Programmez en orienté objet avec C#

20 heures Moyenne

Mis à jour le 10/03/2017

Introduction à la programmation orientée objet

Dans ce chapitre, nous allons essayer de décrire ce qu'est la programmation orientée objet (abrégée souvent en POO). Je dis bien « essayer », car pour être complètement traitée, la POO nécessiterait qu'on lui dédie un ouvrage entier !

Nous allons tâcher d'aller à l'essentiel et de rester proche de la pratique. Ce n'est pas très grave si tous les concepts abstraits ne sont pas parfaitement appréhendés : il est impossible d'apprendre la POO en deux heures. Cela nécessite une longue pratique, beaucoup d'empirisme et des approfondissements théoriques.

Ce qui est important ici, c'est de comprendre les notions de base et de pouvoir les utiliser dans de petits programmes.
Après cette mise au point, attaquons sans plus tarder la programmation orientée objet !

Qu'est-ce qu'un objet ?

Vous avez pu voir précédemment que j'ai utilisé de temps en temps le mot « objet » et que le mot-clé « `new` » est apparu comme par magie... Il a été difficile de ne pas trop en parler et il est temps d'en savoir un peu plus.

Alors qu'est-ce qu'un objet ?

Si on prend le monde réel (si si, vous allez voir, vous connaissez...), nous sommes entourés d'objets : une chaise, une table, une voiture, etc. Ces objets forment un tout.

- Ils possèdent des propriétés (la chaise possède 4 pieds, elle est de couleur bleue, etc.).

- Ces objets peuvent faire des actions (la voiture peut rouler, klaxonner, etc.).
- Ils peuvent également interagir entre eux (l'objet conducteur démarre la voiture, l'objet voiture fait tourner l'objet volant, etc.).

Il faut bien faire attention à distinguer ce qu'est l'objet et ce qu'est la définition d'un objet

La définition de l'objet (ou structure de l'objet) permet d'indiquer ce qui compose un objet, c'est-à-dire quelles sont ses propriétés, ses actions etc. Comme par exemple le fait qu'une chaise ait des pieds ou qu'on puisse s'asseoir dessus.
Par contre, l'objet chaise est bien concret. On peut donc avoir plusieurs objets chaises : on parle également d'instances. Les objets chaises, ce sont bien celles concrètes que l'on voit devant nous autour de l'objet table pour démarrer une partie de belote.

On peut faire l'analogie avec notre dictionnaire qui nous décrit ce qu'est une chaise. Le dictionnaire décrit en quoi consiste l'objet, et l'instance de l'objet représente le concret associé à cette définition. Chaque objet a sa propre vie et est différent d'un autre. Nous pouvons avoir une chaise bleue, une autre rouge, une autre avec des roulettes, une cassée ...

Bon, finalement la notion d'objet est plutôt simple quand on la ramène à ce qu'on connait déjà  .

Sachant qu'un objet en programmation c'est comme un objet du monde réel mais ce n'est pas forcément restreint au matériel. Un chien est un objet. Des concepts comme l'amour ou une idée sont également des objets, tandis qu'on ne dirait pas cela dans le monde réel.

En conclusion :

- La définition (ou structure) d'un objet est un concept abstrait, comme une définition dans le dictionnaire. Cette définition décrit les caractéristiques d'un objet (la chaise a des pieds, l'homme a des jambes, etc.). Cette définition est unique comme une définition dans le dictionnaire.
- Un objet ou une instance est la réalisation concrète de la structure de l'objet. On peut avoir de multiples instances, comme les 100 voitures sur le parking devant chez moi. Elles peuvent avoir des caractéristiques différentes (une voiture bleue, une voiture électrique, une voiture à 5 portes, etc.)

L'encapsulation

Le fait de concevoir une application comme un système d'objets interagissant entre eux apporte une certaine souplesse et une forte abstraction.

Prenons un exemple tout simple : la machine à café du bureau. Nous insérons nos pièces dans le monnayeur, choisissons la boisson et nous nous retrouvons avec un gobelet de la boisson commandée. Nous nous moquons complètement de savoir comment cela fonctionne à l'intérieur et nous pouvons complètement ignorer si le café est en poudre, en grain, comment l'eau est ajoutée, chauffée, comment le sucre est distribué, etc.

Tout ce qui nous importe c'est que le fait de mettre des sous dans la machine nous permet d'obtenir un café qui va nous permettre d'attaquer la journée.

Voilà un bel exemple de programmation orientée objet. Nous manipulons un objet MachineACafe qui a des propriétés (Allumée/éteinte, présence de café, présence de gobelet, ...) et qui sait faire des actions (AccepterMonnaie, DonnerCafe, ...). Et c'est tout ce que nous avons besoin de savoir. On se moque du fonctionnement interne, peu importe ce qu'il se passe à l'intérieur, notre objet nous donne du café, point !

