Typage de la généricité en Java

F. Barthélemy

17 mai 2005

1 Le shéma général

La g´en´ericit´e en Java est un ajout r´ealis´e sur un langage existant depuis d´ejà une dizaine d'ann´ees, avec des contraintes de coçmpatibilit´e ascendante. On doit aborder la question à partir de deux grands principes :

- les règles de typage ne sont pas chang´ees, il y a juste de nouveaux types avec de nouvelles règles pour d´efinir la relation de sous-typage.
- la g´en´ericit´e n'existe pas dans le code compil´e, au niveau de la JVM. Cela entraîne quelques restrictions dans l'usage de la g´en´ericit´e.

1.1 De nouveaux types

Voyons un exemple de classe g´en´erique. Il s'agit d'une classe `Paire` qui mod´elise une paire de deux ´el´ements du même type.

```
class Paire<T>{
    T x, y;
    Paire(T x1, T y1){
        x = x1;
        y = y1;
    }
    T first(){
        return x;
    }
    T second(){
        return y;
    }
    void setFirst(T x1){
        x = x1;
    }
    void setSecond(T y1){
        y = y1;
    }
```

```
    void invert(){
        T tmp = x;
        x = y;
        y = tmp;
    }
    boolean equals(Paire<T> o){
        return o.x.equals(x) && o.y.equals(y);
    }
}
class TestPaire1{
    public static void main(String[] args){
        Paire<String> p = new Paire<String>("toto","le ´ehros");
    }
}
```

On voit dans cet exemple apparaître de nouveaux types par rapport à ceux qu'on avait jusqu'ici.

Rappel des types de Java avant le g´en´ericit´e :

- types primitifs
- types tableaux
- noms de classe
- noms d'interface

Ici, nous avons en plus :

- des variables de type (ex : `T`, utilis´e dans la classe `Paire`)
- des classes g´en´eriques non instanci´ees (ex : `Paire<T>` utilis´e dans la classe, pour `equals`). Les paramètres des classes sont encore des variables.
- des classes g´en´eriques instanci´ees (ex : `Paire<String>` utilis´e pour les instances de la classe). Les paramètres des classes sont remplac´es par des types.

Il faut savoir comment ces nouveaux types se raccordent aux anciens d'une part, entre eux d'autre part, en ce qui concerne l'h´eritage et la relation de sous-typage.

1.2 Relation de sous-typage

Pour m´emoire, voici la relation de sous-typage pour les anciens types : Règles fondamentales :

- Si C est une sous-classe de C', alors $C \subseteq C'$
- Si I est une sous-interface de I', alors $I \subseteq I'$
- Si une classe C impl´emente une interface I, $C \subseteq I$
- Si $T \subseteq T'$, alors $T[] \subseteq T'[]$
- Si $T \subset eqT'$ et $T' \subseteq T''$ alors $T \subseteq T''$

Règles techniques :

- nil est le type de null.
- Si T est un type, alors $T \subseteq T$
- Si T est un type, alors $T[] \subseteq Object$

– Si C est une classe, alors $nil \subseteq C$
– Si T est un type, alors $nil \subseteq T[]$

Et maintenant voici les règles pour les types param´etr´es :

– Si une variable de type T est d´eclar´ee avec une clause `extends C`, alors $T \subseteq C$ (sinon, par d´efaut, $T \subseteq Object$)
– Si une classe $C < T1, \ldots, T_n >$, est d´eclar´ee avec une clause `extends` $C < T_i \ldots, T_j >$, alors $C < T1, \ldots, T_n > \subseteq C' < T_i \ldots, T_j >$ (sinon, par d´efaut $C < T1, \ldots, T_n > \subseteq Object$)
  Note : ici, les T_k sont des variables de types.
– Si une classe $C < T1, \ldots, T_n >$, est d´eclar´ee avec une clause `extends` $C < T_i \ldots, T_j >$, alors pour toute substitution $\theta = [T_1/C_1, \ldots T_n/C_n, T_i/C_i, \ldots T_j/C_j]$ on a $C < T1\theta, \ldots, T_n\theta > \subseteq C' < T_i\theta \ldots, T_j\theta >$

