Année 2008-2009

esil

Les méthodes Java et encapsulation

Nicolas Baudru
mél : nicolas.baudru@esil.univmed.fr
page web : nicolas.baudru.perso.esil.univmed.fr

Nous avons vu qu'une instance d'une classe (un objet d'un type donné) possède :

- des variables d'instances qui correspondent à l'état de l'objet.
- des méthodes qui correspondent au comportement de l'objet.

En pratique le comportement modifie l'état et l'état modifie le comportement. Autrement dit, les méthodes doivent affecter les variables d'instances et les variables d'instance doivent affecter les résultats des méthodes.

Nous allons donc voir en détail comment procéder en Java.

Une classe est un patron d'objets. Elle décrit comment la JVM doit fabriquer un objet de ce type.

On sait que deux objets A et B d'un même type, i.e. construit à partir d'une même classe, ont les mêmes variables d'instance.

Nous savons aussi que la valeur des variables d'instance de A et B peut différer.

Le comportement de deux objets d'une même classe est-t-il le même ?

Exemple

```
class Chien {
  int taille;
  String nom;

  void aboyer() {
    if (taille > 60) {
      System.out.println("Grrr␣Grrr");
    } else if (taille > 15) {
      System.out.println("Ouaf␣Ouaf");
    } else {
      System.out.println("Kai␣Kai");
    }
  }
}
```

Exemple

esil

```
class TestChien {

  public static void main(String[] args){
    Chien mirza = new Chien();
    mirza.taille = 30;
    Chien fido= new Chien();
    fido.taille = 10;

    mirza.aboyer();
    fido.aboyer();
  }
}
```

On compile puis on lance l'exécution de notre programme :

```
% javac Chien.java
% javac TestChien.java
% java TestChien
Ouaf Ouaf
Kai Kai
```

Paramètres des méthodes

Il est possible de transmettre des valeurs aux méthodes :

```
class Chien {
  int taille;
  String nom;

  void aboyer(int nbFois) {
    while (nbFois > 0) {
      System.out.println("Ouaf");
      nbFois = nbFois - 1;
    }
  }
}
```

```
class TestChien {

  public static void main(String[] args){
    Chien fido= new Chien();
    fido.aboyer(3); // demande à fido d aboyer 3 fois
  }
}
```

Paramètres des méthodes

Un code appelant transmet des arguments à une méthode. Ces arguments sont en fait des valeurs (comme un entier) ou des références.

```
fido.aboyer(3); // demande à fido d aboyer 3 fois
```

Les arguments transmis à la méthode sont « copiés » dans les paramètres de cette méthode. Un paramètre n'est rien d'autre qu'une variable (avec un nom et un type) qui prend comme valeur celle passée en argument lors de l'appel de la méthode :

```
void aboyer(int nbFois) {
  // après l'appel de aboyer(3) nbFois vaudra 3
  while (nbFois > 0) {
    System.out.println("Ouaf");
    nbFois = nbFois - 1;
  }
```

Paramètres multiples

Un appel à une méthode doit avoir autant d'arguments que le nombre de paramètre de cette méthode. Chaque argument doit avoir le même type que celui du paramètre correspondant et être transmis dans le bon ordre.

Exemple :

```
class Operation{
  void soustraire (int x, int y) {
    int z = x - y;
    System.out.println("Le résultat est " + z);
  }
}

class TestOperation{
  public static void main(String[] args){
    int n = 7;
    Operation t = new Operation();
    t.soustraire(n,3); //  x<--n et y<--3
    System.out.println("La valeur de n  est " + n);
  }
}
```

Passage par valeur

Exemple :

```
class Operation{
  void soustraire_2 (int x) {
     x = x - 2;
    System.out.println("Le résultat est " + x);
  }
}

class TestOperation{
  public static void main(String[] args){
    int n = 7;
    Operation t = new Operation();
    t.soustraire_2(n); //  x<--n
    System.out.println("La valeur de n  est " + n);
  }
}
```

Quelle est la valeur de n à la fin de l'exécution de ce programme ?

