CHAPITRE 1 :

LES TYPES GÉNÉRIQUES EN JAVA

Mohamed Khairallah KHOUJA

ISET Sfax 2025

OBJECTIFS

A la fin de ce chapitre, vous serez capables de :

- créer une classe générique personnalisée ,
- connaître les règles d'héritage avec les classes génériques,
- utiliser les jokers.

LES TYPES GÉNÉRIQUES

On parle généralement de **programmation générique** lorsqu'un langage permet d'écrire un code source unique utilisable avec des objets ou des variables de types quelconques.

On peut prendre l'exemple d'une méthode de tri applicable à des objets de type quelconque ou encore celui d'une classe permettant de manipuler des ensembles d'objets de type quelconque.

POURQUOI LA GÉNÉRICITÉ

EXEMPLE 1 : CLASSE PAIRE SANS TYPES GÉNÉRIQUES

Objectif : Vous allez créer deux classes : *PaireString et PairePersonne* sans utiliser les types génériques

```
public class PaireString
{
    String premier;
    String second;
    public PaireString (String premier,String second)
    {
        this.premier=premier;
        this.second=second;
    }
    public String getPremier()
    {
        return premier;
    }
    public String getSecond()
    {
        return second;
    }
}
```

```
public class PairePersonne {
        Personne premier;
        Personne second;
        public PairePersonne(Personne premier,Personne
second)
        {
                this.premier=premier;
                this.second=second;
                }
        public Personne getPremier()
        {
                return premier;
                }
        public Personne getSecond()
        {
                return second;
                }
}
```

L'implémentation des deux classes Paire est simple. Mais malgré cette simplicité, une classe PaireX distincte doit être définie pour chaque Objet X.

POURQUOI LA GÉNÉRICITÉ

EXEMPLE 2 : CLASSE PAIRE AVEC N'IMPORTE QUEL TYPE DE DONNÉES (EN UTILISANT LA CLASSE OBJECT)

```
public class Paire {
        Object premier;
        Object second;
        public Paire(Object premier, Object second)
        {
                this.premier=premier;
                this.second=second;
        }
        public Object getPremier()
        {
        return premier;
        }
        public Object getSecond()
        {
                return second;
        }
}
```

Scénario d'exécution :

```
public static void main (String []args) {
        Paire p = new Paire("abc","efg");
        String premier=(String) p.getPremier(); (1)
        Double second = (Double) p.getSecond(); (2)
}
```

Résultat :

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Double
        at TestPaire.main(TestPaire.java:12)
```

Interprétation :

(1) Si vous voulez utiliser les données de l'objet *Paire*, il va falloir faire un **casting**

(2) Certaines erreurs ne sont détectées qu'à l'**exécution** du programme (Casting du Type "Double" sur chaîne de caractère)

POURQUOI LA GÉNÉRICITÉ

EXEMPLE 3 : CLASSE PAIRE AVEC LES TYPES GÉNÉRIQUES

```
public class Paire <T> {
        T premier;
        T second;
        public Paire(T premier, T second){
                this.premier=premier;
                this.second=second;
                }
        public T getPremier(){
                return premier;
                }
        public T getSecond(){
                return second;
                }
}
```

Interprétation :

- Pour créer une version générique de la classe Paire, une variable nommée **T** entourée de chevrons est ajoutée à la définition de la classe.
- Dans ce cas, **T** signifie **type** et désigne n'importe quel type.
- Comme illustré dans l'exemple, le code a été modifié pour utiliser T à la place d'un type spécifique.
- Cette modification autorise la classe Paire à stocker tout type d'objet.

REMARQUE

Certaines lettres sont fréquemment utilisées avec les types génériques. Toute fois vous pouvez utiliser un identificateur à votre choix. Les conventions utilisées sont :

- **T** : Type
- **E** : Elément
- **K** : clé (key)
- **V** : Valeur
- **S, U** : lors de la présence d'un deuxième et troisième type ou plus

NOTION DE CLASSE GÉNÉRIQUE

CLASSE GENERIQUE A UN SEUL PARAMETRE DE TYPE

Reprenez l'exemple de la classe Paire générique avec un seul paramètre de type.

Par souci de simplicité, votre classe ne disposera que des méthodes suivantes :

- un constructeur, recevant les valeurs des deux éléments d'une paire,
- une méthode nommée affiche(), affichant les valeurs des deux éléments d'une paire, et
- une méthode nommée getPremier(), fournissant la valeur du premier élément d'une paire.

NOTION DE CLASSE GÉNÉRIQUE

CLASSE GENERIQUE A UN SEUL PARAMETRE DE TYPE

