

Chapitre VII

Généricité

Principes

- Paramétrer une classe ou une méthode par un type:
 - une pile de X
- (En java toute classe étant dérivée de Object, cela permet d'obtenir une forme de généricité sans contrôle des types
 - une pile d'Object)
- La généricité en Java est un mécanisme "statique" assez complexe qui permet une vérification de types
- la généricité existe dans d'autres langages (exemple C++ et Ada) (mais de façon différente)

Exemple: File

```
public class Cellule<E>{
    private Cellule<E> suivant;
    private E element;
    public Cellule(E val) {
        this.element=val;
    }
    public Cellule(E val, Cellule suivant){
        this.element=val; this.suivant=suivant;
    }
    public E getElement(){ return element;}
    public void setElement(E v){
        element=v;
    }
    public Cellule<E> getSuivant(){ return suivant;}
    public void setSuivant(Cellule<E> s){
        this.suivant=s;
    }
}
```

Suite

```
class File<E>{
    protected Cellule<E> tete;
    protected Cellule<E> queue;
    private int taille=0;
    public boolean estVide(){
        return taille==0;
    }
    public void enfiler(E item){
        Cellule<E> c=new Cellule<E>(item);
        if (estVide())
            tete=queue=c;
        else{
            queue.setSuivant(c);
            queue=c;
        }
        taille++;
    } //..
}
```

suite

```
public E defiler(){
    if (estVide())
        return null;
    Cellule<E> tmp=tete;
    tete=tete.getSuivant();
    taille--;
    return tmp.getElement();
}
public int getTaille(){
    return taille;
}
}
```

Usage

```
Cellule<Integer> cel=new Cellule<Integer>(23);  
//Cellule<Integer> cel=new Cellule<>(23); //java 8  
File<Integer> fi=new File<Integer>();  
//File<Integer> fi=new File<>(); //java 8
```

```
File<String> fs=new File<String>();  
File<Object> fobj=new File<Object>();  
String[] st={"zéro","un","deux",  
            "trois","quatre","cinq"};  
for(int i=0;i<st.length;i++){  
    fs.enfiler(st[i]);  
    fi.enfiler(i);  
}
```

Remarques

- dans `public class Cellule<E>` `E` est un type en paramètre et dans `File<String>` `fs` `String` est un argument type.
- Une déclaration de type générique peut avoir plusieurs paramètres:
 - `Map<K,V>`
- Contrôle de type
 - `fs.enfiler(4)` est refusé à la compilation

Types Bruts (raw types)

- Pour des raisons de compatibilité, on peut instancier une classe générique sans donner de type en argument:

```
File<Integer> fileInt=new File<Integer>();  
File file=fileInt; //ok
```

Mais:

```
File file=new File(); // ok  
File<Integer> fileInt= file; //warning
```

Et:

```
File<Integer> fileInt=new File<Integer>();  
File file= fileInt;  
file.enfiler(5); //warning
```

- (l'effacement de type explique le résultat)

Types génériques, pourquoi?

□ Vérification de type:

```
List myIntList = new LinkedList();  
myIntList.add(new Integer(0));  
Integer x = (Integer) myIntList.iterator().next();
```

Et:

```
List<Integer> myIntList = new LinkedList<Integer>();  
myIntList.add(new Integer(0));  
x=myIntList.iterator().next();
```

Invocation et type en paramètre

```
public interface List <E>{  
    void add(E x);  
    Iterator<E> iterator();  
}  
public interface Iterator<E>{ E next();    boolean hasNext();}
```

List<Integer> pourrait correspondre à (comme en C++):

```
public interface IntegerList {  
    void add(Integer x);  
    Iterator<Integer> iterator();  
}
```

Mais... une déclaration d'un type générique crée un vrai type (qui est compilé comme un tout) et il n'y a pas de type pour List<Integer>

