Avant propos

Les langages de programmations permettent aux informaticiens de prendre quelques distances par rapport au fonctionnement intime des processeurs (en ne travaillant plus directement à partir des adresses mémoire et en évitant la manipulation directe d'instructions élémentaires). Ces langages ont permis d'élaborer une écriture de programmes plus proche de la manière naturelle de poser et de résoudre les problèmes puisque les codes écrits dans ces langages deviennent indépendants des instructions élémentaires propres à chaque type de processeur. Une opération de traduction automatique, dite de compilation, se charge alors de traduire le programme, écrit au départ dans un langage de programmation, dans les instructions élémentaires, seules comprises par le processeur.

La nécessaire distanciation par rapport au fonctionnement du processeur, la volonté de rapprocher la programmation du mode cognitif de résolution des problèmes, les percées de l'intelligence artificielle et de la bio-informatique, conduisirent graduellement à un deuxième style de langage de programmation : *les langages orientés objets*.

L'orienté objet en deux mots

Un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets collaborant par envoi de messages. Le tout s'opère en respectant un principe de distribution des responsabilités on ne peut plus simple, chaque objet s'occupant de ses propres attributs afin de permettre une plus grande stabilité des logiciels en développement.
L'*encapsulation* limite l'impact des modifications d'un code aux seuls objets que ce code concerne, et à aucun autre de ses collaborateurs.

L'orienté objet est en réalité un processus de découpe naturelle et intuitive en des parties plus simples puisqu'elle s'inspire de notre manière cognitive à nous, êtres humains, de découper la réalité qui nous entoure. Dans un gros programme, l'OO permet de tracer les pointillés que les ciseaux doivent suivre là où il semble le plus naturel de les tracer (contours du programme). L'OO ne permet en rien de faire l'économie du procédural, simplement, il complémente celui-ci, en lui superposant un système de découpe plus naturel et facile à mettre en œuvre.

L'OO a permis de repenser trois des chapitres les plus importants de l'informatique de ces deux dernières décennies :

- Le besoin de modélisation graphique (niveau d'abstraction supplémentaire) ;
- Les applications informatiques distribuées (on parle maintenant d'objets distribués) ;
- Le stockage des données qui doit maintenant compter sur les objets.

Chapitre 1 : Principes de base – quel objet pour l'informatique ?

Le trio « entité, attribut, valeur »

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

Stockage des objets en mémoire

Les bases de données relationnelles sont le mode de stockage de données le plus répandu en informatique. La disparition des clés primaires dans la pratique OO fait de la sauvegarde des objets dans les tables un problème épineux de l'informatique d'aujourd'hui.

Le référent d'un objet

Le nom de l'objet informatique, ce qui le rend unique, est également ce qui permet d'y accéder physiquement. Nous appellerons désormais ce nom le *référant 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.

Le référent contient l'adresse physique de l'objet, codée sur 32 bits dans la plupart des ordinateurs aujourd'hui. Le nom d'espaces mémoires disponibles est lié à la taille de l'adresse de façon exponentielle. Sans référent, l'objet situé en mémoire serait inaccessible.
Plusieurs référents peuvent désigner en mémoire un seul et même objet, s'il est nécessaire d'accéder à l'objet dans des contextes d'utilisation différents et qui s'ignorent mutuellement. Cela est faisable en informatique OO, grâce au mécanisme d'adressage indirect qui permet d'offrir plusieurs voies d'accès à un même objet mémoire.

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, désignant ainsi le même objet.

L'objet dans sa composition passive

L'OO, pour des raisons pratiques, encourage à séparer, dans la description de tout objet, la partie utile pour tous les autres objets qui y recourant de la partie nécessaire à son fonctionnement propre. Il faut séparer physiquement ce que les autres objets doivent savoir d'un objet donné, afin de solliciter ses services, et ce que ce dernier requiert pour son fonctionnement, càd la mise en œuvre de ces mêmes services.

