Table des matières

0.1 Introduction

Objective C est une extension du langage C. Définie pour donner au langage C les possibilités de la programmation Objet et ce de façon simple et directe. Les ajouts au C sont peu nombreux et basés sur Smalltalk, un des premier langages objets.

La syntaxe de Objective C est un sur ensemble de la syntaxe du C standard, le compilateur est le même pour des sources en C et en Objective C. Le compilateur (gcc) reconnaît les sources Objective C par l'extension ".m" comme il reconnaît les fichiers C à leur extension ".c". Objective C peut aussi être utilisé comme une extension de C++. Ceci peut sembler un peu superflu comme C++ est déjà orienté objets, mais C++ est un "meilleur C" et pas nécessairement un véritable langage objet. Il manque en effet au C++ le typage dynamique et les liens dynamiques qu'offre Objective C. Il a par contre des possibilités que n'offre pas Objective C. Un mélange des deux peut permettre d'utiliser au mieux les caractéristiques des deux langages.

Comme les langages orientés objet retardent de nombreuses décisions de la compilation à l'exécution ils dépendent donc d'un noyau (run-time) pour exécuter le code compilé. Nous parlerons ici des points importants de ce noyau d'exécution car il sont nécessaire à une bonne compréhension des particularités du langage.

0.1.1 Objets

Les langages orientés objet sont essentiellement basé sur les objets. Un objet associe des données avec les opérations qui peuvent spécifiquement manipuler ces données. Ces opérations sont appelées les méthodes de l'objet. Les données qu'elles manipulent sont appelées les instances. Essentiellement les objets regroupent une structure de donnée (les variables d'instances) et un groupe de fonctions (les méthodes) en une unité de programmation autonome. Ainsi par exemple si vous chercher à modéliser la plomberie d'une maison, vous aurez peut être à créer un objet vanne ayant des méthodes permettant d'ouvrir ou fermer la vanne, de régler le débit d'eau, calculer la quantité d'eau consommée sur une certaine période, etc. Pour répondre à ces services, un objet Vanne aura besoin de variables d'instances, pour suivre des informations comme, le débit d'eau, le fait que la vanne est ouverte ou non, etc.

Un des objets défini, dans un kit graphique, peut afficher une matrice de cellules (Cell) à l'écran. Les cellules peuvent être indifféremment des champs de texte (texte fields), des boutons mutuellement exclusifs, une liste de boutons, un bloc de curseurs, etc. Un objet Matrix va contenir des variables d'instances décrivant, la taille, la position, la police utilisée pour afficher les cellules, que faire quand une cellule est sélectionnée, le nombre de ligne et de colones. La matrice a aussi des méthodes pour changer sa taille, sa position, ajouter ou retirer des cellules, choisir la couleur des interstices entre cellules, suivre le déplacement du curseur sur les différentes cellules, mettre en valeur la sélection de l'utilisateur, etc.

En ObjectiveC, les variables d'instances sont internes à l'objet, vous ne pouvez y accéder qu'a travers l'objet lui-même. Pour prendre connaissance d'une information associé à l'objet il faut qu'il existe dans l'objet une méthode qui fournisse cette information. Les méthodes recouvrent les variables d'instances de l'Objet et les cachent au reste du programme.

Figure 1. Un Objet

De plus un objet ne connaît que les méthodes qui lui ont été explicitement associées, il ne peut ainsi exécuter de méthodes définie sur d'autres objets. D'une façons similaire aux variables locales des fonctions C qui sont invisibles pour le reste du programme, les objets cachent à la fois leurs variables d'instances et l'implémentation de leur méthodes.

0.1.2 Réseaux d'Objets

Un programme consistent en un réseau d'objets interconnectés qui sont appelés pour résoudre les différents éléments du puzzle. Chaque objet a un rôle spécifique à jouer dans le dessin global du programme.

Figure 2. Réseau d'Objets

Pour reprendre notre modélisation de plomberie, le programme de modélisation de la consommation d'eau aura sans doute des objets Tuyaux pour mener l'eau d'une vanne à l'autre, un objet Batiment pour coordonner les différentes Vannes. Des objets Electroménager (lave vaisselle, lave linge, toilettes,...), capable d'ouvrir et fermer les Vannes. Et enfin des utilisateurs capables d'ouvrir des vannes et de mettre en service les machines. Quand on demande à un objet Batiment de fournir sa consommation courante d'eau il peut avoir à demander à chaque Vanne l'état dans lequel elle se trouve. Quand un utilisateur met en marche un machine, cette machine peut avoir à ouvrir une des vannes, pour avoir l'eau nécessaire.

Similairemment, Chaque objet cellule dans une Matrix est une instance d'une sous-classe de Cell (cellule) qui répond aux ordres de Matrix. Quand un objet Matrix reçoit l'ordre de s'afficher (display) sur l'écran, elle commence par envoyer l'ordre de s'afficher aux cellules qu'elle contient. Les cellules elle même peuvent avoir besoin de demander des informations à d'autres objets pour s'afficher.

Pour communiquer entre eux les objets ont besoin de se connaître. A cet effet dans des variables d'instance d'un objet seront stockées les adresses d'autres objets. Ces variables d'instances serviront de receveur pour les message que l'objet envoi. Ainsi une Vanne stockera les tuyaux auxquels elle est connectée. Une matrice conservera une liste de ces cellules. Par contre les cellules ne stockeront pas la matrice à laquelle elles appartiennent.

Ces interconnections définissent la structure de l'application. Elles relient les objets en un réseau de communication, comme des individus sont reliés par leur échanges sociaux, ou comme les composants d'une plomberie sont physiquement connectés. Les langages orientés objets sont bien adaptés pour la modélisation de composants d'un système complexe.

Le réseau d'objet est mis en route par un stimuli extérieur. Si vous écrivez une application interactive avec une interface utilisateur, elle répondra aux actions de l'utilisateur sur le clavier ou la souris. Un programme de décomposition en facteurs premiers débute quand vous lui fournissez un entier cible, sur la ligne de commande. D'autres programmes réagiront à des données reçues sur une ligne téléphonique, ou un autre périphérique.

0.1.3 Le type id

En objective C, les objets sont identifiés par un type distinct, id. Ce type est défini comme un pointeur sur un objet, en réalité, comme un pointeur sur la structure de données de l'objet (variables d'instance). Comme une fonction C ou un tableau, un objet est identifié par son adresse. Tous les objets, indépendamment de leurs variables d'instance ou de leurs méthodes sont de type id.

id unObjet;

Le type id est le type par défaut dans toutes les constructions objet, comme par exemple la valeur de retour d'une méthode. Par contre, pour les constructions purement C, comme la valeur de retour d'une fonction, le type int reste le type par défaut.

Le mot clef nil est défini comme un objet nul, c'est-à-dire, un id de valeur 0.

les constantes id, nil, et les autres types de base d'Objective C sont définis dans le fichier `<objc.h>` .

0.1.4 Dynamic Typing (typage dynamique)

Le type id est non restrictif il ne donne aucune information sur l'objet à part que c'est un objet.

Mais les objets ne sont pas tous identiques! A un moment donné le programme doit trouver plus d'informations sur l'objet qu'il utilise: quelle sont ses variables d'instances, quelles sont les méthodes qu'il peut exécuter et ainsi de suite. Comme le type id ne donne pas d'informations au compilateur, c'est l'objet qui devra les fournir à l'exécution. Ceci est possible car tout objet a une variable d'instance isa (est-un) qui identifie la classe de l'objet (le type de l'objet). Toute Vanne est capable de dire qu'elle est une Vanne, une matrice qu'elle est une matrice, etc.

Les objets sont donc dynamiquement typés à l'exécution, plutôt que typés statiquement à la compilation. Ce qui a de très grandes conséquence sur la conception orienté objets, et est aussi l'élément fondamentale des liaisons dynamiques, voir plus bas.

Pour les classes d'objets voir plus bas la section "Classes".