```
public class Paire <T> {
        T premier;
        T second;
        public Paire(T premier, T second){
                this.premier=premier;
                this.second=second;
                }
        public T getPremier(){
                return premier;
                }
        public void affiche(){
                System.out.print("Première valeur:" + premier + "" );
                System.out.println("Deuxième valeur" + second);
                }
        }
```

Interprétation :

On note la présence d'un "paramètre de type" nommé T. Ce paramètre peut être utilisé là où un type précis peut l'être normalement. Dans cet exemple, on le rencontre dans :

- La définition de la classe : ***class Paire<T>.***
- Les déclarations des champs premier et second.
- L'entête du constructeur, de la méthode ***getPremier()*** et de la méthode ***affiche()***.

NOTION DE CLASSE GÉNÉRIQUE

UTILISATION D'UNE CLASSE GENERIQUE

Lors de la déclaration d'un objet de type Paire, vous devriez préciser le type effectif correspondant à T, de cette manière :

```
public static void main(String[] args) {
        Paire <String> s = new Paire <String>("abc","efg");
        Paire <Personne> p = new Paire <Personne>(new Personne(), new Personne());
        }
```

Le type T doit obligatoirement être de type classe.
L'instruction suivante est rejetée :

```
Paire <int> a = new Paire <int>(5, 4);
```

NOTION DE CLASSE GÉNÉRIQUE

UTILISATION D'UNE CLASSE GENERIQUE

EXEMPLE 4 : EXEMPLE D'UTILISATION DE LA CLASSE PAIRE DANS LA CLASSE TABLEAUALG

```
class TableauAlg {
        public static Paire<String> minmax(String[] chaines) {
                if (chaines==null || chaines.length==0) return null;
                String min = chaines[0];
                String max = chaines[0];
                for (String chaine : chaines) {
                        if (min.compareTo(chaine) > 0) min = chaine;
                        if (max.compareTo(chaine) < 0) max = chaine;
                        }
                return new Paire<String>(min, max);
                }
```

NOTION DE CLASSE GÉNÉRIQUE

TYPES GENERIQUES AVEC INFERENCE DE TYPE LOSANGE

- L'inférence de type losange est une nouvelle fonctionnalité de JDK 7.
- Vous pouvez utiliser la syntaxe en losange pour indiquer que la définition de type de droite est équivalente à celle de gauche.
- Cela évite de saisir plusieurs fois des informations redondantes.
- C'est la partie de gauche de l'expression et non pas la partie droite qui détermine le type.

```
Paire <String> s = new Paire < >("abc","efg");
```

NOTION DE CLASSE GÉNÉRIQUE

CLASSE GENERIQUE A PLUSIEURS PARAMETRES DE TYPE

EXEMPLE 5 : DÉFINITION DE LA CLASSE PAIRE AVEC DEUX PARAMÈTRES

```
public class Paire <T, U> {
        T premier;
        U second;
        public Paire(T premier, U second) {
                this.premier=premier;
                this.second=second;
                }
        public T getPremier()
        {return premier;}
        public U getSecond()
                {return second;}
                public void affiche(){
                        System.out.print("Première
valeur:" + premier + "" );
                        System.out.println("Deuxième
valeur" + second);
                        }
                }
```

NOTION DE CLASSE GÉNÉRIQUE

CLASSE GENERIQUE A PLUSIEURS PARAMETRES DE TYPE

EXEMPLE 6 : UTILISATION DE LA CLASSE PAIRE AVEC DEUX PARAMÈTRES

```
public static void main(String[] args) {
        Integer oi1 = 3;
        Double od1 = 2.5;
        Paire <Integer, Double> ch1 = new Paire <Integer, Double> (oi1, od1);
        ch1.affiche() ;
        }
```

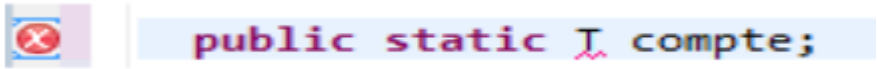
REMARQUES (CAS DES CHAMPS STATIQUES)

- Si l'on définit un champ statique dans une classe générique, il sera unique pour toutes les instances de cette classe, quelle que soit la valeur du paramètre de type.