Typage

- Une invocation ne crée pas un nouveau type:
 - `(fs.getClass()==fi.getClass())` est vrai
 - la classe est ici `File`
 - il s'agit surtout d'un contrôle (effectué à la compilation)
 - à l'exécution `fi` n'a plus aucune information sur quelle invocation a permis sa construction

Conséquences

- Aucune instanciación n'est possible pour un type argument
 - Dans l'exemple: `E v=new E () ;`
est impossible
 - Pas de tableau de `E`

Contrôle du type

- Pourtant, on peut passer un objet d'un type avec paramètre à une méthode.
- Comment se fait le passage des paramètres?
 - « type erasing » le compilateur passe le type le plus général (Object) et utilise le cast pour assurer le contrôle du typage.

Types génériques imbriqués

```
public class FileSimpleChainageb <E>{
    public class Cellule{
        private Cellule suivant;
        private E element;
        public Cellule(E val) {
            this.element=val;
        }
        public Cellule(E val, Cellule suivant){
            this.element=val;
            this.suivant=suivant;
        }
        public E getElement(){
            return element;
        }
        public void setElement(E v){
            element=v;
        }//...
    }
}
```

Suite

```
public Cellule getSuivant(){
    return suivant;
}
public void setSuivant(Cellule s){
    this.suivant=s;
}
}
protected Cellule tete;
protected Cellule queue;
private int taille=0;
public boolean estVide(){
    return taille==0;
}
public int getTaille(){
    return taille;
}
```

Fin...

```
public void enfiler(E item){
    Cellule c=new Cellule(item);
    if (estVide())
        tete=queue=c;
    else{
        queue.setSuivant(c);
        queue=c;
    }
    taille++;
}
public E defiler(){
    if (estVide())
        return null;
    Cellule tmp=tete;
    tete=tete.getSuivant();
    taille--;
    return tmp.getElement();
}
}
```


Sous-typage

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls; //1  
lo.add(new Object()); //2  
String s = ls.get(0); //3 !
```

Si A est une extension de B , $F\langle A \rangle$ n'est pas une extension de $F\langle B \rangle$:
//1 est interdit

Pour les tableaux:

- si A est une extension de B un tableau de A est une extension de tableau de B .
- //1 est autorisé, mais ensuite //2 est interdit

Joker '?'

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) { System.out.println(e);}  
}
```

Ne fonctionne pas avec une `Collection<Integer>`

Une collection d'un type inconnu ('?') :

```
void printCollection(Collection<?> c) {  
    for (Object e : c){ System.out.println(e);}  
}
```

est possible (un type inconnu hérite de `Object`).

Mais

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object()); // erreur compilation  
(un Object ne peut pas être converti en ce type inconnu)
```

Mais

- Ce n'est pas suffisant...
 - On peut vouloir borner le type paramètre:
comparable est une interface générique qui indique la possibilité de comparer des objets
`class valeur implements Comparable<Valeur>{...}`
 - Une SortedCollection est construite avec des éléments de la classe E qui implémentent Comparable<E> d'où:
`interface SortedCollection<E extends Comparable<E>>{}`

Exemple

```
static double somme(List<Number> l){
    double res=0.0;
    for(Number n:l)
        res+=n.doubleValue();
    return res;
}
public static void main(String[] st){
    List<Integer> l= new ArrayList<Integer>();
    for(int i=0;i<10;i++)l.add(i);
    double s=somme(l); //incorrect
}
```

Mais

Type paramètre borné

- Il faut « au moins » un number:
- `List<? extends Number>` une liste constituée de n'importe quel type dérivé de `Number` (une sous-classe de `Number`)

```
static double somme2(List<? extends Number> l){  
    double res=0.0;  
    for(Number n:l)  
        res+=n.doubleValue();  
    return res;  
}
```

Type paramètre borné

- Mais pour:

```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```

- pour ajouter un entier à la liste il faut que la liste soit une liste d' « au plus » un Integer (quelque chose qui peut être affecté par un Integer- une super classe de Integer)

Principes

- `copy(src, dst)` (ou `dst=src`)
 - `src` est un paramètre « in »
 - `dst` est un paramètre « out »
 - pour `src` il faut un « extends » pour pouvoir être copiée: `src` doit être d'une extension du type de `dst`
 - pour `dst` il faut un « super » pour pouvoir être écrit: `dst` doit être d'une super classe du type de `src`.

