Introduction au typage de JAVA

F. Barthélemy

2 mai 2006

1 Un petit jeu

Soit le programme JAVA suivant :

```
class A{
  void m(A a){
    System.out.println("m de A");
  }
  void n(A a){
    System.out.println("n de A");
  }
}


class B extends A{
  void m(A a){
    System.out.println("m de B");
  }
  void n(B b){
    System.out.println("n de B");
  }
  public static void main(String[] argv){
    A a = new B();
    B b = new B();
    a.m(b);
    a.n(b);
  }
}
```

Question : deviner ce qu'affichera l'exécution de la méthode `main` de `B`.
Réponse :

```
m de B
n de A
```

En fait, presque toutes les combinaisons possibles auraient été logiquement concevables. Pour savoir à l'avance, quand on écrit ce programme, ce qui va se produire, il faut connaître la sémantique opérationnelle de JAVA qui elle-même ne peut se décrire qu'en fonction du typage. Connaître les

règles de typage employées est donc indispensable pour comprendre certains phénomènes étranges qui peuvent arriver.

On reprendra le petit jeu un peu plus loin avec d'autres exemples tout aussi surprenants.

2 Le typage

2.1 Erreurs de typage

Le typage est un mécanisme qui permet d'assurer une propriété de **correction partielle** des programmes.

Pour la programmation impérative, cette propriété est que les opérateurs et sous-programmes sont appliqués à des opérandes appartenant au bon domaine de valeurs.

Pour la programmation objet, la propriété essentielle recherchée par le typage est que lorsqu'on appelle une méthode sur un objet, cet objet contient bien cette méthode et les paramètres donnés appartiennent au bon domaine de valeur.

Le typage est généralement réalisé à la compilation pour détecter des erreurs dans les programmes. Dans le cas de Java, une partie du typage est réalisée à l'exécution pour des raisons de sécurité. Le typage à la compilation est effectué après l'analyse syntaxique qui vérifie la syntaxe du programme et avant la génération de code. Le typage s'effectue donc sur un programme correct syntaxiquement.

Le principe du typage consiste à distinguer des ensembles de valeurs (les types) et à faire des raisonnements approximatifs en considérants ces ensembles et non pas les valeurs particulières.

Par exemple, une erreur du genre `5+true` sera détectée. Le + est un opérateur qui attend des opérandes numériques et `true` n'est pas une valeur numérique.

Un exemple d'erreur que le typage ne détecte pas est `150/0`, parce que cette erreur vient de la valeur particulière 0 et non de son type (int) qui est un type correct pour la division. Le typage ne permet donc pas de traiter toutes les erreurs. En java, les erreurs non traitées par le typage sont gérées par le mécanisme d'exception. Les exceptions sont levées à l'exécution.

2.2 Types de variables

Pour typer une expression, il faut connaître le type des variables. Par exemple, l'expression `5+x` peut être bien typée si `x` est de type int, ou mal typé si `x` est de type `boolean`. Le type de la variable est déclaré au moment de la déclaration de la variable. Le typage doit s'en souvenir pour des usages de la variables qui sont dans la portée de la déclaration. Il y a donc un notion de mémoire qui collecte les déclarations de types (pour les variables et les méthodes) et cette mémoire évolue en fonction de la portée des déclarations en questions. Par exemple, une variable d'instance a pour portée la classe qui la contient, ainsi que ses sous-classes. Les paramètres d'une méthode ont pour portée la méthode elle-même.

La mémoire qui contient les déclarations de type s'appelle **l'environnement de typage**.

2.3 Types de méthodes

Les opérateurs et méthodes statiques ont également un type qui résume le type des paramètres et le type du résultat. Par exemple :

$+$: `int` $\times$ `int` $\rightarrow$ int

&& : boolean $\times$ boolean $\rightarrow$ boolean

`Integer.parseInt` : String $\rightarrow$ `int`

`String.charAt` : `int` → char

On note `void` l'absence de valeur (pas de paramètre ou pas de valeur retournée). On peut considérer `void` comme un type, un ensemble de valeurs qui ne contient aucune valeur, l'ensemble vide.