Note: Il est aussi possible de typer statiquement à la compilation un objet en spécifiant sa classe dans le source. Les classes sont des objets particuliers qui peuvent aussi servir de nom de types. Voir "types & classes" et "Options Statiques"

0.2 Messages

Pour qu'un objet fassent quelque chose il faut lui envoyer un message lui disant d'exécuter une méthode. La syntaxe des messages en Objective C est la suivante:

```
[ receveur message ]
```

Le receveur est un objet, et le message lui dit que faire. Cette forme syntaxique est une expression au sens du langage C, remarquer l'usage des crochet identique a celui de smalltalk. Dans du source, le message est simplement le nom d'une méthode et tous les arguments qui vont avec. Quand le message est envoyé, le noyau (d'exécution), sélectionne la méthode appropriée dans le répertoire de méthode du receveur et l'invoque. C'est pourquoi le nom de méthode dans le message est appelé sélecteur de méthode (selector).

L'exemple suivant demande à l'objet maMatrix d'exécuter sa méthode affiche, qui dessine la matrice dans une fenêtre.

```
[maMatrix affiche];
```

Une méthode peut avoir des arguments. Le message suivant dit à la matrice de changer ces coordonnée pour la position (30.0, 50.0) de la fenêtre.

`[maMatrix allerEn:30.0:50.0];`

Le nom de la méthode est allerEn:: remarquer les deux caractères ":" un pour chaque argument. Les arguments sont insérés après chaque ":" qui séparent le nom en morceaux. En effet les ":" n'ont pas besoin d'être tous placés en fin de mot comme il le sont ici. Habituellement un mot décrivant l'argument est placé avant chaque ":". La méthode getRow:andColumn:ofCell: prend trois arguments :

int row, column; [maMatrix getRow:&row andColumn:&column ofCell:someCell];

Cette méthode trouve la position de la cellule someCell dans la matrice et place dans les deux entiers donnés en paramètres la ligne et la colonne où la cellule se trouve.

Les méthodes à nombre variable d'arguments sont aussi possibles. Les arguments surnuméraires étant séparés par des virgules et placés en fin de message. (A l'inverse des ":" les virgules ne font pas partie du nom de la méthode). Dans l'exemple suivant une méthode imaginaire makeGroup: prend un argument obligatoire (group) et ici trois arguments optionnels :

[receveur makeGroup:group, memberOne, memberTwo, memberThree];

Comme les fonctions C les méthodes peuvent retourner des valeurs. l'exemple suivant affecte l'entier retourne par la méthode tag à une variable du même nom.

int tag; tag = [maMatrix tag];

Un message à l'objet nil est valide,

[nil moveTo:100.0:22.5];

mais n'a pas d'effet . Les messages envoyés à l'objet nil retourne simplement nil.

0.2.1 Les variables d'instance du receveur

Une méthode a automatiquement accès aux variables d'instances du receveur. Il n'est pas nécessaire de les passer en argument. Par exemple la méthode tag n'a pas d'argument mais est capable d'accéder à la variable d'instance du receveur. Chaque méthode suppose l'existence du receveur et de ces variables d'instance, sans avoir à les déclarer comme des arguments (ceci vient de ce que les méthodes sont déclarées sur les objets et non l'inverse comme dans les langages procéduraux).

Cette convention simplifie les sources écrit en Objective C. C'est ainsi que la programmation objet définie le rapport entre messages et objets. Les messages apportent de l'information extérieur au receveur, il n'ont pas besoin d'apporter le receveur à lui même. Une méthode à automatiquement et uniquement accès aux variables d'instance de l'objet receveur Pour connaître des informations contenues dans un autre objet, il faut lui envoyer un message lui demandant de révéler les informations demandées. La méthode tag précédente est écrite spécialement à cette intention.

Voir "Définir une Classe" pour plus informations sur l'accès aux variables d'instance.

0.2.2 Polymorphisme

Comme les exemples précédents le montre, les messages peuvent apparaître aux mêmes positions syntaxiques que les appels de fonctions en C. Mais comme les méthodes appartiennent à un objet un autre mécanisme d'appel est utilisé. En particulier, un objet n'a accès qu'aux méthodes définies pour lui. Il ne peut les confondre avec des méthodes appartenant à un autre objet, même si les méthodes ont le même nom. Ce qui veut dire que deux objets peuvent répondre différement au même message. Par exemple la méthode display (affichage) sera très différente en fonction du receveur, chaque objet ayant sa méthode propre pour s'afficher. Un Batiment et une Vanne répondrons différemment à la méthode QuantitéEauConsomée. Cette fonctionnalité, qui joue un rôle significatif dans la conception orientée objets, est couramment appelée polymorphisme. Le polymorphisme et l'éditions de liens dynamique vous permettent d'écrire du code pour toute une famille d'objets sans en connaître a priori le type. Ces objets pourrons même être des objets écrit plus tard par d'autres personnes. Si vous écrivez du code qui envois le message display à un objet, tout objet qui à une telle méthode est un receveur potentiel correct. (exemple: programmation de l'algorithme de Bezou sur des entiers, une fois que les opérateurs +,-,/,* sont définis sur les polynômes, l'algorithme testé et validé sur les entier peut être appliqué tel quel aux polynômes).

0.2.3 Édition de liens dynamique (Dynamic Binding)

Une différence cruciale entre appel de fonction et message est qu'une fonction et ces arguments sont reliés dans le code dés la compilation, alors qu'un message et le receveur ne le sont pas avant l'exécution, et l'envoi du message.

Ainsi, la méthode exacte qui sera invoqué en réponse à un message est déterminé à l'exécution, et non plus quand le code est compilé. La méthode a exécuter dépend du receveur, hors le type de celui-ci n'est connu qu'à la réception du message (typage dynamique), mais pas à la compilation. Ainsi différents receveur peuvent avoir différentes implémentations pour le même sélecteur (polymorphisme). L'information que le compilateur ne connaît pas est fournie à l'exécution par l'objet receveur

Quand un message est envoyé, une routine du noyau d'exécution cherche dans le receveur une méthode ayant le bon sélecteur (selector), défini dans le message, et "appel" la routine avec les arguments définis dans le message, (voir "la gestion des messages" plus bas).

L'édition de liens dynamique et le polymorphisme donne à la programmation objet sa souplesse et sa puissance. On obtient des résultats variés non plus en changeant le message mais en changeant le receveur.

Ceci pouvant être fait dynamiquement à l'exécution. Le receveur peut être choisi "à la volé" et ainsi dépendre d'actions extérieur comme les commandes utilisateur. Dans le générateur d'application NeXT, l'utilisateur va déterminé quel sont les objets qui vont recevoir les ordres copier, couper, coller du menu. Les différents objets réagirons différemment à ces ordres. Comme les méthodes de doivent pas être intégrée à la compilation, les méthodes qui utilisent ces ordres n'ont pas à se concerner de leur existence. De plus chaque application peut définir un comportement spécifique au message copier:.

0.3 Classes

Un programme objet est en général construit à partir d'une grande variété d'objet, de plus il est très fréquent que l'on utilise plusieurs exemplaire d'un même type d'objet, plusieurs cellules, fenêtres, listes, etc. En Objective C on définit les objets en définissant leur classes (class).La définition de la classe est un prototype pour un type d'objet. Dans cette définition sont déclarées les variables d'instance, qui appartiendront à chaque instance de la classe (les objets), sont aussi définie les méthodes que les objets de cette classe peuvent utiliser.

Le compilateur crée un objet de type Class pour chaque classe, cet objet de type Class appelé souvent "factory object" (objet usine) est le seul objet capable de créer des objets appartenant à la classe qu'il défini. Les objets qui travailleront pour votre application sont les objets qui sont instances de vos classes et qui seront fabriqués à l'exécution.

Toutes les instances de la classe (objet du même type) ont tous accès au même ensemble de méthodes et ont tous les mêmes variables d'instances. Les méthodes sont partagées, les variables d'instance sont privées à chaque objet.

0.3.1 Héritage

Les définitions de classes sont additives, chaque nouvelle classe que vous définissez est basée sur une autre classe, à travers laquelle elle hérite de méthodes et de variables d'instance. La nouvelle classe ne fait qu'ajouter ou transformer ce qu'elle hérite. Il n'est pas nécessaire de dupliquer le code hérité.

