ENSAH

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

Type générique

Langage Java

2ème année Génie Informatique

Ecole Nationale des Sciences Appliquées – Al Hoceima

Prof A. Bahri
abahri@uae.ac.ma

A.U 2020/2021

Généricité

Depuis la version 5.0 Java autorise la définition de classes et d'interfaces contenant un (des) paramètre(s) représentant un type(s). Cela permet de décrire une structure qui pourra être personnalisée au moment de l'instanciation à tout type d'objet.

Programmation générique

Exemple: On veut définir une notion de paire d'objets avec deux attributs de même type.

```
Public class PaireEntier{
        private int premier;
        private int second;
        public PaireEntier(int x, int y){
                premier=x;
                second=y;
        }
        public int getPremier(){ return this.premier;}
        public int getSecond(){ return this.second;}
        public void setPremier(int x){ this.premier=x;}
        public void setsecond(int x){ this.second=x;}
        public void interchanger(){
                int temp=this.premier;
                this.premier=this.second;
                this.second=temp;
        }
}
```

Remarque :

- on a créé une classe spécialement pour des paires d'entiers ;
- si on veut des paires de booléens il faudrait réécrire une autre classe (avec un autre nom) qui contiendrait les mêmes méthodes.

Programmation générique

```
Public class PaireObjet{
    private Object premier;
    private Object second;
    public PaireObjet(Object x, Object y){
        premier=x;
        second=y;
    }
    public Object getPremier(){ return this.premier;}
    public Object getSecond(){ return this.second;}
    public void setPremier(Object x){ this.premier=x;}
    public void setsecond(Object x){ this.secondr=x;}
    public void interchanger(){
        Object temp=this.premier;
        this.premier=this.second;
        this.second=temp;
    }
}
```

L'utilisation demande de le transtypage (*typecast*)

```
PaireObjet p1 = new PaireObjet ("Paques",27);
String p1fst = (String) p1.getPremier();
```

Inconvénients :

- les deux attributs peuvent des instances de classes différentes,
- on peut être amené à faire du transtypage (vers le bas),
- on risque des erreurs de transtypage qui ne se détecteront qu'à l'exécution.

Programmation générique

A et B étant deux classes, on peut avoir ce genre d'utilisation

```
A a = new A( ) ;
B b = new B( ) ;
PaireObjet p= new PaireObjet(a,b);
A a2=(A) p.getPremier();
P.interchanger(); // down casting ok
A a3= (A) p.getPremier(); // erreur
```

Programmation générique

Paramétrer les type

- Comme on utilise Object comme type, il est difficile d'avoir un code :
  - – Générique
    (ne pas réécrire le code pour la liste de String, de Integer, de ...)
  - – ET statiquement typé
    (vérifiable par le compilateur)

- La solution est l'utilisation de la généricité, c'est-à-dire l'usage de type paramètre.

Qu'est-ce que la généricité ?

- Dans une fonction «classique», les paramètres sont des valeurs

  ➡ Dans un générique, les paramètres sont des **types**

- Un générique est **un modèle**:
  - Instanciation = création d'un élément à partir d'un modèle

  ➡ Instancier un générique = **fixer le type** de ses paramètres

- Composants pouvant être génériques en Java (depuis Java 5.0)
  - Classes, interfaces, méthodes

Programmation générique

- Définition:
  - Code source unique utilisable avec des objets ou des variables de type quelconque

- Intérêt:
  - Indépendance du code vis-à-vis du type des données manipulées

- Exemple:
  - méthodes de tris applicable à des objets de type quelconque (Point, Double, String, …)

La programmation générique

❑ Définition d'une classe générique simple

- Une classe générique est une classe comprenant une ou plusieurs variables de type = une classe paramétrée, dont le paramètre est un type d'objet

❑ Exemple:

```
Public class Paire<T>{
        private T premier;
        private T second;
        public Paire(){}
        public Paire(T x, T y){ // en-tête du constructeur sans <T>
                premier=x;
                second=y;
        }
        public T getPremier(){ return this.premier;}
        public T getSecond(){ return this.second;}
        public void setPremier( T x){ this.premier=x;}
        public void setsecond( T x){ this.secondr=x;}
        public void interchanger(){
                T temp=this.premier;
                this,premier=this.second;
                this.second=temp;
        }
}
```

Cette définition permet de définir ici des Paire contenant des objets de type (uniforme) mais arbitraire.

