Ministère de l'Enseignement Supérieur et de la
Recherche Scientifique et de la Technologie
Institut Supérieur des
Études Technologiques Bizerte

La généricité

Généricité :Un premier 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 void setPremier(int x){this.premier=x;}
public int getSecond(){return this.second;}
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.

Généricité :1 ère Solution

• Utiliser un type universel « **Object** »

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

Inconvénients :

• les deux attributs peuvent des instances de classes différentes,

• on peut être amené à faire du transtypage ,

• on risque des erreurs de transtypage qui ne se détecteront qu'à l'exécution.

Généricité :1 ère Solution

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(); // downcasting ok
p.interchanger();
A a2 = (A) p.getPremier(); // erreur
```

Généricité :2 ème Solution

La solution est l'utilisation de la généricité, c'est-`a-dire l'usage de type paramétre.

La généricité est une notion de polymorphisme paramétrique.

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

Cette définition permet de définir ici des Paires contenant des objets de type arbitraire.

Généricité :Définition

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.

Généricité :Utilisation

- 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 <...> pour l'instanciation
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
```

- A l'exécution, il n'existe en fait qu'une classe qui est partagée par toutes les instanciations **p.getClass()==p4.getClass()**

Généricité :Conséquences

- Une variable statique n'existe qu'en un exemplaire (et pas en autant d'exemplaires que d'instanciations)

```
c l a s s Pa i r e <A>
}
. . .
. . .
n b I n s t a n c e s ++;
}

Pa i r e <I n t e g e r  > p1 = new Pa i r e (5)

Pa i r e <St r i n g  > p2= new Paire("bonjour");
```

Paire.nbInstances vaut 2 !

- System.out.println(**p1 instanceof Paire); => true**

Généricité :Effacement de type

- Type brut (raw type) = le type paramétré sans ses paramètres

  Paire p7=new Paire() ; !fonctionne !

- Attention: le compilateur ne fait pratiquement pas de vérification en cas de type brut et l'indique par un warning

Généricité :plusieurs types de 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 this.premier;}
public void setPremier(T x){this.premier=x;}
public U getSecond(){return this.second;}
public void setSecond(U y){this.second=y;}
}
...
PaireD<Integer , String> p = PaireD<Integer , String >(1, "bonjour");
```

Généricité :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.

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

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

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

méthodes et généricité

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

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

méthodes et généricité

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

```
public class ClasseA{
...
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 <T> T dernierElement(T[] tab){
return tab[tab.length-1];} // méthode de classe
//<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));
```

méthodes et généricité

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(T x, T y){ // en-tête du constructeur sans <T>
premier =x ; second = y;}
public Paire(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public T getSecond(){return this.second;}
public void setSecond(T y){this.second=y;}
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");
```

Limitation du type paramètre:paramétrage contraint (ou borné)

• Il existe des situations où il peut être utile d'imposer certaines contraintes sur les paramètre 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 interface.**

• 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`

Limitation du type paramètre

Instancier une classe générique à un type quelconque peut empêcher d'écrire certaines méthodes.

Par exemple pour la classe Paire, on voudrait connaitre le plus grands des 2 attributs : cela n'a de sens que si l'instanciation se fait avec un type dont les objets sont comparables donc qui implémente l'interface Comparable avec sa méthode **compareTo**.

Java permet de préciser que le type paramètre doit être ainsi :

```
public class Paire<T extends Comparable> { ...}
```

On peut limiter le type paramètre T par plusieurs interfaces et une classe au plus.

```
public class Paire<T extends Comparable & Cloneable & UneAutreClasse> { ...}
```

Comparable et Cloneable sont des interfaces et UneAutreClasse est une classe.

A l'instanciation le type choisi pour T devra implémenter les 2 interfaces et être une sous-classe de UneAutreClasse.

La classe Paire ne pourra être instanciée qu'avec le type **UneAutreClasse** ou un type dérivé

Héritage de classe générique

Différentes manières de dériver une classe générique

- en conservant les paramètres de type de la classe de base
- en ajoutant de nouveaux paramètres de type
- en introduisant des contraintes sur un ou plusieurs des paramètres de la classe de base
- 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 classe générique

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 paramètre de la classe dérivée

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

✗

Combinaisons de dérivations et d'instanciations

Classe générique dérivée d'une classe non générique

```
class Graphe{}
class GrapheEtiquete<TypeEtiq> extends Graphe{}
```

Classe générique dérivée d'une classe générique

```
class TableHash<TK,TV> extends Dictionnaire<TK,TV>{}
```

Classe dérivée d'une instanciation d'une classe générique

```
class Agenda extends Dictionnaire<Date,String>{}
```

Classe dérivée d'une instanciation partielle d'une classe générique

```
class Agenda<TypeEvt> extends
                    Dictionnaire<Date,TypeEvt>{}
```