« Définitions livre OO Bersini »

Raphael HANNAERT

2015

Objets, attributs, valeurs

Il est possible dans tous les langages informatiques de stocker et de manipuler des objets en mémoire, comme autant d'ensembles de couples attribut/valeur.

La place de l'objet en mémoire

Les objets seront structurellement décrits par un premier ensemble d'attributs de type primitif, tels qu'entier, réel ou caractère, qui permettra, précisément, de déterminer l'espace qu'ils occupent en mémoire.

Bases de données relationnelles

Il s'agit du mode de stockage des données sur support permanent le plus répandu en informatique. Les données sont stockées en tant qu'enregistrement dans des tables, par le biais d'un ensemble de couples attribut/valeur dont une clé primaire essentielle à la singularisation de chaque enregistrement. Des relations sont ensuite établies entre les tables par un mécanisme de jonction entre la clé primaire de la première table et la clé dite étrangère de celle à laquelle on désire la relier. Le fait, par exemple, qu'un conducteur puisse posséder plusieurs voitures se traduit en relationnel par la présence dans la table voiture d'une clé étrangère qui reprend les valeurs de la clé primaire présente dans la table des conducteurs. La disparition de ces clés dans la pratique OO fait de la sauvegarde des objets dans ces tables un problème épineux de l'informatique d'aujourd'hui, comme nous le verrons au chapitre 19.

Espace mémoire

Le référent contient l'adresse physique de l'objet, codée sur 32 bits dans la plupart des ordinateurs aujourd'hui. Le nombre d'espaces mémoire disponibles est lié à la taille de l'adresse de façon exponentielle. Ces dernières années, de plus en plus de processeurs, tant chez Sun, Apple ou Intel, ont fait le choix d'une architecture à 64 bits, ce qui implique notamment une révision profonde de tous les mécanismes d'adressage dans les systèmes d'exploitation. Depuis, les informaticiens peuvent voir l'avenir avec confiance et se sentir à l'aise pour des siècles et des siècles face à l'immensité de l'espace d'adressage qui s'ouvre à eux. Celui-ci devient de 2^{64}, soit 18.446.744.073.709.551.616 octets. Excusez du peu. Aura-t-on jamais suffisamment de données et de programmes à y installer ?

Référent vers un objet unique

Le nom d'un objet informatique, ce qui le rend unique, est également ce qui permet d'y accéder physiquement. Nous appellerons ce nom le « référent de l'objet ». L'information reçue et contenue par ce référent n'est rien d'autre que l'adresse mémoire où cet objet se trouve stocké.

Adressage indirect

C'est la possibilité pour une variable, non pas d'être associée directement à une donnée, mais plutôt à une adresse physique d'un emplacement contenant, lui, cette donnée. Il devient possible de différer le choix de cette adresse pendant l'exécution du programme, tout en utilisant naturellement la variable. Et plusieurs de ces variables peuvent alors pointer vers un même emplacement. Une telle variable est dénommée un pointeur, en C et C++.

Plusieurs référents pour un même objet

Dans le cours de l'écriture d'un programme orienté objet, on accédera couramment à un même objet par plusieurs référents, générés dans différents contextes d'utilisation. Cette multiplication des référents sera un élément déterminant de la gestion mémoire associée à l'objet. On acceptera à ce stade-ci qu'il est utile qu'un objet séjourne en mémoire tant qu'il est possible de le référer. Sans référent un objet est bon pour la poubelle puisque inaccessible. Vous êtes mort, je jour où vous n'êtes même plus un numéro dans aucune base de données.

Changement d'états

Le cycle de vie d'un objet, lors de l'exécution d'un programme orienté objet, se limite à une succession de changements d'états, jusqu'à sa disparition pure et simple de la mémoire centrale.

