Programme de simulation d'algorithmes de résolution de la surcharge pour un mini-Java

TEP

Etudiants : P. Dul, E. Masliah

Table des matières

1 Introduction **2**

2 Principe et application **2**
2.1 Java 1.2 4
2.2 Java 1.5 4
2.3 Java Exam 4

3 Utilisation de SimulAlgoResolutionJVM **4**

4 Conclusion **5**

1 Introduction

Au cours de l'évolution de Java, l'algorithme utilisé lors de la résolution de la surcharge à été modifié. Il est intéressant de bien comprendre le principe et le fonctionnement de ces algorithmes. C'est ce que nous allons détailler dans la suite de ce document.

2 Principe et application

La résolution de la surcharge en Java consiste à déterminer la signature de la méthode qui va être appelée pour un appel de méthode donnée. Cette résolution s'effectue de manière statique, en suivant toujours le même principe, détaillé ci-dessous.

A l'exécution, il ne restera plus qu'à résoudre la redéfinition de la méthode choisie statiquement entre la classe vue statiquement et la classe réelle.

Pour plus de clarté, prenons un exemple.

```java
public class A {
    void m (B x) { System.out.println("A m B"); }
    void m (A x, B y) { System.out.println("A m A B"); }
    void m (String x) { System.out.println("A m String"); }
}
```

```java
public class B extends A {
    void m (A x) { System.out.println("B m A"); }
    void m (B x, A y) { System.out.println("B m B A"); }
}
```

```java
public class C extends B {
   void m (A x) { System.out.println("C m A"); }
   void m (A x, B y) { System.out.println("C m A B"); }
   void m (B x) { System.out.println("C m B"); }
   void m (B x, A y) { System.out.println("C m B A"); }
}
```

```java
public class Exemple1 {
    public static void main(String[] args) {
        B bc = new C();
        bc.m(new A());
    }
}
```

Dans Exemple1, ligne 4, nous effectuons un appel de la méthode m avec un paramètre de type A sur B en statique et sur C en dynamique. Nous allons déterminer quelle méthode va être appelée.

Pour la résolution de surcharge, il faut tout d'abord déterminer l'opération qui est la plus éligible. Pour cela, parmi toutes les méthodes définies pour la classe B :

```
1 public class B extends A {
2     void m (A x) { System.out.println("B m A"); }
3     void m (B x, A y) { System.out.println("B m B A"); }
4     void m (B x) { System.out.println("A m B"); } // From A
5     void m (A x, B y) { System.out.println("A m A B"); } // From A
6     void m (String x) { System.out.println("A m String"); } // From A
7 }
```

Il faut chercher les méthodes qui possèdent le bon nom et la bonne arité :

```
1 public class B extends A {
2     void m (A x) { System.out.println("B m A"); }
3     void m (B x) { System.out.println("A m B"); } // From A
4     void m (String x) { System.out.println("A m String"); } // From A
5 }
```

Parmi ces méthodes, il faut sélectionner les méthodes qui sont compatibles avec la méthode appelée (chacun des paramètres doit être compatible avec le paramètre de la méthode appelée) :

```
1 public class B extends A {
2     void m (A x) { System.out.println("B m A"); }
3     void m (B x) { System.out.println("A m B"); } // From A
4 }
```

On se retrouve avec une liste de méthodes pouvant être appelées à la place de la méthode. Pour chacune de ces méthodes, il faut calculer une valeur qui permettra de déterminer si c'est elle la plus éligible ou non. Cette valeur dépend de l'algorithme choisi.

Ensuite, parmi les méthodes qui ont la plus petite valeur, il faut supprimer les méthodes qui en redéfinissent d'autres. Si il reste plus d'une méthode après ceci, il y a cas d'ambiguïté, sinon la méthode restante est la méthode sélectionnée.