3 Difficultés spécifiques de l'envoi de message

L'envoi de message pose des problèmes spécifiques :

1. souvent, on ignore le *vrai* type de l'objet sur lequel est envoyé un message.
2. parfois, il y a plusieurs méthodes de même nom qui peuvent s'appliquer à des paramètres donnés.

```
class Compte{
    int solde = 0;
    void depot (int n){
        solde = solde + n;
    }//depot
    void retrait (int n){
        solde = solde - n;
    }//retrait
    void print (){
        System.out.println(solde);
    }//print
}//Compte
class CompteSecurise extends Compte{
    void retrait(int n){
        if (solde >= n)
            solde = solde - n;
        else{
            System.out.print("Provision insuffisante:");
            print();
        }//if
    }//retrait
}//CompteSecurise
class Ex{
    static void virement(Compte c1, Compte c2, int montant){
        c1.retrait(montant);
        c2.depot(montant);
    }
    public static void main(String[] args){
        Compte un, deux;
        un = new Compte();
        deux = new CompteSecurise();
        virement(un,un,50);
        virement(deux,un,50);
    }
}
```

Dans cet exemple, il y a deux appels différents à la méthode `virement`, avec deux paramètres ayant des types d'instance différents. Mais la méthode ne sera compilée qu'une fois. On ne peut donc pas prendre en compte le type d'instance effectif du paramètre à la compilation.

```java
class Ex2{
    static void virement(Compte c1, Compte c2, int montant){
        c1.retrait(montant);
        c2.depot(montant);
        System.out.println("methode un");
    }
    static void virement(CompteSecurise c1, Compte c2, int montant){
        c1.retrait(montant);
        c2.depot(montant);
        System.out.println("methode deux");
    }
    static void virement(Compte c1, CompteSecurise c2, int montant){
        c1.retrait(montant);
        c2.depot(montant);
        System.out.println("methode trois");
    }
    static void virement(CompteSecurise c1, CompteSecurise c2, int montant){
        c1.retrait(montant);
        c2.depot(montant);
        System.out.println("methode quatre");
   }
    public static void main(String[] args){
        CompteSecurise un;
        un = new CompteSecurise();
        virement(un,un,50);
    }
}
```

Ici toutes les méthodes `virement` sont applicables à l'appel `virement(un,un,50)`. Il faut une règle permettant de choisir la méthode qui sera exécutée.

4 Hypothèses simplificatrices

Dans un premier temps, nous allons simplifier la description en laissant de côté des choses importantes pour le typage et nous concentrer sur l'essentiel. Nous reviendrons plus tard sur ces hypothèses pour voir en quoi les éléments écartés interviennent dans le typage.

1. pas de types primitifs
2. pas de types tableaux
3. pas d'interfaces, rien que des classes
4. pas de généricité ni d'autres nouveautés de java 1.5.
5. pas d'exception
6. pas de prise en compte des règles de visibilité (private et autres)
7. pas de méthodes statiques

5 Hiérarchie des types

Avec les restrictions données, les types de Java sont de classes. Un objet a plusieurs types, c'est à dire qu'il est une valeur de plusieurs types. Cela se traduit par le fait qu'on peut le mettre dans des

variables déclarées avec différents types, qui ont pour caractéristique d'être sur un chemin de l'arbre d'héritage.

Le typage de Java utilise une notion d'inclusion des types, considérés comme des ensembles de valeurs.

Par exemple, tous les comptes sécurisés sont aussi des comptes et tous les comptes sont des objets. On a donc l'inclusion : $CompteSecurise \subseteq Compte \subseteq Object$.

De façon plus générale, si C_1 est une sous-classe de C_2, alors $C_1 \subseteq C_2$.

La relation d'inclusion d'ensemble est une relation d'ordre partiel, c'est-à-dire une relation réfléxive, transitive et anti-symétrique.

Tous les types d'un objets sont totalement ordonnés et il existe un type le plus petit pour un objet, c'est le type de la classe dont il est instance. On pourra donc appeler ce type indifféremment type d'instance ou type le plus petit de l'objet.

Par exemple, si on crée un objet par l'instanciation `new CompteSecurise()`, le type le plus petit de cet objet est `CompteSecurise`. Ses autres types sont `Compte` et `Object`.

6 Principes du typage en Java

Les grands principes du typage en Java sont :

- le type déclaré pour une variable ou un paramètre doit être l'un des types de la valeur stockée dans cette variable ou ce paramètre.
- à la compilation, on ne prend en compte que les types déclarés (et ceux qui s'en déduisent par calcul), pas les autres types.
- à la compilation, le typage choisit un type pour la méthode de chaque envoi de message. C'est un choix définitif, qui s'appliquera à l'exécution.
- à l'exécution, la méthode sera recherchée dynamiquement dans le type le plus petit de l'objet, en conservant le type choisi pour la méthode à la compilation.