Héritage

Dans notre cognition et dans nos ordinateurs, le rôle premier de l'héritage est de favoriser une économie de représentation et de traitement. La factorisation de ce qui est commun à plusieurs sous-classes dans une même superclasse offre des avantages capitaux. Vous pouvez omettre d'écrire dans la définition de toutes les sous-classes ce qu'elles héritent des superclasses. Il est de bon sens que, moins on écrit d'instructions, plus fiable et plus facile à maintenir sera le code. Si vous apprenez d'une classe quelconque qu'elle est un cas particulier d'une classe générale, vous pouvez lui associer automatiquement toutes les informations caractérisant la classe plus générale et ce, sans les redéfinir. De plus, vous ne recourrez à cette classe plus spécifique que dans des cas bien plus rares, où il vous sera essentiel d'exploiter les informations qui lui sont propres.

Polymorphisme, conséquence directe de l'héritage

Le polymorphisme, conséquence directe de l'héritage, permet à un même message, dont l'existence est prévue dans une superclasse, de s'exécuter différemment, selon que l'objet qui le reçoit est d'une sous-classe ou d'une autre. Cela permet à l'objet responsable de l'envoi du message de ne pas avoir à se préoccuper dans son code de la nature ultime de l'objet qui le reçoit et donc de la façon dont il l'exécutera.

Surcharge de méthode

La manœuvre consistant à surcharger une méthode revient à en créer une nouvelle, dont la signature se différencie de la précédente, uniquement par la liste ou la nature des arguments.

Signature de méthode

La signature de la méthode est ce qui permet de la retrouver dans la mémoire des méthodes. Elle est constituée du nom, de la liste, ainsi que du type des arguments. Toute modification de cette liste pourra donner naissance à une nouvelle méthode, surcharge de la précédente. La nature du `return` ne fait pas partie de cette signature dans la mesure où deux méthodes ayant le même nom et la même liste d'arguments ne peuvent différer par leur `return`.

Le constructeur

Le constructeur est une méthode particulière, portant le même nom que la classe, et qui est définie sans aucun retour. Il a pour mission d'initialiser les attributs d'un objet dès sa création. À la différence des autres méthodes qui s'exécutent alors qu'un objet est déjà créé et sur celui-ci, il n'est appelé que lors de la construction de l'objet, et une version par défaut est toujours fournie par les langages de programmation. La recommandation, classique en programmation, est d'éviter de se reposer sur le « défaut », et, de là, toujours prévoir un constructeur pour chacune des classes créées, même s'il se limite à reproduire le comportement par défaut. Au moins, vous aurez « explicité » celui-ci. Le constructeur est souvent une des méthodes les plus surchargées, selon les valeurs d'attributs qui sont connues à la naissance de l'objet et qui sont passées comme autant d'arguments.

Langage fortement typé

Un langage de programmation est dit fortement typé quand le compilateur vérifie que l'on ne fait avec les objets et les variables du programme que ce qui est autorisé par leur type. Cette vérification a pour effet d'accroître la fiabilité de l'exécution du programme. Java, C++ et C# sont fortement typés. L'étape de compilation y est essentielle. Ce n'est pas le cas, comme nous le verrons plus loin, de Python et PHP 5.

Statique

Les attributs d'une classe dont les valeurs sont communes à tous les objets, et qui deviennent ainsi directement associés à la classe, ainsi que les méthodes pouvant s'exécuter directement à partir de la classe, seront déclarés comme statiques. Ils pourront s'utiliser en l'absence de tout objet. Une méthode statique ne peut utiliser que des attributs statiques, et ne peut appeler en son sein que des méthodes également déclarées comme statiques.

Programmation procédurale

La programmation procédurale s'effectue par un accès collectif et une manipulation globale des objets, dans quelques grands modules fonctionnels qui s'imbriquent mutuellement, là où la programmation orientée objet est confiée à un grand nombre d'objets, se passant successivement le relais, pour l'exécution des seules fonctions qui leur sont affectées.

OO comparé au procédural en performance et temps calcul

Il est parfaitement incorrect de clamer haut et fort que les performances en consommation des ressources informatiques (temps calcul et mémoire) des programmes OO sont supérieures en général à celles de programmes procéduraux remplissant les mêmes tâches. Tout concourt à faire des programmes OO de grands consommateurs de mémoire (prolifération des objets, distribués n'importe où dans la mémoire violant les principes de « localité » informatique) et de temps calcul (retrouver à l'exécution les objets et les méthodes dans la mémoire, puis traduire, au dernier moment, les méthodes dans leur forme exécutable sans parler du garbage collector et autres « casseur » de performance). Les seuls temps et ressources réellement épargnés sont ceux des programmeurs, tant lors du développement que lors de la maintenance et de l'évolution de leur code. L'OO considère simplement, à juste titre nous semble-t-il, que le temps programmeur est plus précieux que le temps machine.

