Les types génériques

Contenu de la section

Quelques généralités

l'objectif

- améliorer le typage du langage en améliorant :
  - l'expressivité du code
  - la robustesse du code (le compilateur vérifie la correction)
  - la lisibilité du code
- **garantir la compatibilité ascendante et descendante avec le code propriétaire**

- **type générique = type paramétré**
- **ATTENTION : type générique de java # template de C++**

un exemple introductif

```
.....
List<String> list = new ArrayList<String>() ;
list.add("un");
list.add("deux");
list.add(new Date()) ;

for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);
    System.out.println("la valeur est : " + s);
}

.....
```

les classes « Collection » sont génériques dans java5

ERREUR : le compilateur vérifie que l'argument de add est une String

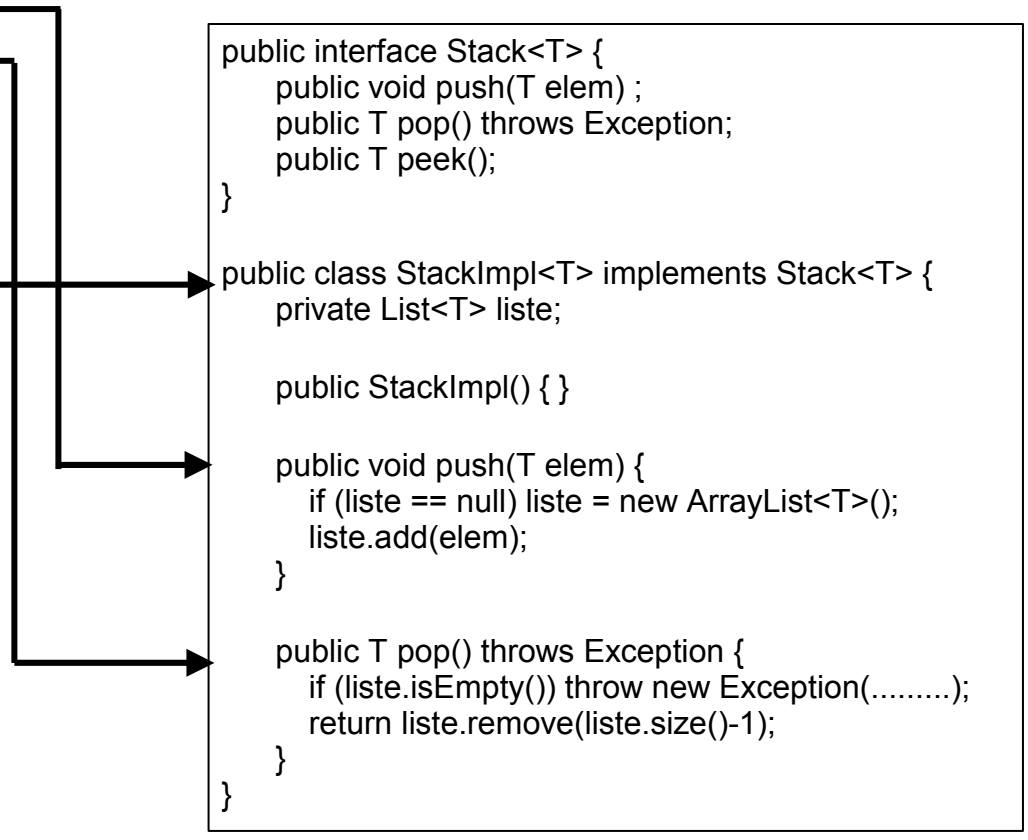
le compilateur ajoute le cast qu'implique la valeur de retour de get()

La définition des types génériques simples (1)

la syntaxe (1)

- déclaration d'une interface/classe paramétré par : Type<T> où T est un type NON primitif
- le paramètre T peut apparaître comme :
  - type d'un argument de méthodes
  - type de retour de méthodes
  - type de champs **non static**
  - type de variables locales

  - la relation d'extension
  - la relation d'implémentation



```
public interface Stack<T> {
      public void push(T elem) ;
      public T pop() throws Exception;
      public T peek();
}

public class StackImpl<T> implements Stack<T> {
      private List<T> liste;

      public StackImpl() { }

      public void push(T elem) {
         if (liste == null) liste = new ArrayList<T>();
         liste.add(elem);
      }

      public T pop() throws Exception {
         if (liste.isEmpty()) throw new Exception(.....);
         return liste.remove(liste.size()-1);
      }
}
```