Exemple :

```
class Paire<T>{
    T x, y;
    Paire(T x1, T y1){
        x = x1;
        y = y1;
    }
    T first(){
        return x;
    }
    T second(){
        return y;
    }
    void setFirst(T x1){
        x = x1;
    }
    void setSecond(T y1){
        y = y1;
    }
    void invert(){
        T tmp = x;
        x = y;
        y = tmp;
    }
    boolean equals(Paire<T> o){
        return o.x.equals(x) && o.y.equals(y);
    }
}
class Triplet<T> extends Paire<T>{
    T z;
    Triplet(T x1, T y1, T z1){
        super(x1,y1);
```

```
        z = z1;
    }
    T third(){
        return z;
    }
    void setThird(T z1){
        z = z1;
    }
}
class TestPaire{
    public static void main(String[] args){
        Paire<String> test = new Paire<String>("ici","la");
        System.out.println(test.first());
        System.out.println(test.second());
        if (test.equals(new Paire<String>("ici","la")))
            System.out.println("Eagux");
        Triplet<String> retest =
            new Triplet<String>("un","deux","trois");
    }
}
```

Voici ce que les règles nous disent, entre autres :

- $Paire < T > \subseteq Object$
- $Triplet < T > \subseteq Paire < T >$
- $Paire < String > \subseteq Object$
- $Triplet < String > \subseteq Paire < String >$
- $Triplet < Integer > \subseteq Paire < Integer >$

Cela semble assez naturel. Ce qui l'est moins au premier abord, ce sont des relations qui ne sont pas donn´ees dans les règles et qui sont donc fausses en Java.

- $Paire < Integer >$ n'est pas un sous-type de $Paire < Object >$ bien que $Integer \subseteq Object$
- $Paire < Integer >$ n'est pas un sous-type de $Paire < T >$ (où T est une variable de type)

1.3 Justification des limites du sous-typage

La relation de sous-typage permet notamment de d´efinir quelles affectations sont bien typ´ees.

Si on avait $Paire < Integer > \subseteq Paire < Object >$, alors le programme suivant serait bien typ´e. Or il n'est clairement pas correct.

```
class PasSousType1{
    public static void main(String[] args){
        Paire<Integer> unePaire = new Paire<Integer>(12,45);
        Paire<Object> laMeme = unePaire;
        unePaire.setFirst("archi");
```

```
    }
}
```

```
> javac PasSousType1.java
PasSousType1.java:4: incompatible types
found   : Paire<java.lang.Integer>
required: Paire<java.lang.Object>
        Paire<Object> laMeme = unePaire;
                               ^
1 error
```

Une relation de sous-typage $Paire < Integer > \subseteq Paire < T >$ autoriserait le programme suivant, auquel on ne sait pas donner de sens.

```
class PasSousType2<T> extends Paire<T>{
    void uneMethode(){
        Paire<T> p = new Paire<String>("arty","olm");
    }
}
```

1.4 Conséquences des relations de sous-typage

Les relations de sous-typage suivantes (d´ejà vues) autorisent un certain nombre de choses.

- $Triplet < T > \subseteq Paire < T >$
- $Triplet < String > \subseteq Paire < String >$
- $Triplet < Integer > \subseteq Paire < Integer >$

Notamment : l'affectation d'une instance de la sous-classe dans la super-classe et la cast de la sous-classe vers la super-classe.

Exemples :

```
class Exemple<T>{
    Paire<T> methode(T val){
        Paire<T> unePaire = new Triplet<T>(val,val,val);
        return unePaire;
    }
}
class TestPaire2{
    public static void main(String[] args){
        Paire<String> unePaire =
            new Triplet<String>("un","deux","trois");
        unePaire.equals(
                        (Paire<String>)
                        new Triplet<String>("u","deu","troi"));
    }
}
```

1.5 Restrictions des constructions dynamiques

La machine virtuelle de Java n'a pas ´et´e modifi´ee. Une classe param´etr´ee est compil´ee en une classe non param´etr´ee (nous verrons plus tard comment). Donc les variables de types et les valeurs utilis´ees pour les substituer sont oubli´ees après la compilation. Certaines constructions qui utilisent les types à l'ex´ecution ne sont donc pas autoris´ees avec des types param´etr´es.

Première chose : la cr´eation d'objet. Un code du genre `new T()` n'est pas permis.

Exemple :

```
class ConstructionDynamique1<T>{
    T x = new T();
}
```

Deuxième chose : un cast vers une sous-classe.

