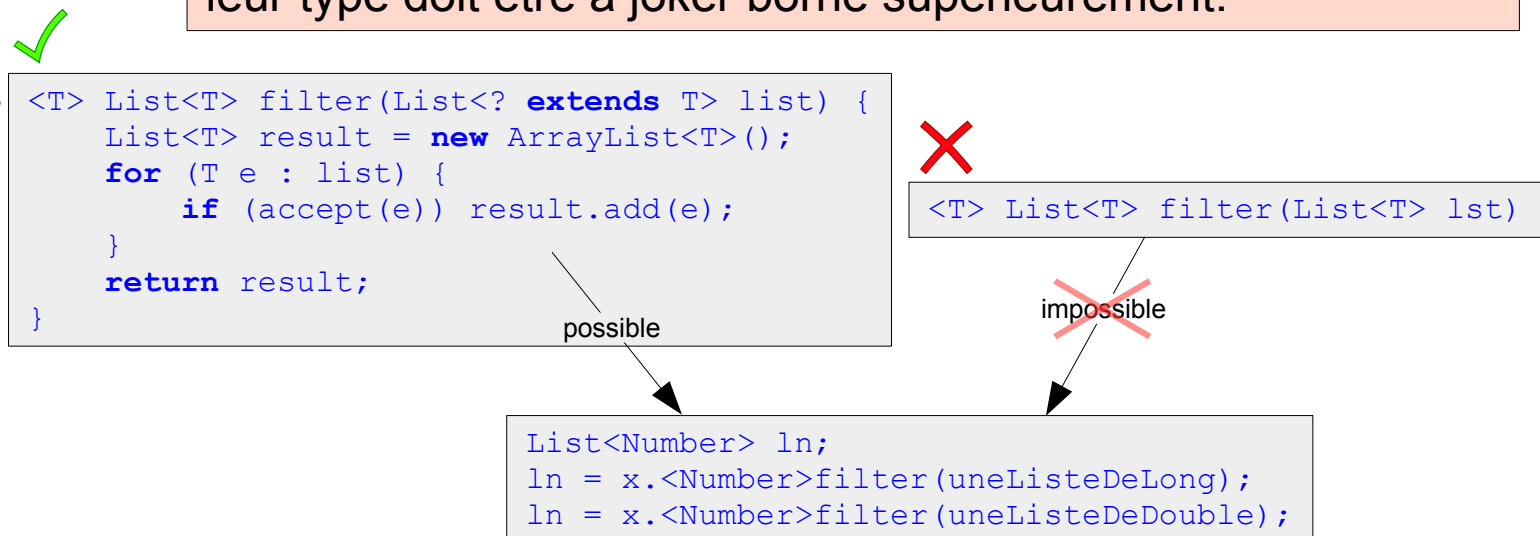
Utilisation des jokers bornés – Règle 1 :

Le type de la valeur retournée ne doit pas être un type à joker borné.



Utilisation des jokers bornés – Règle 2 :

Pour les arguments dont le rôle est “producteur de données”, leur type doit être à joker borné supérieurement.



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

Utilisation des jokers bornés – Règle 3 :

Pour les arguments dont le rôle est “consommateur de données”, leur type doit être à joker borné inférieurement.



```
<T> void fill(List<? super T> lst, int n, T e) {  
    for (int i = 0; i < n; i++) {  
        lst.add(e);  
    }  
}
```

possible



```
<T> void fill(List<T> lst, int n, T e)
```

impossible

```
List<Number> ln = ...;  
x.<Integer>fill(ln, 5, 3);  
x.<Double>fill(ln, 5, 3.14);
```

Utilisation des jokers bornés – Règle 4 :

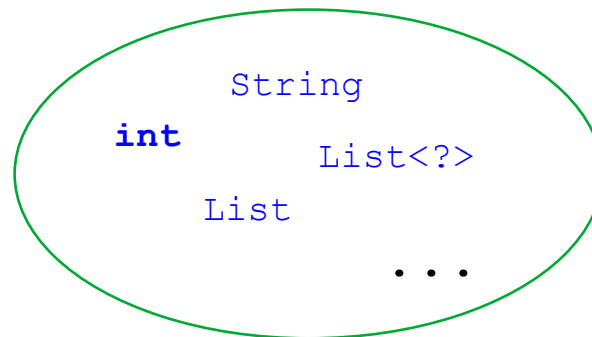
Pour les arguments qui jouent simultanément les deux rôles, leur type ne doit pas être à joker.