```
public class Paire <T, U> {
        T premier;
        U second;
        public static int compte;
        public Paire(T premier, U second){
                this.premier=premier;
                this.second=second;
                compte++;
                }
}
```

Vous verrez la valeur de compte s'accroître à chaque instanciation d'un objet de type Paire

Un champ statique ne peut pas être de type paramétré :



NOTION DE MÉTHODE GÉNÉRIQUE

- La même démarche peut s'appliquer à une méthode, on parle tout simplement de "**méthodes génériques**".
- Les méthodes génériques peuvent être définies dans des classes ordinaires ou des classes génériques.

NOTION DE MÉTHODE GÉNÉRIQUE

EXEMPLE 7 : DÉFINITION ET APPEL D'UNE MÉTHODE GÉNÉRIQUE

```
class TableauAlg {
public static <T> T getMilieu(T [] tableau) {
return tableau[tableau.length / 2];
} }
```

```
public static void main (String[]args){
// Test String
String[]noms={"Mariem","Ali","Mohamed","Salah","Lamia"};
String milieu = TableauAlgo.<String>getMilieu(noms);
// TestInteger
Integer[]number={12, 2, 14, 23, 15};
Integer milieu = TableauAlgo.<Integer>getMilieu(number);
}
```

LIMITATIONS DES VARIABLES DE TYPE

CAS D'UNE CLASSE GENERIQUE

- Il est possible, au moment de la définition de la classe, d'imposer certaines contraintes.
- On pourra imposer à la classe correspondant à un paramètre de type d'être dérivée d'une classe donnée ou d'implémenter une ou plusieurs interfaces.
- Par exemple, en définissant ainsi la classe Paire comme suit :

class Paire <T extends Number> { // definition précédente inchangée }

- Dans ce cas, lors de la compilation de la classe Paire, le mécanisme d'effacement conduira à remplacer le type T, non plus par Object, mais par Number.

```
Paire <Point> p = new Paire<> (new Point(), new Point());

Paire <Double> d = new Paire<> (5, 12);
```

LIMITATIONS DES VARIABLES DE TYPE

CAS D'UNE METHODE GENERIQUE

EXEMPLE 8 : MÉTHODE MIN () *SANS RESTRICTION* DE LA VARIABLE DE TYPE

```
public static <T> T min(T[] tab)
{
        if (tab==null || tab.length==0) return null;
        T pluspetit = tab[0];
        for (T val : tab)
                if (pluspetit.compareTo(val) > 0) pluspetit = val;
        return pluspetit;
}
```

Comment savoir que la classe à laquelle appartient T possède une méthode compareTo() ?

Erreur du compilateur en spécifiant que cette méthode compareTo() n'est pas connue pour un type quelconque T.

LIMITATIONS DES VARIABLES DE TYPE

CAS D'UNE METHODE GENERIQUE

EXEMPLE 9 : MÉTHODE MIN () *AVEC RESTRICTION* DE LA VARIABLE DE TYPE

- La solution consiste à restreindre T à une classe qui implémente l'interface Comparable, une interface standard disposant d'une seule méthode compareTo().
- Pour y parvenir, vous devez donner une limite pour la variable de type T

```
public static <T extends Comparable> T min(T[] tab)
```

LIMITATIONS DES VARIABLES DE TYPE

REGLES GENERALES

- D'une manière générale, dans une définition de classe générique ou de méthode générique, vous pouvez imposer à une classe correspondant à un paramètre de type, non seulement de **dériver d'une classe donnée**, mais aussi **d'implémenter une interface**, comme dans le cas de l'exemple 9.