La définition de types génériques simples (2)

la syntaxe (2)

- possibilité d'avoir plusieurs paramètres
- possibilité d'utiliser des types paramétrés dans les définitions

```
interface IGenericDefinition1 <S,T> {
  public void add(S key,List<T> elem);
  public List<T> get(S key);
}

public class GenericDefinition1 <S,T> implements IGenericDefinition1<S,T> {
  private Map<List<T>> map;

  public GenericDefinition1() { }

  public void add(S key, List<T> elem) {
    if (map == null) historique = new HashMap<S,List<T>>();
        map.put(key, elem);
    }

  public List<T> get(S key) {
    if (map == null) return null;
        return map.get(key);
    }
  }
}
```

Les types génériques et le sous-typage (1)

les sur-types paramétrés

- relation de sur-typage porte sur le type pas sur les paramètres
- TYPE<E> sur-type de TYPE'<E'> si et seulement si TYPE sur-type de TYPE' et E = E'

```
…..
List<Pomme> listl = new ArrayList<Pomme> ();
List<Fruit> listN = listl;          //ERREUR !!!
……
```

List<Pomme> est un sur-type de ArrayList<Pomme>

List<Fruit> n'est pas un sur-type de List<Pomme>
car cela autoriserait : listN.add(new Poire());

➔ **PROBLEME** :

```
public class Inventaire {
  List<List<Object>> store = new ArrrayList<List<Object>>();

  public void addList(List<Object> lst) { store.add(lst); }
  ……
}
```

```
Inventaire inventaire = new Inventaire();
…….
List<Livre> ls = new ArrayList<Livre>();
ls.add(…….);
inventaire.addList(ls);
……
```

INTERDIT : List<Object> n'est pas un sur-type de List<Livre> !!

le joker : n'importe quel type

- le joker (noté ?) signifiant "n'importe quel type"
- , permet de résoudre le problème

```
public class Inventaire {
        List<List<?>> store = new ArrrayList<List<?>>();

        public void addList(List<?> lst) { store.add(lst); }
        ……
}
```

Les types génériques et le sous-typage (2)

les bornes sur les paramètres de type

- il est possible de borner supérieurement un paramètre de type (en particulier le joker) par un type : T extends Type

```
interface Livre { ..... }
class Dictionnaire implements Livre { ..... }
class Roman implements Livre { …… }
class LivreScolaire implements Livre { …… }
```

```
public class InventaireLibrairie {
   public void addList(List<? extends  Livre> lst)
   ......
   ......
}
```

```
InventaireLibrairie inv = new InventaireLibrairie();
.......
List<Dictionnaire> l1= new ArrayList<Dictionnaire>();
List<Roman> l2 = new ArrayList<Roman>();
List<Livre> l3 = new ArrayList<Livre>();
.....
inv.add(l1);      //OK
inv.add(l2);      //OK
inv.add(l3);      //OK
........
```

- il est possible de borner inférieurement le joker par un type : T super Type

```
public class GenericDefinition2  {
   public void copy(Livre[ ] frms, List<? super Livre> lst) {
      if (frms == null || lst == null) return;
      for (int i=0;i<frms.length;i++) lst.add(frms[i]);
   }
}
```

ERREUR !! car cela autoriserait :
List<String> lS = new ArrayList<String>();
Forme[ ] objs = new Forme[10];
new GenericDefinition1().copy(objs,lS);

```
public class GenericDefinition1 {
   public void copy(Livre[ ] a0, List<?> a1) {
      if (a0 == null || a1 == null) return;
      for (int i=0;i<a0.length;i++) a1.add(a0[i]);
   }
}
```

Compléments sur les types génériques et le sous-typage

Les contraintes composites

- il est possible de composer des bornes supérieures via l'opérateur &
- il n'est pas possible de composer des contraintes extends et super
- il n'est pas possible de composer des contraintes inférieures

```
public class GenericDefinition3< T extends LIvre & Comparable> {
   .........
}
```