Association de classes

La manière privilégiée pour permettre à deux classes de communiquer par envoi de message consiste à rajouter aux attributs primitifs de la première un attribut référent du type de la seconde. La classe peut donc, tout à la fois, contenir des attributs et se constituer en nouveau type d'attribut. C'est grâce à ce mécanisme de typage particulier que, partout dans son code, la première classe pourra faire appel aux méthodes disponibles de la seconde.

Communication possible entre objets

Deux objets pourront communiquer si les deux classes correspondantes possèdent entre elles une liaison de type composition, association ou dépendance, la force et la durée de la liaison allant décroissant avec le type de liaison. La communication sera dans les deux premiers cas possible, quelle que soit l'activité entreprise par le premier objet, alors que dans le troisième cas elle se déroulera uniquement durant l'exécution des seules méthodes du premier objet, qui recevront de façon temporaire un référent du second.

Appel imbriqué de méthodes

On imbrique des appels de méthodes l'un dans l'autre quand l'approche procédurale se rappelle à notre bon souvenir. Force est de constater que l'OO ne se départ pas vraiment du procédural. L'intérieur de toutes les méthodes est, de fait, programmé en mode procédural comme une succession d'instructions classiques, assignation, boucle, condition… L'OO vous incite principalement à penser différemment la manière de répartir le travail entre les méthodes et la façon dont les méthodes s'associeront aux données qu'elles manipulent, mais ces manipulations restent entièrement de type procédural. Ces imbrications entre macrofonctions sont la base de la programmation procédurale, ici nous les retrouvons à une échelle réduite, car les fonctions auront préalablement été redistribuées entre les classes.

Passage par valeur ou par référent

En ce qui concerne le passage d'arguments de type prédéfini, le passage par valeur aura pour effet de passer une copie de la variable et laissera inchangée la variable de départ, alors que le passage par référent passera la variable originale, sur laquelle la méthode pourra effectuer ses manipulations.

Message = signature de méthode disponible

Le message se limite uniquement à la signature de la méthode : le type de ce qu'elle retourne, son nom et ses arguments. En aucun cas, l'expéditeur n'a besoin, lors de l'écriture de son appel, de connaître son implémentation ou son corps d'instructions. Cela simplifie la conception et stabilise l'évolution du programme.

Interface

La liste des messages disponibles dans une classe sera appelée l'interface de cette classe.

Encapsulation

L'encapsulation est ce mécanisme syntaxique qui consiste à déclarer comme `private` une large partie des caractéristiques de la classe, tous les attributs et de nombreuses méthodes.

Intégrité des objets

Une première raison justifiant l'encapsulation des attributs dans la classe est d'obliger cette dernière, par l'intermédiaire de ses méthodes, à se charger de préserver l'intégrité de tous ses objets.

La gestion d'exception

Toujours dans la perspective de sécuriser au maximum l'exécution des codes, tous les langages OO que nous présentons intègrent dans leur syntaxe un mécanisme de gestion d'exception dont la pratique est très voisine. Seul change le recours obligatoire ou non à cette gestion, Java étant le plus contraignant en la matière. En Java, la non-prise en compte de l'exception sera signalée et interdite par le compilateur. Une exception est levée quand quelque chose d'imprévu se passe dans le programme. Il est possible alors « d'attraper (try-catch) » cette exception et de prendre une mesure correctrice qui permette de continuer le programme malgré cet événement inat-tendu. Paradoxalement, toute la gestion d'exception consiste à rendre l'inattendu plus attendu qu'il n'y paraît, et de se préparer au maximum à toutes les éventualités problématiques ainsi qu'à la manière de les affronter. À l'instar de Java, il apparaît donc assez cohérent de forcer le programmeur à en faire usage.