La programmation générique

❑ Définition d'une classe générique simple (suite)

- La classe Paire introduit une variable de type T, incluse entre <> après le nom de la classe

- Une classe générique (paramétrée) peut posséder plusieurs variables de type
  - utiliser l'opérateur virgule pour les séparer

- Les variables de type peuvent être utilisées tout au long de la définition de la classe pour spécifier le type de retour des méthodes, le type des attributs, ou même le type de certaines variables locales

- rien n'empêche également d'utiliser des attributs ou d'autres éléments avec des types bien définis, c'est-à-dire non paramétrable, comme int, String, etc.

La programmation générique

❑ Utilisation de la classe générique

- L'attribut premier utilise une variable de type
- Ainsi, comme son nom l'indique premier pourra être de n'importe quel type
- C'est celui qui utilisera cette classe qui spécifiera le type qu'il désire utiliser pour décrire l'objet de cette classe Paire
- Ainsi, vous instanciez une classe paramétrée en précisant le type que vous désirez prendre

La programmation générique

Utilisation de la classe générique (suite)

- Par exemple :

  Paire<String> ordre ;

- Ici, ordre est un objet de type Paire<String>

- Le paramètre de la classe Paire, ou exprimé autrement, la variable de type est String

- Ainsi, au moment de cette déclaration, à l'intérieur de la classe Paire, tout ce qui fait référence à T est remplacé par le véritable type, c'est-à-dire String

La programmation générique

- Exemple d'utilisation
  - Le programme suivant met en œuvre la classe Paire
  - La méthode minmax() statique parcourt un tableau de chaînes de caractères et calcule en même temps la valeur minimale et la valeur maximale
  - Elle utilise un objet Paire pour renvoyer les deux résultats

```
public class Test {
    public static void main(String[] args) {
            String[] phrase = {"Marie", "possède", "une", "petite", "lampe"};
            Paire<String> extrêmes = TableauAlg.minmax(phrase);
            System.out.println("min = "+extrêmes.getPremier());
            System.out.println("max = "+extrêmes.getDeuxième());
    }
}
```

La programmation générique

Instanciation/invocation

- une classe générique doit être instanciée pour être utilisée
- on ne peut pas utiliser un type primitif pour l'instanciation, il faut utiliser les classes enveloppantes
- on ne peut pas instancier avec un type générique
- une classe instanciée ne peut pas servir de type de base pour un tableau

```
Paire <String> p= new Paire<String> ("bonjour" , "Monsieur" )  //oui
// le constructeur doit contenir <....>
Paire <>p2=new Paire<>();     // non
Paire<int> p3=new  Paire<int>(1,2);   // non
Paire<Integer> p4=new  Paire<Integer>(1,2);   // oui
Paire<Paire> p5=new  Paire<Paire>();   // non
Paire<Paire<String>> p6=new  Paire<Paire<String>>(p,p);   // oui
Paire<Integer>[]tab=new Paire<Integer>[10];  //non
```

plusieurs types paramètres

- On peut utiliser plusieurs types paramètres

```
Public class PaireD<T,U>{
        private T premier;
        private U second;
        public PaireD(T x, U y){// en-tête du constructeur sans <T,U>
                premier =x;
                second= y;
        }
        public PaireD(){}
        public T getPremier(){ return premier;}
        public U getSecond(){ return second;}
        public void setPremier(T x){ this,premier=x;}
        public void setSecond(U y){ this,second=y;}
}
...
PaireD<Integer, String> p= new PaireD<Integer, String> (1,  " bonjour ");
```

La programmation générique

Utilisation du type paramètre

- le type paramètre peut être utilisé pour déclarer des variables (attributs) sauf dans une méthode de classe

- le type paramètre ne peut pas servir à construire un objet.

```
Publc class Paire<T> {
        . . .
        T var ; // oui
        T var = new T( ) ; // non
        T[ ] tab ; // oui
        T[ ] tab = new T[ 10 ] ; // non
        . . .
}
```

Méthodes et généricité

- Une méthode de classe (static) ne peut pas utiliser une variable du type paramètre dans une classe générique.

```
Publc class UneClasseGenerique<T> {
        . . .
        Public static void methodeDeClasse(){
                T var; // erreur à la compilation
                . . .
        }
}
```

