Démo 2 : Modificateurs d'accès et Packages (IFT1025, H08)

Mathieu Lemoine

Rappels

Les restrictions d'accès en Java sont posées au niveau des classes.

Les Modificateurs d'accès vus jusqu'à présent :

- **public** : Aucune restriction d'accès.
- **private** : Accès autorisé uniquement à la classe propriétaire.

L'héritage ...

L'héritage permet d'étendre une classe.

En Java, utilisation de **extends** :

```
public class Fille extends Mere { [...] }
```

En CL, utilsation de la « liste d'héritage » :

```
(defclass Fille (Mere)
(...))
```

Relation de type « est-un » (*is-a*).
La classe héritante (classe *fille*) reçoit **toutes** les méthodes membres et **toutes** les variables d'instance de la classe héritée (classe *mère*).

L'héritage ... et l'accès aux variables héritées

Problème :
Comment accèder aux/manipuler les variables héritées ?

Via les méthodes **public** ?
Oui, mais tout ne peux pas être fait ainsi... Le but est d'étendre le fonctionnement interne de la classe.

Pourquoi ne pas modifier le comportement de **private** pour permettre l'accès aux classes filles ? L'encapsulation serait brisée ! ! ! !

D'où l'apparition de **protected**.

protected ... ou pas ?

protected permet de restreindre l'accès aux classes filles de la classe propriétaire.

Comment choisir le niveau d'accès d'une méthode membre ?

Si la méthode n'a pas besoin d'être **public** et que :

- on veut donner un accès plus important aux classes filles.
- on veut que les classes filles puissent redéfinir un comportement de la classe mère.

Alors, **protected** est approprié...
Sinon, on reste **private**...

Exemple

Comment organiser/modéliser un ensemble de classe représentant des formes (Triangle, Carré, Rectangle, Cercle, Disque).

Packages

Qu'est-ce qu'un package ?
Un moyen de regrouper des classes devant coopérer étroitement ensemble. On indique le package d'une classe sur la première du fichier source, avec la ligne « **package** pack; ».

Pour accèder à une classe à l'intérieur d'autre package deux possibilités :

- Écrire le nom canonique de la classe « pack1.pack2.MyClass ».
- utiliser « **import** pack1.pack2.MyClass; » au début du fichier, puis utiliser la classe comme si de rien était.

Pourquoi utiliser des package ?

- Pour préserver l'encapsulation des classes à l'intérieur d'un même module.
- Pour permettre aux classes à l'intérieur du module de coopérer plus finement entre elles.
- Pour pouvoir plus facilement déployer (distribuer) un module.

Packages (2)

Le but des packages étant de privilégier l'interaction entre les classes, il faut pouvoir fournir aux classes des fonctionnalitées qui ne peuvent être appellées qu'à l'intérieur de leur package.

Le dernier modificateur d'accès disponible en Java est appellé *package-private*, « interne au package » (? ?).

Il permet de déclarer des méthodes accessibles uniquement aux classes présentent dans le même package que la classe propriétaire.

Le dernier modificateur d'accès : ϕ

C'est le modificateur par défaut, ce qui veut dire qu'il sera appliqué si aucun autre modificateur n'est indiqué.

Son niveau de liberté s'intercale entre **protected** et **private**. Autrement dit, les méthodes indiquées **protected**, sont accessibles aux autres classes du package, et les méthodes *package-private* ne sont pas accessibles aux classes filles.

En vérité, il est très rarement *nécessaire* de l'utiliser, et peu souvent utile !

Récapitulatif

- L'héritage permet d'étendre les fonctionnalités d'une classe et de factoriser son code.
- Les packages permettent de regrouper des classes ayant une utilité commune.
- L'encapsulation apposée entre les classes se fait sur 4 niveaux :
  - **private** : Seules les instances de la classe propriétaire y ont accès.
  - ϕ : Seules les classes à l'intérieur d'un même package y ont accès.
  - **protected** : Seules les classes filles et les classes à l'intérieur d'un même package y ont accès.
  - **public** : Tout le monde y a accès.

Common Lisp : Héritage et Packages, une autre approche...

