

# La généricité

(d'après le cours de Philippe GENOUD )



Cours de Programmation Orientée Objet Avancée

ISET Bizerte

CHALOUAH Anissa

# Généricité : motivation

- ▶ Supposons que l'on développe du code pour gérer une file d'attente (FIFO First In First Out) et que l'on veuille utiliser ce code pour
  - ▷ *une file d'entiers*
  - ▷ *une file de chaînes de caractères (String)*
  - ▷ *une file d'objets Personne*

## Comment procéder ?



# Généricité : motivation

## 1<sup>ère</sup> solution

Écrire une classe pour chaque type de valeur que l'on peut mettre dans la file



# Généricité : motivation

```
class ElementInt {  
    private final int value;  
    private ElementInt next;  
  
    ElementInt(int val) {  
        this.value = val; next = null; }  
  
    void inserer(ElementInt elt) {  
        this.next = elt; }  
  
    int getValue() { return value; }  
  
    ElementInt getNext() { return next; }  
}
```

```
public class FileAttenteInt {  
  
    private ElementInt tête = null;  
    private ElementInt queue = null;  
  
    public void ajouter(int val) {  
        ElementInt elt = new ElementInt(val);  
        if (estVide()) { tête = queue = elt; }  
        else { queue.inserer(elt); queue = elt; }  
    }  
  
    public boolean estVide() { return tête == null; }  
  
    public int retirer() {  
        int val = -1;  
        if (!estVide()) {  
            val = tête.getValue();  
            tête = tête.getNext();  
        }  
        return val; }  
}
```

# Généricité : motivation

```
class ElementPersonne {  
  
    private final Personne value;  
    private ElementPersonne next;  
  
    ElementPersonne(Personne val) {  
        this.value = val; next = null; }  
  
    void inserer(ElementPersonne elt) {  
        this.next = elt; }  
  
    int getValue() { return value; }  
  
    ElementPersonne getNext() { return next; }  
}
```

```
public class FileAttentePersonne {  
  
    private ElementPersonne tête = null;  
    private ElementPersonne queue = null;  
  
    public void ajouter(Personne val) {  
        ElementPersonne elt = new ElementPersonne(val);  
        if (estVide()) { tête = queue = elt; }  
        else { queue.inserer(elt); queue = elt; }  
    }  
  
    public boolean estVide() { return tête == null; }  
  
    public int retirer() {  
        int val = -1;  
        if (!estVide()) {  
            val = tête.getValue();  
            tête = tête.getNext();  
        }  
        return val; }  
}
```

# Généricité : motivation

► **1ère solution** : écrire une classe pour chaque type de valeur que l'on peut mettre dans la file

- ❌ Duplication du code, source d'erreurs à l'écriture et lors de modifications du programme
- ❌ Nécessité de prévoir toutes les combinaisons possibles pour une application

# Généricité : motivation

## 2<sup>ème</sup> solution

Utiliser un type "universel", **Object** en Java



# Généricité : motivation

```
class Element {  
  
    private final Object value;  
    private Element next;  
  
    Element(Object val) {  
        this.value = val; next = null;  
    }  
  
    void inserer(Element elt) {  
        this.next = elt; }  
  
    public Object getValue() {  
        return value; }  
  
    public Element getNext() {  
        return next; }  
}
```

*Toute classe héritant de **Object**, il est possible d'utiliser FileAttente pour stocker n'importe quel type d'objet*

```
public class FileAttente {  
  
    private Element tête = null;  
    private Element queue = null;  
  
    public void ajouter(Object val) {  
        Element elt = new Element(val);  
        if (estVide()) { tête = queue = elt; }  
        else { queue.inserer(elt); queue = elt; } }  
  
    public boolean estVide() { return tête == null; }  
  
    public Object retirer() {  
        Object val = null;  
        if (!estVide()) {  
            val = tête.getValue();  
            tête = tête.getNext();  
        }  
        return val; }  
}
```



# Généricité : motivation

► 2ème solution : utiliser un type "universel", **Object** en Java

```
FileAttente f1 = new FileAttente();  
f1.ajouter(new Personne(...));
```

```
FileAttente f2 = new FileAttente();  
f2.ajouter(new Voiture(...));
```