Entre eux, les objets peuvent entrer dans une relation de type composition, où certains objets se trouvent contenus dans d'autres et ne sont accessibles qu'à partir de ces autres. Leur existence dépend entièrement de celle des objets qui les contiennent.

L'objet dans sa version active

Les objets changent donc d'état, continûment, mais tout en préservant leur identité, en restant ces mêmes objets qu'ils ont toujours été. Les objets sont dynamiques, le changement d'un attribut n'affecte en rien l'adresse de l'objet, et donc son identité. Le cycle de vie d'un objet, lors de l'exécution d'un programme OO, se limite à une succession de changement d'états, jusqu'à sa disparition pure et simple de la mémoire centrale.

Introduction à la notion de classes

Une nouvelle structure de données voit le jour en OO : la classe, qui, de fait, a pour principale raison d'être d'unir en son sein tous les attributs de l'objet et toutes les opérations qui y accèdent et qui portent sur ceux-là. Opération que l'on désignera par le nom de méthode, et qui regroupe un ensemble d'instructions portant sur les attributs de l'objet. Le type primitif entier est souvent appelé dans les langages de programmation « int », le type réel double ou float, le caractère « char ».
On lie la méthode f(x) à l'objet « a », sur lequel elle doit s'appliquer, au moyen d'une instruction comme : a.f(x) où le « . » est le lien entre l'objet précis et la méthode à exécuter sur cet objet.

Différencier langage orienté objet et langage manipulant des objets
De nombreux langages de programmation, surtout de scripts, rendent possible l'exécution de méthodes sur des objets dont les classes préexistent au développement. Le programmeur ne crée jamais de nouvelles classes mais se contente d'exécuter les méthodes de celle-ci sur des objets. Ainsi, on se contente de manipuler des objets sans jamais avoir créé de classe, préalablement.

Des objets en interaction

La raison d'être des parenthèses, qui même lorsqu'elles ne continent rien, sont obligatoires puisqu'elles peuvent recevoir des arguments afin de paramétrer le fonctionnement des méthodes.

La méthode est un regroupement d'instructions semblable aux procédures, fonctions et routines dans tous les langages de programmation, à ceci près qu'une méthode s'exécute sur un objet précis (comme si celui-ci lui était, implicitement, passé comme un argument additionnel).

Envoi de message

Le seul mode de communication entre deux objets revient à la possibilité pour le premier de déclencher une méthode sur le second, méthode déclarée et définie dans la classe de celui-ci. On appellera ce mécanisme de communication un *« envoie de message »* du premier objet vers le second. On conservera cette même dénomination y compris pour des objets s'exécutant sur des processeurs très éloignés géographiquement, situation entraînant réellement un envoi physique de message d'un processeur à l'autre.

Héritage et taxonomie

Une pratique clé de l'OO est d'organiser les classes entre elles de manière hiérarchique ou taxonomique, des plus générales aux plus spécifiques. On parlera d'un mécanisme « d'héritage » entre les classes. Un objet, instance d'une classe, sera à la fois instance de cette classe mais également de toutes celles qui la généralisent et dont elle hérite. Tout autre objet ayant besoin de ses services choisira de le traiter selon le niveau hiérarchique le plus approprié.

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 classe offre des avantages capitaux. On peut omettre d'écrire dans la définition de toutes les sous-classes ce qu'elles héritent des classes. Il est de bon sens que, moins on écrit d'instructions, plus fiable et plus facile à maintenir sera le code. Si on apprend d'une classe quelconque qu'elle est un cas particulier d'une classe générale, on peut lui associer automatiquement toutes les informations caractérisant la classe la plus générale et ce, sans les redéfinir.

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 de la nature des objets qui le reçoivent.

Chapitre 2 : un objet sans classe … n'as pas de classe

Constitution d'une classe d'objets

Toute définition de classe induit trois informations : d'abord, le nom de la classe, ensuite ses attributs et leur type, enfin ses méthodes. Dans les langages OO, à l'instar des langages procéduraux, on fait la distinction entre une fonction (déclarée avec retour comme tout fonction mathématique f(x) en général) et une procédure (déclarée sans retour et qui se borne à modifier des aspects du code sans que cette action soit intégrée à l'intérieur même d'une instruction).
La présence des *arguments* (entre parenthèses) dans la définition de la méthode permet de moduler le comportement du corps d'instructions de cette méthode selon la valeur prises par ses arguments. C'est l'équivalent du x dans les fonctions mathématiques f(x).