2.1 Java 1.2

L'algorithme de Java 1.2 dit que la valeur d'une méthode est le produit cartésien de sa classe de définition et des types des paramètres de la méthode. Ainsi, dans notre exemple, les deux méthodes retenues ont la même valeur. De plus, elles ne se redéfinissent pas. Nous sommes donc dans un cas d'ambiguïté et dans l'impossibilité de choisir la fonction à appeler. La classe Exemple1 ne compile pas.

2.2 Java 1.5

L'algorithme de Java 1.5 dit que la valeur d'une méthode est le produit cartésien des types des paramètres de la méthode. Ainsi, dans notre exemple, la méthode avec comme paramètre A a une valeur inférieur à la méthode avec le paramètre B. La méthode sélectionnée est donc la m(A) de la classe B.

A l'exécution, comme cette méthode a été redéfinie dans la classe C, cela sera la méthode m(A) de la classe C qui sera appelée.

2.3 Java Exam

L'algorithme de Java Exam diffère des algorithmes de résolution de Java car il n'a pas la même définition d'une méthode compatible. En effet, pour lui, les méthodes compatibles doivent avoir exactement le même type de paramètres (et non pas être un sous type). De ce fait, nous avons donc :

```
1 public class B extends A {
2     void m (A x) { System.out.println("B m A"); }
3 }
```

La méthode sélectionnée est donc m(A) de la classe B.

A l'exécution, comme cette méthode a été redéfinie dans la classe C, cela sera la méthode m(A) de la classe C qui sera appelée.

3 Utilisation de SimulAlgoResolutionJVM

Ce programme requiert Java 1.5 pour fonctionner. Pour le lancer, taper

```
java -jar SimulAlgoResolutionJVM.jar
```

Le programme SimulAlgoResolutionJVM prend comme premier paramètre un fichier xml respectant la BNF (Backus-Naur Form) suivante :

```
start ::= <myJVM>
<myJVM> ::= "<myJVM>" <classes> <runs> "</myJVM>"
<classes> ::= "<classes>" <class> | <interface> "</classes>"
<class> ::= "<class " <name> ">" [<extend>] [<implements>] <method>*
"</class>"
<interface> ::= "<interface " <name> ">" [<extends>] "</interface>"
<name> ::= "name=" <string>
<extends> ::= "<extends>" <extend>* "</extends>"
<extend> ::= "<extend " <name> ">" "</extend>"
<implements> ::= "<implements>" <implement>* "</implements>"
<implement> ::= "<implement " <name> ">" "</implement>"
<method> ::= "<method " <name> ">" [<parameters>] [<return>] "</method>"
<parameters> ::= "<parameters>" <parameter>* "</parameters>"
<parameter> ::= "<parameter " <type>  "></parameter>"
<return> ::= "<return " <type> "></return>"
<type> ::= "type=" <string>
<runs> ::= "<runs>" <run> "</runs>"
<run> ::+ "<run>" <object> "</run>"
<object> ::+  "<object " <typeReel> " " <typeAffiche> ">" <methodRun>
"</object>"
<typeReel> ::= "typeReel=" <string>
<typeAffiche> ::= "typeAffiche=" <string>
<methodRun> :: "<method " <name> ">" <parametersRun> "</method>"
<parametersRun> ::= "<parameters>" <parameterRun> "</parameters>"
<parameterRun> ::= "<parameter " <typeReel> " " <typeAffiche>
"></parameter>"
<string> ::= <alpha>+
<alpha> ::= 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9'
```

Le deuxième paramètre est optionnel. Il peut prendre la valeur "1.2" pour lancer l'algorithme Java 1.2, "1.5" pour lancer l'algorithme Java 1.5, "exam" pour lancer l'algorithme Java Exam ou encore "all" (valeur pas défaut) pour lancer les 3 algorithme.

4 Conclusion

La résolution de surcharge est un élément essentiel dans un langage objet. En effet, cela permet entre autre de pouvoir écrire plusieurs méthodes avec le même nom mais avec

des paramètres différents. Sans surcharge, chaque méthode devrait avoir un nom différent. Cela serait plus contraignant, mais éviterait les ambiguïtés.