# Généricité : motivation

```
FileAttente f1 = new FileAttente();  
f1.ajouter(new Personne(...));  
...  
// on veut récupérer le nom de la personne  
// en tête de file.  
String nom = (Personne) (f1.retirer()).getNom();
```

Transtypage

Object

```
f1.ajouter("Hello");
```

```
String nom = (Personne) (f1.retirer()).getNom();
```

ClassCastException

- code lourd, moins lisible, plus difficile à maintenir
- risques d'erreurs d'exécution.



obligation pour le programmeur d'effectuer un transtypage lorsqu'il accède aux éléments de la file d'attente



pas de contrôle sur les valeurs rangées dans la file d'attente



# Généricité : motivation



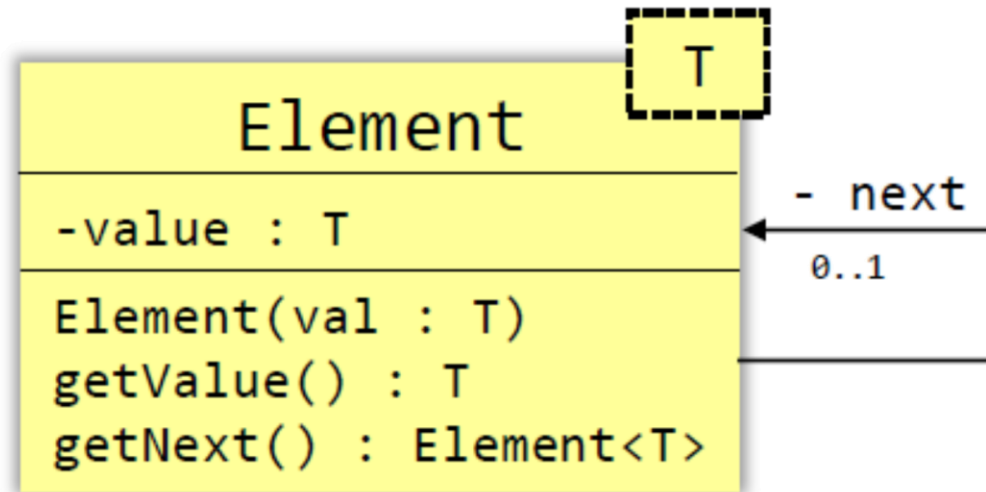
- ▶ Mais ça, c'était avant...
- ▶ depuis la version 1.5 de java (Tiger), cela n'est qu'un mauvais souvenir grâce à l'introduction (enfin !) de la généricité dans le langage Java.

# Généricité : Principe

- ▶ La généricité (ou **polymorphisme paramétrique**) permet de faire des **abstractions sur les types**.
  - ▷ utiliser des types génériques pour paramétrer une définition de classe ou d'interface ou de méthode
  - ▷ Réutiliser le même code avec des types de données différents.
- ▶ Présente dans de nombreux langages de programmation avant introduction en Java : Eiffel, Ada, C++, Haskell, ...

# Généricité : Représentation UML

- La notation UML pour une classe générique se présente comme suit :



## ► Les classes **Element** et **FileAttente** sont paramétrées en fonction d'un type formel **T**

► *T est utilisé comme type d'une ou plusieurs caractéristiques. Les classes manipulent des informations de type T. Elles ignorent tout de ces informations.*

```
public class FileAttente <T> {  
  
    private Element <T> tête = null;  
    private Element <T> queue = null;  
  
    public void ajouter( T val ) {  
        Element <T> elt = new Element <T>(val);  
        if (estVide()) {  
            tête = queue = elt;  
        } else {  
            queue.inserer(elt);  
            queue = elt;  
        }  
    }  
  
    public boolean estVide() {  
        return tête == null;  
    }  
  
    public T retirer() {  
        T val = null;  
        if (!estVide()) {  
            val = tête.getValue();  
            tête = tête.getNext();  
        }  
        return val;  
    }  
}
```

paramètre de type : représente un type inconnu au moment de la compilation

Dans le code du type générique, le paramètre de type peut être utilisé comme les autres types :

pour déclarer des variables ①

pour définir le type de retour ② ou le type des paramètres de méthodes ③

comme argument d'autres types génériques ④


```
class Element <T> {  
  
    private final T value;  
    private Element next;  
  
    Element( T val ) {  
        this.value = val;  
        next = null;  
    }  
  
    void inserer( Element <T> elt ) {  
        this.next = elt;  
    }  
  
    public T getValue() {  
        return value;  
    }  
  
    public Element <T> getNext() {  
        return next;  
    }  
}
```