Identification et surcharge des méthodes par leur signature

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.

La *signature de la méthode* est ce qui permet de la retrouver dans la a mémoire des méthodes. Elle est constituée du nom, de la liste, ainsi que du type d'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. A la différence des autres méthodes, 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, on a « 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.

La classe comme garante de son bon usage

Comme le *compilateur* a pour fonction critique de générer un code « exécutable », et que son utilisateur exige le moins d'inattendu possible, il prendra garde de vérifier que rien de ce qui est écrit par le programmateur ne puisse être source d'imprévu et d'erreur. La classe disparaît lors de l'exécution pour donner place aux objets, tout en s'étant assurée par avance que tout ce que feraient les objets est conforme.

Ainsi, un langage de programmation est dit fortement « typé » lorsque 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.

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 déclarées 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.

Liaison naturelle et dynamique de développement

La *classe*, par le fait qu'elle s'assimile à un petit programme à part entière, constitue un module idéal pour le découpage du logiciel en ses différents fichiers. La liaison sémantique entre les classes, rendue possible si la première intègre en son code un appel à la seconde, devrait suffire à relier de façon dynamique, pendant la compilation et l'exécution du code, les fichiers dans lesquels ces classes sont écrites. C'est principalement dans Java que cette logique de découpe et d'organisation sémantique du code en ses classes isomorphes à la découpe et l'organisation physique en fichiers sont le plus scrupuleusement forcées par la syntaxe. C'est un très bon point en faveur du Java.

Chapitre 3 : Du faire savoir au savoir faire … du procédural à l'OO

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 ou la programmation OO est confiée à un grand nombre d'objets, se passant successivement le relais pour l'exécution des seules fonctions qui leur sont affectées.

Conception procédurale vs conception objet

La conception procédurale découpe l'analyse du problème en de grandes fonctions, imbriquées et portant toutes sur l'essentiel des données du problème. En procédural, les grandes fonctions accomplies par le logiciel, en s'imbriquant l'une dans l'autre, sont la voie de la modularité et de l'évolution de toute l'approche.

Or, on perçoit aisément qu'il est plus facile de parvenir à une décomposition naturelle du problème en des modules relativement indépendants. Dans la pratique OO, on cherche d'abord à identifier les acteurs du problème et à les transformer en classe, regroupant leurs caractéristiques structurelles et comportementales. Ce ne sont plus les grandes fonctions qui guident la construction modulaire du logiciel mais les acteurs/classes eux-mêmes.

La différence principale se situe au niveau du fait qu'en OO, le découpage logiciel s'effectue à partir des grands acteurs de la situation, et non plus des grandes activités propres à la situation. Cela permet donc un découpage plus naturel et cela facilite également la maintenance et la mise à jour du logiciel.

Dépendance fonctionnelle vs. dépendance logicielle
Alors que l'exécution d'un programme OO repose pour l'essentiel sur un jeu d'interaction entre classes dépendantes, tout est syntaxiquement mis en œuvre lors du développement logiciel pour maintenir une grande indépendance entre les classes. Cette indépendance au cours du développement favorise tant la répartition des taches entre les programmateurs que la stabilité des codes durant leur maintenance et leur évolution.

Il est parfaitement incorrect de clamer haut et fort que les performances en consommation des ressources informatiques des programmes OO sont supérieures en générale à celles de programmes procéduraux remplissant les mêmes taches. Tout concourt à faire des programmes OO de grands consommateurs de mémoire (prolifération des objets) et de temps de calcul (retrouver les méthodes et les objets en mémoire et ensuite les traduire en procédural). Les seuls temps et ressources réellement épargnés sont ceux des programmateurs.