L'ensemble de ces liens d'héritage relie les classes dans un arbre généalogique où la classe Object est à la racine. Toute classe (sauf Objet) a une super-classe juste au dessus, et toute classe peut être la super-classe d'un nombre quelconque de sous-classes. Exemple Figure 3.

Figure 3. Classes du Kit Graphique

Ce schéma montre que Matrix est une sous-classe de Control, elle même sous-classe de View, View sous-classe de Responder, et Responder sous-classe d'Object. L'héritage est cumulatif. Ainsi Matrix a des méthodes et des variables d'instances définies dans Control, View, Responder, et Object. Ainsi que ceux spécialement défini pour Matrix. Tous cela pour dire qu'un objet Matrix est aussi un objet Control,View, Responder, et un objet Object.

Toute classe (sauf Object) peut être vue comme un spécialisation ou adaptation d'une autre classe. Les sous-classes successive transforme ce qui est au total hérité. La classe Matrix ne défini que ce qui est nécessaire pour transformer un Control en Matrix.

Quand vous définissez une classe, elle est reliée à la hiérarchie de classe par la simple déclaration de sa super-classe (sauf si vous définissez une nouvelle classe racine!). Un grand nombre de super classes sont disponibles. [Sous NeXTSTEP l'environnement de développement fournit la classe Objet et plus d'une centaine de classes. Certaines utilisable tel quel, d'autre qui peuvent être adaptées à vos besoins, en définissant une sous-classe. Certaines classes du Kit définissent tous vos besoins; D'autres sont écrites pour être utilisées comme super classes. Ainsi des classes très complexes pourrons êtres définies en écrivant très peu de code.]

0.3.2 La classe Object

La classe Object est la seule classe racine (sans super-classe), la seule est unique qui est dans la généalogie de toutes les autres. Ceci parce qu'elle définit les éléments de base nécessaires au langage et aux interactions entre objets. Elle fournit aux sous-classes et aux instances de ces sous-classes, le comportement d'objet et la possibilité d'interagir avec le noyau d'exécution.

Une classe toute neuve qui n'a pas besoin d'hérité d'une autre classe sera de toute façon placé dans l'arborescence des classes, comme une sous classe d'Object. Les instances d'une classe ont besoin de se comportées comme des objets d'objectiveC pendant l'exécution. Hériter des capacités de la classe Object est plus facile que de les réinventer/réécrire à chaque nouvelle classe.

0.3.3 Héritage de variables d'instance

Quand un objet classe créé une nouvelle instance (un nouvel objet de la classe spécifiée), le nouvel objet contient les variables d'instances défini dans toute la généalogie de l'objet ainsi tout objet aura une variable d'instance isa qui est définie dans la racine soit la classe Object.

La Figure 4 présente une partie des variables d'instance de la classe Matrix et les classes dont elles proviennent. Remarquer le comportement additif.

Figure 4. Variables d'Instance de Matrix

Une classe n'a donc pas nécessairement à déclarer de variables d'instances propres, elle peut simplement définir de nouvelles méthodes, et profiter en cas de besoin des variables héritées.

0.3.4 Héritage des méthodes

Un objet n'accède qu'aux méthodes définies dans sa classe et les surclasses de sa généalogie. Une Matrix peut utiliser les méthodes définies dans Control, View, Responder, et Object, en plus des méthodes définies dans sa propre classe. On profite donc de tous le code écrit dans les super classes, c'est un gain important de productivité. Ainsi l'utilisation d'un lot de classes prédéfinies peut accélérer grandement l'écriture du code comme il suffit d'adapter les classes au besoins.

Les objets de type Class héritent uniquement des méthodes.

0.3.5 Recouvrement d'un méthode par une autre

Une exception fondamentale au mécanisme d'héritage: Dans une nouvelle classe vous définissez une méthode qui a le même nom qu'une méthode définie dans une super-classe. Cette nouvelle méthode recouvre l'original, c'est-à-dire que les objets de cette nouvelle classe utiliserons la nouvelle méthode, plutôt que l'original.

La méthode originale est toujours accessible par un appel spécifique (voir "Messages à self et super").

La méthode redéfinie peut elle même appeler l'original. Ainsi la nouvelle version ne fait que rajouter des fonctionnalités à la méthode, au lieu de la remplacer en entier. Ainsi l'implémentation d'une méthode pourra se trouver distribuée sur un ensemble de classes (exemple: free).

A l'inverse il n'est pas possible de recouvrir des variables d'instances (le compilateur refusera), en effet, la création d'un objet passe par l'allocation d'une structure dans laquel les positions des différentes varaibles d'instances est fixé, du fait de l'inclusion de types définis par les classes, il n'est pas possible dans une sous classe de déplacer ou enlever un des champs de la structure.

0.3.6 Modèles (super classes abstraites)

Certaine classes que l'on nomme modèles sont définies uniquement pour que d'autres classes en hérite. Ces super classes abstraites regroupent des méthodes et des variables d'instances qui seront utilisées de façon commune par un ensemble de sous-classes. La super-classe en elle même est incomplète, mais contient beaucoup de code réutilisable, ce qui réduit le travail dans l'écriture des sous-classes. La classe Object est l'exemple numéro 1 des classes abstraites, les programmes utilisent un grand nombre de sous-classes D'Object mais jamais d'instance de la classe Object. En effet un objet de la classe Object est peu intéressant,du fait qu'il n'a pas de variables d'instance.

Sous NeXTSTEP, dans les bibliothèques de classes définies dans le générateur d'applications, vous trouverez un ensemble de classes abstraites, qui aide à définir la structure de votre application. Quand vous créez des sous-classes de ces super classes, les instances de ces classes s'intégreront sans effort dans la structure d'application définie par le générateur et interagiront sans problème avec les autres classes de la bibliothèque.

0.3.7 Types & Classes

Une définition de classe est la spécification d'une espèce d'objet. Pratiquement, la classe définie un type de donné, une structure C. Le nom d'une classe peut être utilisé dans le code pour spécifié le type d'un objet en lieu et place de id. Exemple :

```
Matrix *maMatrix;
```

Ceci est du typage statique, le compilateur connaît le type de l'objet, en opposition au typage dynamique avec le type id.

Le type id est un pointeur sur un objet. De même les objets sont statiquement typés comme des pointeurs vers une classe (vers une structure définie par la classe). Un objet est toujours typé par un pointeur, le typage statique explicite le pointeur, le typage par id le cache.

Un objet peut être typé statiquement avec sa propre classe ou toute classe de sa généalogie. Comme un objet Matrix est aussi une View il peut être statiquement typé en View :

```
View *maMatrix;
```

L'objet maMatrix est plus qu'une View, il a les variables d'instance d'une View et d'un Control mais c'est néanmoins une View. Pour la vérification de type le compilateur considère que maMatrix est une View mais à l'exécution maMatrix sera manipulé comme une Matrix.

Voir "Options Statiques," pour plus d'information sur le typage statique et ces avantages.

0.3.8 Les objets Class

Une définition de classe contient des informations diverses, essentiellement sur les instances de la classe :

– le nom de la classe et de sa super classe

- une description des ses variables d'instances
- le prototypage des méthodes
- l'implémentation des méthodes

Cette information est compilée et stockée dans des structure de données, et mise à disposition du noyau d'exécution. Le compilateur crée un seul objet pour représenter la classe. Cet objet a accès à toutes les informations pertinentes sur la classe, soit essentiellement, le type des variables d'instance, et la possibilité de créer de nouvelles instances conformes à la description.

Bien qu'un objet classe contiennent le prototype des instances, il n'est pas lui même instance de la classe. Il n'a pas de variables d'instances propre. Par contre des méthodes peuvent être définies spécifiquement pour cet objet (méthodes de classe, a opposer aux méthodes d'instances). Un objet classe hérites des méthodes des classes définies dans sa généalogie, comme les instances héritent des variables d'instance de la généalogie.

