

# Introduction à la programmation objet

*Thinking in Java*

University of Geneva  
www.miralab.ch

## La programmation par objet PPO

32

Tous les langages de programmation fournissent des abstractions.

On peut dire que la complexité des problèmes qu'on est capable de résoudre est directement proportionnelle au type et à la qualité de nos capacités d'abstraction.

Le langage assembleur est une petite abstraction de la machine sous-jacente. (Fotran, Basic, C)

⇒ Ces langages sont de nettes améliorations par rapport à l'assembleur, mais **leur abstraction première requiert une réflexion en termes de structure ordinateur plutôt qu'à la structure du problème qu'on essaye de résoudre.**

L'autre alternative à la modélisation de la machine est de modéliser le problème qu'on tente de résoudre

LISP sous forme de listes

PROLOG convertit tous les problèmes en chaînes de décisions.

Des langages ont été créés en vue de programmer par contrainte, ou pour programmer en ne manipulant que des symboles graphiques (très restrictif ☹)

University of Geneva  
www.miralab.ch

LCI  
Introduction au langage C

33

L'approche orientée objet va un cran plus loin en fournissant des outils au programmeur pour représenter des éléments dans l'espace problème.

Nous nous référons aux éléments dans l'espace problème et leur représentation dans l'espace solution en tant qu'« objets »

L'idée est que

le programme est autorisé à s'adapter à l'esprit du problème en ajoutant de nouveaux types d'objet, de façon à ce que, quand on lit le code décrivant la solution, on lit aussi quelque chose qui décrit le problème.

la POO permet de décrire le problème avec les termes mêmes du problème plutôt qu'avec les termes de la machine où la solution sera mise en œuvre.

Chaque objet ressemble à un mini-ordinateur ; il a un état, et il a à sa disposition des opérations qu'on peut lui demander d'exécuter. Cependant, là encore on retrouve une analogie avec les objets du monde réel - ils ont tous des caractéristiques et des comportements.

34

# 1. Toute chose est un objet.

Il faut penser à un objet comme à une variable améliorée : il stocke des données, mais on peut « effectuer des requêtes » sur cet objet, lui demander de faire des opérations sur lui-même. En théorie, on peut prendre n'importe quel composant conceptuel du problème qu'on essaye de résoudre (un chien, un immeuble, un service administratif, etc...) et le représenter en tant qu'objet dans le programme.

# 2. Un programme est un ensemble d'objets se disant les uns aux autres quoi faire en s'envoyant des messages.

Pour qu'un objet effectue une requête, on « envoie un message » à cet objet. Plus concrètement, on peut penser à un message comme à un appel de fonction appartenant à un objet particulier.

# 3. Chaque objet a son propre espace de mémoire composé d'autres objets.

Dit d'une autre manière, on crée un nouveau type d'objet en créant un paquetage contenant des objets déjà existants. Ainsi, la complexité d'un programme est cachée par la simplicité des objets mis en œuvre.

# 4. Chaque objet est d'un type précis.

Dans le jargon de la POO, chaque objet est une *instance* d'une *classe*, où « classe » est synonyme de « type ». La plus importante caractéristique distinctive d'une classe est : « Quels messages peut-on lui envoyer ? ».

# 5. Tous les objets d'un type particulier peuvent recevoir le même message.

C'est une caractéristique lourde de signification, comme vous le verrez plus tard. Parce qu'un objet de type « cercle » est aussi un objet de type « forme géométrique », un cercle se doit d'accepter les messages destinés aux formes géométriques. Cela veut dire qu'on peut écrire du code parlant aux formes géométriques qui sera accepté par tout ce qui correspond à la description d'une forme géométrique. Cette *substituabilité* est l'un des concepts les plus puissants de la programmation orientée objet.

## Un objet dispose d'une interface

35

une classe décrit un ensemble d'objets partageant des caractéristiques communes (données) et des comportements (fonctionnalités)

Une fois qu'une classe est créée, on peut créer autant d'objets de cette classe qu'on veut et les manipuler comme s'ils étaient les éléments du problème qu'on tente de résoudre.

Mais comment utiliser un objet ?

Il faut pouvoir lui demander d'exécuter une requête, telle que terminer une transaction, dessiner quelque chose à l'écran, ou allumer un interrupteur. Et chaque objet ne peut traiter que certaines requêtes. Les requêtes qu'un objet est capable de traiter sont définies par son *interface*, et son type est ce qui détermine son interface. Prenons l'exemple d'une ampoule électrique :

Non du type	Ampoule
Interface	<pre>allumer() eteindre() intensifier() diminuer()</pre>



L'interface précise *quelles* opérations on peut effectuer sur un objet particulier. Cependant, il doit exister du code quelque part pour satisfaire cette requête. Ceci, avec les données cachées, constitue l'*implémentation*.

University of Geneva  
www.miralab.ch

```
Ampoule amp = new Ampoule();
amp.allumer();
```

## Limites d'accès

36

Java utilise trois mots clefs pour fixer des limites au sein d'une classe : **public**, **private** et **protected**. Leur signification et leur utilisation est relativement explicite.

Ces *spécificateurs d'accès* déterminent qui peut utiliser les définitions qui suivent.

**public** veut dire que les définitions suivantes sont disponibles pour tout le monde.

Le mot clef **private**, au contraire, veut dire que personne, le créateur de la classe et les fonctions internes de ce type mis à part, ne peut accéder à ces définitions.

**private** est un mur de briques entre le créateur de la classe et le programmeur client. Si quelqu'un tente d'accéder à un membre défini comme **private**, ils récupéreront une erreur lors de la compilation.

**protected** se comporte comme **private**, en moins restrictif : une classe dérivée a accès aux membres **protected**, mais pas aux membres **private**. L'héritage sera introduit bientôt.

Java dispose enfin d'un accès « par défaut », utilisé si aucun de ces spécificateurs n'est mentionné. Cet accès est souvent appelé accès « amical » car les classes peuvent accéder aux membres amicaux des autres classes du même package, mais en dehors du package ces mêmes membres amicaux se comportent comme des attributs **private**.

University of Geneva  
www.miralab.ch

LCI  
Introduction au langage C