Chapitre 4 : les objets parlent aux objets

Association de classes

La manière privilégiée pour permettre à deux classes de communiquer par envoie 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 se mécanisme de typage particulier que, partout dans son code, la première classe pourra faire appel aux méthodes disponibles dans 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.

Réaction en chaîne

Tout processus d'exécution OO consiste essentiellement en une succession d'envois de messages en cascade, d'objets en objet, messages qui, selon le degré de parallélisme mis en œuvre, seront plus ou moins imbriqués.

Chapitre 5 : collaboration entre classes

Une classe, un fichier

Dans sa pratique, et bien plus que les trois autres langages, Java force la séparation des classes en fichier distincts. Si on ne le fait pas lors de l'écriture des sources, Java le fera pour nous, comme résultat de la compilation de ces sources. En conséquence de quoi, autant le précéder, par une écriture des classes séparée en fichiers. Cette pratique tend à se généraliser à tous les développements OO.

Fichiers .dll

Les fichiers portant l'extension « .dll » sont des fichiers caractéristiques des plates-formes Windows et qui permettent de relier dynamiquement plusieurs fichiers exécutables.

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 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 microfonctions 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.

« import » en Java

Les classes étant regroupées en paquetage imbriqués, il est indispensable, lors de leur utilisation, soit de spécifier leur nom complet : « paquatage.classe » (d'abord le nom du paquetage, ensuite le nom de la classe), soit d'indiquer, au début du code, le paquetage qui doit être utilisé afin de retrouver les classes exploitées dans le fichier. Cela se fait par l'addition, au début des codes, de l'instruction « import ».

Chapitre 6 : méthodes ou messages ?

Passage par valeur ou 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.

Passage par référent

La programmation OO favorise dans sa pratique le passage des arguments objets comme référent plutôt que comme valeur. Les langages comme Java en ont d'ailleurs fait leur mode légitime de fonctionnement.

Une méthode est –elle d'office un message ?

Pas vraiment, et ce pour plusieurs raisons. Le message ramène la méthode à sa seule signature, en réalité, à la signature de sa 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. La liste des messages disponibles dans une classe sera appelée *l'interface de la classe*.

Chapitre 7 : l'encapsulation des attributs

Attribut public et private

Un attribut ou une méthode sera private, si l'on souhaite restreindre son accès à la seule classe dans laquelle il est déclaré. Il sera public si son accès est possible, par ou dans toute autre classe.

Encapsulation : pourquoi faire ?

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.

- *pour préserver l'intégrité des objets* : les méthodes de la classe filtrent l'usage que l'on fait des attributs de la classe. Ainsi, la classe est obliger de préserver l'intégrité de ses objets, par l'intermédiaire de ses méthodes.
- *la gestion d'exception* : une exception est levée quand quelque chose d'anormal se passe dans le programme. Il est alors possible d'attraper cette exception et de prendre une mesure correctrice qui permette de continuer le programme malgré cet événement inattendu. Paradoxalement, la gestion d'exception permet de rendre l'inattendu plus attendu qu'il n'y parait …

Stabilisation des développements

Il y a une seconde raison de déclarer les attributs comme private, comme 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.

Chapitre 8 : les classes et leur jardin secret

Encapsulation des méthodes

En plus des attributs, une bonne partie des méthodes d'une classe doit être déclarée comme private. On différencie les méthodes private des méthodes public en déclarant que les premières sont responsables de l'implémentation de la classe, alors que les secondes le sont de l'interface de la classe.
L'interface ne reprend de toutes les méthodes de la classe, que les seuls possibles messages, càd les signatures des méthodes publiques. Ce sont ces seules signatures qui ne peuvent évoluer dans le temps, car même corps des méthodes identifiées par ces signatures peut être modifié, sans conséquence sur les autres classes.

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 incite, par ailleurs, à rendre tout cela plus clair, par l'utilisation explicite des interfaces.

Désolidarisation des 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 d'encapsulation. 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.

