IFT 1010 - Programmation 1

Objets 3

Sébastien Roy & François Duranleau

Département d'informatique et de recherche opérationelle
Université de Montréal
hiver 2003

Au programme

[Tasso : 7,8] et [Niño : 2.3-5]

- Protection : `public` vs `private`
- Contexte d'appel et le mot-clé `this`
- Multiples constructeurs
- Constructeur par défaut

Rappel

On a vu deux types d'objets :

Composites

Tous les attributs sont publiques. *Rarement* utilisé en pratique. Lorsqu'employé, ils servent typiquement comme simple extension au types de base, et la modification d'un attribut ne doit pas invalider l'objet.

Ex. : Un point générique.

Non composites

Tous les attributs sont protégés (privés). On considère l'objet plutôt comme une entité avec laquelle on effectue des requêtes et des commandes. L'objet a le plein *contrôle* de ses attributs.

`public` *vs* `private`

Première question existentielle de la journée : qu'est–ce `public` et `private` veulent *vraiment* dire ?

$\rightarrow$ Ces mots clés définissent les droits d'*accès* ou d'*opération* aux différents endroits dans un programme.

`public`
: L'accès ou l'opération est permise en tout temps.

`private`
: L'accès ou l'opération est permise *seulement dans le code même* de la classe.

Ces règles d'accès s'applique sur *tout* (variables d'instance ou de classe, méthodes, constructeurs).

Exemple

```
// Bidon.java
public class Bidon
{
  public int contenant; // accessible de partout
  private int contenu;  // accessible seulement dans la classe

  public Bidon() {
    init(); // opération légale, on est dans la classe
  }

  private void init() {
    contenant = 1;

    // accès légale, init() est dans la classe Bidon
    contenu = 2;
  }
}
```

```
// TestBidon.java
public class TestBidon
{
  public static void main( String[] arg )
  {
    Bidon b = new Bidon(); // constructeur publique => ok

    b.init(); // méthode privée => illégale! NE COMPILE PAS!

    // publique donc accès permis:
    System.out.println( "contenant = " + b.contenant );

    // privé donc accès non permis. NE COMPILE PAS!
    System.out.println( "contenu = " + b.contenu );
  }
}
```

$\Rightarrow$ Un accès illégal provoque une erreur de *compilation*.

Attributs privés

Deuxième question existentielle de la journée : pourquoi mettre des attributs privés ?

$\rightarrow$ (**Rappel**) pour protégér les données de l'objet, de sorte qu'il ait plein contrôle sur la valeur de ses attributs (son état).

$\Rightarrow$ (**Rappel**) on procède par requêtes et commandes (méthodes publiques) pour consulter ou modifier l'objet.

Troisième question existentielle de la journée : pourquoi mettre un attribut privé s'il n'est pas utilisé dans les méthodes de la classe ?

$\rightarrow$ À RIEN ! :) C'est de la mauvaise programmation !

Méthodes privées

Quatrième question existentielle de la journée : pourquoi mettre des méthodes privés ?

$\longrightarrow$ pour définir des opérations que seul l'objet même peut faire, car autrement l'opération pourrait invalider son état, ou alors parce qu'il s'agit d'une opération interne qui n'a pas de raison d'être publique.

$\Rightarrow$ tout comme la protection des attributs, on protège certaines opérations pour que l'objet contrôle son état.

Ex. : Conservation en mémoire du calcul d'une requête dont la valeur dépend des attributs de l'objet qui peuvent changer.

Exemple

Soit un cercle avec son rayon comme attribut.

$\longrightarrow$ Une requête possible sur le cercle est sa circonférence.

$\Rightarrow$ la requête effectue le calcul.

$\Rightarrow$ Si on veut éviter qu'il soit fait à chaque appel, il faut le garder en mémoire.

$\longrightarrow$ Mais si on change le rayon, la circonférence est maintenant invalide et le calcul devra être fait à nouveau.