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

Type réifiable :

Type qui ne perd pas d'information durant la phase d'effacement.



~~List<String> T~~
~~List<Object> ...~~
~~List<? extends Number>~~
~~List<? extends Object>~~

Type tableau instanciable : Un type tableau est instanciable si ses éléments de base sont d'un type réifiable.

~~new G<T>[2]~~

Exercice traité en TD...

Supposons valide la classe suivante

```
class G<T> {  
    G<T>[] m() {  
        G<T>[] x = new G<T>[2];  
        return x;  
    }  
    T p() { ... }  
}
```

⚠ code incorrect

Montrer que le test suivant passerait la compilation mais lèverait une exception à l'exécution (laquelle et où ?)

```
G<Integer> gi = new G<Integer>();  
G<Integer>[] tgi = gi.m();  
Object[] tab = tgi;  
G<String> gs = new G<String>();  
tab[0] = gs;  
Integer n = tgi[0].p();
```

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

Un type d'exception ne peut pas être générique.

```
class Exc<T> extends Exception { ... }
```

```
try {  
    gt.test(exc);  
} catch (Exc<Integer> e) {  
    ...  
} catch (Exc<String> e) {  
    ...  
}
```

⚠ code incorrect

effacement

Impossible !

```
try {  
    gt.test(exc);  
} catch (Exc e) {  
    ...  
} catch (Exc e) {  
    ...  
}
```

Généricité

III. Généricité contrainte

1. Contrainte générique
2. Bornes multiples

IV. Joker

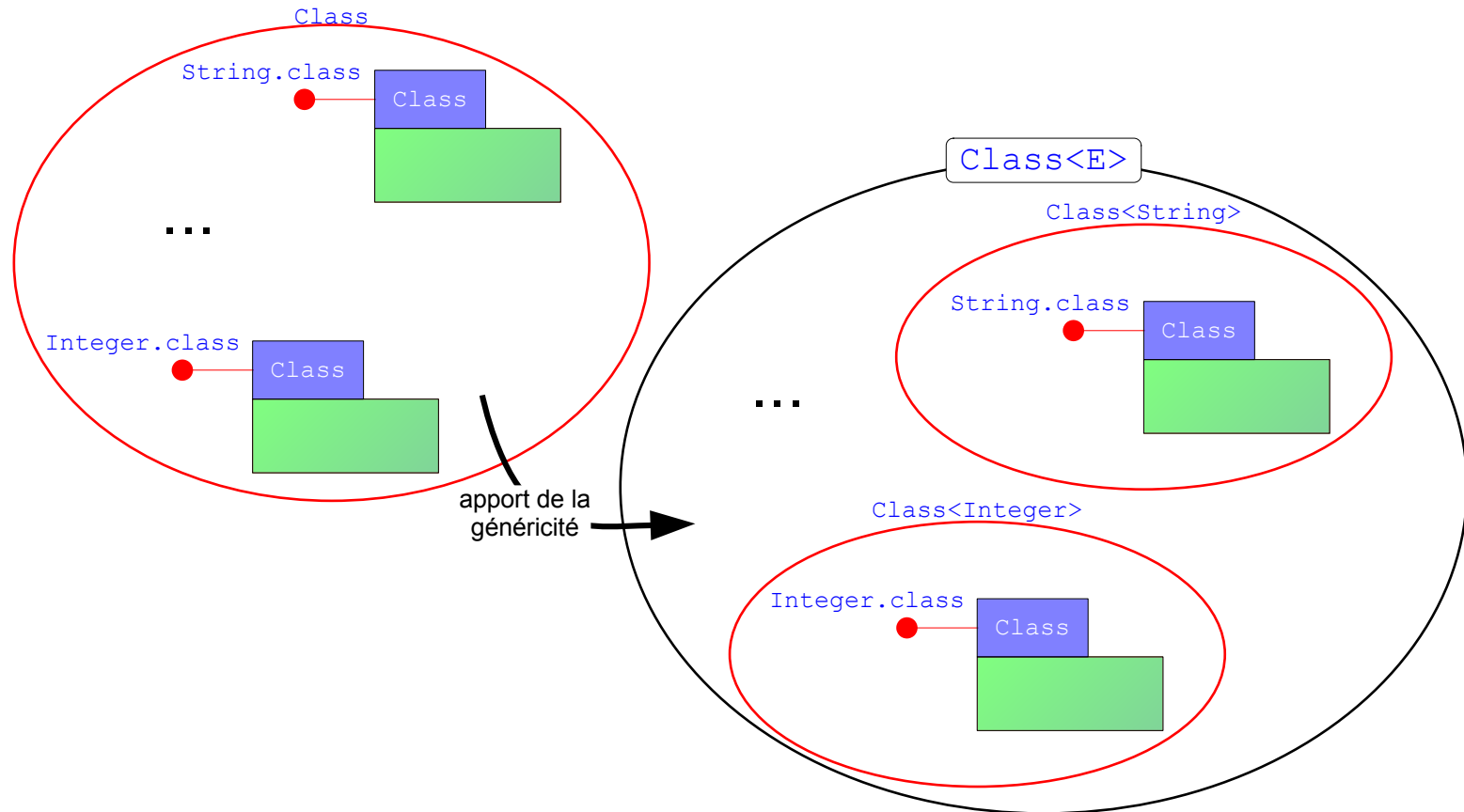
1. Définition
2. Bornes du joker
3. Types à joker borné
4. Sous-typage
5. Instanciation des types paramétrés

