L'Orienté Objet : notions et vocabulaire fondamentaux

Cible : tout langage OO.

1. Les deux grands mondes/types/techniques de programmation

Types / Caractéristiques	Procédural ou structuré	Orienté Objet (OO)
Langages	C	C++ / Java
Eléments prépondérants	Fonctions/Procédures	Objets (ensembles groupés de variables et fonctions membres)
Motivation	Actions/Verbes décrivant des traitements particuliers	Sujets dotés de comportements propres
Exemple	Calculer la vitesse d'une voiture	
Philosophie d'appel	calculVitesse(voiture)	Voiture.calculerVitesse()
Code	Fondu et dispersé	Regroupé dans le même ensemble que les données

2. Les avantages de l'OO

- L'OO supplante peu à peu le procédural dans les grands programmes car il présente d'énormes avantages: **facilité d'organisation, réutilisation, méthode plus intuitive, possibilité d'héritage, facilité de correction, projets plus faciles à gérer**. L'intérêt principal de l'OO réside dans le fait que l'on ne décrit plus par le code des actions à réaliser de façon linéaire mais par des ensembles cohérents appelés objets.

- L'OO est facilement concevable car il décrit des entités comme il en existe dans le monde réel. Ainsi, l'objet *Voiture* qui implémente la méthode *Voiture.rouler* et possède la propriété *Voiture.Cylindree* est facilement concevable et peut être utilisée -par exemple- dans une simulation de course automobile où on fera interagir plusieurs objets *Voiture*. Les modèles à objets ont été créés pour modéliser le monde réel. "**Dans un modèle à objets, toute entité du monde réel est un objet, et réciproquement, tout objet représente une entité du monde réel**".

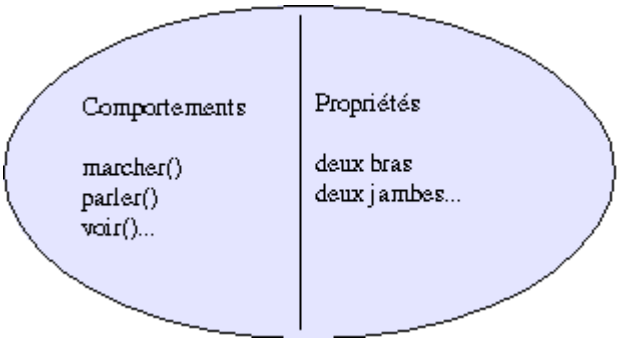
- L'OO permet en quelque sorte de **factoriser le code en ensembles logiques**. Du point de vue de la programmation, L'OO permet d'écrire des programmes facilement lisibles avec un minimum d'expérience, de taille minimale et à la correction aisée. Ces programmes sont, de plus, souvent très stables.

- Les informations concernant un domaine étant centralisées en objets, il est facile de **sécuriser le programme** en interdisant ou autorisant l'accès à ces objets aux autres parties du programme.
- En général, en OO, cohabitent **deux types de développeurs** : ceux qui conçoivent les objets et ceux qui utilisent ces objets dans leurs programmes. De cette façon, la programmation objet permet de *diviser la complexité*. La programmation objet permet d'obtenir des projets stratifiés. Chaque protagoniste ne travaille que sur des implémentations le concernant.

3. Les concepts généraux

Notion de classe

Une classe encapsule, c'est-à-dire **regroupe des propriétés et des comportements**. Par exemple, la classe Humain *définit* des propriétés (deux bras, deux jambes...) et des comportements (marcher, parler, voir...).

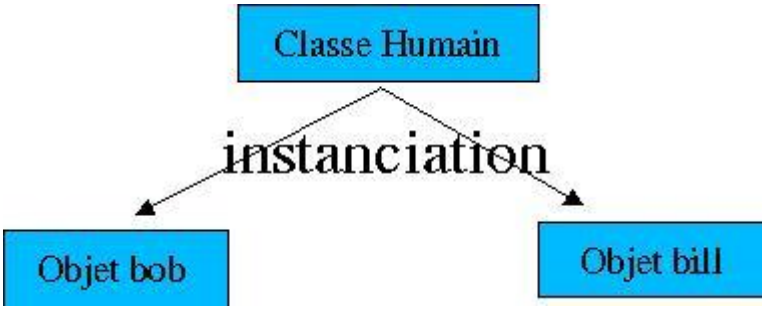


classe Humain

En OO, les comportements sont appelés ***méthodes*** et les propriétés ***variables d'instance***. Notons que les propriétés des classes peuvent elles-mêmes être des objets. Par exemple, la propriété *bras* de notre exemple peut être un objet de type *'Bras'*. Si nous prenons l'exemple d'une voiture, la classe 'Voiture' sera l'ensemble des plans de la voiture.

