

Programmation Orientée Objet - Licence TIS CM2/9

Lancelot PECQUET

Lancelot.Pecquet@math.univ-poitiers.fr



Poitiers, le 13/01/2006

- 1 Structure d'une classe
- 2 Constructeurs
- 3 Encapsulation
- 4 Conversion en chaîne et égalité
- 5 Gestion mémoire, destructeurs et composition

Rappel sur la séance précédente

La fois précédente, nous avons vu :

- 1 une introduction générale à la POO
- 2 une présentation du langage Java
- 3 les notions de classe, d'agrégation et d'encapsulation

Aujourd'hui, nous voyons en détail :

- la structure des classes
- les constructeurs et les destructeurs
- les différents aspects de l'encapsulation

Une classe minimale

Diagramme de la classe point en UML

nom de la classe	point
champs	x y
méthodes	get_x() get_y() set_x() set_y()

Version Java minimale (point.java)

```
1 class point{
2   int x,y; // abscisse et ordonnee entieres
3
4   // accesseurs:
5   int get_x() {return x;}
6   int get_y() {return y;}
7
8   // mutateurs:
9   void set_x(int x){this.x=x;} // this pour lever l'ambiguite sur x
10  void set_y(int y){this.y=y;} // this pour lever l'ambiguite sur y
11 }
```

Invocation d'une méthode

Syntaxe

Soit C une classe. Pour invoquer une méthode f de C sur une instance M , la syntaxe est :

$M.f(\text{liste des arguments})$.

Exemple sur une instance p de point

```
1 int a = p.get_x(); // a est l'abscisse de x
2 p.set_y(12); // positionne l'ordonnee de p a 12
3 System.out.println(a); // Affichage de a
```

NB

Typage statique des champs et des méthodes.

Polymorphisme \implies choix retardé du corps de méthode.

Mot-clé static

static :

- ① champ : ne dépend pas d'une instance de la classe (attaché à la classe elle-même) ;
- ② méthode : ne fait appel qu'à des champ statiques.

```
1 class votant{
2     static int nombre_votants;
3     static int getNombreVotants(){return nombre_votants;}
4     boolean a_vote = false;
5     void vote(){if(!a_vote){nombre_votants++;} a_vote = true;}
6 }
7 ...
8 votant A;
9 votant B;
10 votant.getNombreVotants() // 0
11 A.vote();
12 B.vote();
13 votant.getNombreVotants() // 2
```

Mot-clé final

final :

- 1 champ : constant (tentative de modification → erreur de compilation)
- 2 méthode : non redéfinissable dans une classe héritière
- 3 classe : qu'elle ne pourra pas avoir d'héritière

```
1 final int x = 1; // constante entiere  
2 x = 2; // erreur a la compilation
```

Classes enveloppantes de types primitifs

- int → Integer
- float → Float
- ...
- vue des types primitifs comme des objets
- constantes statiques associées e.g. :
 - Float.NaN, Float.POSITIVE_INFINITY (IEEE754)
 - Long.MAX_VALUE (le plus grand entier long)
- méthodes associées e.g. Integer.parseInt(s) qui renvoie un int issu de la String s

Questions ?



Définition des constructeurs de la classe point

Définition

Un **constructeur** est une méthode qui porte le nom de la classe et n'a pas de type de retour (même pas void!). Les constructeurs servent à fabriquer les instances de la classe.

NB

Contrainte technique due à l'héritage (expliquée plus tard) : toutes les méthodes utilisées dans les constructeurs (typiquement accesseurs et mutateurs) doivent être **final**.

Définition des constructeurs de la classe point

Constructeurs de point dans point.java

```
1 class point{
2   int x,y;
3   ...
4   // constructeurs:
5   point(int x, int y){set.x(x); set.y(y);}
6   point(){this(0,0);} // this designe un autre constructeur
7   point(point p){this(p.get.x(),p.get.y())} // constructeur "de copie"
8   ...}
```

Utilisation des constructeurs de point dans Main.java

Le fichier point.java

```
1 class point{
2   int x,y;
3   ...
4   // constructeurs
5   point(int x, int y){set.x(x); set.y(y);}
6   point(){this(0,0);} // this designe un autre constructeur
7   point(point p){this(p.get.x(),p.get.y())} // constructeur "de copie"
8   ...}
```

Le fichier Main.java (dans le même répertoire que point.java)

```
1 class Main{
2   public static void main(String[] args){
3     point p = new point(3,4); // un nouveau point p, x=3, y=4
4     point o = new point(); // l'origine (appel du constructeur precedent)
5     point q = new point(p); // une copie q de p
6     System.out.println(p.get.x()); // Affichage de l'abscisse de p
7   }
8 }
```

Questions ?