Stabilisation des développements

Il y a une seconde raison de déclarer les attributs comme `private`, commune aux méthodes : c'est d'éviter que tout changement dans le typage ou le stockage de ceux-ci ne se répercute sur les autres classes.

Méthode *private*

En plus des attributs, une bonne partie des méthodes d'une classe doit être déclarée comme `private`.

Interaction avec l'interface plutôt qu'avec la classe

Lorsqu'une classe interagit avec une autre, il est plus correct de dire qu'elle interagit avec l'interface de cette dernière. Une bonne pratique de l'OO vous incite, par ailleurs, à rendre tout cela plus clair, par l'utilisation explicite des interfaces comme médiateurs entre les classes.

Désolidariser les modules

Alors que les deux niveaux extrêmes de l'encapsulation – « private » : fermé à tous et « public » : ouvert à tous – sont communs à tous les langages de programmation OO, ceux-ci se différencient beaucoup par le nombre et la nature des niveaux intermédiaires. De manière générale, cette pratique de l'encapsulation permet, tout à la fois, une meilleure modularisation et une plus grande stabilisation des codes, en désolidarisant autant que faire se peut les réalisations des différents modules.

Le garbage collector ou ramasse-miettes

Il s'agit de ce mécanisme puissant, existant dans Java, C#, Python et PHP 5, qui permet de libérer le programmeur du souci de la suppression explicite des objets encombrants. Il s'effectue au prix d'une exploration continue de la mémoire, simultanée à l'exécution du programme, à la recherche des compteurs de référents nuls (un compteur de référents existe pour chaque objet) et des structures relationnelles cycliques. Une manière classique de l'accélérer est de limiter son exploration aux objets les plus récemment créés. Ce ramasse-miettes est manipulable de l'intérieur du programme et peut être, de fait, appelé ou simplement désactivé (dans les trois langages, ce sont des méthodes envoyées sur « gc » qui s'en occupent). Les partisans du C++ mettent en avant le coût énorme en temps de calcul et en performance occasionné par le fonctionnement du « ramasse-miettes ». Mais à nouveau, lorsqu'il est question de comparer le C++ aux autres langages (notez que des librairies existent qui permettent de rajouter un mécanime de garbage collector au C++), la chose s'avère délicate car les forces et les faiblesses ne portent en rien sur les mêmes aspects. C++ est un langage puissant et rapide, mais uniquement pour ceux qui ont choisi d'en maîtriser toute la puissance et la vitesse. Mettez une Ferrari dans les mains d'un conducteur qui n'a d'autres besoins et petits plaisirs que des sièges confortables, une voiture large et silencieuse ainsi qu'une complète sécurité, vous n'en ferez pas un homme heureux. Mettez un appareil photo aussi sophistiqué qu'un Hasselblad dans les mains d'un amateur qui n'a d'autres priorités que de faire rapidement et sur-le-champ des photos assez spontanées de ses vacances et les photos seront toutes ratées.

Association entre classes

Il y a association entre deux classes, dirigée ou non, lorsqu'une des deux classes sert de type à un attribut de l'autre, et que dans le code de cette dernière apparaît un envoi de message vers la première. Sans cet envoi de message, point n'est besoin d'association. Plus simplement encore, on peut se représenter l'association comme un « tube à message » fonctionnant dans un sens ou dans les deux.

Composition

Bien que le lien d'agrégation et de composition servent à reproduire, tous deux, une relation de type « un tout et ses parties », le lien de composition rend, de surcroît, l'existence des objets tributaires de l'existence de ceux qui les contiennent. L'implantation de cette relation dans les langages de programmation dépendra de la manière très différente dont les langages de programmation gèrent l'occupation mémoire pendant l'exécution d'un programme. Nous retrouvons le besoin pour les programmeurs C++ de redoubler d'attention par l'utilisation du `delete`. Pour les programmeurs des autres langages, ils devront s'assurer que le seul référent de l'objet contenu est possédé par l'objet contenant.