Rappel à propos du cast : si $T \subseteq T'$, alors on peut faire une conversion de type de T' vers T, mais cela peut provoquer une erreur à l'ex´ecution.

Exemple :

```
class RappelCast{
    public static void main(String[] args){
        Compte c = new CompteRemunere();
        CompteSecurise c2 = (CompteSecurise) c;
    }
}
```

Une v´erification de type est faite à l'ex´ecution pour v´erifier que le type d'instance est un sous-type du type impos´e par le cast.

```
> javac RappelCast.java
> java RappelCast
Exception in thread "main" java.lang.ClassCastException: CompteRemunere
        at RappelCast.main(RappelCast.java:7)
```

Comme les types param´etr´es n'existent pas à l'ex´ecution, on ne peut pas faire de cast vers une sous-classe d'un tel type. En revanche, dans l'autre sens, de la sous-classe vers la super-classe, le contrôle se fait à la compilation. Il n'y a donc pas de problème.

```
class ConstructionDynamique2<T>{
    Triplet<T> convert(Object o){
        return (Triplet<T>) o;
    }
}
```

```
> javac ConstructionDynamique2.java
Note: ConstructionDynamique2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
> javac -Xlint:unchecked ConstructionDynamique2.java
ConstructionDynamique2.java:3: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: Triplet<T>
        return (Triplet<T>) o;
                            ^
1 warning
```

Troisième chose : un v´erification de type d'instance dynamique au moyen de l'op´erateur `instanceof`.

```
class ConstructionDynamique3{
    boolean estPaireInteger(Object o){
        return o instanceof Paire<Integer>;
    }
}
```

```
> javac ConstructionDynamique3.java
ConstructionDynamique3.java:3: illegal generic type for instanceof
        return o instanceof Paire<Integer>;
                            ^
1 error
```

1.6 Traduction des classes génériques

La g´en´ericit´e a pour seul effet de changer des types pour les variables et m´ethodes d'une classe. Ces types sont connus une fois la substitution des paramètres effectu´ee, c'est à dire dans le code qui utilise la classe param´etr´ee.

```
class Paire<T>{
    T x, y;
    Paire(T x1, T y1){
        x = x1;
        y = y1;
    }
    T first(){
        return x;
    }
    T second(){
        return y;
    }
    void setFirst(T x1){
        x = x1;
    }
```

```
    void setSecond(T y1){
        y = y1;
    }
    void invert(){
        T tmp = x;
        x = y;
        y = tmp;
    }
    boolean equals(Paire<T> o){
        return o.x.equals(x) && o.y.equals(y);
    }
}
class TestPaire3{
    public static void main(String[] args){
        Paire<String> p = new Paire<String>("toto","le´ehros");
        System.out.println(p.x);
        p.setSecond("artu");
        System.out.println(p.first());
    }
}
```

Se traduit en :

```
class Paire{
    Object x, y;
    Paire(Object x1, Object y1){
        x = x1;
        y = y1;
    }
    Object first(){
        return x;
    }
    Object second(){
        return y;
    }
    void setFirst(Object x1){
        x = x1;
    }
    void setSecond(Object y1){
        y = y1;
    }
    void invert(){
        Object tmp = x;
        x = y;
        y = tmp;
    }
```

```
    boolean equals(Paire o){
        return o.x.equals(x) && o.y.equals(y);
    }
}
class TestPaire3{
    public static void main(String[] args){
        Paire p = new Paire("toto","le ´ehros");
        System.out.println((String) p.x);
        p.setSecond("artu");
        System.out.println((String) p.first());
    }
}
```

Dans certains cas li´es à l'h´eritage et à la red´efinition, la traduction est un peu plus complexe et fait intervenir une *méthode-pont* qui caste les paramètres.

En voici un exemple :

```
class Compte{
    int solde;
}
interface Test<T>{
    public boolean propriete(T x);
}
class DansLeRouge implements Test<Compte>{
    public boolean propriete(Compte c){
        return c.solde < 0;
    }
}
```

```
interface Test{
    public boolean propriete(Object x);
}
class DansLeRouge implements Test{
    public boolean propriete(Compte c){
        return c.solde < 0;
    }
    public boolean propriete(Object x){
        return this.propriete((Compte) x);
    }
}
```