


Concept. et prog. objet


\1 TD n°2 : polymorphisme

Détails

Écrit par stéphane Domas

Catégorie : M3105 - Concept. et prog, objet avancée (/index.php/menu-cours-s3/menu-mmi3-test)

 Publication : 21 octobre 2014

 Affichages : 28

1°/ Principe

- Le polymorphisme est la capacité d'une variable objet de prendre plusieurs formes, ou, traduit en langage informatique, plusieurs types.
- Dans un langage impératif fortement typé comme le C, c'est normalement impossible.
- En POO, c'est possible car une variable peut avoir un type à la compilation et un autre type (voire plusieurs) lors de l'exécution, à une condition :

Une variable **déclarée** de type A peut, lors de l'exécution, référencer une instance de A, OU de toute autre sous-classe de A.

- En effet, le principe d'instanciation fait que les objets sont créés lors de l'exécution et pas directement lors de la phase de compilation (NB : c'est malgré tout possible en C++ car on est pas obligé d'utiliser new).
- De part le polymorphisme, il est donc possible d'appeler le constructeur de la classe B et de mettre le résultat dans une variable déclarée de type A, pourvu que B soit une sous-classe de A.
- Fort heureusement, sauf exception, on est pas obligé d'attendre l'exécution pour vérifier si la règle est respectée lors d'une affectation. Le compilateur peut la plupart du temps vérifier cette règle en se basant sur les types des données à gauche et droite du égal. Par exemple :

```
1  class A { ... }
2  class B extends A { ... }
3  class C extends A { ... }
4
5  A a = null;
6  B b = null;
7  C c = null;
8  a = new A(); // OK : variable type A <- instance de A
9  a = new B(); // OK : variable type A <- instance de B sous-classe de A
10 a = new C(); // OK : variable type A <- instance de C sous-classe de A
11
12 b = new B(); // OK
13 b = new A(); // ERREUR COMPIL.
14 b = new C(); // ERREUR COMPIL.
15
16 c = new C(); // OK
17 c = new A(); // ERREUR COMPIL.
18 c = new B(); // ERREUR COMPIL.
```

2°/ Implications

- Le polymorphisme n'a généralement aucun intérêt si on ne l'utilise pas conjointement avec le principe de redéfinition de membres.
- En effet, une règle inviolable POUR LE COMPILATEUR est qu'un objet ne peut manipuler que les membres de sa classe ou bien ceux dont il hérite. Par exemple :

```

1  class A {
2      public void print();
3      ...
4  }
5  class B extends A {
6      public void toto();
7      ...
8  }
9
10 A a = new B();
11 a.print(); // OK : variable type A appelle méthode de A
12 a.toto(); // ERREUR COMPIL. : variable type A appelle méthode de B
13
14 B b = new B();
15 b.print(); // OK : variable type B appelle méthode héritée de A
16 b.toto(); // OK : variable type B appelle méthode de B

```

- Que se passe-t-il si `print()` est redéfinie dans B et que l'on manipule une variable polymorphe ? Dans ce cas, il faut devenir un peu schizophrène et voir les choses de deux points de vue : celui de compilateur et celui de l'exécution.

```

1  class A {
2      public void print();
3      ...
4  }
5  class B extends A {
6      public void print();
7      public void toto();
8      ...
9  }
10
11 A a1 = new A();
12 a1.print(); // pour COMPIL. : OK car variable type A appelle méthode de A
13              // pour EXEC. : appel de la méthode print() de A
14
15 B b = new B();
16 b.print(); // pour COMPIL. : OK car variable type B appelle méthode de B
17              // pour EXEC. : appel de la méthode print() de B
18
19 A a2 = new B(); // utilisation du polymorphisme
20 a2.print(); // pour COMPIL. : OK car variable type A appelle méthode de A
21              // pour EXEC. : appel de la méthode print() de B

```

- Dans l'exemple ci-dessus, on constate qu'il se passe une chose différente du point de vue compilation et exécution.
- En fait, le compilateur se base toujours sur le type de déclaration alors que pendant l'exécution, la JVM se base toujours le type d'instance d'un objet pour décider quelle méthode appeler.
- Ce choix est automatique et ne peut pas être contourné.