C'est ce qu'on appelle l'**encapsulation**. Cela permet de protéger l'information contenue dans notre objet et de le rendre manipulable uniquement par ses actions ou propriétés. Ainsi, l'utilisateur ne peut pas accéder au café ni au sucre ou encore moins à la monnaie. Notre objet est ainsi protégé et fonctionne un peu comme une boite noire. L'intérêt est que si la personne qui entretient la machine met du café en grain à la place du café soluble, c'est invisible pour l'utilisateur qui se soucie simplement de mettre ses pièces dans la machine.

> L'encapsulation protège donc les données de l'objet et son fonctionnement interne.

Héritage

Un autre élément important dans la programmation orientée objet que nous allons aborder est l'héritage.

> Ah bon ? Les objets aussi peuvent mourir et transmettre leur patrimoine ?

Eh bien c'est presque comme en droit, à part que l'objet ne meurt pas et qu'il n'y pas de taxe sur l'héritage.
C'est-à-dire qu'un objet dit « père » peut transmettre certaines de ses caractéristiques à un autre objet dit « fils ».

Pour cela, on pourra définir une relation d'héritage entre eux. S'il y a une relation d'héritage entre un objet père et un objet fils, alors **l'objet fils hérite de l'objet père**. On dit également que l'objet fils est une **spécialisation** de l'objet père ou qu'**il dérive de l'objet père**.

En langage plus courant on peut également dire que l'objet fils est « une sorte » d'objet père.

> Des exemples !!

On dit souvent qu'un petit exemple vaut bien un long discours, alors prenons par exemple l'objet « chien » et imaginons ses caractéristiques tirées du monde réel en utilisant l'héritage :

- L'objet « chien » est une sorte d'objet « mammifère »
- L'objet « mammifère » est une sorte d'objet « animal »

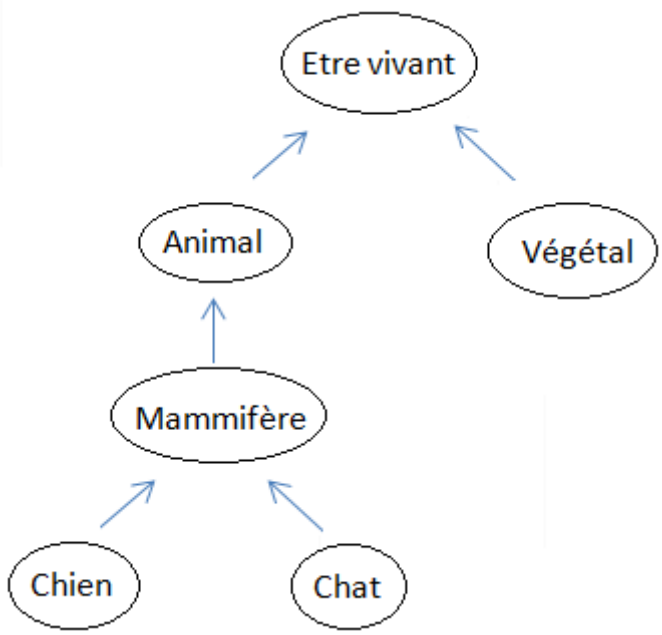
- L'objet « animal » est une sorte d'objet « être vivant »

Chaque père est un peu plus général que son fils. Et inversement, chaque fils est un peu plus spécialisé que son père. Avec l'exemple du dessus, un mammifère est un peu plus général qu'un chien, l'être vivant étant encore plus général qu'un mammifère.

Il est possible pour un père d'avoir plusieurs fils, par contre, l'inverse est impossible, un fils ne peut pas avoir plusieurs pères. Et oui, c'est triste mais c'est comme ça, c'est le règne du père célibataire avec plusieurs enfants à charge !

Ainsi, un objet « chat » peut également être un fils de l'objet « mammifère ». Un objet « végétal » peut également être fils de l'objet « être vivant ».

Ce qu'on peut reproduire sur le schéma suivant. Chaque bulle représentant un objet et chaque flèche représente l'héritage entre les objets.



On peut définir une sorte de hiérarchie entre les objets, un peu comme on le ferait avec un arbre généalogique. La différence est qu'un objet héritant d'un autre peut obtenir certains ou tous les comportements de l'objet qu'il spécialise, alors qu'un petit enfant n'hérite pas forcément des yeux bleus de sa mère ou du côté bougon de son grand-père, le hasard de la nature faisant le reste.

Pour bien comprendre cet héritage de comportement, empruntons à nouveau les exemples du monde réel.