Première conséquence de cette application de l'héritage

Il ne peut y avoir dans la sous-classe, par rapport à sa superclasse, que des caractéristiques additionnelles ou des précisions. Ce que l'héritage permet d'abord, c'est de rajouter dans la sous-classe de nouveaux attributs et de nouvelles méthodes, les seuls à préciser dans la déclaration des sous-classes.

Principe de substitution : un héritier peut représenter la famille

Partout où un objet, instance d'une superclasse apparaît, on peut, sans que cela pose le moindre problème, lui substituer un objet quelconque, instance d'une sous-classe. Tout message compris par un objet d'une superclasse le sera obligatoirement par tous les objets issus des sous-classes. L'inverse est évidemment faux. Toute Ferrari peut se comporter comme une voiture, tout portable comme un ordinateur et tout livre d'informatique OO comme un livre. Vous pouvez le jeter par la fenêtre comme tout livre en effet. C'est pour cette simple raison qu'il sera toujours possible de typer statiquement un objet par une superclasse bien que, lors de sa création et de son utilisation, il sera plus précisément instance d'une sous-classe de celle-ci, par exemple « `SuperClasse unObjet = new SousClasse()` » (le compilateur ne bronche pas, même si l'objet typé superclasse sera finalement créé comme instance de la sous-classe ; c'est aussi la raison pour laquelle vous devez écrire deux fois le nom de la classe dans l'instruction de création d'objet) ou encore :

```
SuperClasse unObjet = new SuperClasse() ;
SousClasse unObjetSpecifique = new SousClasse() ;
unObjet = unObjetSpecifique ; // l'inverse serait refusé par le compilateur
```

Tout message autorisé par le compilateur lorsqu'il est censé s'exécuter sur un objet d'une superclasse peut tout autant s'exécuter sur un objet de toutes ses sous-classes. La Ferrari peut prendre des passagers ou simplement démarrer. L'inverse n'est pas vrai. Demandez à une voiture quelconque (une Mazda, par exemple) d'atteindre 300 km/h dans les 5 secondes et à un livre quelconque (la Bible, par exemple) de vous révéler les subtiles secrets de l'héritage en OO.

« Casting explicite » versus « casting implicite »

Comme nous le verrons dans le chapitre suivant, le « casting » permet à une variable typée d'une certaine façon lors de sa création d'adopter un autre type le temps d'une manœuvre. Conscient des risques que nous prenons en recourant à cette pratique, le compilateur l'accepte si nous recourons à un « casting explicite ». C'est le cas en programmation classique quand on veut, par exemple, assigner une variable réelle à une variable entière. Dans le cas du principe de substitution, les informaticiens parlent souvent d'un « casting implicite », signifiant par là, qu'il n'y a pas lieu de contrer l'opposition du compilateur quand nous faisons passer un objet d'une sous-classe pour un objet d'une superclasse. Le compilateur (gardien du temple de la bonne syntaxe et de la bonne pratique) ne bronchera pas, car cette démarche est tout à fait naturelle et acceptée d'office. En revanche, faire passer un objet d'une superclasse pour celui d'une sous-classe requiert la présence d'un « casting explicite », par lequel vous prenez l'entière responsabilité de ce comportement risqué. Le compilateur vous met en garde (vous êtes en effet sur le point de commettre une bêtise) mais, ensuite, vous en faites ce que vous voulez en votre âme et conscience. Par exemple, si vous transférez un réel dans un entier (la superclasse dans la sous-classe), c'est en effet une bêtise, car vous perdez la partie décimale. En général, vous en êtes pleinement conscients et assumez la responsabilité de cette perte d'information. Les puristes de l'informatique détestent la pratique du casting (explicite évidemment) qui est toujous évitable par un typage plus fin et une écriture de code plus soignée. Un mauvais casting sera responsable d'une erreur à l'exécution du code lorsque l'on assigne à une classe un type qui ne lui correspond pas, et ce alors que le compilateur a laissé faire.

La place de l'héritage

L'héritage trouve parfaitement sa place dans une démarche logicielle dont le souci principal devient la clarté d'écriture, la réutilisation de l'existant, la fidélité au réel, et une maintenance de code qui n'est pas mise à mal par des évolutions continues, dues notamment à l'addition de nouvelles classes.