Sous-typage

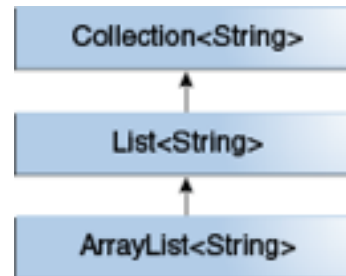
- `List<?>` est une super classe de `List<Object>`

Et:

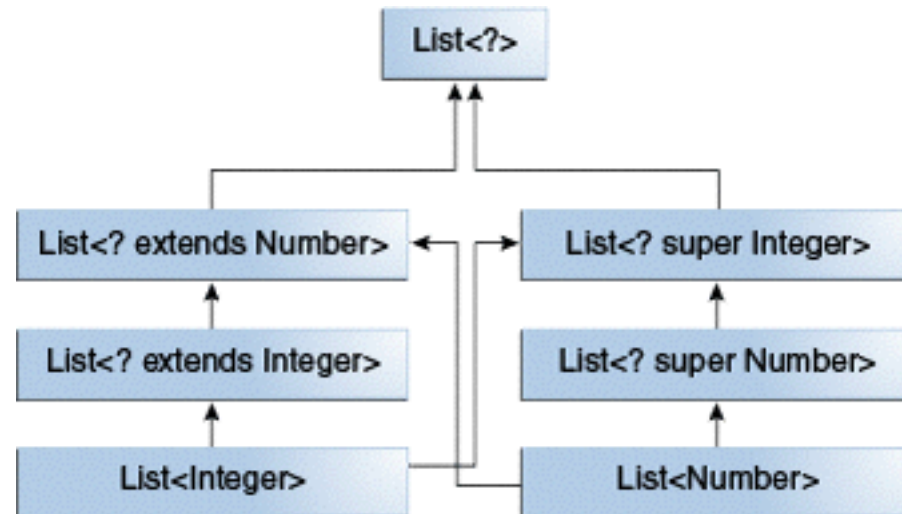
- `List<?>`
 - `List<? extends Number>`
 - `List<Number>`
 - `List<? extends Integer>`
 - `List<Integer>`

Hiérarchie

□ on a bien sûr:



□ et:



Mais attention

```
File<?> str=new File<String>();  
str.enfiler("un");
```

provoque une erreur à la compilation:

```
enfiler(capture of ?) in generique.File<capture of ?>  
cannot be applied to (java.lang.String)
```

de même:

```
File<? extends Number> num=new File<Number>();  
num.enfiler(Integer.valueOf(12));
```

en effet `File<? extends Number>` peut être par exemple une `File` d'un type dérivé de `Number`.

Par contre:

```
File<? super Number> num=new File<Number>();  
num.enfiler(Integer.valueOf(12));
```

est correct

Quelques explications

- ? peut correspondre à n'importe quel type
enfiler(a) où a est de type A ne peut fonctionner
si le type correspondant à ? est dérivé de A
- de même ? dans <? extends X> ne peut
fonctionner car si ? est Y dérivé de X il faut un
paramètre d'une classe dérivée de Y
- par contre ? dans <? super X> ne pouvant
correspondre qu'à une classe "avant" X, tout Z
dérivé de X fonctionne

Mais

- inversement pour la valeur retournée (avec la méthode défiler par exemple)
 - pour `<?>` quelque soit le type `X` correspondant on peut l'affecter à `Object` et à `X`
 - idem pour `<? extends X>`
 - mais pour `<? super Y>` si `Z` correspond à ? pour `T` un type quelconque on ne peut savoir si `T` peut être affecté par un `Z`

Méthodes génériques

- Supposons que l'on veuille convertir en tableau une File de E
 - on ne peut pas créer un tableau de E

Méthodes génériques...