Méthodes et généricité

- Une méthode peut être paramétrée par un type, qu'elle soit dans une classe générique ou non
- L'appel de la méthode nécessite de l'instancier par un type, sauf si le compilateur peut réaliser une *inférence de type*

```
public class MyClass{
    public static <T> void permute(T[] tab, int i, int j){
        T temp = tab[i];
        tab[i] = tab[j];
        tab[j] = temp;
    }
}
  ...
String[] tabs = new String[]{"toto","titi","tutu"};
Float[] tabf = new Float[]{3.14f,27.12f};
MyClass.<String>permute(tabs,0,2);
MyClass.permute(tabf,0,1);
```

La programmation générique

Méthodes et généricité (suite)

- Une méthode (de classe ou d'instance) peut être générique dans une classe non générique. Elle utilise alors son propre type paramètre.

```
publc class ClasseA<T> {
        . . .
        Public <T>  T premierElement( T[] tab){
                return tab[0]; // méthode d'instance
        }
        //<T> est placé après les modificateurs et avant le type renvoyé
        public static <U> U  dernierElement( U[] tab){//méthode de classe
                return tab[ tab.length-1];
        }
         //<T> est placé après les modificateurs et avant le type renvoyé
        . . .
}
```

- Pour utiliser une telle méthode on doit préfixer le nom de la méthode par le type d'instanciation entre < et >.

```
ClasseA a= new ClasseA();
String [] t= { "game " , " of " , " thrones " };
System.out.println(a.<String> premierElement(t));
System.out.println(ClasseA.<String> dernierElement(t));
```

La programmation générique

Méthodes et généricité (suite)

Une méthode (de classe ou d'instance) peut être générique dans une classe générique. Elle peut utiliser le type paramètre de la classe et son propre type paramètre.

```
Public class Paire<T>{
        private T premier;
        private T second;
        public Paire(){}
        public Paire(T x, T y){ // en-tête du constructeur sans <T>
                premier=x;
                second=y;
        }
        public T getPremier(){ return this.premier;}
        public T getSecond(){ return this.second;}
        public void setPremier( T x){ this.premier=x;}
        public void setsecond( T x){ this.secondr=x;}
        public void interchanger(){
                T temp=this.premier;
                this,premier=this.second;
                this.second=temp;
        }
        public <U> void voir (U var){
                System,out,println(" qui est là ? "+ var);
                System,out,println(" le premier est  "+ this.premier);
        }
        . . .
        Paire<Integer> p = new Paire<Integer> (1,2);
        p.<String> voir ("un ami");
}
```

La programmation générique

Exercice :

Réécrire les méthodes equals et toString pour les deux classes Paire et PaireD.

Généricité et héritage

une classe générique peut étendre une classe (générique ou pas)

```
Public class Triplet<T> extends Paire<T>{
        T troisieme ;
        . . .
}
```

- Triplet< T > est une sous classe de Paire< T >
- Triplet< String > est une sous classe de Paire< String >
- Triplet< String > n'est pas une sous classe de Paire< T >
- Triplet< String > n'est pas une sous classe de Triplet< Object > bien que String soit une sous classe de Object

Ce dernier point interdit donc une affectation du genre

```
Triplet<Integer> t = new Triplet<Shor t>();
```

La programmation générique

Limites pour les variables de type

- Par moment, une classe paramétrée ou une méthode paramétrée doit placer des restrictions sur des variables de type.

- Visualisons le problème au travers d'un exemple :

```
class TableauAlg {
      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;
                   }
     }
}
```

La programmation générique

Limites pour les variables de type (suite)

- Mais dans cette méthode générique min() un problème demeure: En effet, la variable plus petite possède un type T, ce qui signifie qu'il pourrait s'agir d'un objet d'une classe ordinaire

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

- D'ailleurs, si on écrit ce code, on obtient une erreur du compilateur en spécifiant que cette méthode compareTo() n'est pas connue pour un type quelconque T

La programmation générique

Limites pour les variables de type (suite)

- La solution consiste à restreindre T à une classe qui implémente l'interface Comparable
- Pour y parvenir, on doit donner une limite pour la variable de type T :

```
class TableauAlg {
        public static <T extends Comparable> T min(T[] tab) {
                ...
        }
}
```

Limites pour les variables de type (suite)

- Désormais, la méthode générique min() ne peut être appelée qu'avec des tableaux de classes qui implémentent l'interface Comparable, comme String, Date, etc.
- Pour y parvenir, on doit donner une limite pour la variable de type T :

• *Appeler min() avec un tabelau de Rectangle produit une erreur de compilation car la classe Rectangle n'implémente pas Comparable*