# Utilisation d'une classe générique simple

- Lors de l'instanciation de **FileAttente<T>** le type formel **T** est remplacé par un type (Classe ou Interface) existant Le type est passé en argument

```
FileAttente<Personne> f1 = new FileAttente<Personne> ();  
f1.ajouter(new Personne(...));
```

```
FileAttente<Voiture> f2 = new FileAttente<> ();  
f2.ajouter(new Voiture(...));
```



*Depuis la version Diamond (java 7),  
on n'est pas obligé de mentionner les  
arguments de type dans le  
constructeur*

# Utilisation d'une classe générique simple



sans généricité (<JDK 5)

```
FileAttente f1 = new FileAttente();
f1.ajouter(new Personne(...));
...
// on veut récupérer le nom de la personne
// en tête de file.
String nom = (Personne)(f1.retirer()).getNom();
```



Transtypage obligatoire

```
f1.ajouter("Hello");
String nom = (Personne) (f1.retirer()).getNom();
```

**ClassCastException**



Erreur à l'exécution



avec généricité (JDK 5+)

```
f1.ajouter(new Personne(...));
...
// on veut récupérer le nom de la personne
// en tête de file.
String nom = f1.retirer().getNom();
```



Plus de transtypage



Erreur détectée dès la compilation

```
f1.ajouter("Hello");
String nom = f1.retirer().getNom();
```

Type incorrect

La généricité simplifie la programmation, évite de dupliquer du code et le rend plus robuste



# Types Génériques Simples

► **Type générique** : Classe ou interface paramétrée par un ou plusieurs types

classe générique à deux paramètres

```
public class Paire<T1, T2> {  
    private final T1 first;  
    private final T2 second;  
    public Paire(T1 first, T2 second)  
    {  
        this.first = first;  
        this.second = second;  
    }  
    public T1 getFirst() { return  
first; }  
    public T2 getSecond() { return  
second; }  
}
```

Interface générique

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

# Instanciation d'un type générique

- ▶ Instanciation (invocation) d'un type générique consiste à valuer les paramètres de type
- ▶ Les arguments de type (paramètres effectifs) peuvent être :

▷ *des classes (concrètes ou abstraites)*

```
Paire<String, Forme> c1 = new Paire<>("Cercle 1", new Cercle(0,0,10));
```

▷ *des interfaces*

```
Paire<String, IDessinable> c2 = new Paire<>("Visage 1", new VisageRond());
```

▷ *des types paramétrés*

```
Paire<String, Paire<String, Forme>> = new Paire<>("Paire 3", c1);
```

▷ *des paramètres de type*

```
public class FileAttente<E> {  
    private Element<E> tête = null;  
    ...  
}
```

# Compilation de code générique

```
public class Paire<T1,T2> {  
    private final T1 first;  
    private final T2 second;
```

toutes les informations  
de type placées entre  
chevrons sont effacées

```
    public Paire(T1 first, T2 second) {  
        this.first = first;  
        this.second = second;  
    }
```

```
    public T1 getFirst() {  
        return first;  
    }
```

```
    public T2 getSecond() {  
        return second;  
    }  
}
```

Les variables de type  
sont remplacées par  
Object (en fait leur  
borne supérieure en  
cas de généricité  
contrainte)

```
public class Paire {  
    private final Object first;  
    private final Object second;
```

```
    public Paire0(Object first, Object second) {  
        this.first = first;  
        this.second = second;  
    }
```

```
    public Object getFirst() {  
        return first;  
    }
```

```
    public Object getSecond() {  
        return second;  
    }  
}
```

javac

Le type `Paire<T1,T2>` a été  
remplacé par un type brut (*raw type*)  
en substituant `Object` à `T1` et `T2`