7 Typage de l'affectation

Une affectation dans une variable `var = expression;` est correctement typée si le type de l'expression (donné par déclaration ou déduit des déclarations) est inclus dans le type déclaré pour `var`.

Exemples :

```
CompteSecurise c1 = new CompteSecurise();
Compte c2 = c1;
c2 = new CompteSecurise();
String s = "attention".toUpperCase();
```

8 Typage de l'envoi de message

Le typage d'un envoi de message à la compilation se fait en trois étapes.

8.1 Etape 1 : détermination de la classe où chercher la méthode

Trois cas :

– l'objet sur lequel est appélé la méthode est implicite. Exemple : `depot(100)`. L'objet implicite est `this` et son type est celui de la classe qui contient cet appel. On va chercher la méthode dans cette classe.
– l'appel utilise `super` (par exemple `super.retrait(100)`). Dans ce cas-là, on va chercher la méthode dans la super-classe de la classe courante.
– l'appel se fait sur un objet calculé par une expression. Exemples : `(new Compte()).depot(50)` ou `c.depot(50)`. Dans ce cas, on regarde le type déduit des déclarations pour l'expression et on cherche dans ce type-là (cette classe) la méthode appelée.

Exemple :

```
Compte c2 = new CompteSecurise();
c2.retrait(50);
```

On va chercher la méthode `retrait` dans la classe `Compte`, la classe utilisée dans la déclaration de `c2` et non pas dans la classe `CompteSecurise` qui est le type d'instance de `c2`.

8.2 Recherche des méthodes applicables

Dans la classe déterminée à l'étape précédente, on cherche toutes les méthodes qui peuvent s'appliquer :
– qui ont le bon nom
– qui ont le bon nombre de paramètres (le même que dans l'appel)
– dont chaque paramètre est déclarée avec un type plus grand que l'argument correspondant donné dans l'appel.

Les méthodes peuvent être déclarées dans le type ou héritées des super-classes. On ne fait pas de distinction.

On collecte les types de toutes ces méthodes.

8.3 Etape 3 : détermination du type le plus petit

L'étape précédente a isolé un certain nombre de méthodes et collecté leurs types.

Il faut déterminer le type le plus petit pour les paramètres, dans cet ensemble. Un tel type n'existe pas forcément le cas. S'il n'y a pas exactement un type le plus petit dans cet ensemble (soit qu'il n'y en ait pas, soit qu'il y a en ait plusieurs) l'envoi de message est mal typé et le programme ne se compile pas.

Le type le plus petit est choisi.

8.4 Exécution de l'envoi de message

A l'exécution, la méthode est recherchée dans **le type le plus petit**, le **type d'instance de l'objet**. Dans cette classe, une méthode ayant le bon nom et **le type déterminé à l'étape 3** de la compilation est recherchée.

La méthode choisie peut être une autre méthode que celle qui a servi à la compilation, si il y a eu redéfinition de cette méthode dans une sous-classe en conservant le même type. C'est la liaison tardive. S'il n'y a pas eu de redéfinition, c'est la méthode utilisée à la compilation qui est utilisée. Elle est présente dans la classe par héritage, dans le cas où le type le plus petit n'est pas égal au type déclaré ou déduit de l'objet.

8.5 Exemples

Reprenons le petit jeu de la section 1. Il y a deux envois de messages.

```
A a = new B();
B b = new B();
a.m(b);
a.n(b);
```

Pour le premier envoi de message `a.m(b)` :

- étape 1 : la classe où chercher la méthode est celle déclarée pour `a`, c'est à dire `A`.
- étape 2 : il n'y a qu'une méthode de nom `m` dans `A`. Elle a pour type `A` → `void`.
- étape 3 : le type fixé est `A` → `void`.
- à l'exécution, la recherche de la méthode m de type `A` → `void` se fait dans le type le plus petit ou type d'instance de `a`, c'est à dire `B`. Il y a là la méthode *m de B* qui a le bon type. C'est elle qui est exécutée.

Pour le deuxième envoi de message, `a.n(b)` :

- étape 1 : la classe où chercher la méthode est celle déclarée pour `a`, c'est à dire `A`.
- étape 2 : il n'y a qu'une méthode de nom `n` dans `A`. Elle a pour type `A` → `void`.
- étape 3 : le type fixé est `A` → `void`.
- à l'exécution, la recherche de la méthode `n` de type `A` → `void` se fait dans le type le plus petit ou type d'instance de `a`, c'est à dire `B`. La méthode *n de B* ne convient pas, car elle n'a pas le bon type. La méthode *n de A* convient. Elle a le bon type et elle est présente dans la classe `B` par héritage. C'est elle qui est choisie.