• *Nous pouvons nous demander pourquoi utiliser le mot clé extends plutôt que le mot clé implements dans cette situation (après tout, Comparable est une interface)*

La programmation générique

- La notation :
  <T extends TypeLimitant>
- indique que T doit être un sous-type du **Typelimitant**

- T et le typelimitant peuvent être une classe ou une interface

- Le mot clé extends a été choisi car il constitue une approximation raisonnable du concept de sous-type et que les concepteurs Java ne souhaitaient pas ajouter un nouveau mot clé (comme sub)

La programmation générique

- Une variable peut avoir plusieurs limites :
  <T extends Comparable & Serializable >
- Les types limitants sont séparés par des esperluettes (&) car les virgules sont utilisées pour séparer les variables de type
- Comme pour l'héritage Java, on peut disposer d'autant d'interfaces qu'on le souhaite, mais une seule des limites peut être une classe
- De plus, si on dispose d'une classe agissant comme élément limitant, elle doit figurer en première place dans la liste

La programmation générique

Exemple d'utilisation

- La méthode générique minMax() de la classe TableauALg calcule le minimum et le maximum d'un tableau générique et renvoie un Paire<T>

```
public class Test {
	public static void main(String[] args) {
		String[] phrase = {"Marie", "possède", "une", "petite", "lampe"};
		Paire<String> extrêmes = TableauAlg.minmax(phrase);
		System.out.println("min = "+extrêmes.getPremier());
		System.out.println("max = "+extrêmes.getSecond());
	}
}
```

La programmation générique

Exemple d'utilisation

```
class TableauAlg {
    public static <T extends Comparable>
    Paire<T> minmax(T[] tab) {
          if (tab==null || tab.length==0)
                return null;
          T min = tab[0];
          T max = tab[0];
          for (T élément : tab) {
                if (min.compareTo(élément) > 0)
                            min = élément;
                if (max.compareTo(élément) < 0)
                            max = élément;
           }
           return new Paire<T>(min, max);
    }
}
```

```
Public class Paire<T>{
                private T premier;
                private T second;
                public Paire(){
                            premier=null;
                            second=null;
                }
                public Paire(T x, T y){
                            premier=x;

second=y;
                }
                public T getPremier(){ return this.premier;}
                public T getSecond(){ return this.second;}
                public void setPremier( T x){ this.premier=x;}
                public void setsecond( T x){ this.secondr=x;}
                public void interchanger(){
                            T temp=this.premier;
                            this,premier=this.second;
                            this.second=temp;
                }

}
```

La programmation générique

Règles d'héritage pour les types génériques

- Soit une classe de base (ancêtre) Personne et une sous-classe (classe dérivée) Elève qui hérite donc de Personne

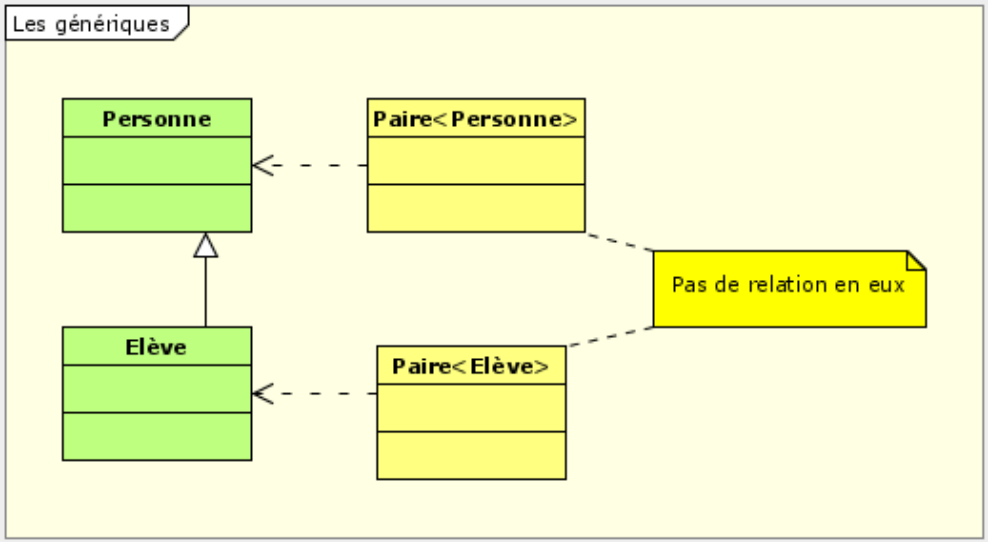
- La question qui se pose "Est-ce que Paire<Elève> est une sous-classe de Paire<Personne> ?". la réponse est "Non "