Exemple (le code)

```
public class Cercle
{
  private double rayon; // le rayon du cercle
  private double circ;  // résultat du calcul de la circonférence

  public Cercle( double unRayon )
  {
    changeRayon( unRayon );   // initialise le rayon
    circ = -1; // négatif veut dire qu'il faudrat faire le calcul
  }

  public double litRayon() { return rayon; }

  public void changeRayon( double unRayon )
  {
    if( unRayon < 0 ) // validation du rayon en entrée
      unRayon = 0;
    rayon = unRayon;
  }
```

```
  public double litCirconference()
  {
    if( circ < 0 ) // pas encore calculé
      circ = 2 * Math.PI * litRayon(); // => mise en mémoire
    return circ;
  }
}
```

Le code n'est pas complet. Il y a un problème si on fait ceci :

```
Cercle c = new Cercle( 0.5 );
System.out.println( c.litCirconference() ); // affiche 3.1415...
c.changeRayon( 1 );
System.out.println( c.litCirconference() ); // affiche encore 3.1415...!
```

Pourquoi ?

$\longrightarrow$ Lorsqu'on change le rayon, `circ` reste inchangé et donc au second appel de `litCirconference()`, `circ` n'est pas négatif, et alors le calcul n'est pas refait.

Exemple (prise 2)

$\Rightarrow$ Il faut remettre une valeur négative dans `circ` lorsqu'on change le rayon. On va appeler cette opération `invalideCalcul`. Voici le nouveau code :

```
public class Cercle
{
  private double rayon; // le rayon du cercle
  private double circ;  // résultat du calcul de la circonférence

  public Cercle( double unRayon )
  {
    changeRayon( unRayon ); // initialise le rayon
    // circ sera initialisé dans changeRayon()
  }

  public double litRayon() { return rayon; }
```

```
  public void changeRayon( double unRayon )
  {
    if( unRayon < 0 )  // validation du rayon en entrée
      unRayon = 0;
    rayon = unRayon;
    // Nouveaté:
    invalideCalcul(); // re-initialise les valeurs calcules
  }

  public double litCirconference()
  {
    if( circ < 0 ) // pas encore calculé
      circ = 2 * Math.PI * litRayon(); // => mise en mémoire
    return circ;
  }

  // Nouveaté:
  private void invalideCalcul()
  {
    circ = -1;
  }
}
```

Exemple

La méthode `invalideCalcul()` est privé car il s'agit d'une opération *interne*.

Pourquoi une méthode pour une opération aussi simple ? Pourquoi ne pas écrire directement `circ = -1` dans la méthode `changeRayon()` ?

- $\longrightarrow$ Parce que c'est plus élégant.
- $\longrightarrow$ Mais surtout, parce que c'est du code plus facilement *extensible* ! Si, par exemple, on voulait ajouter le calcul de l'aire du cercle, la seule place à changer pour invalider le calcul de l'aire est dans cette méthode.

Exemple

Mais `invalideCalcul()` est appelée seulement dans $\mathrm{changeRayon}()$, alors la question se pose encore.

$\rightarrow$ Et si on avait un cylindre au lieu d'un cercle ? On aurait le volume et l'aire du cylindre qui dépendent de ses deux attributs : le rayon et la hauteur.

$\Rightarrow$ Dès qu'on change l'un ou l'autre des attributs, les calculs sont invalidés.

$\Rightarrow$ On appel la méthode `invalideCalcul()` à tout changement des attributs au lieu d'écrire directement les instructions d'invalidations à chaque endroit et risquer d'en oublier.

Exercice : Écrire la classe `Cylindre`.

Méthodes privés

Cinquième question existentielle de la journée : pourquoi mettre une méthode privée si elle n'est pas utilisé dans les méthodes de la classe ?

$\rightarrow$ À RIEN ! C'est soit de la mauvaise programmation, ou alors une méthode en attente d'être utilisée dans des développements futurs.

Autres protections

– Et si on omet le mot clé `public` ou `private` ?

$\longrightarrow$ Il s'agit d'un autre mode de protection relié à la notion de $package$, dont on ne parlera pas dans ce cours.

– Qu'est-ce que le `public` devant une classe ?

$\longrightarrow$ Signifie que la classe est accessible dans tout $package$. Encore une fois, on ne parlera pas de cette notion dans ce cours.

Remarque : Ce type de protection relié au $package$ n'est pas commun à tous les langages.

Statique *vs* non statique

Sixième question existentielle de la journée : comment une méthode fait-elle pour distinguer les attributs des différents objets qui l'appellent ?

