Interfaces

Fabrice Rossi

TELECOM ParisTech

Janvier 2009

Avantages de la P.O.O.

P.O.O. = Programmation Orientée Objet

- focalisée sur les données (par opposition aux « algorithmes ») :
  - un objet : représentation (attributs) + comportement (méthodes)
  - simplification des appels (paramètre implicite `this`)
- **abstraction** :
  - permet d'écrire un programme sans connaître exactement l'objet auquel il sera appliqué
  - mécanisme de contrat : le programme propose un contrat aux objets et s'appliquent à tous ceux qui le réalisent
  - en Java : contrat = interface
- et toutes sortes d'autres choses...

Exemple sans abstraction

StatsRedondant

```
1  public class StatsRedondant {
2      public static double min(double[] x) {
3          double result = x[0];
4          for(int i = 1; i < x.length; i++) {
5              if(result < x[i]) {
6                  result = x[i];
7              }
8          }
9          return result;
10     }
11
12     public static double max(double[] x) {
13         double result = x[0];
14         for(int i = 1; i < x.length; i++) {
15             if(result > x[i]) {
16                 result = x[i];
17             }
18         }
19         return result;
20     }
21 }
```

Analyse de l'exemple

- le code est **très redondant**
- une seule différence entre les deux méthodes : < contre >
- problèmes :
  - perte de temps : écrire plusieurs fois la même chose
  - difficulté de compréhension : il faut étudier attentivement le code pour être certain d'avoir exactement la même chose
  - bugs éventuels à corriger plusieurs fois
- dans l'exemple :
  - bug sur un tableau vide
  - on peut multiplier les méthodes redondantes (somme, moyenne, etc.)

Solution

- processus d'abstraction :
  - regrouper les éléments communs aux deux méthodes
  - s'appuyer sur un objet pour représenter les parties uniques
- dans l'exemple :
  - algorithme général :
    1. initialiser le résultat
    2. parcourir toutes les cases du tableau en mettant à jour le résultat
    3. renvoyer le résultat final
  - parties uniques :
    - initialisation (cf tableau vide)
    - mise à jour du résultat
    - résultat final (cf la moyenne par ex.)

Exemple avec abstraction

Stats

```
1  public class Stats {
2      public static double acc(double[] x, Calcul c) {
3          c.init();
4          for(int i = 0; i < x.length; i++) {
5              c.update(x[i]);
6          }
7          return c.result();
8      }
9  }
```

- une seule méthode générique
- ce qui change est dans un « objet » `Calcul` :
  - le code des méthodes d'un objet est fixé par sa classe
  - il faut donc **plusieurs** classes `Calcul`

Interface

- `Calcul` n'est pas une classe, c'est un **contrat** :
  - on doit avoir une méthode `void init()` (pas de paramètre, pas de résultat)
  - et une méthode `void update(double)` (pas de résultat)
  - et enfin une méthode `double result()` (pas de paramètre)
- en Java, c'est une `interface` :

Calcul

```
1  public interface Calcul {
2      public void init();
3
4      public void update(double t);
5
6      public double result();
7  }
```

- patron pour une classe

Exemple

CalculMax

```
1  public class CalculMax implements Calcul {
2      private double result;
3
4      public void init() {
5          result = Double.NEGATIVE_INFINITY;
6      }
7
8      public void update(double t) {
9          if(t > result) {
10             result = t;
11         }
12     }
13
14     public double result() {
15         return result;
16     }
17 }
```

`implements` indique que la classe `CalculMax` s'engage à remplir le contrat `Calcul`

Utilisation

Demo

```
1  public class Demo {
2      public static void main(String[] args) {
3          double[] tab = {0.1, 0.5, 5};
4          CalculMax algo = new CalculMax();
5          System.out.println(Stats.acc(tab,algo));
6      }
7  }
```

