

# Java : généricité

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille  
Chercheur en programmation par contrainte (IA)  
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



# Plan

- 1 Introduction
- 2 Premier exemple
- 3 Exemple avec deux types génériques
- 4 Exemple avec plusieurs attributs avec le même type générique
- 5 Exemple avec l'héritage

## Concept objet

- Classe = { attributs } + { méthodes }
- **Java** : langage fortement typé  $\Rightarrow$  Les attributs ont **forcément** un nom et un type
- Le type d'attribut ne change pas pour toutes les instances d'une classe
- Les méthodes permettent généralement d'effectuer des opérations sur les attributs tout en respectant leurs spécificités (type, taille...)

## Problématique

- Et si on a besoin d'une classe dont les méthodes effectuent les mêmes opérations quel que soit le type d'attributs
  - **somme** pour **entiers** ou **réels**,
  - **concaténation** pour **chaînes de caractères**,
  - **ou logique** pour **booléens**...
  - ...
- **Impossible sans définir plusieurs classes (une pour chaque type) et une interface ou en imposant le type `Object` et en utilisant plusieurs cast...**

une solution plus élégante avec la généricité et l'opérateur diamond <>

- Introduit en **Java 5** et simplifié en **Java 7**
- Concept : ne pas définir de type pour les attributs
- Définir un type générique qui ne sera pas précisé à la création de la classe
- À l'instanciation de la classe, on précise le type à utiliser par cette classe
- **On peut donc choisir pour chaque instance le type que l'on souhaite utiliser.**

## Une première classe avec un type générique

```
public class Exemple <T> {  
    private T var;  
  
    public T getVar() {  
        return var;  
    }  
  
    public void setVar(T var) {  
        this.var = var;  
    }  
  
}
```

# Java

Dans `main`, créons des instances de la même classe avec des types différents

```
Exemple<Integer> entier = new Exemple<Integer>();
entier.setVar(10);
System.out.println(entier.getVar().getClass().getSimpleName());
Exemple<String> chaine = new Exemple<String>();
chaine.setVar("Bonjour");
System.out.println(chaine.getVar().getClass().getSimpleName());
```

© Achref EL

# Java

Dans `main`, créons des instances de la même classe avec des types différents

```
Exemple<Integer> entier = new Exemple<Integer>();  
entier.setVar(10);  
System.out.println(entier.getVar().getClass().getSimpleName());  
Exemple<String> chaine = new Exemple<String>();  
chaine.setVar("Bonjour");  
System.out.println(chaine.getVar().getClass().getSimpleName());
```

## Résultat

```
Integer  
String
```



# Java

## Exemple avec deux types génériques

```
public class Exemple<T, S> {  
  
    private T var1;  
    private S var2;  
  
    public T getVar1() {  
        return var1;  
    }  
  
    public void setVar1(T var1) {  
        this.var1 = var1;  
    }  
  
    public S getVar2() {  
        return var2;  
    }  
  
    public void setVar2(S var2) {  
        this.var2 = var2;  
    }  
}
```

# Java

Testons cela dans `main`

```
Exemple<Integer, String> couple = new Exemple<Integer, String>();  
couple.setVar1(10);  
couple.setVar2("Bonjour");  
System.out.println(couple.getVar1().getClass().getSimpleName());  
System.out.println(couple.getVar2().getClass().getSimpleName());
```

© Achref EL M...

# Java

Testons cela dans `main`

```
Exemple<Integer, String> couple = new Exemple<Integer, String>();  
couple.setVar1(10);  
couple.setVar2("Bonjour");  
System.out.println(couple.getVar1().getClass().getSimpleName());  
System.out.println(couple.getVar2().getClass().getSimpleName());
```

Résultat

```
Integer  
String
```

# Java

Considérons la classe `Operation` suivante

```
public class Operation <T>{  
  
    private T var1;  
    private T var2;  
  
    public Operation(T var1, T var2) {  
        this.var1 = var1;  
        this.var2 = var2;  
    }  
}
```

© Achref EL MOU

# Java

Considérons la classe `Operation` suivante

```
public class Operation <T>{  
  
    private T var1;  
    private T var2;  
  
    public Operation(T var1, T var2) {  
        this.var1 = var1;  
        this.var2 = var2;  
    }  
}
```

## Exercice

Écrire une méthode `plus` dans `Operation` qui affiche

- la somme si `var1` et `var2` sont de type `Float` ou `Integer`,
- la concaténation si `var1` et `var2` sont de type `String`,
- le OU logique si `var1` et `var2` sont de type `Boolean`.