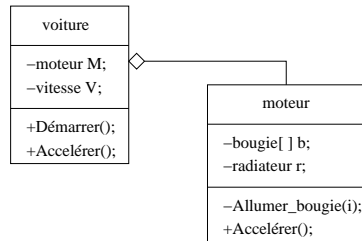


Principe de l'encapsulation

Pas d'accès direct à l'état de l'objet → **message**.

- sûreté : modification \implies invocation **explicite** des mutateurs
- respecte la structure interne de l'objet
- masquage de l'implémentation

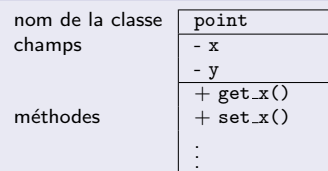
Exemple



- dans une voiture, il n'y a pas d'interrupteur pour allumer/éteindre à la demande chaque bougie (!)
- on modifie indirectement l'état des bougies en passant des messages au moteur ("accélère à telle vitesse", etc)

Encapsulation de point

Diagramme de la classe point en UML (version 2)



Version Java (point.java)

```
1 class point{
2     private int x,y; // abscisse et ordonnee entieres
3
4     public int get_x(){return x;}
5     public void set_x(int x){this.x=x;}
6     ...
7     public point(int x, int y){set_x(x); set_y(y);}
8     public point(){this(0,0);}
9     ...}
```

Modes d'encapsulation

- 1 **private** : depuis la classe uniquement
 - champ : préférentiellement (encapsulation de l'état de l'objet)
 - méthode : choix retenu pour les méthodes internes à l'objet
- 2 **package** : depuis le package uniquement (défaut)
 - champ : pratique et parfois raisonnable si le package est petit
 - méthode : idem
- 3 **protected** : package + classes héritières
 - champ : peut être plus efficace mais attention à l'encapsulation
 - méthode : minimal pour être accessibles par héritage
- 4 **public** : depuis partout
 - champ : rare car les données ne sont plus encapsulées
 - méthode : nécessaire pour interagir avec l'objet
 - classe : pour la rendre utilisable en dehors du paquetage

Utilisation des accesseurs et mutateurs

Fichier point.java

```
1 class point{
2     private int x,y; // abscisse et ordonnee entieres
3
4     public int get_x(){return x;}
5     public void set_x(int x){this.x=x;}
6     ...}
```

Fichier Main.java

```
1 point p;
2 ...
3 int a,b;
4 a = p.x; // provoque une erreur a la compilation
5 a = p.get_x(); // recuperation sans risque de l'abscisse de p
6 p.set_x(3); // modification explicite de l'abscisse de p
```

Masquage de l'implémentation

- une fois définis les mutateurs/accesseurs, la structure interne des données peut être masquée.
- par exemple, pour représenter un complexe $z = x + iy$:

```
1 class complex{
2     private double x,y;
3
4     public double get_partie_relle(){return x;}
5     public double get_partie_imaginaire(){return y;}
6     public double get_module(){return Math.Sqrt(x*x + y*y);}
7     public double get_argument(){return Math.Atan(y/x);}
8 }
```

Masquage de l'implémentation

- précédemment représentation de z par $\text{re}(z)$ et $\text{im}(z)$
- autre représentation par $\rho = |z|$ et $\theta = \arg(z)$ ($z = \rho e^{i\theta}$) :

```
1 class complex{
2     private double rho,theta;
3
4     public double get_partie_relle(){return rho*Math.Cos(theta);}
5     public double get_partie_imaginaire(){return rho*Math.Sin(theta);}
6     public double get_module(){return rho;}
7     public double get_argument(){return theta;}
8 }
```

- dans les deux cas les accesseurs et mutateurs ont même signature (mais pas même implémentation)

Questions ?