01001  
10011

Paire.class

```
Paire<String, String> c1;  
c1 = new Paire<>("Hello","World");
```

```
String s = c1.getFirst();
```



insertion d'opérations de  
transtypage si nécessaire

```
String s = (String) c1.getFirst();
```

```
Personne p = c1.getFirst();
```



rejeté dès la compilation : une `String`  
n'est pas une `Personne`

javac

01001  
10011

compilation de code générique  
basée sur le mécanisme  
d'effacement (*erasure*)

# Conséquences de l'effacement du type : Limitations

- ▶ à l'exécution, il n'existe qu'une classe (le type brut) partagée par toutes les instanciations du type générique

```
Paire<String,String> c1;  
Paire<Personne,Personne> p1;  
c1 = new Paire<>("Hello","World");  
p1 = new Paire<>(new Personne("DURAND", "Sophie"),new Personne("DUPONT", "Jean"));  
System.out.println( c1.getClass() == p1.getClass() ); → true
```

- ▶ mais cela induit un certain nombre de limitations

- ▷ *les membres statiques d'une classe paramétrée sont partagés par toutes les instanciations de celle-ci*

```
public class Paire<T1,T2> {  
    private static int nbInstances = 0;  
    private final T1 first;  
    private final T2 second;  
  
    public Paire(T first, T2 second) {  
        nbInstances++;  
        this.first = first;  
        this.second = second;  
    }  
  
    public static getNbInstances() {  
        return nbInstances;  
    }  
  
    ...  
}
```

```
Paire<String,String> c1;  
Paire<Personne,Personne> p1;  
c1 = new Paire<>("Hello","World");  
p1 = new Paire<>(new Personne("DURAND", "Sophie"),  
                new Personne("DUPONT", "Jean"));  
System.out.println( Paire.getNbInstances() );
```

→ 2

# Conséquences de l'effacement du type : Limitations

- ▶ Les **membres statiques** d'une classe paramétrée **ne peuvent pas** utiliser ses **paramètres de type**
- ▶ au sein d'une classe paramétrée on ne peut pas utiliser les paramètre de type **pour instancier un objet (simple ou tableau)**

```
public class C<T> {  
    ✓ T x1;  
    ✓ T[] tab;  
    ✗ static T x2;  
    ✗  
    ✓ T method1(T param) {  
        ...  
    }  
  
    ✗ static T method2(T param) {  
        ...  
    }  
  
    void method3(...) {  
        ...  
        ✗ x1 = new T();  
        ✗ tab = new T[10] ;  
    }  
    ...  
}
```

# Conséquences de l'effacement du type : Limitations

- ▶ seul le type brut est connu à l'exécution

```
Paire<String, String> c1 = new Paire<>("Hello","World");  
Paire<String, Personne> p1 = new Paire<>("personne 1", new Personne("DURAND", "Sophie"));  
System.out.println( c1.getClass() == p1.getClass() );  
System.out.println( c1 instanceof Paire );  
System.out.println( c1 instanceof Paire );
```

- ▶ on ne peut utiliser un type générique instancié dans un contexte de vérification de type

```
✗ if ( c1 instanceof Paire<String, String> ) { ... }
```

- ▶ on ne peut pas instancier de tableaux d'un type générique

```
Paire<Personne,Personne>[] duos;  
✗ duos = new Paire<Personne,Personne>[100] ;  
  
✓ duos = new Paire[100]; // mais utiliser le type brut est possible  
✓ duos[0] = p1;  
✗ duos[1] = c1;
```

# Méthodes Génériques

- ▶ Méthodes génériques : méthodes qui définissent leur propres paramètres de type.
  - ▷ *similaire aux types génériques mais la portée du paramètre de type est limitée à la méthode où il est déclaré.*
  - ▷ *méthodes génériques peuvent être définies dans des types non génériques*

# Méthodes Génériques : Exemple de la méthode copyOf

exemple : méthode `copyOf` de la classe `java.util.Arrays`  
permet de créer un tableau d'une taille donnée qui est une copie d'un tableau passé en paramètre

Arrays propose des méthodes utilitaires pour faire de recherches, tris, copies de tableaux...

Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `null`. Such indices will exist if and only if the specified length is greater than that of the original array. The resulting array is of exactly the same class as the original array.

## Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

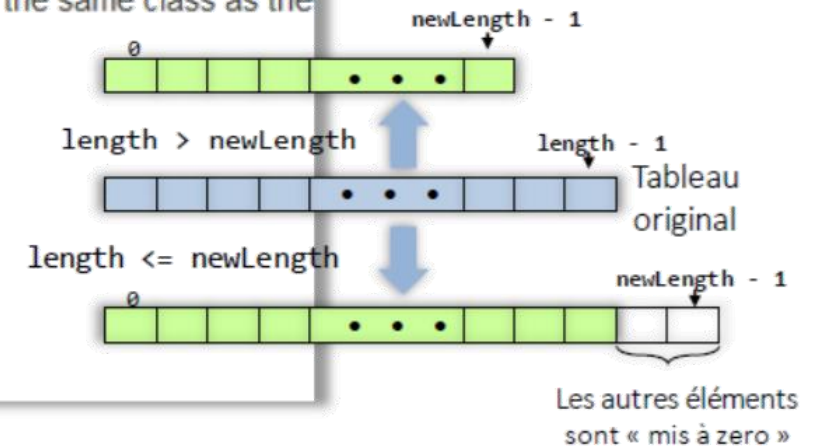
`T[]` un type tableau

`int`

## Returns:

le même type tableau que pour `original`

a copy of the original array, truncated or padded with nulls to obtain the specified length



Quelle signature pour cette méthode ?



- déclaration de `copyOf`

Il faut que ces types soient les mêmes

```
public static TypeTableau copyOf(TypeTableau original, int newLength)
```

Pour les types simples :  
surcharge de la méthode

```
public int[] copyOf(int[] original, int newLength)
```

```
public float[] copyOf(float[] original, int newLength)
```

...

```
public boolean[] copyOf(boolean[] original, int newLength)
```

Mais pour les types objets ? `Object[]`  $\neq$  `Personne[]`

→ définition d'une méthode générique

`T` : un type java (classe ou interface)

```
public static <T> T[] copyOf(T[] original, int newLength)
```

Comme pour les classes génériques, le paramètre de type doit être déclaré,  
déclaration entre `< >` avant le type de retour de la méthode

- invocation d'une méthode générique

```
Personne[] tab1 = new Personne[200];
```

...

```
Personne[] tab2 = Arrays.<Personne[]>copyOf(tab1, 100);
```

```
Personne[] tab2 = Arrays.copyOf(tab1, 100);
```

argument de type  
peut être ignoré  
à l'invocation...  
mais attention  
on verra plus  
tard les consé-  
quences

# Méthodes Génériques

## Méthode générique dans une classe non générique.

```
public class Class1{  
    private int x;  
    ...  
  
    public Class1(...) {  
        ...  
    }  
  
    public <T1,T2> T1 methodX(T1 p1, T2 p2) {  
        ...  
    }  
  
    ...  
}
```

teste si la seconde composante de la paire (this) est égale à la seconde composante d'une autre paire dont la première composante n'est pas forcément du même type que celle this

## Méthode générique dans une classe générique.

```
public class Paire<T1, T2> {  
    private final T1 first;  
    private final T2 second;  
  
    public Paire(T1 first, T2 second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public T1 getFirst(){  
        return first;  
    }  
  
    public T2 getSecond(){  
        return second;  
    }  
  
    public <T3> boolean sameScnd(Paire<T1,T3> p) {  
        return getSecond().equals(p.getSecond());  
    }  
}
```

Le (les) paramètre(s) de type de la méthode générique ne fait (font) pas partie des paramètres de type de la classe.

# Généricité : paramétrage contraint (ou borné)

- Il existe des situations où il peut être utile d'imposer certaines contraintes sur les paramètres de type (pour une classe, interface ou une méthode générique).

## Généricité contrainte (bounded type parameters)

- impose à un argument de type **d'être dérivé (sous-classe) d'une classe donnée ou d'implémenter une ou plusieurs interfaces**.