# Java

## Première solution

```
public class Operation <T>{

    private T var1;
    private T var2;

    public Operation(T var1, T var2) {
        this.var1 = var1;
        this.var2 = var2;
    }

    public void plus () {
        if (var1.getClass().getSimpleName().equals("Double")) {
            System.out.println((Double)var1 + (Double)var2);
        }
        else if (var1.getClass().getSimpleName().equals("Integer")) {
            System.out.println((Integer)var1 + (Integer)var2);
        }
        else if (var1.getClass().getSimpleName().equals("Boolean")) {
            System.out.println((Boolean)var1 || (Boolean)var2)
        }
        else {
            System.out.println(var1.toString() + var2.toString());
        }
    }
}
```

# Java

## Deuxième solution

```
public class Operation <T>{

    private T var1;
    private T var2;

    public Operation(T var1, T var2) {
        this.var1 = var1;
        this.var2 = var2;
    }

    public void plus() {
        if (var1 instanceof Double v1 && var2 instanceof Double v2) {
            System.out.println(v1 + v2);
        } else if (var1 instanceof Integer v1 && var2 instanceof Integer v2) {
            System.out.println(v1 + v2);
        } else if (var1 instanceof Boolean v1 && var2 instanceof Boolean v2) {
            System.out.println(v1 || v2);
        } else {
            System.out.println(var1.toString() + var2.toString());
        }
    }
}
```

# Java

## Testons cela dans `main`

```
Operation<Integer> operation1 = new Operation<Integer>(5, 3);  
operation1.plus();  
Operation<String> operation2 = new Operation<String>("bon", "jour");  
operation2.plus();  
Operation<Double> operation3 = new Operation<Double>(5.2, 3.8);  
operation3.plus();  
Operation<Boolean> operation4 = new Operation<Boolean>(true, false);  
operation4.plus();
```



# Java

## Testons cela dans `main`

```
Operation<Integer> operation1 = new Operation<Integer>(5, 3);  
operation1.plus();  
Operation<String> operation2 = new Operation<String>("bon", "jour");  
operation2.plus();  
Operation<Double> operation3 = new Operation<Double>(5.2, 3.8);  
operation3.plus();  
Operation<Boolean> operation4 = new Operation<Boolean>(true, false);  
operation4.plus();
```

## Résultat

```
8  
bonjour  
9.0  
true
```

**Considérons la classe `Personne` suivante**

```
package org.eclipse.model;

public class Personne {
    private String nom;
    private String prenom;

    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    @Override
    public String toString() {
        return "Personne [nom=" + nom + ", prenom=" + prenom + "];"
    }
}
```

# Java

La classe `Etudiant` qui hérite de la classe `Personne`

```
package org.eclipse.model;

public class Etudiant extends Personne {

    private String niveau;

    public String getNiveau() {
        return niveau;
    }

    public void setNiveau(String niveau) {
        this.niveau = niveau;
    }

    @Override
    public String toString() {
        return "Etudiant [niveau=" + niveau + " " + super.toString() + " ]";
    }
}
```

# Java

## La classe Vehicule

```
package org.eclipse.model;

public class Vehicule {

    private int marque;

    public int getMarque() {
        return marque;
    }

    public void setMarque(int marque) {
        this.marque = marque;
    }
    @Override
    public String toString() {
        return "Vehicule [marque=" + marque + "]";
    }
}
```

# Java

## La classe Humain

```
package org.eclipse.model;

public class Humain <T> {

    private T var;

    public T getVar() {
        return var;
    }

    public void setVar(T var) {
        this.var = var;
    }
}
```

# Java

En testant le `main` suivant, aucun message d'erreur n'est signalé

```
package org.eclipse.test;

import org.eclipse.model.Etudiant;
import org.eclipse.model.Humain;
import org.eclipse.model.Personne;
import org.eclipse.model.Vehicule;

public class Main {
    public static void main(String[] args) {
        Humain<Personne> humain = new Humain<>();
        Humain<Etudiant> humain2 = new Humain<>();
        Humain<Vehicule> humain3 = new Humain<>();
    }
}
```

# Java

## Modifions la classe Humain

```
package org.eclipse.model;

public class Humain <T extends Personne> {

    private T var;

    public T getVar() {
        return var;
    }

    public void setVar(T var) {
        this.var = var;
    }
}
```

# Java

On ne change rien dans `main`

```
public class Main {  
    public static void main(String[] args) {  
        Humain<Personne> humain = new Humain<>();  
        Humain<Etudiant> humain2 = new Humain<>();  
        Humain<Vehicule> humain3 = new Humain<>();  
    }  
}
```

© Acti



# Java

On ne change rien dans `main`

```
public class Main {  
    public static void main(String[] args) {  
        Humain<Personne> humain = new Humain<>();  
        Humain<Etudiant> humain2 = new Humain<>();  
        Humain<Vehicule> humain3 = new Humain<>();  
    }  
}
```

## Remarque

Dernière instruction soulignée en rouge car `Vehicule` n'hérite pas de `Personne`.

## Pour les collections

- `<?>` autorise tout
- `<? extends Personne>` autorise seulement les objets de la classe `Personne` ou ceux dont la classe mère est `Personne`
- `<? super Personne>` autorise seulement les objets de la classe `Personne` ou ceux qui ont la classe `Personne` comme classe fille