Conversion d'un objet en String

System.out.println(p) appelle automatiquement toString() :

Fichier point.java

```
1 class point{...
2   public String toString(){return "(" + x + "," + y + ")";} // concatenation avec +
3 ...}
```

Fichier Main.java : conversion en String ≠ affichage

```
1 point p = new point (3,4); System.out.println(p);
```

Affichage

```
1 (3, 4)
```

Note sur la sémantique de l'égalité

Sémantique par valeurs pour les types primitifs

```
1 int x = 5;  
2 int y = 5;  
3 boolean b = (x == y); // true car x et y ont la même valeur
```

Sémantique par référence pour les types non-primitifs

```
1 point p = new point(3,4);  
2 point q = new point(3,4);  
3 boolean b = (p == q); // false car p et q occupent un espace différent en mémoire
```

ou encore :

```
1 Integer X = new Integer(5);  
2 Integer Y = new Integer(5);  
3 boolean b = (X == Y); // false car X et Y pour la même raison
```

Questions ?



Programmation de haut niveau

La conception dans un langage orienté objets implique de réfléchir à un niveau d'abstraction élevé et d'éviter les soucis de bas niveau tels que :

- processeur
- OS
- gestion mémoire
- ...

Allocation, désallocation : *garbage collection*

- À la demande du programme, l'allocateur trouve un bloc mémoire B , initialement atteignable :
 - soit le programme a une référence directe sur B (« racine »);
 - soit il existe un bloc atteignable contenant une référence B .
- Les blocs non-référencés sont inutiles :
 - identification ;
 - libération de la mémoire.

Gestion automatique de la mémoire

- principe connu depuis les années 60 (Lisp, Smalltalk, ML...)
- avantages :
 - fiabilité accrue (plus de seg. fault, fuites de mémoire...)
 - gain de temps de développement
- inconvénients :
 - performances parfois diminuées (pauses)
 - configuration du comportement du GC parfois complexe

Pratique Java : finalize() et System.gc()

Quand un objet M devient inaccessible (sortie du scope...), Java :

- le détruit en appelant $M.finalize()$
- récupérant la mémoire à nouveau libérée avec le GC

Java a une stratégie d'optimisation des finalisations et collections pour les objets inutilisés mais on peut la contourner :

- `System.runFinalization()` force les `finalize()`
- `System.gc()` force le GC à récupérer les cellules mémoire

Le destructeur `finalize()`

Certaines classes peuvent avoir besoin de redéfinir `finalize()`, par exemple pour :

- effacer des fichiers temporaires
- libérer des ressources matérielles
- modifier le comportement du programme à la mort d'un objet lorsqu'un objet disparaît
- ...

Exemple d'utilisation de `finalize()`

Exemple :

```
class point{...
    // compteur du nombre de points crees:
    private static nb.points = 0;
    ...
    // le constructeur incremente le compteur de points a chaque creation:
    point(int x,int y){
        set_x(x.get_x()); set_y(y.get_y());
        set_nb_points((get_nb_points()+1));
    }

    // ce destructeur decremente le compteur de points:
    protected void finalize() throws Throwable{
        set_nb_points((get_nb_points()-1));
        super.finalize(); // Peut lever un Throwable
    }
}
```

Exemple d'utilisation de `finalize()` (suite)

Source du `main()`

```
public static void main(String[] args){  
    point p1 = new point();  
    System.out.println("1?_" + point.n);  
    for(int i=0;i<10;i++){point p = new point();}  
    // p n'est plus dans le scope:  
    System.out.println("1?_" + point.n);  
}
```

Sortie

```
1 1? 1  
2 1? 11
```

Exemple d'utilisation de `finalize()` (fin)

Source du `main()`

```
public static void main(String[] args){  
    point p1 = new point();  
    System.out.println("1?_" + point.n);  
    for(int i=0;i<10;i++){point p = new point();}  
    // p n'est plus dans le scope:  
    System.gc(); // On force l'appel au GC  
    System.out.println("1?_" + point.n);  
}
```

Sortie

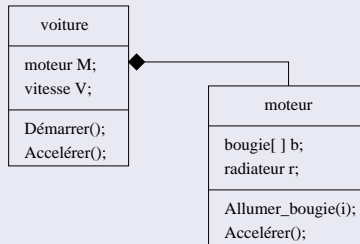
```
1 1? 1  
2 1? 1
```

Une variante de l'aggrégation : la composition

Définition

Lorsqu'une classe C_1 possède un champ de type C_2 tel que la durée de vie des instances de C_2 est celle des instances de C_1 , on dit que l'aggrégation correspondante est une **composition**.

Exemple : le moteur part à la casse avec la voiture



Questions ?



Conclusion

Aujourd'hui, nous avons vu :

- 1 Structure d'une classe
- 2 Constructeurs
- 3 Encapsulation
- 4 Conversion en chaîne et égalité
- 5 Gestion mémoire, destructeurs et composition

La séance prochaine, nous verrons l'héritage simple.