- Par exemple, le code suivant ne sera pas compilé :

• Elève[] élèves = ... ;
• Paire<Personne> personne = TableauAlg.minmax(élèves) ;

Règles d'héritage pour les types génériques

- La méthode minmax() renvoie un Paire<Elève>, et non pas un Paire<Personne>, et il n'est pas possible d'affecter l'une à l'autre
- En général, il n'existe pas de relation entre Paire<S> et Paire<T>, quels que soient les éléments auxquels S et T sont reliés



La programmation générique

Types joker

- Comme nous venons juste de le découvrir au niveau de l'héritage, un système de type trop rigide n'est pas toujours très agréable à utiliser. Les concepteurs Java ont:
- inventé une "sortie de secours" ingénieuse (tout en étant sûre) : le type joker.
- - Par exemple, le type joker :
  - **<? extends Personne>** remplace toute paire générique dont le paramètre type est une sous-classe de Personne, comme Paire<Elève>, mais pas, bien entendu, Paire<String>

La programmation générique

Types joker

- Imaginons que nous voulions écrire une méthode qui affiche des paires de personnes, comme ceci :

```
public static void afficheBinômes(Paire<Personne> personnes) {
        Personne premier = personnes.getPremier();
        Personne second= personnes.getSecond();
        System.out.println(premier.getNom()+" et "+deuxième.getNom()+" sont ensembles.");
}
```

- Avec:

La programmation générique

Types joker

```
Public class Paire<T>{
        private T premier;
        private T second;
        public Paire(){
                premier=null;
                second=null;
        }
        public Paire(T x, T y){
                premier=x;
                second=y;
        }
        public T getPremier(){ return this.premier;}
        public T getSecond(){ return this.second;}
        public void setPremier( T x){ this.premier=x;}
        public void setsecond( T x){ this.secondr=x;}
        public void interchanger(){
                T temp=this.premier;
                this,premier=this.second;
                this.second=temp;
        }

}
```

```
class Personne {
        private String nom, prénom;
        public String getNom() { return nom; }
        public String getPrénom() { return
        prénom; }
        public Personne(String nom, String
        prénom) { this.nom=nom;
            this.prénom=prénom;}
        }
class Elève extends Personne {
        private double[] notes = new
        double[10];
        private int nombreNote = 0;
        public Elève(String nom, String
        prénom) { super(nom, prénom); }
        public void ajoutNote(double note) {
            if(nombreNote<10)
             notes[nombreNote++] = note; }
}
```

La programmation générique

Types joker

- Comme nous l'avons vu précédemment, nous ne pouvons pas passer un Paire<Elève> à cette méthode ***afficheBinôme()***
- La solution est simple pour résoudre ce problème, il suffit d'utiliser un joker :

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

Ici, nous indiquons que nous pouvons utiliser n'importe quelle classe qui fait partie de l'héritage de Personne, classe de base comprise

La programmation générique

Voici, donc une utilisation possible :

```
public class Main {
      public static void main(String[] args) {
                Personne personne1 = new Personne("Lagafe", "Gaston");
                Personne personne2 = new Personne("Talon", "Achile");
                Paire<Personne> bynômePersonne = new Paire<Personne>(personne1,  personne2);
                Elève élève1 = new Elève("Guillemet", "Virgule");
                Elève élève2 = new Elève("Mouse", "Mickey");
                Paire<Elève> bynômeElève = new Paire<Elève>(élève1,élève2);
                afficheBinômes(bynômePersonne);
                afficheBinômes(bynômeElève);
       }
     public static void afficheBinômes(Paire<? Extends Personne> personnes) {
                Personne premier = personnes.getPremier();
                Personne second= personnes.getSecond();
                System.out.println(premier.getNom()+" et "+deuxième.getNom()+" sont ensembles.");
     }
}
```

Pour résumer

Objectifs du polymorphisme paramétrique

- éviter des duplications de code ;
- éviter des *typecast* et des contrôles dynamiques
- Effectuer des contrôles à la compilation (statiques)
- faciliter l'écriture d'un code générique et réutilisable
- 2004 : Java 1.5 (Tiger) - JDK 5.0
  - Paramétrage des classes et des interfaces
  - L'API des collections devient générique