Le nom de classe est utilisé comme receveur dans un message envoyé à l'objet class. Dans l'exemple suivant la classe Matrix renvoi sont numéro de version en utilisant une méthode héritée de Object :

```
int quelle =[Matrix version];
```

Tous les objets class sont de type Class. Dans l'exemple suivant la méthode class retourne la classe de l'objet receveur :

```
Class classeDeMonObjet = [monObject class];
```

Comme les autres objets, les objets Class peuvent être manipulés par des variables de type id :

```
id laClasse = [monObject class];
```

Les objets de type Class sont ainsi des objets à part entière, qui peuvent être typés dynamiquement et qui peuvent recevoir des messages et hériter des méthodes d'autres classes. Il sont spéciaux en ce qu'ils sont construit par le compilateur, n'ont pas de structures de données associées (a part celle définie dans la classe Class) et sont des agents capables de produire des instances pendant l'exécution.

Remarque: Le compilateur fabrique pour chaque objet de type Class un "objet métaclasse". Cet objet décrit la classe comme les classes décrivent leurs instances. Mais ces "métaclasse" ne sont manipulées que par le noyau d'exécution (a l'inverse d'autres langages ou elle sont utilisables).

0.3.9 La création d'instances

Une des fonctions principales des classes est de fabriquer de nouvelles instances. Le code suivant dit à l'objet classe Matrix de fabriquer une nouvelle instance et de l'assigner à la variable maMatrix :
`id maMatrix; maMatrix = [Matrix alloc];` La méthode alloc alloue dynamiquement de la mémoire pour les variables d'instance du nouvel objet et les initialise toutes à zéro (toutes sauf la variable isa qui connecte la nouvelle instance à sa classe qui est initialisé par alloc). Pour qu'un objet soit utilisable il est souvent nécessaire qu'il soit plus subtilement initialisé, que linitialisation réalisée par alloc, c'est le rôle de la méthode init. L'initialisation vient en générale juste après l'allocation comme dans le code suivant :

```
maMatrix = [[Matrix alloc] init];
```

Cette ligne de code, ou une toute proche, sera nécessaire avant que maMatrix puisse répondre à tous les messages que nous avons décrit plus haut. La méthode alloc retourne une nouvelle instance de la classe, la méthode init retourne l'instance initialisée. Tout objet class à une méthode comme alloc qui lui permet de produire de nouvelles instances. Toute instance a au moins une méthode comme init qui permet de l'initialisée. Si l'ont pense qu'une classe est la réalisation d'un type abstrait, la méthode init permet de placer l'instance dans un état conforme au axiomes du type abstrait. Les méthodes d'initialisation ont souvent des arguments qui permettent de spécifier les initialisations a réalisées, par exemple pour une Matrix la méthode initFrame:mode:cellClass:numRows:numColumns: est la plus souvent utilisée. Elles commencent toutes par "init".

Par convention tous les identificateurs de classe (Matrix) commence par une Majuscule, et les instances par une minuscule (maMatrix).

0.3.10 Les variables de classe

Toute instance a sa propre copie de ces variables d'instance déclarées dans la classe, chaque objet contrôle ses propres données, l'ensemble des valeur des varaibles d'instance forme ce que l'on appel l'état de l'instance.

L'objet classe par contre n'a pas de variable, seulement une structure de donnée interne initialisée à la définition (par le compilateur), rien de directement manipulable. L'objet classe n'a accès à aucune des variables d'instance de ces instances (ni lecture, ni écriture).

Ainsi pour que l'ensemble des instances d'une classe puissent partager des données il est nécessaire d'introduire des variables externes d'un certain type. Certaines classes définissent des variables statiques (mot clef static du C) et des méthodes pour les manipuler. (Déclarer une variable static dans le même fichier que la définition de classe, limite sa portée à la classe et à ses instances, la variable n'est pas héritée par les sous-classes, sauf si elles sont définies dans le même fichier.)

Ces variables static permettent d'ajouter des fonctionnalités à l'objet classe en plus de celle d'usine à instances, et lui permet de devenir un objet à part entière. Un objet classe peut être utilisé pour coordonner ses propres instances, distribuer des instances contenus dans une liste ou toute autre action primordiale pour votre application. Dans le cas limite ou vous n'avez besoin que d'un objet d'une classe donnée, vous pouvez déclarer toutes les variables en statique et utiliser que des méthodes de classe, cela vous épargnera l'allocation et l'initialisation d'une instance.

Remarque : L'utilisation de static n'est pas obligatoire mais ce choix est fait pour conserver l'encapsulation des variables dans les objets.

0.3.11 Nom de classes dans le source

Dans le source le nom des classes peux être utilisé dans deux contextes différents qui reflètent le rôle dual des classes en temps qu'objet et en temps que type de données.

Placé en receveur le nom de classe désigne l'objet classe, utilisation que nous venons de décrire dans les deux exemples précédents. Il est possible d'utiliser le nom de classe pour désigne l'objet class en position de receveur dans un message. Dans les autres contextes il faudras utiliser la méthode class pour obtenir l'identifiant de la classe, ainsi dans le test suivant on cherche à savoir si un objet est une Matrix pour cela on le teste avec la méthode isKindOf: qui prend en paramètre une class, il faut pour cella demander à l'objet class Matrix de nous dévoilé sont identifiant de classe :

```
if ( [anObject isKindOf:[Matrix class]] )
. . .
```

Il aurait été illégal d'utiliser simplement Matrix comme argument. Le nom de classe ne peuvent être qu'un receveur.

Le nom de classe peut aussi servir de type, comme dans l'exemple :

```
Matrix *unObjet;
unObjet = [[Matrix alloc] init];
```

Ici les deux usages du nom de classe sont utilisés, l'objet unObjet est typé statiquement en tant que Matrix, le compilateur suppose qu'il a la structure d'une matrix et qu'il réagit aux méthodes définies dans la classe Matrix (et sa généalogie). Le typage statique permet au compilateur de faire de meilleur vérifications de type et permet de réaliser certaines optimisations (recherche de la méthode), voir "Options Statiques" pour des détails.

Seules les instances peuvent être typées statiquement, les objet classe sont automatiquement de type Class!

0.4 Définir une classe

L'essentiel de la programmation objet est l'écriture de code pour de nouveaux objets, c'est à dire la définition de nouvelles classes. En Objective C les classes sont définie en deux parties :

- une interface qui déclare : les méthodes, les variables d'instances, la super classe.
- une implémentation qui définie réellement la classe (le source des méthodes).

Bien que le compilateur ne le demande pas l'interface et l'implémentation sont en général placés dans deux fichiers différents. En effet l'interface doit être accessible à tous ceux qui utilise la classe. Les utilisateurs n'ont pas besoin de l'implémentation et peut être préférez vous ne pas distribuer trop facilement vos sources.

Un seul fichier peut déclarer ou implémenter plusieurs classes. Mais l'usage veut que l'on place chaque interface dans un fichier différents, ainsi que chaque implémentation. Ce découpage explicite mieux le status indépendant des différentes classes, et donne un rôle important au concept d'interface qui est un concept clef dans la programmation à plusieurs.. Le nom de ces fichiers est corrélé généralement au nom de la classe quand il ne sont pas tout simplement identiques. L'implémentation est un fichier d'extension ".m" ce qui indique du code ObjectiveC. L'interface peut avoir l'extension de votre choix mais la norme est à l'usage de l'extension ".h" comme en C pour les fichiers d'inclusion. Ainsi la classe Matrix aura sont interface dans le fichier Matrix.h et sont implémentation dans le fichier Matrix.m.

Cette séparation de l'interface et de l'implémentation est adaptée à la philosophie objet. Un objet est une entité autonome qui résout les problèmes comme une boite noire, une fois que vous avez définie les interactions de l'extérieur avec l'objet, vous prouver changer l'implémentation sans danger pour le reste de l'application. Reciproquement si l'interface doit changer (ce qui doit être éviter au maximum) un mécanisme de protection des fichiers d'interace peut vous obliger à signaler la modification soit à l'utilisateur soit à l'implementeur suivant votre propre cituation.