Notion d'objet

Un objet est une **instance de classe**, c'est-à-dire un **exemplaire utilisable** créé à partir de cette classe et en valorisant certaines propriétés. Par exemple, *bob* ou *bill* sont des instances de la classe Humain, c'est-à-dire des humains ayant des propriétés spécifiques (*bob* est un humain aux yeux noirs et *bill* un humain aux yeux marrons). Notons que le concept de classe est abstrait : personne n'a jamais vu d'*Humain* dans la nature alors que le concept d'objet est fondamentalement concret : il est concevable de croiser des personnes physiques, c'est-à-dire des instances d'*Humain*.



Tout objet possède un type: *bob* est de type *Humain*. L'instance d'une classe est du type de sa classe. C'est le type initial de l'objet. La classe est le *moule,* le *plan,* la *recette* de tout objet. La plupart du temps, les objets sont initialisés lors de leur instanciation. Le système exécute alors automatiquement le ***constructeur*** de la classe. Le constructeur est exécuté à l'instanciation de l'objet. Il prend en argument des propriétés du nouvel objet. L'instanciation d'une voiture rouge se fait par *Voiture(rouge)*. Une classe peut avoir **plusieurs constructeurs**. Dans notre exemple, on peut instancier une *Voiture* en ne lui précisant que sa couleur (exemple précédant) ou en lui précisant en plus sa marque de fabrique : *Voiture(rouge, Renault).* Toujours dans notre exemple de voiture, une instance de Voiture sera physiquement une voiture - unique - qui a été construite à partir du plan 'Voiture'.

Notion de référence sur un objet

Dans les programmes, un objet est une zone mémoire qui est pointée par une ou plusieurs références. Une fois l'objet instancié, il correspond en réalité à une zone mémoire donnée. Cette zone mémoire, pour être utilisée doit être référencée. Par exemple, *bob* est une référence sur l'objet de type *Humain* correspondant et vaut l'adresse mémoire *E515:FB80*. Instancions l'objet *bill* : il y a réservation d'une seconde place mémoire référencée (pointée) par la référence *bill* valant *E515:FF90*. Un objet peut posséder plusieurs références: par exemple, si on fait *bill = bob*, cela signifie qu'ils référencent tous les deux le même objet, la même zone mémoire. Dans ce cas, l'objet bill est donc perdu car il n'est plus référencé. Les références bill et bob désignent maintenant le même objet. Dans la suite de ce document et pour améliorer la compréhension, nous désignerons par abus de langage une référence sur un objet par le terme

'*objet*'. Par exemple: "*l'objet Bill de type Humain"* au lieu de *"la référence Bill sur l'objet correspondant de type Humain".* Dans l'exemple de la voiture, une référence sur une voiture pourrait être sa plaque d'immatriculation.

4. Les propriétés de base de l'OO

L'encapsulation

C'est le fait de regrouper dans une même entité des données et les codes qui les manipulent de manière à empêcher/ restreindre/contrôler les modifications de l'extérieur.

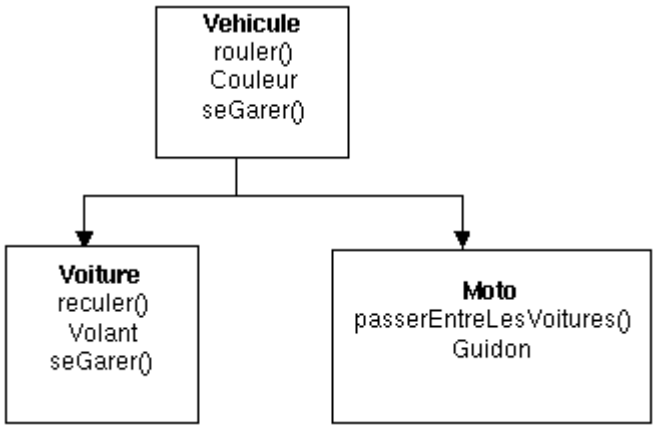
L'héritage

L'héritage permet de **spécialiser** une classe qui possédera non seulement les propriétés et méthodes de sa mère mais également d'autres méthodes spécifiques ou redéfinies. Le terme est *faire dériver la classe en une classe fille*. Dans l'objet fille, on trouve:

- De nouvelles méthodes ou propriétés
- Des méthodes ou propriétés qui *surchargent*, c'est-à-dire redéfinissent celles de la classe mère.
- Les propriétés et méthodes de la classe mère qui n'ont pas été surchargées

Remarque: Les méthodes et propriétés peuvent être héritées à un niveau n, c'est-à-dire qu'un objet peut utiliser une méthode de la mère de sa mère et ainsi de suite. Exemple d'héritage :

Les classes *Voiture* et *Moto* dérivent de la classe *Véhicule.*



Voiture :

Conserve la méthode *rouler()* de *Véhicule*
Conserve la propriété *Couleur* de *Véhicule*
Surcharge la méthode *seGarer* de *Véhicule*
Implémente la méthode *reculer()*
Possède la nouvelle propriété *Volant*

Moto:

Conserve la méthode seGarer() de *Véhicule*
Conserve la méthode *rouler()* de *Véhicule*
Conserve la propriété *Couleur* de *Véhicule*
Implémente la méthode *passerEntreLesVoitures()*
Possède la nouvelle propriété *Guidon*

Le polymorphisme et le transtypage

Les objets sont dits *polymorphes* car ils possèdent plusieurs types: le type de leurs classes et les types des classes ascendantes. Par exemple, *uneHonda*, instance de *Moto* aura comme type initial *Moto* mais une *Moto* possède par héritage le type *Véhicule*. L'objet *uneHonda* est bien un *Véhicule*. Dans certains cas, il est possible de forcer le programme à 'voir' un objet comme un type différent de son type initial, c'est le *transtypage* ou *cast.* Ce transtypage ne modifie pas l'objet mais indique seulement la façon de le voir. Il y a transtypage implicite de la fille vers la mère: une *Voiture* est implicitement de type *Véhicule*.

Attention! Une erreur courante est de penser qu'il y a cast de la mère vers la fille: Une classe mère n'est pas du type de sa fille! Un *Véhicule* n'est pas forcémment une *Voiture*. Dans certains cas, nous avons besoin d'affecter une variable d'un type à une autre d'un autre type. De nombreux problèmes métaphysiques s'en suivent généralement. La bonne façon de raisonner lorsque l'on veut affecter un type à un autre est de procéder comme suit:

On fait *A* <- *B* ; A possède un type initial, le type que l'on a donné à l'instanciation. Cette affectation est valide si et seulement si B est du type initial de A. Autrement dit, il faut que B soit de même type initial que A ou d'un type dérivé de celui de A. Exemple:

monVehicule est une instance de type *Véhicule*;
maVoiture est une instance de type *Voiture*;
maVoiture <- *monVehicule* : est-ce une affectation valide?

maVoiture est de type initial *Voiture* ; *monVehicule* est de type initial *Véhicule* et un *Véhicule* n'est pas forcement une *Voiture* donc cette affectation n'est pas valide. En revanche étudions l'affectation : *monVehicule* <- *maVoiture* :

maVehicule est de type initial *Véhicule* et *maVoiture* est également de type *Véhicule* car *Voiture* dérive de *Véhicule*; donc cette affectation est valide !

Polymorphisme paramétrique

Le terme polymorphisme est également souvent associé par abus de langage au concept de *polymorphisme paramétrique* : un objet peut comporter plusieurs méthodes de même nom et possédant des arguments différents. Ce sont des méthodes polymorphiques. Nous en avons vu un exemple avec les constructeurs.

Voiture.rouler(chemin) la façon dont la voiture roule ne dépend que du chemin.
Voiture.rouler(chemin, meteo) la façon dont la voiture roule dépend du chemin et de la météo.
Voiture.rouler(chemin, meteo, circulation) la façon dont la voiture roule dépend du chemin, de la météo et de la circulation.

Source : https://florat.net//tutorial-java/chapitre01.html