- comme `CalculMax implements Calcul`, on peut utiliser un objet `CalculMax` quand on attend un `Calcul`
- déroulement de `Stats.acc(tab,algo)` :
  - appel de `algo.init()` : `algo.result` $=-\infty$
  - appel de `algo.update(tab[i])` pour tout `i` : mise à jour de `algo.result`
  - appel de `algo.result()` : renvoie la valeur de `algo.result`

Exercice

Calcul de la moyenne

Calcul de la moyenne

CalculMoyenne

```
1  public class CalculMoyenne implements Calcul {
2      private double result;
3
4      private int nb;
5
6      public void init() {
7          result = 0;
8          nb = 0;
9      }
10
11     public void update(double t) {
12         result += t;
13         nb++;
14     }
15
16     public double result() {
17         return result/nb;
18     }
19 }
```

Cas général

Définition d'une interface

- une `interface` contient :
  - des déclarations de méthodes (pas de code)
  - des constantes
- une classe peut implémenter une ou plusieurs interfaces :
  - sous la forme `class A implements B,C,D`
  - la classe doit contenir le code de toutes les méthodes des interfaces
- interprétation :
  - `interface` : contrat sous forme de méthodes disponibles
  - `implements` : remplir le contrat, c.-à-d. implémenter les méthodes

Cas général

Utilisation d'une interface

- une interface peut remplacer une classe partout
- sauf pour une création d'objet :
  - par exemple `new Calcul()` est impossible
  - une interface est un contrat, pas une classe : pas de représentation, pas de code !
- si une variable `a` est de type `A` une interface, on peut appeler les méthodes et les constantes déclarées dans `A`
- dans une variable `a` de type `A`, on peut mettre une référence vers un objet d'une classe `B` qui implémente `A` :
  - c'est le mécanisme fondamental
  - type d'une variable $\neq$ type de l'objet pointé par la variable

Exemple

Dummy

```
1  public interface Dummy {
2      public int val();
3  }
```

One

```
1  public class One implements Dummy {
2      public int val() {
3          return 1;
4      }
5  }
```

Val

```
1  public class Val implements Dummy {
2      private int content;
3
4      public Val(int content) {
5          this.content = content;
6      }
7
8      public int val() {
9          return content;
10     }
11 }
```

Exemple

DemoDummy

```
1  public class DemoDummy {
2      public static void main(String[] args) {
3          Dummy variable;
4          variable = new One();
5          System.out.println(variable.val());
6          variable = new Val(5);
7          System.out.println(variable.val());
8          variable = new Val(7);
9          System.out.println(variable.val());
10     }
11 }
```

Exemple

DemoDummy

```
1  public class DemoDummy {
2      public static void main(String[] args) {
3          Dummy variable;
4          variable = new One();
5          System.out.println(variable.val());
6          variable = new Val(5);
7          System.out.println(variable.val());
8          variable = new Val(7);
9          System.out.println(variable.val());
10     }
11 }
```

- Affichage :

  1
  5
  7

Mécanisme sous-jacent

- typage $\neq$ édition de liens :
- à la compilation, on vérifie les types :
  - autorisation d'appels de méthodes, de constructeurs, etc.
  - autorisation d'accès (cf l'**encapsulation** dans la suite du cours)
- à l'exécution, on doit avoir associé un appel de méthode au code à exécuter :
  - soit pendant l'édition de liens : **statiquement**
  - soit pendant l'exécution elle-même : **dynamiquement** (tardivement)

Édition de liens dynamique

- Si `a` est de type `A` une interface, l'édition de liens est généralement dynamique
- algorithme :
  1. accéder à l'objet référencé par `a`
  2. déterminer le type réel de l'objet : par exemple la classe `B` qui implémente l'interface `A`
  3. exécuter l'implémentation contenue dans `B`
- en pratique :
  - chaque objet contient un pointeur vers un objet qui représente sa classe
  - l'objet classe contient des pointeurs vers les méthodes
  - on peut aussi avoir directement un pointeur vers la liste des méthodes dans chaque objet