Du fait de certaines spécifictés propre à CL (le Dispatch Multiple, notamment), les méthodes « appartenant » à une classe ne sont pas réellement à l'intérieur de celle-ci.
L'héritage ne s'effectue donc qu'au niveau des données membres (*slots*), et le mécanisme d'héritage est légèrement différent.

CL ne propose pas d'encapsulation coercitive (i.e. qui interdit l'accès aux données encapsulées).

L'encapsulation existe cependant...

Voyons rapidement comment et pourquoi CL s'est engagé dans cette voie.

Le Dispatch Multiple en CL

En CL, une classe est avant tout constituée de données (*slots*).

Cela est dû au Dispatch Multiple des méthodes.
En Java, les méthodes subbissent un dispatch simple, c'est-à-dire que pour choisir la méthode à appeller réellement, on ne s'intéresse qu'à un seul des arguments (celui qui est en position privilégiée : avant le point).
On appelle également cela l'« envoie de message » (*Message Passing*).

En CL, il est possible de spécialiser une méthode selon chacun de ses arguments. C'est ça le Dispatch Multiple.

Le Dispatch Multiple en CL (2)

Avec cette approche, il est plus difficile de dire qu'une méthode « appartient » à une classe, et quasi-impossible de trouver une syntaxe permmettant de montrer un envoie de message (i.e. mettant en valeur un des arguments).

C'est pour ça que les classes CL ne contiennent pas de méthodes.

Le second avantage de cette fonctionalité, c'est qu'on peut créer des méthodes spécialisées sur des classes auxquelles on n'a pas d'accès privilégié. Ce qui serait (presque) impossible en Java, ou dans les autres langage utilisant l'envoie de messages.

L'héritage en CL

L'héritage en CL est donc primitivement qu'un ajout de slot à la classe mère.
C'est la définition de méthodes spécialisées pour la classe fille qui créeront un nouveau comportement pour cette dernière.
S'il n'y a pas de méthodes spécialisées pour la classe fille, ce sont les méthodes définies pour les classes mères qui s'appliqueront.
Cependant, CL propose un mécanisme que Java (par exemple) ne propose pas : L'héritage multiple.

L'héritage en CL

Autrement dit, une classe peut hériter (directement) de plusieurs autres classes. Souvent, la première classe héritée est la classe principale, et les suivantes sont des ajouts (*mixin*).

Par exemple : la classe *Rectangle-Colore* pourrait hériter de *Rectangle* en premier lieu, et de *Coloration* en second lieu. *Coloration* n'est pas vraiment une classe destinée à être instanciée, mais plutôt une fonctionnalitée à propager. Elle contiendrait par exemple un slot 'couleur' et des méthodes manipulant la couleur seraient définies.

L'encapsulation de CL

Comme dit plutôt, CL ne propose pas d'encapsulation coercitive. Autrement, il est toujours possible d'accéder à un élément encapusler.
La philosophie de Lisp pourrait être résumée ainsi : « Voilà, je te donne du code qui pourrait être utile, et le manuel avec. Si tu veux faire des choses qui sont pas prévues dans le manuel, tu peux. Mais t'as intérêt à savoir ce que tu fais. Compte pas sur moi pour t'aider si ça marche pas... »

Bien entendu, il s'agit là d'un dialogue entre programmeurs, et un logiciel final sans gestion d'erreur serait totalement inutile...
En résumé l'utilisateur du code est libre de ses mouvements et de ses choix.

Les Packages de CL

Ainsi le système de package de CL permet d'indiquer quelles méthodes, variables, parties du code, on destine à l'utilisation publique (on parle de symboles *exportés*), et quelles parties n'y sont pas destinés (on parle alors de symboles *internes*).

Pour accèder à un symbole exporté, il suffit de faire : pack:symb.
Si on souhaite accèder à un symbole non exporté, il faut doubler les ' :' : pack::symb.

Il est très fortement déconseillé d'utiliser un symbol non exporté, à moins d'en avoir réellement besoin et de savoir ce que l'on fait.
Gardez à l'esprit que le fonctionement interne d'un package ou d'une classe peut être modifié *à tout moment*.

La bonne nouvelle du jour !

Demain, après le lab :
Rencontre du MSLUG (Montréal Scheme and Lisp UserGroup),
Au »Saint-Élizabeth« .

Vous êtes tous les bienvenus !