- Méthode générique: paramétrer la méthode par le type, mais...

```
public <E> E[] toArray(File<E> f){  
    E[] tab=new E[f.getTaille()]; //non  
    for(int i=0;i<f.getTaille();i++)  
        tab[i]=f.defiler();  
}
```

- On peut aussi utiliser la classe Object

Object

```
public static <E> Object[] toArray(File<E> f){  
    Object[] tab=new Object[f.getTaille()];  
    for(int i=0;i<f.getTaille();i++)  
        tab[i]=f.defiler();  
    return tab;  
}
```

toArray est une méthode générique <E> est le type paramètre de la méthode:

mais on perd l'avantage du contrôle de type.

enTableau1

```
public E[] enTableau1(E[] tab){
    Object[] tmp = tab;
    int i=0;
    for(Cellule<E> c= tete; c != null && i< tab.length;
        c=c.getSuivant())
        tab[i++] = c.getElement();
    return tab;
}
```

- (ici on suppose `Tableau1` est une nouvelle méthode de `File`)

```
File<String> fs=new File<String>();
String[] u;
u=fs.enTableau1(new String[fs.getTaille()]);
```


enTableau

- Mais,
 - il faut que le tableau passé en paramètre soit un tableau de E, alors qu'un tableau d'une super-classe de E devrait fonctionner (si F est une superclasse de E un tableau de F peut contenir des objets E).
 - avec une méthode générique:

enTableau

```
public <T> T[] enTableau(T[] tab){
    Object[] tmp = tab;
    int i=0;
    for(Cellule<E> c= tete; c != null && i< tab.length;
                                   c=c.getSuivant())
        tmp[i++] = c.getElement();
    return tmp;
}
```

- la déclaration impose que le type du tableau retourné soit du type du tableau de l'argument
- Notons que tmp est un tableau d'Object ce qui est nécessaire pour le getSuivant
- Notons que normalement il faudrait que T soit une superclasse de E (à l'exécution il peut y avoir une erreur).
- Notons enfin que 'T' ne sert pas dans le corps de la méthode.

Remarque

```
public <T> T[] enTableaubis(T[] tab){
    int i=0;
    for(Cellule<E> c= tete;
        c != null && i< tab.length;
        c=c.getSuivant())
        tab[i++] = (T)c.getElement();
    return tab;
}
```

- **provoque un warning** "Cellule.java uses unchecked or unsafe operations".
- **(l'"effacement de type" ne permet pas de vérifier le type)**

Avec Reflection...

- Une autre solution peut être, si on veut créer un vrai tableau, d'utiliser `Array.newInstance` de la classe: `java.lang.reflect`

Exemple avec Reflection

```
public E[] enTableau2(Class<E> type){
    int taille = getTaille();
    E[] arr=(E[])Array.newInstance(type,taille);
    int i=0;
    for(Cellule<E> c= tete; c != null && i< taille;
        c=c.getSuivant())
        arr[i++] = c.getElement();
    return arr;
}
```

- on crée ainsi un tableau de "E"
- "unchecked warning": le cast (E[]) n'a pas le sens usuel
- pour fs déclaré comme précédemment on aura:

```
String[] u=fs.enTableau2(String.class); //ok
```

```
Object[] v=fs.enTableau2(Object.class); //non
```

- car le type doit être exact

Avec une méthode générique

```
public <T> T[] enTableau3(Class<T> type){
    int taille = getTaille();
    T[] arr=(T[])Array.newInstance(type,taille);
    int i=0;
    Object[] tmp=arr;
    for(Cellule<E> c= tete; c != null && i< taille;
    c=c.getSuivant())
        tmp[i++] = c.getElement();
    return arr;
}
```