Passage par valeur (suite)

1 On déclare une méthode avec deux paramètres de type int :

```
... void soustraire_2( int x ) { ...
```

2 On déclare une variable de type int et on lui affecte la valeur 7 :

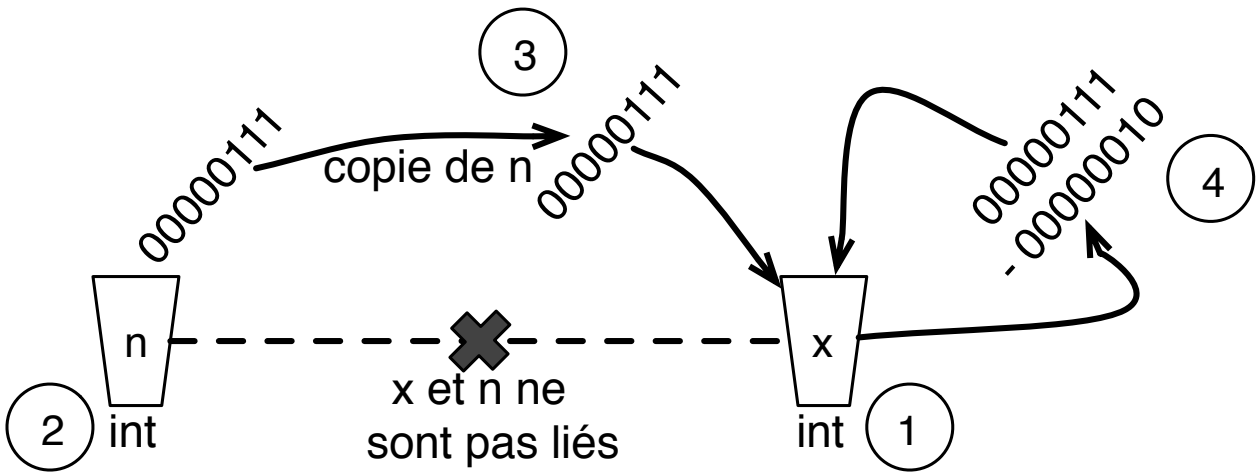
```
... int n = 7; ...
```

3 On appelle la méthode soustraire() en lui passant en paramètres n. Les bits contenus dans n sont copiés dans x :

```
... t.soustraire_2(n) ...
```

4 On modifie la valeur de x. Celle de n ne change pas !

```
... x = x - 2; .
```



Types de retour

Toute méthode a un type de retour qui correspond au type renvoyé par la méthode. Si rien n'est renvoyé alors le type est void. Parler du polymorphisme

```
class Operation{
  int soustraire_bis (int x, int y) {
     int z =  x - y;
     retourn z;
  }
}

class TestOperation{
  public static void main(String[] args){
     int n = 7;
     int res;
     Operation t = new Operation();
     res = t.soustraire_bis(n, 3);
     System.out.println("La valeur de res  est " + res);
  }
}
```

Quelques questions

Que se passe-t-il si l'argument est un objet ?

Qu'est ce qu'une méthode doit retourner ?

Une méthode peut-elle retourner plusieurs valeurs ?

Que faire de la valeur de retour d'une méthode ?

Dans tout ce que nous avons vu jusqu'ici les données n'étaient pas protégées.

Exemple : dans la classe chien, tout le monde peut modifier la taille du chien à l'aide de l'opérateur « . » et affecter des valeurs non pertinantes, voire même des valeurs pouvant faire « bugguer » votre programme :

```
fido.taille = 0;
```

Pour résoudre ce problème, deux choses à faire :

- protéger les variables d'instance d'une classe A à l'aide du modificateur d'accès private : seules les méthodes de la classe A peuvent dorénavent modifier ces variables d'instances.