0.4.1 L'interface

Une déclaration d'interface commence par la directive de compilation @interface et finie par la directive @end. (toutes les directives de compilation de ObjectiveC commence par @ pour les différentier de celles du préprocesseur cpp du C).

```
@interface NomClasse : SaSuperClasse
{
déclarations des variables d'instance
}
prototypes des méthodes
@end
```

La première ligne déclare la nouvelle classe et crée le lien avec sa superclasse. La superclasse définie la proportion de la nouvelle classe qui est héritée dans la hiérarchie des classes (voir héritage plus haut). Si le ":" et le nom de la superclasse sont oubliés la nouvelle classe est une classe racine (rivale de Object).

La zone entre déclare les variables d'instance, la structure de données qui sera présente dans chaque instance. Par exemple une partie des variables d'instance de Matrix :

```
id      selectedCell;
int     numRows;
int     numCols;
```

```
float   backgroundGray;
id      cellClass;
```

Les méthodes de la classe sont déclarées après l'accolade fermente et avant la directive @end. Les méthodes de classe utilisables par les objets classe sont précédées du signe + :

`+ alloc;` Les méthodes utilisables par les instances sont précédées par le signe - :

`- display;`

Bien que ce ne soit pas recommandé vous pouvez déclarer des méthodes de classe et d'instances avec le même nom. Une méthode peut avoir le même nom qu'une variable d'instance, c'est plus fréquent surtout quand la méthode renvoi la valeur de la variable d'instance. Ainsi la classe Matrix a une méthode selectedCell associée avec sa variable d'instance selectedCell.

La valeur de retour des méthodes est déclarée avec la syntaxe de coercition du :

`- (int)tag;` de même pour les arguments des méthodes :

`- setTag:(int)anInt;`

Si le type n'est pas explicité, le type par défaut est utilisé, c'est-à-dire un id. Les méthodes alloc, display, et setTag: détaillées au dessus retourne toutes des id.

Quand il y a plus d'un argument il sont déclarés après chaque ":" comme dans un message, par exemple :

```
- moveTo:(NXCoord)x:(NXCoord)y;
- getRow:(int *)aRow andColumn:(int *)aColumn ofCell:aCell;
```

(NXCoord est un type flotant utilisé pour les coordonnées sur l'écran.)

Pour les méthodes à nombre variable d'argument on utilisera une virgule et une ellipse, comme pour les fonctions C :

```
- makeGroup:group, ...;
```

0.4.2 Importer une interface

Le fichier d'interface doit être importé (#include,#import) dans tous fichiers qui utilise la classe. Tout module qui crée des instances de la classe, envoie des messages à la classe, ou mentionne une variable d'instance de la classe. L'interface est en général incluse par la directive #impor :

```
#import "Matrix.h"
```

Cette directive est équivalent à #include, sauf quelle s'assure qu'un même fichier n'est jamais inclut plus d'une fois. C'est pourquoi elle est utilisée de préférence à #include. Le gcc de fillmore préfère que l'on utilise #include et que l'on place dans le fichier une séquence de directives comme ceci :

```
#ifndef _nomfichier_
#define _nomfichier_
/* contenu du fichier */
#endif
```

Mais vous pouver utiliser #import et éliminer les warning avec -Wno-import.

Comme une classe est construite à partir d'une super-classe il faut importer l'interface de la super classe avant d'écrire l'interface :

```
#import "SaSuperClasse.h"

@interface NomClasse: SaSuperClasse
{
        d'{e}clarations des variables d'instance
}
prototypes des m'ethodes
@end
```

Cette convention implique que toute interface inclut automatiquement les interfaces de sa généalogie. Quand un module d'implémentation inclut une interface, il inclut automatiquement toutes les interfaces de la généalogie de la classe dont il vient d'importée l'interface.

0.4.3 Faire référence à d'autres classes

Un fichier d'interface déclare une classe et en important sa super classe contient implicitement la déclarations de toutes les classes de sa généalogie. Si une interface mentionne une classe qui n'est pas dans sa généalogie elle doit alors être déclarée explicitement avec la directive @class :

```
@class Matrix, List;
```

Ceci informe le compilateur que Matrix et List sont des noms de classes, sans importer leur déclarations.

Un fichier d'interface mentionne des noms de classes quand il type statiquement les variables d'instance ou les arguments ou les valeurs de retour des méthodes . Par exemple :

```
- (Matrix *)matrixFromList:(List *)aList;
```

Comme ces déclarations utilisent simplement le nom des classes comme un type sans dépendre de l'interface de ces classes (méthodes et variables), la directive @class est suffisante. Par contre si la class est effectivement utilisée, (création d'instances, envoi de messages), alors l'interface de la class doit être importé. Typiquement, dans un fichier interface on utilise la directive @class, dans un fichier d'implémentation, on importe l'interface de la classe utilisée (comme en général on va envoyer des messages aux objets et que l'on doit créer ceux-ci). La directive @class est le moyen le plus simple pour déclarer l'existence d'une classe sans risquer des problèmes avec l'inclusion de fichiers qui eux même importent d'autres fichiers, en particulier si la classe UneClasse a des variables de classes statiques, et si elle importe l'interface d'une autre classe UneAutreClasse qui elle même importe l'interface de UneClasse aucunes des deux classes ne pourra compiler correctement.

0.4.4 Séparer l'interface de l'implémentation

Le but de l'interface est de déclarer la nouvelle classe pour les modules qui utilise cette classe (et les programmeurs qui l'utilisent!). Elle contient toutes les informations dont le compilateur a besoin pour travailler avec la classes (les programmeurs apprécierons sans doute une petite documentation).

- La liste des méthodes du fichier interface permet aux autres modules de connaître les messages qui peuvent êtres envoyés aux objets de cette classe. Toutes les méthodes qui peuvent être utilisées à l'extérieur de la définition de la classe (ensemble de l'interface et de l'implémentation) sont déclarées dans le fichier d'interface, les méthodes (fonctions) internes à la classe peuvent être omises.
- Elle fournit aux compilateurs et aux programmeurs la liste des variables d'instances, pour la création des objets mais aussi pour connaître les variables héritées par les sous-classes. Bien que les variables d'instance sont plus liées à l'implémentation qu'a l'interface du fait de l'encapsulation, elles doivent néanmoins être indiquées dans l'interface. Ceci par ce que

le compilateur doit connaître la structure d'un objet quant il est manipulé, et non pas uniquement là ou il est déclaré. Le programmeur lui peut généralement ignorer les variables d'instances, sauf quand il crée une sous-classe.

– Finalement, l'interface décrit pour les utilisateurs, les classes connectées et/ou héritées qui sont nécessaires.

0.4.5 L'implémentation

La définitions d'un classe est structuré comme sa déclaration, elle commence par la directive @implémentation et finie par @end : `@implementation NomClasse: SaSuperClasse`

```
déclarations de variables d'instances

définitions des méthodes
@end
```

Tout fichier d'implémentation importe sont fichier d'interface, il n'est donc pas nécessaire de répéter les déclarations déjà réalisées. Le fichier d'implémentation peu donc omettre sans danger :

– Le nom de la superclasse

– la déclaration des variables d'instance

Ce qui simplifie l'implémentation et la dévoue seulement à la définition des méthodes :

```
   #import "NomClasse.h"
@implémentation NomClasse
définitions des méthodes
@end
```

Les méthodes d'une classe sont définies comme des fonctions C, avec un bloc défini par des accolades. Avant le bloc elles sont déclarées avec la même syntaxe que pour l'interface :

```
   + alloc
{
}

- (int)tag
{
}
- moveTo:(NXCoord)x:(NXCoord)y
{
}
```

Pour les méthodes avec un nombre variable d'argument on utilise les même technique qu'en C :

```
   #import <stdarg.h>

- getGroup:group, ...
{ va_list ap;
va_start(ap, group);
. . . }
```

0.4.6 Accès aux variables d'instance

Par défaut, les variables d'instances d'un objet sont dans la portée des méthodes de cet objet. Elles sont simplement utilisables par leur nom. Bien que le compilateur fabrique une structure équivalente aux structures du C pour stocker ces variables, la structure réelle est cachée. Il n'est donc pas nécessaire d'utilisé les opérateurs de champ (.) ou les indirections (-¿) pour manipuler les données de l'objet. Par exemple, la définition de méthode suivante manipule la variable d'instance tag du receveur :

```
- setTag:(int)anInt
{
    tag = anInt;
    . . .
}
```

Ni l'objet ni sa variable d'instance tag ne sont déclarés comme argument de la méthode, pourtant elles sont dans la portée de la méthode. Ceci simplifie la syntaxe des méthodes et permet un raccourci utile dans l'écriture des sources en ObjectiveC.