- L'être vivant peut par exemple faire l'action « vivre ».
- Le mammifère possède des yeux.
- Le chien, qui est une sorte d'être vivant et une sorte de mammifère, peut également faire l'action « vivre » et aura des yeux.
- Le chat qui est une autre sorte d'être vivant peut lui aussi faire l'action « vivre » et aura également des yeux.

On voit bien ici que le chat et le chien héritent des comportements de leurs parents et grands-parents en étant capables de vivre et d'avoir des yeux.
Par contre, l'action « aboyer » est spécifique au chien. Ce qui veut dire que ni le chat, ni le dauphin ne seront capables d'aboyer. Il n'y a que dans les dessins animés de Tex Avery que ceci est possible ! Évidemment, il n'y a pas de notion d'héritage entre le chien et le chat et l'action aboyer est définie au niveau du comportement du chien. Ceci implique également que seul un objet qui est une sorte de chien, par exemple l'objet Labrador ou l'objet Chihuahua, pourra hériter du comportement « aboyer », car il y a une relation d'héritage entre eux.
Finalement, c'est plutôt logique.
Rappelons juste avant de terminer ce paragraphe qu'un objet ne peut pas hériter de plusieurs objets. Il ne peut hériter que d'un seul objet. Le C# ne permet pas ce qu'on appelle **l'héritage multiple**, *a contrario* d'autres langages comme le C++ par exemple.

Voilà globalement pour la notion d'héritage. Je dis globalement car il y a certaines subtilités que je n'ai pas abordées mais ce n'est pas trop grave, vous verrez dans les chapitres suivants comment le C# utilise la notion d'héritage et ce qu'il y a vraiment besoin de savoir. Ne vous inquiétez pas si certaines notions sont encore un peu floues, vous comprendrez surement mieux grâce à la pratique.

Polymorphisme - Substitution

Polymorphisme

Le mot **polymorphisme** suggère qu'une chose peut prendre plusieurs formes. Sous ce terme un peu barbare se cachent plusieurs notions de l'orienté objet qui sont souvent sources d'erreurs. Je vais volontairement passer rapidement sur certains points qui ne vont pas nous servir pour me concentrer sur ceux qui sont importants pour ce tutoriel. J'espère que vous ne m'en voudrez pas et que vous m'accorderez quelques dérogations qui pourraient déplaire aux puristes.

En fait, on peut dire qu'une manifestation du polymorphisme est la capacité pour un objet de faire une même action avec différents types d'intervenants. C'est ce qu'on appelle le polymorphisme « ad hoc » ou le polymorphisme « paramétré ». Par exemple, notre objet voiture peut rouler sur la route, rouler sur l'autoroute, rouler sur la terre si elle est équipée de pneus adéquats, rouler au fond de l'eau si elle est amphibie, etc ...
Concrètement ici, je fais interagir un objet « voiture » avec un objet « autoroute » ou un objet « terre »... par l'action qui consiste à « rouler ». Cela peut paraitre anodin décrit ainsi, mais nous verrons ce que cela implique avec le C#.

Substitution

La **substitution** est une autre manifestation du polymorphisme. Il s'agit de la capacité d'un objet fils à redéfinir des caractéristiques ou des actions d'un objet père.

Prenons par exemple un objet mammifère qui sait faire l'action « se déplacer ». Les objets qui dérivent du mammifère peuvent potentiellement avoir à se déplacer d'une manière différente. Par exemple, l'objet homme va se déplacer sur ses deux jambes et donc différemment de l'objet

dauphin qui se déplacera grâce à ses nageoires ou bien encore différemment de l'objet « homme accidenté » qui va avoir besoin de béquilles pour s'aider dans son déplacement.
Tous ces mammifères sont capables de se déplacer, mais chacun va le faire d'une manière différente. Ceci est donc possible grâce à la substitution qui permet de redéfinir un comportement hérité. Ainsi, chaque fils sera libre de réécrire son propre comportement, si celui de son père ne lui convient pas.

Interfaces

Un autre concept important de la programmation orientée objet est la notion d'interface.

> L'interface est un contrat que s'engage à respecter un objet. Il indique en général un comportement.

Prenons un exemple dans notre monde réel et connu : les prises de courant. Elles fournissent de l'électricité à 220V avec deux trous et (souvent) une prise de terre. Peu importe ce qu'il y a derrière, du courant alternatif de la centrale du coin, un transformateur, quelqu'un qui pédale,... nous saurons à coup sûr que nous pouvons brancher nos appareils électriques car ces prises s'engagent à nous fournir du courant alternatif avec le branchement adéquat.
Elles respectent le contrat ; elles sont « branchables ». Ce dernier terme est un peu barbare et peut faire mal aux oreilles. Mais vous verrez que suffixer des interfaces par un « able » est très courant ( ) et permet d'être plus précis sur la sémantique de l'interface. « Able » est également un suffixe qui fonctionne en anglais et les interfaces du framework .NET finissent pour la plupart par « able ».