Type borné

- On cherche le max dans un tableau d'un type E: il faut que le type E implémente Comparable.

```
static <E extends Comparable<E>> E max(E[] tab) {  
    E tmp = null;  
    if (tab.length > 0) {  
        tmp = tab[0];  
        for (int i = 1; i < tab.length; i++) {  
            if (tmp.compareTo(tab[i]) < 0) {  
                tmp = tab[i];  
            }  
        }  
    }  
    return tmp;  
}
```

Inférence de type

- Comment invoquer une méthode générique?
- l'inférence de type en Java permet de trouver le type adapté le « plus spécifique »
- Exemple:

```
static <T> T identite(T obj){  
    return obj;  
}
```


Invocations

- On peut explicitement préciser le type:

```
String s1="Bonjour";  
String s2= Main.<String>identite(s1);
```

- Mais le compilateur peut, lui-même, trouver le type le plus spécifique:

```
String s1=identite("Bonjour");
```

- On aura:

```
Object o1=identite(s1);           //ok  
Object o2=identite((Object)s1);   //ok  
s2=identite((Object) s1);         //non!!!  
s2=(String)identite((Object) s1); //ok
```

Comment ça marche?

- Mécanisme de l'effacement ("erasure")
- Pour chaque type générique il n'y a qu'une classe:
Cellule<String> et Cellule<Integer> sont la même classe Cellule
- Effacement:
 - Cellule<String> -> Cellule
 - Cellule est un type brut
 - Pour une variable type:
 - <E> -> Object
 - <E extends Number> -> Number
- Le compilateur remplace chaque type paramètre par son effacement

Comment ça marche?

- Si le résultat de l'effacement du type générique ne correspond pas au type variable, le compilateur génère un cast:
 - par effacement le type variable de `File<E>` est `Object`
 - pour un "defiler" sur un objet `File<String>` le compilateur insère un cast sur `String`

Comment ça marche?

- A cause de l'effacement, rien de ce qui nécessite de connaître la valeur d'un argument type n'est autorisé. Par exemple:
 - on ne peut pas instancier un paramètre type: pas de `new T()` ou de `new T[]`
 - on ne peut pas utiliser `instanceof` pour une instance de type paramètre
 - on ne peut pas créer de tableau sur un type paramétré sauf s'il est non borné : `new List<String>[10]` est interdit mais `new List<?>[10]` est possible.

Comment ça marche?

- les "cast" sont possibles mais n'ont pas la même signification que pour les types non paramétrés:
 - le cast est remplacé par un cast vers le type obtenu par effacement et génère un "unchecked warning" à la compilation.
 - Exemple:
 - on peut caster un paramètre type `File<?>` vers un `File<String>` pour un enfiler (ce qui génère le warning)
 - A l'exécution si le type effectif est `File<Number>` cela passe... mais le defiler provoquera un `ClassCastException`.

Comment ça marche?

□ Exemple:

```
List<String> l=new ArrayList<String>();  
Object o=identite(l);  
List<String> l1=(List<String>)o; // warning  
List<String> l2=(List)o; //warning  
List<?> l3=(List) o; //ok  
List<?> l4=(List<?>)o; //ok
```

Application: surcharge

- avoir la même signature s'étend aux méthodes avec variables types
- même signature pour des variables types = même type et même borne (modulo bien sûr renommage!)
- signatures équivalentes par annulation: mêmes signatures obtenues par effacement

Surcharge

```
class Base<T>{  
    void m(int x){};  
    void m(T t){};  
    void m(String s){};  
    <N extends Number> void m(N n){};  
    void m(File<?> q){};  
}
```

```
m(int)  
m(Object)  
m(String)  
m(Number)  
m(File)
```


Et héritage...

□ exemple:

```
class D<T> extends Base<T>{  
    void m(Integer i){} //nouveau  
    void m(Object t){}  // redéfinit m(T t)  
    void m(Number n){}  // redéfinit m(N n)  
}
```