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



Règle de compilation :

`CT(e.getClass()) = Class<? extends |CT(e)||>`

```
public class Object {
    public final Class<?> getClass();
    ...
}
```

```
List<String> ls = new ArrayList<String>();
Class<? extends List> c = ls.getClass();
```

le compilateur tolère l'absence de transtypage !

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

```
enum Civ { UKN, MR, MRS, MS }
```

classe
synthétisée

```
final class Civ extends Enum<Civ> {
    public static final Civ UKN;
    public static final Civ MR;
    public static final Civ MRS;
    public static final Civ MS;

    private static final Civ[] ENUM_VALUES;
    static {
        UKN = new Civ("UKN", 0);
        MR = new Civ("MR", 1);
        MS = new Civ("MRS", 2);
        MRS = new Civ("MS", 3);
        ENUM_VALUES = new Civ[] {
            UKN, MR, MRS, MS
        };
    }

    public static Civ[] values() {
        // retourne une copie de ENUM_VALUES
    }

    public static Civ valueOf(String n) {
        // retourne la constante de nom n
        // ou null sinon
    }
}
```

sérialisation : codage d'un objet sous la forme d'une suite d'octets pour le sauvegarder dans un flux (disque, réseau, ...).

```
public abstract class Enum<E> extends Enum<E>>
    extends Object
    implements Comparable<E>, Serializable
```

ce que l'on veut
comprendre dans
cet entête

```
...class Enum<E> extends Enum<E>>...implements Comparable<E>...
```


Généricité

III. Généricité contrainte

1. Contrainte générique
2. Bornes multiples

IV. Joker

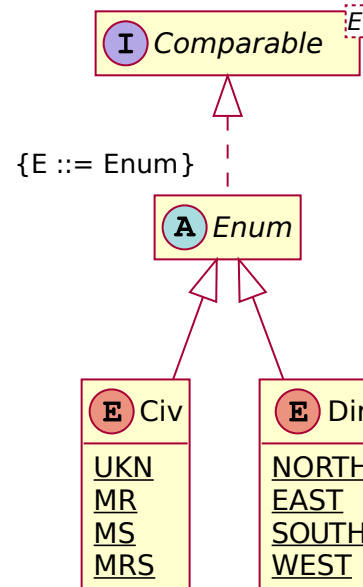
1. Définition
2. Bornes du joker
3. Types à joker borné
4. Sous-typage
5. Instanciation des types paramétrés

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



`class Enum implements Comparable<Enum>`

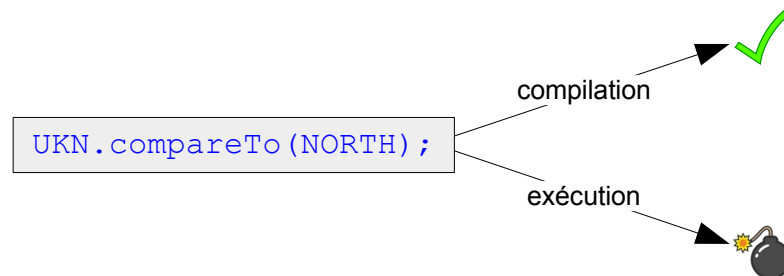
incorrect !

```

final class Civ extends Enum {
    public int compareTo(Enum other) {
        if (!(other instanceof Civ)) {
            throw new IllegalArgumentException();
        }
        ...
    }
    ...
}
    
```

```

final class Dir extends Enum {
    public int compareTo(Enum other) {
        if (!(other instanceof Dir)) {
            throw new IllegalArgumentException();
        }
        ...
    }
    ...
}
    
```



un élément ne devrait pouvoir être comparé **que** avec un autre élément de même type