Messages et nivaux d'abstraction

L'héritage permet à une classe, communiquant avec une série d'autres classes, de leur parler à différents niveaux d'abstraction, sans qu'il soit toujours besoin de connaître leur nature ultime (on retrouvera ce point essentiel dans l'explication du polymorphisme), ce qui facilite l'écriture, la gestion et la stabilisation du code.

Protected

Un attribut ou une méthode déclaré `protected` dans une classe devient accessible dans toutes les sous-classes. La charte de la bonne programmation OO déconseille l'utilisation de « protected ». D'ailleurs, Python ne possède pas ce niveau d'encapsulation.

Héritage et constructeur

La sous-classe confiera au constructeur de la superclasse (qu'elle appellera par l'entremise de `super()` en Java et `base` en C# et `parent` en PHP 5) le soin d'initialiser les attributs qu'elle hérite de celle-ci. C'est une excellente pratique de programmation OO que de confier explicitement au constructeur de la superclasse le soin de l'initialisation des attributs de cette dernière. D'ailleurs, si vous ne le faites pas, Java, C# et C++ le font par défaut, en appelant implicitement un constructeur sans argument.

La redéfinition des méthodes

Une méthode redéfinie dans une sous-classe possédera la même signature que celle définie dans la superclasse avec, comme unique différence possible, un mode d'accès moins restrictif. En général, une méthode définie `protected` ou `public` dans la superclasse sera redéfinie comme `protected` ou `public` dans la sous-classe.

Polymorphisme et casting

Une conséquence du polymorphisme est le recours au « casting » qui vise à récupérer des fonctionnalités propres à l'une ou l'autre sous-classe lors de l'exécution d'un programme. Sa pratique est parfois délicate et demande une attention soutenue, car elle peut mener à des erreurs pendant la phase d'exécution. Java et C#, par exemple, par l'introduction de la généricité dans leurs dernières versions, tentent de diminuer ce recours. L'absence de typage statique dans Python et PHP 5 est bien évidemment une manière de contourner cette problématique, bien que cette absence ne les mette pas non plus à l'abri d'avatars ne survenant malheureusement qu'à l'exécution, lorsqu'on s'y attend le moins.

Polymorphisme contre case-switch

On dit souvent que toute programmation qui exige qu'un objet, à la réception d'un message, teste sa nature intime par l'intermédiaire d'un `case-switch` ou d'une succession de `if-then` afin de savoir quelle methode exécuter, est une partie de code idéale pour réaliser un « polymorphisme ». Lorsque nous donnons cours de programmation OO, nous ne testons pas, au préalable les prérequis de chacun de nos élèves, de façon à adopter le cours pour chaucun. De même, chacun d'entre eux, à la réception du message que nous leur délivrons, ne s'interroge pas sur ses capacités propres afin de savoir comment digérer la matière. Une programmation polymorphique vous permet en effet d'éviter cette succesion de tests. Chaque sous-classe d'étudiant (n'y voyez rien de péjoratif) recevra ce cours à la manière de sa sous-classe. Un programme conçu de telle sorte évitera évidemment les réécritures qu'une série de tests ou un `case-switch` entraînerait suite au rajout d'une sous-classe.

Classe abstraite

Une classe abstraite en Java, C##, PHP 5 et C++ (dans ce cas, en présence d'une méthode virtuelle pure) ne peut donner naissance à des objets. Elle a comme unique rôle de factoriser des méthodes et des attributs communs aux sous-classes. Si une méthode est abstraite dans cette classe, il sera indispensable de redéfinir cette méthode dans les sous-classes, sauf à maintenir l'abstraction pour les sous-classes et à opérer la concrétisation quelques niveaux en dessous.

Traitement en surface et en profondeur

Les objets se référant mutuellement en mémoire, et constituant ainsi un graphe connecté dans cette même mémoire, il sera important, dans tout traitement qu'ils subissent, de penser à prolonger ces traitements le long du graphe ou pas, différenciant ainsi un traitement en surface d'un traitement en profondeur.