- Attention "**?**" n'est pas équivalent à **Object**

```
List<?> l = new ArrayList<Object>();                    //OK mais ....
//l.add(new Object());                                  //erreur
//l.add("bonjour");                                     //erreur
//List<?> l1 = new ArrayList<?>();                      //erreur
List<?> l1 = new ArrayList<String>();
//l1.add("bonjour");                                    //erreur

List<List<?>> list0 = new ArrayList<List<?>>();
//List<List<?>> list = new ArrayList<List>();
list0.add(new ArrayList<String>());
list0.add(new ArrayList());
List<List> list1 = new ArrayList<List>();
//List<List> list1 = new ArrayList<List<?>>();
list1.add(new ArrayList<String>());
list1.add(new ArrayList());
//List<List<?>> list3 = new ArrayList<List<Object>>();
```

L'interopérabilité avec les types "non génériques"

Les règles

- interopérabilité assurée pour permettre l'utilisation de code propriétaire
- génération de warning, lorsque la correction dy typage n'est pas garantie

```
public class GenericDefinition3 {
  public static void main(String[ ] args {
     List<String> lS0 = new ArrayList<String>();
     List l0 = lS0;                    //WARNING à la compilation
     List l1 = new ArrayList();
     l1.add(new Integer(2));           //OK
     List<String> lS1 = l1;            //WARNING à la compilation...
  }
}
```

dangereux parce que permet :
l0.add(new Integer(2)); //OK
String s0 = lS0.get(0); //OK à la compilation
//mais ClassCastException à l'exécution

dangereux parce que permet :
String s1 = lS1.get(0); //OK à la compilation
//mais ClassCastException à l'exécution

- TYPE se comporte comme TYPE< ?> avec des contraintes relachées
- on peut utiliser un TYPE<E> là où un TYPE est attendu
- on peut utiliser un TYPE là ou un TYPE<E> est attendu

Les mécanismes sous-jacents (1)

Le mécanisme d'effacement

- les paramètres de type sont tous effacés à la compilation
  - une seule classe au runtime indépendante des valeurs des paramètres de type
  - une seule instance de Class indépendante des valeurs des paramètres de type

```
List<Integer> listI = new ArrayList<Integer>();
List<String> listS = new ArrayList<String>();
.....
if (listI.getClass() == listS.getClass()) {
    System.out.println("type runtime : " + listS.getClass()) ;
}
```

listI et listS sont de type List au runtime

```
type runtime : class java.util.ArrayList
```

- le type d'un champs static ne peut pas être paramétré

Si cela était autorisé

```
package generics;
import java.util.List;

public class AA<T> {
    public static T static1;          //erreur
    public static List<T> static2;    //erreur
}
```

```
package generics;
   public class TestAA {
   public static void main(String[] args) {
        AA<String>.static1= "bonjour";
        AA<Date>.static1 = new Date();
        String str1 = AA<String>.static1;          //ClassCastException ....
        AA<String>.static2.add("bonjour");
        Date date = AA<Date>.static2.get(0);       //ClassCastException ....
   }
}
```

Les mécanismes sous-jacents (2)

Le mécanisme d'effacement (suite)

- toute référence au paramètre de type est IMPOSSIBLE à l'exécution
  - instanceof comportant un paramètre de type ➔ ERROR

```
public class GenericDefinition6 <S,T> {
  Hashtable table = new Hashtable();

  public void m(S key) {
     Object elem =  table.get(key);
     if (elem instanceof T) { ..... }                    //ERREUR
     if (elem instanceof List<T>) { …… }                 //ERREUR
     if (elem instanceof List<Number> { ..... }          //ERREUR
  }
}
```

résultat de l'exécution

- définition de méthodes ayant même signature après effacement :: définition multiple ➔ ERREUR

```
interface IGenericDefinition6 <S,T extends S, U> {
   public void add(List<S> elem, U key);
   public void add(List<T> elem, U key);
   public List<T> get(S key);
}
```

il faut utiliser le type non paramétré (i.e. List)

- l'API réflexive ignore les paramètres de type [utilisation de la signature sans les paramètres de type]

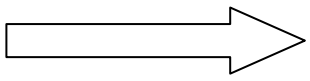
Les mécanismes sous-jacents (3)

La traduction "Generic Java"