Conclusion : quand on veut redéfinir une méthode, la remplacer par une autre, il faut impérativement conserver le même type.

8.6 Exemple de choix

Dans cet exemple, on suppose qu'il existe une classe `Forme` avec des sous-classes `Cercle, Carre, Triangle`. Dans un autre classe, on a l'API suivante :

```
class Calcul{
   int distance(Forme f1, Forme f2); // méthode 1
   int distance(Forme f1, Carre f2); // méthode 2
   int distance(Carre f1, Forme f2); // méthode 3
   int distance(Triangle f1, Carre f2); // méthode 4
   int distance(Forme f1, Cercle f2); // méthode 5
}
```

On étudie l'envoi de message :

```
Cercle c;
Triangle t;
Carre r;
...
(new Calcul()).distance(t,r);
```

- étape 1 : la classe où chercher la méthode est le type de `(new Calcul())`, c'est-à-dire la classe `Calcul`.

- étape 2 : on cherche dans `Calcul` les méthodes `distance` à deux paramètres dont le premier est plus grand que `Triangle` et le deuxième plus grand que `Carre`. Il y en a trois : les méthodes 1, 2 et 4.
- étape 3 : parmi les types des méthodes 1, 2 et 4, on cherche le plus petit, en ce qui concerne les paramètres (le type retourné n'est pas pris en compte). Ces types sont :
  - méthode 1 : `Forme` $\times$ `Forme` $\rightarrow$ `int`
  - méthode 2 : `Forme` $\times$ `Carre` $\rightarrow$ `int`
  - méthode 4 : `Triangle` $\times$ `Carre` $\rightarrow$ `int`

  C'est le type de la méthode 4 le plus petit, le plus précis. C'est lui qui est choisi.

8.7 Exemple d'ambiguïté

En utilisant les mêmes classes et les mêmes variables, l'envoi de message :

```
(new Calcul()).distance(r,r)
```

pose un problème.

- étape 1 : la classe où chercher la méthode est le type de `(new Calcul())`, c'est-à-dire la classe `Calcul`.
- étape 2 : on cherche dans `Calcul` les méthodes `distance` à deux paramètres dont le premier est plus grand que `Carre` et le deuxième plus grand que `Carre`. Il y en a trois : les méthodes 1, 2 et 3.
- étape 3 : parmi les types des méthodes 1, 2 et 3, on cherche le plus petit, en ce qui concerne les paramètres (le type retourné n'est pas pris en compte). Ces types sont :
  - méthode 1 : `Forme` $\times$ `Forme` $\rightarrow$ `int`
  - méthode 2 : `Forme` $\times$ `Carre` $\rightarrow$ `int`
  - méthode 3 : `Carre` $\times$ `Forme` $\rightarrow$ `int`

  Les types des méthodes 2 et 3 ne sont pas plus précis l'un que l'autre. L'un est plus précis sur un argument, l'autre sur un autre. Il n'y a pas un type plus précis que l'autre, le choix est impossible. Il y a une ambiguïté qui produit une erreur à la compilation.

Le message du compilateur est le suivant :

```
Calcul.java:15: reference to distance is ambiguous, both
method distance(Forme,Carre) in Calcul and
method distance(Carre,Forme) in Calcul match
        (new Calcul()).distance(r,r);
        ^

1 error
```

9 Typage du Cast

La conversion de type explicite est typée en deux temps : d'abord à la compilation, puis, de façon plus précise, à l'exécution.

A la compilation : on peut demander une conversion d'un type T dans un Type U si T est inclus dans U ou si U est inclus dans T.

A l'exécution, on peut réaliser la conversion de type si le nouveau type fait partie des types de l'objet. Sinon, une exception est levée.

Voyons un exemple qui reprend l'exemple des formes déjà utilisé.

```
Forme f = new Cercle();
Cercle c = (Cercle) f;
Carre r = (Carre) f;
```

Les deux conversions de type sont compilées correctement parce que $Cercle \subseteq Forme$ et $Carre \subseteq Forme$.

A l'exécution, l'objet contenu dans `f` a les types `Cercle, Forme` et `Object`. Il n'a pas le type `Carre`, donc la deuxième conversion provoque une erreur.

```
Exception in thread "main" java.lang.ClassCastException: Cercle
        at Calcul.test(Calcul.java:18)
        at Calcul.main(Calcul.java:21)
```