Les variables d'instance de la classe de l'objet ne sont pas les seules auxquelles une méthode peut référer. En effet, la méthode peut manipuler toutes les variables d'instance définies dans sa généalogie, aussi longtemps que deux conditions sont vérifiées:

- Les variables d'instance doivent être dans la portée de la définition de la classe. Soit simplement une des variables d'instance de la généalogie ou directement dans la classe (la portée est vue dans la section suivante).
- Le compilateur sait à quel objet appartient la variable.

Quand la variable d'instance appartient au receveur (comme dans la méthode setTag:), il n'y a pas de problèmes. Le receveur est implicite mais évident, c'est ce type que définie la classe. Quand la variable d'instance n'appartient pas au receveur, alors le type de l'objet doit être explicité pour le compilateur par du typage statique. Dans ce cas on utilisera l'opérateur d'indirection (-¿). Supposons par exemple que la classe NouvelleClasse déclare une variable statiquement typé dans ses variables d'instance :

```
@interface NouvelleClasse: Object
{
    NouvelleClasse *compagnon;
    int tag;
    char *name;
}
```

Tant que les variables d'instance de l'objet sont dans la portée (comme ici), une des méthodes de NouvelleClasse peut les modifier directement :

```
- creerCompagnon
{
    if ( !compagnon ) {
        compagnon = [[NouvelleClasse alloc] init];
        compagnon->tag = tag;
        compagnon->name = name;
    }
    return compagnon;
}
```

0.4.7 La portée des variables d'instance

Bien que déclarée dans l'interface les variables d'instance sont plus une question d'implémentation que d'utilisation. L'interface d'un objet dépend plus de ces méthodes que de la structure de ses données. Il y a souvent une correspondance entre une méthode et une variable comme dans l'exemple suivant :

```
- (int)tag
```

```
{
    return tag;
}
```
Mais ce n'est pas obligatoire. Des méthodes peuvent retourner des informations qui ne sont pas contenues dans les variables d'instances et certaines variables d'instance peuvent contenir des informations que l'objet ne veut pas révéler.

A longueur des révisions d'une classe, le choix des variables d'instance peut changer, bien que les méthodes déclarées restent inchangées. Tant que les messages sont le véhicules des interactions avec les instances, ces changements n'affecterons pas son interface.

Pour assurer l'encapsulation des variables d'instances, le compilateur limite leurs portée. Mais pour plus de flexibilité, il permet de spécifier la portée sur trois niveau différents comme c'est illustré Figure 5.

Figure 5. La portée des variables d'instance

Chaque niveau est indiqué par une directive de compilation.

Directive Sens

@private La variable d'instance n'est accessible que dans la classe qui la déclare.

@protected La variable d'instance n'est accessible que dans la classe qui la déclare et dans les classes qui en hérite.

@public La variable d'instance est accessible de partout.

Une directive s'applique a toutes les variables d'instance listé après, jusqu'à la prochaine directive ou la fin de la liste. Dans l'exemple suivant, les instances squelette et placard sont privées, gardeAVue est protégée, invitation public.

```
@interface UneClass : Object
{
@private
    id squelette;
    float placard;
@protected
    int gardeAVue;
@public
    char * invitation;
}
```
Par défaut les variables sont protégée (@protected). Toutes les variables d'instance qu'une classe déclare (quelque soit la directive) sont visible dans l'implémentation de la classe elle même. Par exemple, une classe qui déclare une variable d'instance capitaine, peut y référer dans une définition de méthode :

```
- remplacerCapitaine: leNouveau
{
    id leVieux = capitaine;
    capitaine = leNouveau;
    return leVieux;
}
```
Il est clair que si une classe ne pouvai pas accéder à ces propres variables d'instance, celles ci seraient sans aucune utilité.

Normalement, une classe a aussi accès aux variables d'instance dont elle hérite. Il est naturel que l'ensemble de la structure de donnée de l'objet soit dans la portée des méthodes de la classe, surtout si vous pensez profiter de l'héritage. La possibilité de manipuler une variable est en général hérité avec celle ci. La méthode remplacerCapitaine: peut être définie dans toute classe qui hérite de la variable capitaine.

Des raisons pour lesquelles vous voudriez retirer l'accès d'une variable aux sous classes:

- Une fois qu'une sous classe a accès à une variable, la classe qui a déclaré cette variable est tenue de la conserver et de ne pas changer son utilisation. En effet il n'est pas possible d'éliminer une variable ou de changer son rôle sans par inadvertance détruire la sous classe.

- De l'autre coté une sous classe peut par inadvertance changer la valeur d'une variable dont elle hérite et crée des erreurs cachées (bugs) dans la classe qui a déclaré la variable, surtout si la variable participe à des interdépendance interne à la classe.

Pour limiter la portée d'une variable à la classe qui la déclare utiliser la directive @private.

A l'autre extrême, déclarer une variable @public, lui donne une porté générale (comme le champ d'une structure C). Normalement pour connaître la valeur d'une variable d'instance, on envoi un message demandant l'information. Pour les variables publiques on peu y accédé avec la même syntaxe que pour un champ de structure en langage C.

UneClass *mutin = [[UneClass alloc] init]; mutin-¿captain = nil;

Remarquer que l'objet doit être statiquement typé.

Rendre une variable d'instance @public va à l'encontre du principe d'encapsulation, ou les objets protègent leurs variables contre la lecture et les erreurs d'inadvertance. Les variables publiques doivent donc être évitées au maximum sauf dans des cas particuliers extraordinaires.

0.5 La gestion des messages

En ObjectiveC les messages ne sont pas liés aux méthodes avant l'exécution. Le compilateur converti les messages, `[receveur message]` en un appel à la fonction, obi_msgSend(). Cette fonction prend le receveur et le nom de la méthode mentionné dans le message, le sélecteur (selector), comme ces deux arguments principaux :

`objc_msgSend(receveur, selector)` Tous les arguments du message sont aussi fournis à la fonction objc_msgSend() :

`objc_msgSend(receveur, selector, arg1, arg2, . . .)`

Cette fonction de message fait le nécessaire pour le lien dynamique:

- Elle cherche en premier la procédure (le code de la méthode) à la quelle se réfère le sélecteur. Comme plusieurs classes peuvent répondre au même sélecteur, le code sélectionné dépend donc de la classe du receveur.
- Elle appelle la procédure, avec la structure de donnée définie par le receveur (ces variables d'instances), ainsi que les arguments d'appel spécifiés pour la méthode.
- Finalement, elle retourne la valeur de retour de la procédure.

Remarque, c'est le compilateur qui créer les appels à la fonction de message. Vous ne devez jamais l'appeler directement dans le code que vous écrivez.

La clef du mécanisme de message se trouve dans les structure de donnée que le compilateur créé pour chaque classe et chaque objet. Chaque objet classe contient deux éléments essentiels:

- Un pointeur sur sa super classe.
- Une table d'indirection de classe; Cette table a des entrées qui associe les sélecteurs aux adresses des procédures qu'ils nomment. Le sélecteur moveTo:: est associé à l'adresse de (la procédure qui réalise) moveTo::, le sélecteur display est associé à l'adresse de display, etc.

A la création d'un objet, une zone mémoire est alloué pour le représenté, ses variables d'instances sont initialisées. La première de ces variables (définie dans Object) est un pointeur sur sa structure de classe. Ce pointeur isa donne à l'objet l'accès à sa classe, et par sa classe à toutes les classes dont il hérite.

Ces éléments des structure de classe et d'objet sont illustrés Figure 6.

Figure 6. Messagerie

Quand un message est envoyé à un objet, la fonction obi_msgSend() suit le pointeur isa de l'objet jusqu'a la structure de classe ou elle cherche le sélecteur dans la table d'indirection de classe.

