

IFT 1015 - Objets 2

Professeur:
Stefan Monnier

B. Kégl, S. Roy, F. Duranleau, S. Monnier
Département d'informatique et de recherche opérationnelle
Université de Montréal

hiver 2006

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

On a vu deux types d'objets:

Composites

Tous les attributs sont publiques.

Rarement utilisé en pratique. Lorsqu'employés, ils servent typiquement comme simple extension aux 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?

public **vs** private

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

→ 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'appliquent sur *tout* (variables d'instance ou de classe, méthodes, constructeurs).

Exemple depuis l'intérieur

```
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 légal, init() est dans la classe Bidon.
        contenu = 2;
    }
}
```

Exemple depuis l'extérieur

```
public class TestBidon
{
    public static void main (String[] arg)
    {
        Bidon b = new Bidon(); // Constructeur publique => OK.
        b.init(); // Méthode privée => 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);
    }
}
```

⇒ Un accès illégal provoque une erreur de *compilation*.

Attributs privés

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

Attributs privés

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

- (**Rappel**) pour protéger les données de l'objet, de sorte qu'il ait plein contrôle sur la valeur de ses attributs (son état).
- ⇒ (**Rappel**) on procède par requêtes et commandes (méthodes publiques) pour consulter ou modifier l'objet.

Attributs privés

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

- (**Rappel**) pour protéger les données de l'objet, de sorte qu'il ait plein contrôle sur la valeur de ses attributs (son état).
- ⇒ (**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?

Attributs privés

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

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

⇒ (**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?

→ À RIEN! :) C'est de la mauvaise programmation!

Méthodes privées

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

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

- 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.
- ⇒ 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: mémorisation de circ

Soit un cercle avec son rayon comme attribut.

- Une requête possible sur le cercle est sa circonférence.
- ⇒ la requête effectue le calcul.
- ⇒ Si on veut éviter qu'il soit fait à chaque appel, il faut le garder en mémoire.
- Mais si on change le rayon, la circonférence est maintenant invalide et le calcul devra être fait à nouveau.

Exemple: mémoisation de circ (le code)

```
public class Cercle {
    private double rayon, circ;

    public double litRayon () { return rayon; }
    public void changeRayon (double unRayon)
    { rayon = unRayon < 0 ? 0 : unRayon; } // Validation.

    public Cercle (double unRayon)
    { changeRayon(unRayon); circ = -1; }

    public double litCirconference ()
    {
        if (circ < 0) circ = 2 * Math.PI * litRayon();
        return circ;
    } }
}
```

Exemple: mémoisation de circ (le bug)

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 3.141  
c.changeRayon(1);  
System.out.println(c.litCirconference()); // affiche encor
```

Pourquoi?

Exemple: mémorisation de `circ` (le bug)

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 3.141  
c.changeRayon(1);  
System.out.println(c.litCirconference()); // affiche encore
```

Pourquoi?

→ 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: mémorisation de circ (fix)

⇒ Il faut remettre une valeur négative dans `circ` chaque fois que l'on change `rayon`. On va appeler cette opération `invalidateCalcul`.

Voici le nouveau code:

Exemple: mémorisation de circ (code fix)

```
public class Cercle {
    private double rayon, circ;
    public double litRayon () { return rayon; }
    public void changeRayon (double unRayon)
    { rayon = unRayon < 0 ? 0 : unRayon; circ = -1; }

    public Cercle (double unRayon)
    { changeRayon(unRayon); }

    private void invalideCalcul () { circ = -1; }
    public double litCirconference ()
    {
        if (circ < 0) circ = 2 * Math.PI * litRayon();
        return circ;
    }
}
```

Exemple: mémoisation de circ (discussion 1)

La méthode `invalidateCalcul` est privée 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`?

Exemple: mémorisation de `circ` (discussion 1)

La méthode `invalidateCalcul` est privée 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`?

- Parce que c'est plus élégant (documentation).
- 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: mémorisation de circ (discussion 2)

Mais `invalidateCalcul` est appelée seulement dans `changeRayon`, alors la question se pose encore.

- 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.
- ⇒ Dès qu'on change l'un ou l'autre des attributs, les calculs sont invalidés.
- ⇒ On appelle la méthode `invalidateCalcul` 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?

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?

→ 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`?

→ 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?

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

Remarque: Cette notion de *package* est spécifique à Java.

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?

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

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?

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

→ C'est le *contexte d'appel!*

Dans un appel de méthode

```
x.methode(a)
```

le *contexte d'appel* est défini par `x`, qui peut être un objet ou une classe:

- Pour une méthode statique, `x` doit être une classe.
- Pour une méthode non-statique, `x` doit être un objet, et dans `methode` les attributs manipulés seront ceux de `x`.

Le mot-clé `this`

Chaque méthode **non statique** (incluant les constructeurs) a un paramètre implicite nommé `this` correspondant au contexte d'appel.

⇒ pour tout accès à un attribut ou à une autre méthode:

`attribut` \iff `this.attribut`

`methode()` \iff `this.methode()`

E.g.:

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

\iff

```
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.

Expliciter `this`: avant

On peut considérer les méthodes comme équivalentes à des fonction statiques avec un argument qui porterait le nom `this`.

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);
    }
}
```

Expliciter `this`: après

```
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);
    }
}
```

Et l'usage de cette class:

```
Bidon b = new Bidon(4);
int x = b.litAttrib();
b.methode(5);
```

⇒

```
Bidon b = new Bidon(4);
int x = 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.

- ⇒ Maintenant, il n'y a plus de fonction non statique!
- ⇒ Allourdit considérablement l'écriture du code!
- ⇒ Mauvais style!
- ⇒ Engendre des problèmes liés 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 connaît pas encore la référence?

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

⇒ 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

```
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:

→ 0, 0

Pourquoi?

Réponse:

→ 0, 0

Pourquoi?

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

⇒ Il faut soit renommer les paramètres, ou alors se souvenir de l'existence de `this` pour régler le conflit de visibilité.

`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?

→ Il arrive fréquemment de vouloir plus d'une façon d'initialiser un objet.

Exemple 1: Un point. On peut vouloir un point nul ou un 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 point.

    public Point () { } // Constructeur par défaut.

    /** 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; }
}
```

Voici quelque exemple d'instanciation de `Point`s:

```
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.

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

⇒ 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.
Point p2 = new Point();    // NE COMPILER PAS.
```

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