$\Rightarrow$ Septième question existentielle de la journée : qu'est-ce qui différencie vraiment les méthodes statiques des méthodes non statiques ?

$\longrightarrow$ (**Rappel**) C'est le contexte d'appel !

Contexte d'appel

Soit `ref` une référence sur un objet quelconque. Lorsqu'on écrit

```
ref.methode() ;
```

`ref` devant le point définit le *contexte d'appel*. Dans la méthode, tous les attributs manipulés seront ceux de l'objet de la référence `ref`.

Attention aux lecteurs de [Tasso]

Dans [Tasso : 7,8], on dit qu'une instance d'un objet contient non seulement les attributs mais également le $byte\ code$ (code compilé) de chaque méthode.

C'est complètement faux !

Évidemment, comme chaque instance aurait une copie du code de ses méthodes, on aurait un méchanisme facile pour savoir quel attribut modifier puisque chaque méthode serait adaptée à chaque instance.

Mais ça n'a pas de sens de faire ça ! Il y aurait une trop grande consommation de mémoire.

Dans les faits, chaque méthode non statique a un paramètre implicite qui donne la référence sur l'objet appelant.

Le mot-clé `this`

– Chaque méthode non statique (incluant les constructeurs) a un paramètre implicite correspondant au contexte d'appel, *i.e.* la référence sur l'objet appelant.

– Ce contexte porte un nom : le *mot-clé* `this`.

$\Rightarrow$ Dans une méthode, pour tout accès à un attribut ou à une autre méthode, on a les *équivalences* suivantes :

$$\texttt{attribut} \iff \texttt{this.attribut}$$
$$\texttt{methode()} \iff \texttt{this.methode()}$$

Exemple :

```
public void
changeRayon( double r )
{
  rayon = r;
  invalideCalcul();
}
```

$\Leftrightarrow$

```
public void
changeRayon( double r )
{
  this.rayon = r;
  this.invalideCalcul();
}
```

`this` et fonctions statiques

– Dans une fonction statique, `this` n'est pas défini !

⇒ D'où le fait qu'on ne peut pas accéder à des variables d'instances dans ces fonctions.

⇒ Elles sont indépendante du contexte d'appel.

Autre équivalence

On peut considérer les méthodes comme équivalentes à des fonction statiques avec un argument qui porterait le nom `this`, une référence sur le type de la classe.

Soit la classe suivante :

```
public class Bidon
{
  private int attrib;

  public Bidon( int a ) { attrib = a; }

  public int litAttrib() { return attrib; }

  public void methode( int b ) {
    System.out.println( attrib + b );
  }
}
```

Voici l'équivalence :

```
public class Bidon
{
  private int attrib;

  public Bidon( int a ) { attrib = a; }

  public static int litAttrib( Bidon this ) { return this.attrib; }

  public static void methode( Bidon this, int b ) {
    System.out.println( this.attrib + b );
  }
}
```

Voici un usage :

```
Bidon b = new Bidon( 4 );
System.out.println(
    b.litAttrib() );
b.methode( 5 );
```

idem, avec l'équivalence :

```
Bidon b = new Bidon( 4 );
System.out.println(
    Bidon.litAttrib( b ) );
Bidon.methode( b, 5 );
```

Cependant, l'équivalence ne compile pas car `this` est un mot réservé.

Et si nommait `this` autrement ?

On pourrait réécrire l'équivalence en appelant `this` autrement, par exemple `ceci`. Alors le tout compilerait.

- $\Rightarrow$ Maintenant, il n'y a plus de fonction non statique !
- $\Rightarrow$ Allourdit considérablement l'écriture du code !
- $\Rightarrow$ Mauvais style !
- $\Rightarrow$ Engendre des problèmes reliés à l'héritage de classes (qui n'est pas sujet au cours).

`this` et les constructeurs

Si `this` est un paramètre implicite aux méthodes, alors comment pourrait-il l'être dans un constructeur puisqu'*a priori* on ne connait pas encore la référence ?

Rappel : Un constructeur n'est en fait qu'une méthode spéciale appelée immédiatement après l'instanciation de l'objet.