```
class Paire <T extends Comparable & Cloneable >
{
// T doit implémenter les interfaces Comparable & Cloneable
}
```

c'est la première interface citée qui sera utilisée par le compilateur en remplacement du type T

LIMITATIONS DES VARIABLES DE TYPE

REGLES GENERALES

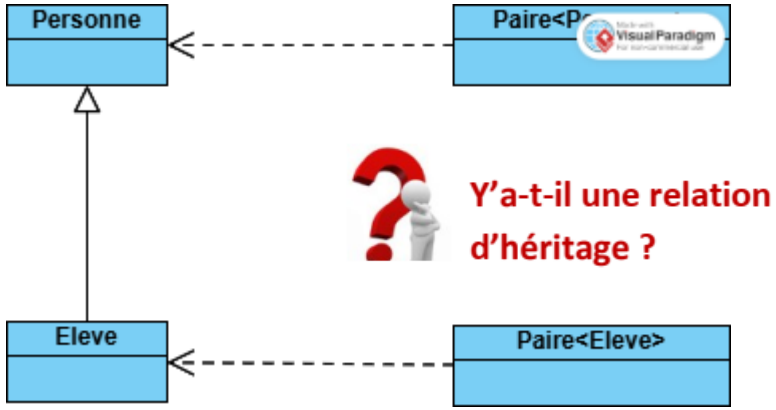
- Il est possible de combiner les deux possibilités : une classe et une ou plusieurs interfaces, comme dans l'exemple suivant.

```
class Couple <T extends Personne & Comparable & Cloneable >
{ // T doit étendre la classe Personne et implémenter les interfaces Comparable & Cloneable
        }
```

Dans ce cas, il ne doit y avoir qu'une seule classe et elle devra être citée en premier : c'est elle qui sera utilisée par le compilateur dans ses traductions.

RÈGLES D'HÉRITAGE POUR LES TYPES GÉNÉRIQUES

SI LA VARIABLE DE TYPE S DERIVE DE T




```
Eleve[] eleves = … ;
Paire<Personne> personne = TableauAlg.minmax(eleves) ;
```

La méthode minmax() de la classe TableauAlg renvoie un Paire<Elève>, et non pas un Paire<Personne>, et il n'est pas possible d'affecter l'une à l'autre.

Il n'existe pas de relation entre Paire<S> et Paire <T>, quel que soit le type de relation auquel S et T sont relié.

RÈGLES D'HÉRITAGE POUR LES TYPES GÉNÉRIQUES

DERIVATION D'UNE CLASSE GENERIQUE

Il est possible de créer une classe dérivée d'une classe générique et ceci de différentes manières, suivant que l'on conserve ou non les paramètres de type ou que l'on en ajoute de nouveaux.

Soit la classe générique class C <T> :

- ***1ier cas : La classe dérivée conserve les paramètres de type de la classe de base, sans en ajouter d'autres***.

Exemple 1 : class D <T> extends C <T> { ..... }
Ici, C et D utilisent le même paramètre de type. Ainsi D<String> dérive de C<String>

Exemple 2 : class D<T, U> extends C<T, U>
Ici, D<String, Double> dérive de C<String, Double>.

RÈGLES D'HÉRITAGE POUR LES TYPES GÉNÉRIQUES

DERIVATION D'UNE CLASSE GENERIQUE

- ***2ième Cas : La classe dérivée utilise les mêmes paramètres de type que la classe de base, en en ajoutant de nouveaux.***

Exemple : class D <T, U> extends C <T> { ..... }
Ici, outre le paramètre de type T de C, D possède un paramètre supplémentaire U.
D<String, Doule> dérive de C<String>, D<Integer, Double> dérive de C<Integer>.

- ***3ième Cas : La classe dérivée introduit des limitations sur un ou plusieurs des paramètres de type de la classe de base.***

Exemple : class D <T extends Number> extends C<T>
Ici, on peut utiliser D<Double>, qui dérive alors de C<Double>, en revanche, on ne peut pas utiliser D<String> (alors qu'on peut utiliser C<String>).

RÈGLES D'HÉRITAGE POUR LES TYPES GÉNÉRIQUES

DERIVATION D'UNE CLASSE GENERIQUE

- ***4ième Cas : La classe de base n'est pas générique, la classe dérivée l'est.***

Exemple : class D<T> extends X
Ici, X est une classe usuelle. D<String>, D<Double>, D<Point> dérivent toutes de X.

- ***5ième Cas : La classe de base est une instance particulière d'une classe générique.***

Exemple : Il s'agit en fait d'un cas particulier du cas précédent, dans lequel X est une instance particulière de C<T> (avec T = String). D<Double>, D<Point> et même D<String> dérivent toutes de C<String>.

LES JOKERS

- Vous venez de voir que, si T' dérive de T, C<T'> ne dérive pas de C<T>.
- Cependant, il existe une relation intéressante entre ces deux classes puisqu'il reste toujours possible d'utiliser un objet de type C<T'> comme on le ferait d'un objet de type C<T>, pour peu qu'on ne cherche pas à en modifier la valeur.
- Les concepteurs JAVA ont inventé le concept de **Joker** pour exploiter cette relation d'héritage.
- Exemple de Joker :

<?extends Personne> : remplace toute paire générique dont le paramètre type est une sousclasse de Personne, comme Paire<Eleve>, mais pas, bien entendu, Paire<String>

LES JOKERS

EXEMPLE 10 : JOKER APPLIQUÉ À UNE MÉTHODE