Si le sélecteur ne s'y trouve pas, la fonction obi_msgSend() suit le pointeur vers la superclasse où de nouveau elle recherche le sélecteur de la méthode. Les échecs successifs font que la fonction remonte la hiérarchie de super classes jusqu'à la classe Object, qui Elle, signale l'échec. Une fois que le sélecteur est localisé, elle appelle la fonction donnée par la table sur la structure de données de l'objet receveur.

Pour accélérer ces recherches, à chaque message le noyau d'exécution, enregistre les sélecteurs utilisés. En enregistrement est crée par classe, et il peut contenir à la fois les méthodes de la classe mais aussi les méthodes de la généalogie. Avant de recherche effectivement dans la table d'indirection, la fonction objc_msgSend() consulte ces enregistrements dans la classe de l'objet receveur (selon la théorie que si une méthode a été utilisé elle le sera de nouveau). Si le sélecteur est dans l'enregistrement, alors le message est presque aussi rapide qu'un appel de fonction. Une fois qu'un programme a tourné suffisamment longtemps presque tous les appels se feront par enregistrement. Ces enregistrements sont alloués dynamiquement pour accepter de nouveaux messages pendant l'exécution du programme.

0.5.1 Sélecteurs (selector)

Pour des raisons d'efficacité les noms ASCII ne sont pas utilisés comme sélecteurs de méthodes dans le code compilé. A leurs place, le compilateur et le noyau d'exécution utilisent une table de conversion qui associe à chaque nom ASCII de méthode un unique identifieur qui servira de représentant de la méthode. Ces représentants sont du type SEL, ce qui permet de les différencier des autres types de données. Un sélecteur valide n'est jamais nul.

Un représentant contient des champs définis par le compilateur qui aide le noyau d'exécution. Il est donc impératif que vous laissiez au système le travail de définir la valeur de ces sélecteurs. La directive @selector() permet de référer le sélecteur compilé dans du source ObjectiveC, plutôt que d'utiliser le nom complet. Dans l'exemple suivant le représentant de moveTo:: est placé dans la variable action:

SEL action; action = @selector(moveTo::);

Il est plus efficace de placer les représentants dans des variables de type SEL à la compilation avec la directive @selector() mais parfois il est nécessaire de convertir en cours d'exécution une chaîne en représentant. Ceci peut être fait avec la fonction sel_getUid() :

action = sel_getUid("moveTo::");

Cette fonction et d'autre fonction du noyau d'exécution sont décrite dans le manuel général.

0.5.2 Changer le message en cours d'exécution

Les méthodes perform:,perform:with:, et perform:with:with:, définies dans la classe Object, prennent des SEL comme premier paramètre. Ces trois méthodes sont parfaitement intégrées dans le système de messages. Par exemple:

[voisin perform:@selector(discute:) with:unAmis];

est équivalent à :

[voisin discute: unAmis];

Ces méthodes permettent de faire varier le message pendant l'exécution, comme il est possible de faire varier l'objet receveur. Ainsi une variable peut être utilisé à la fois pour le receveur et le sélecteur.

id cible = [unObjet donneLe Receveur]; SEL action = [unObjet donneLeSelecteur]; [cible perform:action];

Dans cet exemple deux méthodes fictives sélectionnent en cours d'exécution, d'une part un receveur cible, d'autre part une action. En suite ce receveur reçoit le message action grâce à au message perform:.

Remarque: les méthodes perform retourne des id. Si la méthode exécuté retourne un autre type il faut réaliser une coercition sur la valeur retournée. Mais attention ce n'est pas valable pour tous les types, la méthode doit renvoyer un type compatible avec les pointeurs.

0.5.3 Le type des arguments et de la valeur de retour des méthodes

Les sélecteurs compilés identifient les noms de méthodes par leur implémentations. La méthode display de Matrix aura le même sélecteur que la méthode display dans d'autres classes. Ceci est essentiel pour le polymorphisme, cela permet à des méthodes comme perform: et perform:with: d'envoyer le même messages à des receveur appartenant à des classes différentes.

La fonction de message a accès aux implémentations uniquement à travers les sélecteurs, ainsi elle traite toutes les méthodes avec le même sélecteur de la même façon. Elle trouve le type de la valeur de retour et des arguments à partir du sélecteur. Ainsi, à part pour les messages envoyés à des receveurs statiquement typés, l'édition de liens dynamique demande que toutes les méthodes de même nom aient le même type de valeur de retour et le même type d'arguments. (Les receveur statiquement typé sont une exception, comme le compilateur peut trouver l'implémentation à partir de la classe du receveur.)

Les méthodes de classe et d'instance avec le même nom ont des sélecteurs compilés différents. Et peuvent avoir des arguments et valeur de retour différents.

0.5.4 Arguments cachés

Quand la fonction de message trouve la procédure qui implémente une méthode, elle appelle cette procédure avec tous les arguments du message. Elle passe aussi deux arguments cachés qui sont :

- l'objet receveur. C'est ainsi que les variables d'instances sont dans la porté de la méthode.
- le sélecteur de la méthode.

Ces arguments sont caché dans le sens qu'ils ne sont pas déclarés dans le source qui défini la méthode. Ils sont insérés dans le code par le compilateur. Bien que ces arguments ne soit pas explicitement déclarés il sont quand même manipulables.

Les méthodes réfères au receveur par la variable self et au sélecteur par _cmd. Dans l'exemple suivant _cmd est le sélecteur de la méthode etrange et self est l'instance qui reçoit le message etrange.

```
- etrange
{
    id  cible = leReceveur();
    SEL action = laMethode();

    if ( cible == self || action == \_cmd )
        return nil;
    return [cible perform:action];
}
```

La variable self est la plus utile, c'est grâce à elle que les variables d'instances sont dans la portée de la méthode.

L'usage est de faire retourner self aux méthodes qui n'ont pas de valeur de retour spécifique. Ceci permet d'écrire des messages imbriqués dans le source, comme par exemple :

```
[[[maMatrix setMode:NX_RADIOMODE] setEnabled:YES] setTag:99];
```

La variable self est détaillé dans la section suivante.

0.5.5 Messages à self et super

La syntaxe des messages pose un problème dès qu'une méthode doit référer le receveur (l'objet qui reçoit le message). Supposons que l'on définisse une méthode reCentre qui change les coordonnées de l'objet sur lequel elle s'applique. Pour cela elle va utiliser la méthode moveTo:: pour réaliser le déplacement. Ce message doit donc être envoyé à l'objet receveur du message reCentre. Il n'est pas possible avant l'envoi du message reCentre de savoir quel objet l'on doit déplacer avec le message moveTo::. En effet le message reCentre peut être envoyé à tout un ensemble d'objets différents, de classes différentes, de nom différents :

- reCentre [??? moveTo:30.0:50.0]; . . .

Pour résoudre ce problème ObjectiveC fournit deux termes qui se réfère à l'objet qui réalise la méthode se sont :

– self

– super

La méthode reCentre s'écrit donc soit :

```
- reCentre
{
  [self moveTo:30.0:50.0];
  . . .
}
```
ou:
```
- reCentre
{
  [super moveTo:30.0:50.0];
  . . .
}
```

Ici self et super tous deux réfèrent l'objet receveur du message reCentre, quelque soit la classe de l'objet. Mais les deux termes sont bien différents, self est un des arguments cachés de la fonction de traitement des messages, c'est une variable locale qui peut être utilisée librement dans l'implémentation d'une méthode, comme les variables d'instance. super peut se substituer à self uniquement en tant que receveur dans un message. En position de receveur la différence des deux termes est dans le traitement du message :

– Pour self l'implémentation du sélecteur est recherchée de façon usuelle, en commençant par la table de la classe du receveur.

– Pour super l'implémentation du sélecteur est recherchée en commençant dans la table de la superclasse.

La distinction entre self est super est réalisée à la compilation, avec l'utilisation de objc_msgSend dans le cas de super, et d'une autre fonction qui recherche directement dans la superclasse de l'objet receveur.