- créer des méthodes permettant de manipuler ces variables d'instances tel que vous le souhaitez, appelées accesseurs (get) et mutateurs (set), et les rendre accéssibles à tout le monde en utilisant le modificateur d'accès public.

Exemple : Dans la classe Chien : encapsulez toutes les variables et créez les méthodes accesseurs et mutateurs associées :

```
class Chien {
  private int taille;
  private String nom;

  public void setTaille(int t){
    if ( t > 9 ) {
      taille = t;
    }
  }

  public int getTaille(){ return taille; }

  public void setNom(String n){ nom = n; }

  public String getNom(){ return nom; }

  void aboyer(int nbFois) { ... }
}
```

Exemple : A l'extérieur de la classe Chien : Au lieu d'utiliser directement les variables d'instances de la classe Chien, utiliser les accesseurs et les mutateurs.

```
class TestChien {

  public static void main(String[] args){
    Chien c = new Chien();
    c.setTaille(60);
    c.setNom("Fido");
    c.aboyer(3); // demande à fido d aboyer 3 fois
    system.out.println("la taille de " + c.getNom()
            + "est :" + c.getTaille());
  }
}
```

L'encapsulation (encore)

Exemple : Avec un tableau :

```
public class TestChien {

  public static void main(String[] args){
    Chien[] meute = new Chien[3];

    meute[0] = new Chien();
    meute[1] = new Chien();
    meute[2] = new Chien();

    meute[0].setNom("Beethoven"); // meute[0].nom = "Beethove
    meute[1].setNom("Médor"); // meute[1].nom = "Médor";
    meute[2].setNom("Mirza"); // meute[2].nom = "Mirza";

    int i = 0;
    while ( i < meute.length) {
      meute[i].aboyer();
      i = i + 1;
    }
  }
}
```

Initialisation de variables d'instance

```
class Chien {
  private int taille;
  private String nom;

  public int getTaille(){ return taille; }

  public String getNom(){ return nom; }
}

class TestChien {

  public static void main(String[] args){
    Chien c = new Chien();
    system.out.println("Nom du chien :" + c.getNom());
    system.out.println("Sa taille :" + c.getTaille());
  }
}
```

Ce programme compile-t-il ?

Si oui, quel est le résultat de ce programme ?

Initialisation de variables d'instance (suite)

Il n'est pas nécessaire d'initialiser les variables d'instance, car elles ont toujours une valeur par défaut :

type	valeur par défaut
entiers	0
décimaux	0.0
booléens	false
références	null

Si la valeur par défaut n'est pas celle que vous souhaitez, alors il faut initialiser la variable d'instance en même temps que sa déclaration :

```
class Chien {
  private int taille = 54;
  private String nom = "Mirza";
  ...
}
```

Et pour les variables locales

Les variables d'instance sont déclarées dans une classe.

Les variables locales sont déclarées dans une méthode.

Contrairement aux variables d'instance, une variable locale doit toujours être initialisée avant usage. Les variables locales n'ont pas de valeur par défaut.

```
class Foo {
  public void go() {
    int x;
    int z = x + 3;
  }
}
```

Ce programme ne compilera pas.

Et pour les paramètres des méthodes ?

Ce que nous avons vu ...

- Les méthodes d'une classe A peuvent modifier les variables d'instance de A. Inversement les variables d'instance de A peuvent modifier le résultat des méthodes de A.
- Les méthodes peuvent être paramètrées. Dans ce cas, l'appel d'une méthode doit être réalisée avec des arguments correspondant aux paramètres de la méthodes.
- Les arguments sont passés par valeur.
- Encapsulez vos classe : les variables d'instance doivent être protégées (modificateur private), les accesseurs et mutateurs doivent être publics (modificateur public).
- Les variables d'instances sont initialisées par défaut. Les variables locales doivent être explicitement initialisées.