```
public static void afficheBinomes (Paire<? extends Personne> personnes) {
        Personne premier = personnes.getPremier();
        Personne deuxieme = personnes.getDeuxieme();
        System.out.println(premier.getNom()+" et "+deuxieme.getNom()+" sont
ensembles.");  }
```

La méthode afficheBinomes() permet d'afficher une paire de Personne. Il est impossible de faire passer un Paire <Eleve> à la méthode afficheBinome. Ainsi, la solution consiste à utiliser un joker : **<? extends Personne>**.

Ceci permet d'indiquer qu'il est possible d'utiliser n'importe quelle classe qui fait partie de l'héritage de Personne, classe de base comprise.

LES JOKERS

EXEMPLE 11 : UTILISATION POSSIBLE DE LA MÉTHODE AFFICHEBINOME

```
public class Main {
	public static void main(String[] args) {
		Personne personne1 = new Personne("Med", "Khouja");
		Personne personne2 = new Personne("Ali", "Omar");
		Paire<Personne> binomePersonne = new Paire<Personne>(personne1,personne2);
		Eleve eleve1 = new Eleve("Yasmine", "Khouja");
		Eleve eleve2 = new Eleve("Olfa", "Arbi");
		Paire<Eleve> binomeEleve = new Paire<Eleve>(eleve1, eleve2);
		afficheBinomes(binomePersonne);
		afficheBinomes(binomeEleve);
}}
```

EXERCICE 1

On suppose qu'on a défini une classe générique nommée C :

class C <T> { ..... }, ainsi qu'une classe ordinaire nommée X.

Pour chacune des définitions suivantes, donner les relations d'héritage existant entre les classes mentionnées en commentaires :

1. class D<T> extends C<T> { ..... } // C<Object>, C<Double>, D<Object>, D<Double>
2. class D<T, U> extends C<T> { ..... } // C<Double>, D(Double, Integer), D(Double, Double), D(Integer, Double)
3. class D<T extends Number> extends C<T> { ..... } // D<Double>, C<Double>, D<String>, C<String>
4. class D<T> extends X { ..... } // D<Double>, X, D<String>
5. class D<T> extends C<String> // D<String>, D<Integer>, C<String>, C<Integer>

EXERCICE 2 (1)

Dans une application destinée à une entreprise, on souhaite modéliser les vendeurs à l'aide d'une classe adaptée. Tous les types de vendeurs peuvent travailler dans cette entreprise et vendre n'importe quel produit à différents types de clients (particuliers, entreprises, institutions, associations, etc.).

1.Classe Générique Vendeur<T> :

- La classe doit être générique pour pouvoir gérer différents types de clients.
- Elle doit permettre de :
  - **Gérer les ventes** : ajouter une vente, supprimer une vente et afficher la liste des ventes réalisées par le vendeur.
  - **Gérer les clients** : ajouter un client, supprimer un client et afficher la liste des clients du vendeur.

EXERCICE 2 (2)

2.Classe VendeurAuDetail :

- VendeurAuDetail est une sous-classe de **Vendeur** représentant les vendeurs qui n'ont pour clients que des **particuliers**.
- Les particuliers sont modélisés par la classe suivante :

```
public class Particulier {
    private String nom;

    public Particulier(String n) {
        this.nom = n;
    }

    public String toString() {
        return this.nom;
    }
}
```

EXERCICE 2 (3)

3.Programme de Test :

- Le programme doit :
  - Créer des instances de Vendeur pour gérer différents types de clients (entreprises, institutions, etc.).
  - Créer des instances de VendeurAuDetail pour gérer des clients particuliers.
  - Ajouter des ventes et des clients à chaque vendeur.
  - Afficher les ventes et les clients pour vérifier le bon fonctionnement du système.