À noter que les interfaces ne fournissent qu'un contrat, elles ne fournissent pas d'implémentation c'est-à-dire pas de code C#.
Les interfaces indiquent que les objets qui choisissent de respecter ce contrat auront forcément telle action ou telle caractéristique mais elles n'indiquent pas comment faire, c'est-à-dire qu'elles n'ont pas de code C# associé. Chaque objet respectant cette interface (on parle d'objet **implémentant** une interface) sera responsable de coder la fonctionnalité associée au contrat.

Pour manipuler ces prises, nous pourrons utiliser cette interface en disant : « allez hop, tous les branchables, venez par ici, on a besoin de votre courant ». Peu importe que l'objet implémentant cette interface soit une prise murale, une prise reliée à une dynamo ou autre, nous pourrons manipuler ces objets par leur interface et donc brancher nos prises permettant d'alimenter nos appareils.

Contrairement à l'héritage, un objet est capable d'implémenter plusieurs interfaces. Par exemple, une pompe à chaleur peut être « Chauffante » et « Refroidissante ». Notez qu'en français, nous pourrons également utiliser le suffixe « ante ». En anglais, nous aurons plus souvent « able ».

Nous en avons terminé avec la théorie sur les interfaces. Il est fort probable que vous ne saisissiez pas encore tout l'intérêt des interfaces ou ce qu'elles sont exactement. Nous allons y revenir avec

des exemples concrets et vous verrez des utilisations des interfaces dans le cadre du framework .NET qui vous éclaireront d'avantage.

À quoi sert la programmation orientée objet ?

Nous avons décrit plusieurs concepts de la programmation orientée objet mais nous n'avons pas encore dit à quoi elle allait nous servir.

En fait, on peut dire que la POO est une façon de développer une application qui consiste à représenter (on dit également « **modéliser** ») une application informatique sous la forme d'objets, ayant des propriétés et pouvant interagir entre eux. La modélisation orientée objet est proche de la réalité ce qui fait qu'il sera relativement facile de modéliser une application de cette façon. De plus, les personnes non-techniques pourront comprendre et éventuellement participer à cette modélisation.

Cette façon de modéliser les choses permet également de découper une grosse application, généralement floue, en une multitude d'objets interagissant entre eux. Cela permet de découper un gros problème en plus petits afin de le résoudre plus facilement.

Utiliser une approche orientée objet améliore également la maintenabilité. Plus le temps passe et plus une application est difficile à maintenir. Il devient difficile de corriger des choses sans tout casser ou d'ajouter des fonctionnalités sans provoquer une régression par ailleurs. L'orienté objet nous aide ici à limiter la casse en proposant une approche où les modifications internes à un objet n'affectent pas tout le reste du code, grâce notamment à l'encapsulation.

Un autre avantage de la POO est la réutilisabilité. Des objets peuvent être réutilisés ou même étendus grâce à l'héritage. C'est le cas par exemple de la bibliothèque de classes du framework .NET que nous avons déjà utilisée. Cette bibliothèque nous fournit par exemple tous les objets permettant de construire des applications graphiques. Pas besoin de réinventer toute la mécanique pour gérer des fenêtres dans une application, le framework .NET sait déjà faire tout ça. Nous avons juste besoin d'utiliser un objet « fenêtre », dans lequel nous pourrons mettre un objet « bouton » et un objet « zone de texte ». Ces objets héritent tous des mêmes comportements, comme le fait d'être cliquable ou sélectionnable, etc. De même, des composants tout faits et prêts à l'emploi peuvent être vendus par des entreprises tierces (système de log, contrôles utilisateurs améliorés, etc ...).

Il faut savoir que la POO, c'est beaucoup plus que ça et nous en verrons des subtilités plus loin, mais comprendre ce qu'est un objet est globalement suffisant pour une grande partie du tutoriel.

En résumé

- L'approche orientée objet permet de modéliser son application sous la forme d'interactions entre objets.
- Les objets ont des propriétés et peuvent faire des actions.
- Ils masquent la complexité d'une implémentation grâce à l'encapsulation.
- Les objets peuvent hériter de fonctionnalités d'autres objets s'il y a une relation d'héritage entre eux.

PROGRAMMEZ EN ORIENTÉ OBJET AVEC C#

CRÉEZ VOTRE PREMIER OBJET

Le professeur

Nicolas Hilaire

Expert .NET, artisan logiciel, plusieurs fois honoré du titre Microsoft MVP, mais également curieux des autres technologies.


Télécharger dans
l'App Store