$\Rightarrow$ Si on pouvait appeler directement un constructeur (mais on ne peut pas), on aurait l'équivalence suivante :

```
Bidon b = new Bidon( val );   ⇔   Bidon b = new Bidon();
                                  b.Bidon( val );
```

d'où l'existence de `this` dans pour les constructeurs.

Attributs et visibilité

Soit la classe suivante :

```
public class Point
{
  public double x, y;

  public Point( double x, double y )
  {
    double tmp_x = x;
    double tmp_y = y;

    x = tmp_x;
    y = tmp_y;
  }
}
```

Qu'est-ce qui sera affiché par les instructions suivantes ?

```
Point p = new Point( 10, 5 );
System.out.println( p.x + ", " + p.y );
```

Réponse :

$\longrightarrow$ `0, 0` Pourquoi ?

Parce que seul les `x` et `y` locaux au constructeur (les paramètres) sont modifiés.

$\Rightarrow$ Il faut soit renommer les paramètres, ou alors se souvenir de l'existence de `this` pour régler le conflit de visibilité, similairement aux variables de classes vs les variables locales.

`this` et visibilité (exemple)

Voici maintenant le code corrigé avec l'usage de `this` :

```
public class Point
{
  public double x, y;

  public Point( double x, double y )
  {
    this.x = x;
    this.y = y;
  }
}
```

Est-ce mieux que de changer le nom des paramètres ? Pas vraiment, mais ce n'est pas inélégant non plus.

Remarque : Évidemment, ceci s'applique pour les méthodes aussi.

Multiples constructeurs

Rappel : Il est possible d'écrire des fonctions qui portent le même nom mais qui ont des paramètres différents.

Exemples :

```
public static int max( int a, int b ) { ... }
public static double max( double a, double b ) { ... }
public static int max( int a, int b, int c ) { ... }
...
```

Il est possible de faire de même avec les constructeurs, $i.e.$ une classe peut avoir plus d'un constructeur avec des paramètres différents.

Multiples constructeurs : pourquoi ?

$\longrightarrow$ Il arrive fréquemment de vouloir plus d'une façon d'initialiser un objet.

Exemple 1 : Un point. On peut vouloir un point nul ou ou point initialisé avec des valeurs.

Exemple 2 : Un colis postal. On peut vouloir instancier avec des valeurs par défaut pour ensuite lire ses valeurs au clavier, ou alors l'instancier avec tous ses attributs initialisés.

Exemple 3 : La classe `String` de Java ! !

Exemple

Voici un exemple pour un point avec *trois* constructeurs :

```
public class Point {
  public double x, y; // coordonnées du points

  /** Construit un point nul. */
  public Point() { x = 0; y = 0; }

  /** Construit un point de coordonnées (x, y) . */
  public Point( double x, double y ) {
    this.x = x;
    this.y = y;
  }

  /** Construit un point de même coordonnées que p. */
  public Point( Point p ) {
    x = p.x;
    y = p.y;
  }
}
```

Exemple

Voici quelque exemple d'instanciation de `Points` :

```
Point p1 = new Point();
Point p2 = new Point( 4, 6 );
Point p3 = new Point( p2 );
```

Remarque : Il est souvent pratique d'avoir un constructeur de la forme suivante :

```
public Classe( Classe c ) { ... }
```

pour créer une copie d'un objet de même type.

Constructeur par défaut

Si une classe ne définit pas de constructeur, le compilateur en crée un par défaut qui est équivalent à ceci :

```
public Classe() {}
```

$i.e.$ un constructeur qui ne fait rien.

$\rightarrow$ **Cependant**, dès qu'une classe définit un constructeur, ce constructeur par défaut *n'existe plus* !

$\Rightarrow$ Si on le veut toujours, il faut le définir *explicitement*.

Exemple

```
public class Point
{
  public double x, y;

  public Point( double x, double y ) {
    this.x = x;
    this.y = y;
  }
}
```

Appels :

```
Point p1 = new Point( 1, 2 ); // ok, le constructeur existe
Point p2 = new Point();       // NE COMPILE PAS, le constructeur
                              // n'existe plus
```

D'où le fait qu'on a définit un constructeur sans argument dans l'exemple du `Point` à trois constructeurs.