Chapitre 9 : vie et mort des objets

L'OO coûte cher en mémoire

L'existence de pratique informatique comme l'OO permettent de s'affranchir des soucis d'optimisation de place dans la RAM puisque la pratique OO ne regarde pas trop à la dépense.

Bien que la pratique de l'OO soit une grande consommatrice de mémoire RAM, et que celle-ci va s'accroissant dans les ordinateurs suivant la loi de Moore (doublement de capacité tous les 16 mois), elle reste une ressource extrêmement précieuse.

Gestion de la mémoire pile

Ce système de gestion de la mémoire s'est donc extrêmement ingénieux, car il est fondamentalement économe, gère de façon adéquate le temps de vie des variables intermédiaires par leur participation dans des fonctionnalités précises, garde rassemblées les variables qui agissent de concert, et synchronise le mécanisme d'empilement et de dépilement des variables avec l'emboîtement des méthodes.

La vie des objets indépendante de ce qu'ils font

L'OO, se détachant d'une vision procédurale de la programmation, tend à rendre indépendante la gestion mémoire occupée par les objets de leur participation dans l'une ou l'autre opération. Cette nouvelle gestion mémoire résultera d'un suivi des différentes transformations subies par l'objet et sera, soit laissée à la responsabilité du programmeur, soit automatisée.

Le garbage collector (« ramasse-miette »)

Mécanisme puissant, existant dans certains langages de programmation OO, 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 compte 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 garbage collector est manipulable de l'intérieur du programme et peut être, de fait, appelé ou simplement désactivé.

Chapitre 10 : UML 2 (Unified Modeling Langague)

UML 2

UML 2 permet, au moyen de ses 13 diagrammes, de représenter le cahier des charges du projet, les classes et la manière dont elles s'agencent entre elles. Il permet, finalement, d'organiser les fichiers qui constituent le projet, ainsi que de penser leur stockage et leur exécution dans les processeurs.

UML 2 entre le coup de pouce au tableau noir et un vrai langage de programmation

Alors que la plupart des utilisateurs d'UML le voient aujourd'hui comme un complément et une assistance graphique à l'écriture du code, ses créateurs et ses avocats espèrent le voir évoluer vers un véritable langage de programmation, capable d'universaliser et de chapeauter tous ceux qui existent aujourd'hui.

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 ».

Chapitre 11 : l'héritage

Première conséquence de l'application de l'héritage

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

Principe de substitution : un héritier peut représenter la famille

Partout où un objet, instance d'une superclasse apparaît, 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 également par un objet issu des sous-classes. L'inverse est évidemment faux.

Casting explicite vs. Casting implicite

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. Si le compilateur l'accepte, nous recourons à un « casting explicite ». 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 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 on prend l'entière responsabilité de ce comportement risqué.

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 pour des évolutions continues, dues notamment à l'addition de nouvelles classes.

Messages et niveaux d'abstraction

L'héritage permet à une classe, communiquant avec une série d'autre 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.

Encapsulation protected

Un attribut ou une méthode déclarée « protected » dans une classe devient accessible dans toutes les sous-classes. La charte de la bonne programmation OO déconseille l'utilisation de « protected ».

Héritage et constructeur

La sous-classe confiera au constructeur de la superclasse (qu'elle appellera par l'entremise de *super()*) 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.

Chapitre 12 : redéfinition des méthodes

La base du polymorphisme

L'héritage offre la possibilité pour une classe de s'adresser à une autre, en sollicitant de sa part un service qu'elle est capable d'exécuter de mille manières différentes, selon que ce service, nommé toujours d'une seule et même façon, se trouve redéfinit dans autant de sous-classes, toutes héritant du destinataire du message. C'est la base du polymorphisme. Cela permet à notre première classe de traiter toutes ses classes interlocutrices comme une seule, et de ne modifier en rien son comportement si on ajoute une de celles-ci ou si une de celles-ci modifie sa manière de répondre aux messages de la première.

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.

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.

Chapitre 13 : abstraite, cette classe est sans objet

Classe abstraite

Une classe abstraite en Java 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.