3°/ Complications

3.1°/ Polymorphisme et héritage

- Comment s'applique le polymorphisme lorsqu'une classe hérite d'une méthode redéfinie par sa super-classe. Par exemple :

```

1  class X {
2      ...
3      public void print();
4  }
5  class Y extends X {
6      ...
7      public void print();
8  }
9  class Z extends Y {
10     ... // Z ne redéfinit pas print() et en hérite de Y
11 }
12
13 X x = new Z();
14 x.print(); // instance de Z, qui appelle de la méthode héritée de Y

```

3.2°/ Erreurs lors de l'exécution

- Il a été dit plus haut que le compilateur sait détecter si la règle du polymorphisme s'applique correctement ou non. Dans certains cas, ce n'est pas vrai, notamment lorsque l'on crée des structures de données contenant des objets polymorphes.
- Par exemple, soit une classe A dont hérite B et C. On peut donc parfaitement créer une collection d'objet A et stocker dedans des instances de A, B et C

```

1  List<A> list = new ArrayList<A>();
2  list.add(new A());
3  list.add(new B());
4  list.add(new C());
5  ...
6  // appel d'une méthode qui mélange les objets stockés dans list
7  ...
8  for(int i=0;i<list.size();i++) {
9      A a = list.get(i); // COMPIL. OK : variable type A <- type A renvoyé par get
10                     // EXEC. OK : si l'objet instance de B/C -> polymorphisme
11 }
12 for(int i=0;i<list.size();i++) {
13     B b = list.get(i); // ERREUR COMPIL.: variable type B <- type A renvoyé par get
14 }
15 for(int i=0;i<list.size();i++) {
16     B b = (B)(list.get(i)); // COMPIL. OK : variable type B <- type A renvoyé par get, transtypage
17                     // ERREUR EXEC.: pour certains i, transtypage en B d'une instance de A
18 }

```

- Si le compilateur était suffisamment puissant, il pourrait simuler l'exécution et voir que certaines cases contiennent des instances de A ou C. Il est donc impossible de les mettre dans une variable de type B. Mais ce n'est pas son travail : il se contente de vérifier les types à gauche et droite du égal et s'ils ne respectent pas la règle, de signaler une erreur. C'est donc lors de l'exécution que la ligne 16 vaut provoquer une erreur.

3.3°/ Polymorphisme à la rescousse

- Dans l'exemple ci-dessus, on a clairement un problème dans le premier `for` si on veut appeler une méthode qui n'existe **que dans B** : `a` est de classe A donc le compilateur n'accepte que les appels aux méthodes de A.
- Si on voulait appeler `toto()`, il faudrait écrire un truc ignoble, du genre :

```

1  for(int i=0;i<list.size();i++) {
2      A a = list.get(i);
3      if (a instanceof B) {
4          B b = (B)a;
5          b.toto();
6      }
7  }

```

- Ce code est tout sauf conforme aux préceptes de la POO. La solution est donnée par le principe de redéfinition et d'appel automatique à la méthode de l'instance.
- En fait, on définit toto() dans A, mais avec un code vide. Ensuite, on la redéfinit dans B :

```

1  class A {
2      protected void toto() { // pas de code }
3      ...
4  }
5  class B extends A {
6      private void toto() { ... }
7      ...
8  }
9  class C extends A {
10     ... // pas de redéfinition de toto() -> héritage de A
11 }
12
13 List<A> list = new ArrayList<A>();
14 list.add(new A());
15 list.add(new B());
16 list.add(new C());
17 ...
18 // appel d'une méthode qui mélange les objets stockés dans list
19 ...
20 for(int i=0;i<list.size();i++) {
21     A a = list.get(i);
22     a.toto(); // COMPIL. OK : toto() méthode de A
23             // EXEC OK. : si a instancé de A/C -> appel méthode toto() de A qui ne fait rien
24             //             si a instancé de B -> appel méthode toto() de B
25     ...
26 }

```

Remarque : on verra dans le TD n°3 que cette solution n'est pas totalement satisfaisante et qu'il existe un concept POO qui permet d'écrire un code encore plus propre.

3.4°/ Cas bizarres

- En POO, il est parfaitement possible pour un constructeur de faire appel à une méthode redéfinie. Par exemple :

```
1  class A {
2      protected int val;
3      public A(int val) {
4          setVal(val);
5      }
6      public void setVal(int val) {
7          this.val = 2*val;
8      }
9  }
10
11  class B extends A {
12      public B(int val) {
13          super(val);
14      }
15      public void setVal(int val) {
16          this.val = 4*val;
17      }
18  }
19
20  A a1 = new A(10); // a1.val contient 20
21  A a2 = new B(10); // a2.val contient 40
```

- Le résultat peut paraître bizarre au regard du processus de construction de a2.
- En effet, l'appel à new B() commence par faire appel au constructeur de A. Quand on est dans celui-ci, on est donc en train de construire un objet de la classe A. On pourrait donc s'attendre à ce que ce soit la méthode setVal() de A qui soit appelée. Cela serait le cas en C++, mais en Java, la JVM appelle la méthode de l'objet final à construire, donc celle de B.