- le compilateur infère la borne supérieure du paramètre de type à la génération du byte code

```
package generics;
interface A {
    public void m1();
    public void m2();
}
public class Test2<T extends A> {
    T refT;

    public Test2(T r) { refT = r; refT.m1(); }
    T getRefT() { return refT; }
}
```



```
public class generics.Test2 {
 generics.A refT;

 public Test2( generics.A);
   0  aload_0 [this]
   1  invokespecial java.lang.Object() [13]
   4  aload_0 [this]
   5  aload_1 [r]
   6  putfield generics.Test2.refT : generics.A [16]
   9  aload_0 [this]
  10  getfield generics.Test2.refT :  generics.A [16]
  13  invokeinterface generics.A.m1() : void [18] [nargs: 1]
  18  return
    Line numbers:
      [pc: 0, line: 9]
    Local variable table:
      [pc: 0, pc: 19] local: this index: 0 type: generics.Test2
      [pc: 0, pc: 19] local: r index: 1 type:  generics.A
    Local variable type table:
      [pc: 0, pc: 19] local: this index: 0 type: generics.Test2<T>
      [pc: 0, pc: 19] local: r index: 1 type: T

 generics.A getRefT();
   0  aload_0 [this]
   1  getfield generics.Test2.refT :  generics.A [16]
   4  areturn
    Line numbers:
      [pc: 0, line: 10]
    Local variable table:
      [pc: 0, pc: 5] local: this index: 0 type: generics.Test2
    Local variable type table:
      [pc: 0, pc: 5] local: this index: 0 type: generics.Test2<T>
```

- le compilateur rajoute les cast nécessaires à l'utilisation des classes

```
List<String> lst = new ArrayList<String>();
.......
String str = lst.get(0);
.....
```

Le compilateur rajoute le cast :
String str = (String) lst.get(0);

L'utilisation de types génériques

Quelques contraintes

Le compilateur doit pouvoir générer un byte code valide pour toutes valeurs du paramètre

- INSTANCIER un paramètre de type génère une erreur à la compilation
- INSTANCIER un tableau de types paramétrés génère une erreur à la compilation
- CAST faisant intervenir un type paramètré génère un warning à la compilation

```
public class Exemple<T> {
   ......
   List<T> list = new ArrayList<T>();
   T ref2 = null;                              //OK : référence
   list.add(ref2);
   if (ref2 != null) System.out.println(ref2.toString());
   //ref2 = new T() ;                          //ERREUR : instanciation
   ref2 = (T) "bonjour";                       //cast obligatoire mais WARNING !!!
   List<String>[ ] list1;                      //OK : référence
   //list1 = new List<String>[10] ;            //ERREUR : instanciation
   list1 = new List[10] ;                      //WARNING !!!
   list1[0] = new ArrayList<String>();
   list1[0].add(args[0]);
   List<T>[ ] list2;                           //OK : reference
   //list2 = new List<String>[10] ;            //ERREUR : instanciation
   list2 = new List[10] ;                      //WARNING !!!
   list2[0] = (List<T>) new ArrayList<T>();
   list2[0].add((T) args[0]);
   T[ ] ref1 = null;                           //OK mais !!!!
   //ref1 = new T[10] ;                        //ERREUR
   ref1 = (T[]) new String[10];                //cast obligatoire mais WARNING !!!
   ref1[0] = ref2;
   ..........
```

Compléments : les méthodes génériques

la syntaxe

- possibilité de définir des méthodes dont le type des arguments est déterminé à l'exécution par INFERENCE
- les paramètres de type sont déclarés avant le type de retour

```
public Exemple2 {
  public static <T> void copy(T[ ] a0, List<T> a1) {
     if (a0 == null || a1 == null) return ;
     for (int i = 0; i <a0.length; i++) {
          a1.add(a0[i]);
     }
}
```

TYPE1 sur-type de TYPE2 => TYPE1[] sur-type de TYPE2[ ]

```
public TestExemple2 {
   public static void main(String[ ] args) {
      Object[ ] tObj = new Object[20];
      String[ ] tStr = new String[20];
      Integer[ ] tInt = new Integer[20];
      List<Object> lObj = new List<Object>();
      List<String> lStr = new List<String>();
      List<Integer> lInt = new List<Integer>();
      Exemple2.copy(tObj,lObj);          //T inféré est Object
      Exemple2.copy(tStr,lStr);          //T inféré est String
      Exemple2.copy(tInt,lObj);          //T inféré est Object
   }
}
```