Généricité

III. Généricité contrainte

1. Contrainte générique
2. Bornes multiples

IV. Joker

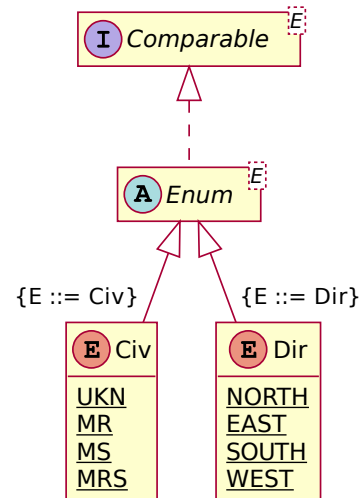
1. Définition
2. Bornes du joker
3. Types à joker borné
4. Sous-typage
5. Instanciation des types paramétrés

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

class Enum<E> implements Comparable<E>

incorrect !



```

class Enum<E> implements Comparable<E> {
    public int compareTo(E other) {
        ...
        return other.ordinal() - this.ordinal();
    }
    ...
}
    
```

compilation



ordinal() n'est pas définie dans le type E
E ~~<~~ Enum<E>

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

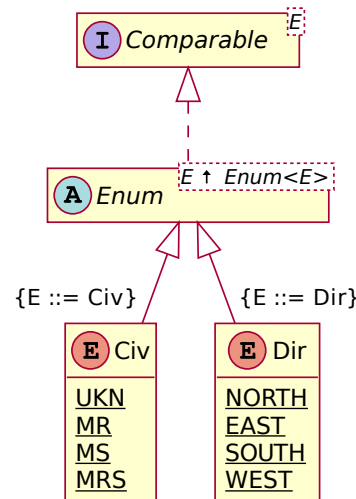
V. Méthodes génériques

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

class Enum<E **extends** Enum<E>> **implements** Comparable<E>



```
class Enum<E extends Enum<E>> implements Comparable<E> {
    public int compareTo(E other) {
        ...
        return other.ordinal() - this.ordinal();
    }
    ...
}
```

compilation



mais il reste encore le pb suivant

```
class Civ extends Enum<Civ> { ... }
class Dir extends Enum<Civ> { ... }
```



Règle de définition des types énumérés :
Seules les classes définies avec le mot clé **enum** ont la possibilité de dériver Enum.
Leur code est synthétisé par le compilateur.

```
enum Civ { ... }
enum Dir { ... }
```

classes
synthétisées

```
final class Civ extends Enum<Civ> { ... }
final class Dir extends Enum<Dir> { ... }
```



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

```
Double computeValue(String[] tokens) {
    Contract.checkCondition(tokens != null && tokens.length > 0);

    Queue<Double> stack = Collections.asLifoQueue(new ArrayDeque<Double>());
    for (String t : tokens) {
        Operator op = Operator.valueOfSymbol(t);
        if (op != null) {
            double v2 = stack.element();
            stack.remove();
            double v1 = stack.element();
            stack.remove();
            stack.add(op.eval(v1, v2));
        } else {
            Double v = Double.valueOf(t);
            stack.add(v);
        }
    }
    return stack.element();
}
```

```
String[] tokens = new String[] { "2", "4", "+", "7", "*" };
Double val = obj.computeValue(tokens);
assert val == 42;
```

```
enum Operator {
    PLUS("+") {double eval(double x,double y){return x+y;}},
    MINUS("-") {double eval(double x,double y){return x-y;}},
    TIMES("*") {double eval(double x,double y){return x*y;}},
    DIVIDED_BY("/") {double eval(double x,double y){return x/y;}};
    private static final Map<String, Operator> OPERATORS;
    static {
        OPERATORS = new HashMap<String, Operator>();
        for (Operator op : Operator.values()) {
            OPERATORS.put(op.symbol, op);
        }
    }
    private final String symbol;
    Operator(String s) { symbol = s; }
    abstract double eval(double x, double y);
    public String toString() { return symbol; }
    static Operator valueOfSymbol(String c) {
        return OPERATORS.get(c);
    }
}
```