Le multithreading, une réplique à moindre échelle du multitâche

Le multitâche vous permet d'utiliser différentes applications, Word, Firefox, votre jeu favori, votre système de courrier électronique… avec la même apparence de simultanéité que la proie et le prédateur s'abreuvant de concert. Le multitâche et sa version réduite, le multithreading, s'exécutent de la même façon. Cela se déroule de la manière suivante : le gestionnaire entame une première mini-tâche (thread), en exécute quelques instructions, puis est interrompu pour donner la main à une deuxième mini-tâche. Avant de passer la main, il mémorise tout ce qui lui est nécessaire pour pouvoir reprendre l'exécution de la première mini-tâche, à peine délaissée, par exemple, l'adresse de l'instruction suivante à exécuter ou les adresses de fichiers et autres périphériques avec lesquels cette mini-tâche interagissait. En substance, il mémorise tout le contexte d'exécution de cette mini-tâche. Après avoir donné un peu de son temps à toutes les mini-tâches qui s'exécutent à la queue leu leu et pourtant parallèlement, il revient à la première, en commençant, avant toute chose, par restituer son contexte d'exécution.

Traitement en surface et en profondeur

Les objets se référant mutuellement en mémoire, et constituant ainsi un graphe connecté dans cette même mémoire, il sera important, dans tout traitement qu'ils subissent, de penser à prolonger ces traitements le long du graphe ou pas, différenciant ainsi un traitement en surface d'un traitement en profondeur.

Le multithreading, une réplique à moindre échelle du multitâche

Le multitâche vous permet d'utiliser différentes applications, Word, Firefox, votre jeu favori, votre système de courrier électronique… avec la même apparence de simultanéité que la proie et le prédateur s'abreuvant de concert. Le multitâche et sa version réduite, le multithreading, s'exécutent de la même façon. Cela se déroule de la manière suivante : le gestionnaire entame une première mini-tâche (thread), en exécute quelques instructions, puis est interrompu pour donner la main à une deuxième mini-tâche. Avant de passer la main, il mémorise tout ce qui lui est nécessaire pour pouvoir reprendre l'exécution de la première mini-tâche, à peine délaissée, par exemple, l'adresse de l'instruction suivante à exécuter ou les adresses de fichiers et autres périphériques avec lesquels cette mini-tâche interagissait. En substance, il mémorise tout le contexte d'exécution de cette mini-tâche. Après avoir donné un peu de son temps à toutes les mini-tâches qui s'exécutent à la queue leu leu et pourtant parallèlement, il revient à la première, en commençant, avant toute chose, par restituer son contexte d'exécution.

Implémentation d'interfaces

Quand, en Java, C# et PHP 5, une classe implémente ou hérite d'une interface, elle est contrainte et forcée de concrétiser les méthodes qu'elle hérite de celle-ci. Ne pas le faire génère une erreur lors de la compilation ou directement à l'exécution pour PHP 5. L'interface oblige à utiliser et à redéfinir ses méthodes. En Java, c'est par le biais des interfaces que sont implémentées les fonctionnalités GUI et multithread. Nous verrons que C# et Python procèdent différemment.

Généricité

L'esprit de la généricité est partagé par tous les langages OO. Il s'agit d'autoriser le codage de fonctionnalités universelles, applicables sur tout type d'objet, et de pouvoir très facilement récupérer ces fonctionnalités pour un type donné cette fois. C++ l'a réalisé par un jeu d'écriture, qui permet au compilateur de particulariser un code générique à un type particulier. Lors de l'exécution, la fonctionnalité ne sera plus générique mais bien instanciée pour le type en question. En revanche, Java et C# recouraient jusqu'à présent à la classe `Object`, pour permettre le codage de cette fonctionnalité générique. Lors de l'exécution, il fallait s'assurer que le type particulier soit celui dans lequel les objets ont été « castés ». L'approche est plus intuitive et plus simple à mettre en œuvre, mais plus vulnérable en raison de la présence de « casting ». Ces deux langages proposent actuellement un mécanisme de généricité très proche de l'esprit du C++ qui résoud une fois pour toutes les problèmes de « casting » intempestif (ou, en son absence d'apparition de l'exception invalide `cast`) et permet d'éviter toutes les erreurs d'exécution qui peuvent en découler.