## Exemple avec une classe générique

```
public class Couple<T extends Personne> {  
  
    private final T first;  
    private final T second;  
  
    public Couple(T first, T second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public T getFirst() { return first; }  
    public T getSecond() { return second; }  
  
    public int getAgeMoyen() {  
        return (first.getAge() + second.getAge()) / 2;  
    }  
}
```

possibilité d'invoquer les méthodes définies dans la borne du type paramétré

la classe couple ne pourra être instanciée qu'avec le type Personne ou un type dérivé

Couple<Personne> cp; ✓  
Couple<Etudiant> ce; ✓

Couple<Point> cpt; ✗

erreur de compilation  
un point n'est pas une personne



A la compilation, le mécanisme d'effacement de type consiste à remplacer T par sa borne supérieure

```
public class Couple {  
    private final Personne first;  
    private final Personne second;  
  
    public Couple(Personne first, Personne second) {  
        ...  
    }  
    ...  
}
```

type brut (raw type)

ce.getFirst();

(Personne) ce.getFirst();

# Généricité : paramétrage contraint (ou borné)

- ▶ borne supérieure peut être soit une classe (concrète ou abstraite) soit une interface

`<T1 extends T2>`

`extends` est interprété avec un sens général, si **T1** est une classe et **T2** une interface `extends` doit être compris comme `implements`

# Généricité : paramétrage contraint (ou borné)

- possibilité de définir des bornes multiples (plusieurs interfaces, une classe et une ou plusieurs interfaces)

```
<T extends B1 & B2 & B3>
```

T est un sous type de tous les types listés et séparés par &

```
Class A { /* ... */ }  
interface B { /* ... */ }  
interface C { /* ... */ }
```

si l'une des bornes est une classe (et il ne peut y en avoir qu'une seule, car héritage simple en Java), celle-ci doit être spécifiée en premier

```
class D <T extends A & B & C> {  
    ...  
}
```



```
class D <T extends B & 1 & C> {  
    ...  
}
```



A la compilation, le mécanisme d'effacement de type consiste à remplacer **T** par la première de ses bornes supérieures (ici **A**)

# héritage de classes génériques

- Dans l'exemple précédent (Couple) n'aurait-il pas été possible de réutiliser le code de la classe générique Paire<T1,T2> ?

```
public class Paire<T1, T2> {  
  
    private final T1 first;  
    private final T2 second;  
  
    public Paire(T1 first, T2 second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public T1 getFirst(){  
        return first;  
    }  
  
    public T2 getSecond(){  
        return second;  
    }  
}
```

```
public class Couple<T extends Personne> {  
  
    private final T first;  
    private final T second;  
  
    public Couple(T first, T second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public T getFirst() { return first; }  
    public T getSecond() { return second; }  
  
    public int getAgeMoyen() {  
        return (first.getAge() +  
                second.getAge()) / 2;  
    }  
}
```

```
public class Couple<T extends Personne> extends Paire<T, T> {  
  
    Couple(T val1, T val2) {  
        super(val1, val2);  
    }  
  
    public int getAgeMoyen() {  
        return (getFirst().getAge() + getSecond().getAge()) / 2;  
    }  
}
```

Il est possible de sous typer une classe ou une interface générique en l'étendant ou en l'implémentant

# héritage de classes génériques

Différentes manières de dériver une classe générique

- a) en conservant les paramètres de type de la classe de base
- b) en ajoutant de nouveaux paramètres de type
- c) en introduisant des contraintes sur un ou plusieurs des paramètres de la classe de base
- d) en spécifiant une instance particulière de la classe de base

```
class ClasseA <T> { ... }
```

```
class ClasseB<T> extends ClasseA<T>{  
... }
```

```
class ClasseB<T,U> extends ClasseA<T>{  
... }
```

```
class ClasseB<T extends TypeC> extends ClasseA<T>{  
... }
```

```
class ClasseB extends ClasseA<String>{  
... }
```

```
class ClasseB<T> extends ClasseA<String>{  
... }
```



# héritage de classes génériques

## situations incorrectes

- ▶ ClasseB doit disposer au moins du paramètre T

l'inverse est possible, une classe générique peut hériter d'une classe non générique `class ClasseB<T> extends ClasseC{ ... }` ✓

```
class ClasseB extends ClasseA<T>{  
    ...  
}
```



- ▶ Les contraintes doivent être exprimées sur les paramètres de la classe dérivée

```
class ClasseB<T> extends ClasseA<T extends TypeC>{  
    ...  
}
```