0.5.6 Un exemple

La distinction entre self et super se fait clairement dans une hiérarchie de trois classes. Supposons par exemple, l'existence d'une instance de la classe Bas. La superclasse de Bas est Milieu, la super classe de Milieu est Haut. Les trois classes ont une méthode négocier différente, qu'elles peuvent utiliserpour une variété d'utilisations. De plus, milieu a une méthode ambitieuse appelée PaixDurable, qui utilise aussi la méthode négocier.Voir Figure 7.

On envoi maintenant à notre instance de Bas, le message PaixDurable qui va envoyer à la même instance de la Classe Bas le message négocier, Deux possibilités si le source de Paix Durable est le suivant: - PaixDurable ... [self négocier]; ... le système de messages va trouver la méthode définie dans Bas. Si le source est le suivant : - PaixDurable ... [super négocier]; ... le système de messages va trouver la méthode définie dans Haut! Ansi la méthode définie dans Milieu n'est jamais atteinte à partir d'une instance de Bas !

Cet exemple illustre la possibilité de cours circuiter une méthode qui peut avoir été redéfinie dans une sous classe.

Cette exemple semble suggérer l'existence d'un défaut dans le système de messages, du fait qu'il est impossible d'accéder à la méthode négocier de la classe Milieu dans une instance de la classe Bas. Mais dans les circonstances décrite c'est le bon choix en effet : - l'auteur de la classe Bas a intentionnellement caché la méthode définie dans Milieu, en créant un nouvelle méthode pour cette classe Bas (et ces sousclasses). - l'auteur de la classe Milieu a utiliser super dans la méthode Paix Durable pour assurer que la méthode négocier de la classe Haut soit utiliser et aucune autre !

Pour utiliser la méthode négocier de la classe Milieu il suffit d'avoir une instance de cette classe et de lui envoyer directement le message négocier.

Figure 7. Haut, Milieu et Bas

0.5.7 Utilisation de super

Les messages à super permettent aux implémentations de méthodes d'être distribuées sur plusieurs classes. Vous pouvez aussi cours circuiter une méthode existante, la modifier, ou y ajouter des fonctionalités tout en gardant la méthode originale :

- négocier . . . return [super négocier];

Pour certaines taches, chaque classe dans la hiérarchie implémente une méthode qui fait une partie du travail et passe le message à sa super classe grâce à super pour que la méthode de la super classe réalise le reste. La méthode init, qui initialise les nouveaux objets d'une classe et la méthode write: qui permet de préparer les objets pour archivage sur un canal de sortie (stream), sont construits pour fonctionner de cette façon. Toutes les méthodes write: ont pour tache de réaliser l'écriture des variables d'instance définies dans la classe. Avant de réaliser cette écriture elle envoie à sa super classe un message write: pour que celle-ci archive ces variables d'instance et ainsi de suite jusqu'a la classe racine (Object). Comme chaque version de write est écriture de la même façons les variables d'instance sont écrites sur le canal de sortie dans l'ordre d'héritage: - write:(type_stream *)stream [super write:stream]; . . . return self;

Il est aussi possible de concentrer l'algorithmique d'une méthode dans une superclasse et que les autres classes incorpore la méthode à travers des appels à super. Par exemple, toutes les méthodes de classe qui créé de nouvelles instances doivent allouer de la place pour le nouvel Objet et initialiser sont pointeurs isa (qui type l'objet) vers la classe de l'objet (structure de classe contenant la classe de l'objet). Ceci est typiquement laissé à la méthode alloc définie sur la classe Object. Si par un hasard (rare) vous voulez modifier cette méthode il est encore possible de conserver la fonctionnalité de base en réalisant un message à super.

0.5.8 Redéfinir self

Le mot clef super est simplement un drapeau spécifiant au compilateur ou il doit commencer la recherche de la méthode a réalisée. Par contre self est une variable qui peut être utilisé de nombreuses façons et peut même être affecté d'une nouvelle valeur.

Ce dernier cas ce rencontre fréquemment dans les méthodes de classes. Les méthodes de classes sont en générale concernées par des instances de la classe plutôt que par l'objet class lui même. Par exemple, une méthode peut combiner l'allocation et l'initialisation d'une instance:

+ newTag:(int)anInt return [[self alloc] initTag:anInt];

Dans une telle méthode, il est tentant d'envoyer des messages à l'instance et d'appeler l'instance self, de la même façons que pour une méthode d'instance. Mais ce serais une erreur. En effet self et super tout deux font référence à l'objet receveur, l'objet qui reçoit le message qui lui dit de réaliser une méthode, or dans une méthode d'instance self désigne l'instance mais dans une méthode de class self désigne l'objet classe ! Ainsi pour pouvoir envoyer un message d'application d'une méthode d'instance il faut redéfinir self comme étant une instance :

+ newTag:(int)anInt andColor:(NXColor)aColor self = [[self alloc] initTag:anInt]; [self setColor:aColor]; return self;

La méthode + newTag:andColor: est une méthode de class (signe +) donc initialement self est l'objet classe, c'est cet objet classe qui reçoit le message alloc. self est en suite redéfini pour comme l'instance renvoyée par la méthode alloc et initialisé par initTag:. C'est cette nouvelle instance qui reçoit le message setColor:. Pour éviter toute confusion il est préférable d'utiliser une autre variable que self pour désigner une instance dans une méthode de class :

+ newTag:(int)anInt andColor:(NXColor)aColor id newInstance = [[self alloc] initTag:anInt]; [newInstance setColor:aColor]; return newInstance;

Remarque: Dans ces exemples, la méthode de classe envoi des messages (initTag: et setColor:) pour initialiser l'instance. Elle n'assigne pas de nouvelles valeurs directement dans les variables d'instance comme le fait une méthode d'instance:

tag = anInt; color = NX_REDCOLOR;

Seules les variables d'instance du receveur du message peuvent être positionnée directement. Comme l'objet class n'a pas de variables d'instances, cette syntaxe ne peut être utilisée. Par contre si la variable newInstance avait été typé statiquement, l'affectation directe est possible avec la syntaxe suivante : newInstance-¿tag = anInt;

Voir "Accès aux Variables d'instance" plus haut.

0.6 Catégories

Il est possible d'ajouter des méthodes à une classe en les déclarant dans un autre fichier d'interface sous un nom de catégorie et un fichier d'implémentation de la même catégorie. Le nom de catégorie indique que c'est un ajout de méthodes à la classe et non pas une nouvelle classe.

Les méthodes de catégories sont écrites de la même façons que les autres méthodes, il n'y a pas de différence à l'exécution et elles sont héritées de la même façons par toutes les sous-classes. Il n'y a pas de limites au nombre de catégories que l'on peut ajouter à une classe, mais chaque nom de catégorie doit être différent, et chacunes doit définir un ensemble disjoint de méthodes.

0.6.1 Additions à une Classe

La déclaration d'une catégorie est similaire à une déclaration d'interface de classe, sauf que la super classe est remplacée par par le nom de la catégorie entre parenthèses. La catégorie doit importer le fichier d'interface de la classe:

#import "ClassName.h"

@interface ClassName (Category) déclaration des nouvelles méthodes @end

Noter bien que la catégories n'ajoute pas de nouvelles variables d'instance, elle ne contient que des déclarations de méthodes. Par contre toutes les variables d'instance accessibles par la classe sont accessibles aux méthodes de la catégorie. Ce qui inclut toutes les variables d'instance définie dans la classe. Le fichier d'implémentation doit évidement importer sont propre fichier d'interface. En supposant que le fichier d'interfaces et le fichier d'implémentation sont nommés d'après le nom de la catégorie, un fichier d'implémentation se présente sous la forme suivante:

/* Category.m */ #import "Category.h"

@implémentation ClassName (Category) définition des méthodes @end

Les méthodes ajoutées par une catégorie peuvent soit étendre les fonctionnalités d'une classe, soit cours- circuiter les méthodes héritées par la classe et même cours-circuiter les méthodes déclarées dans la classe mais par contre ce n'est pas possible entre deux catégories (en effet comment choisir entre les deux méthodes). Attention une catégorie n'est pas un substitut pour une sous-classe. (Elles ont plus d'utilité comme outil d'extention de classe, plutôt que de redéfinition de classe). Il est plus prudent que les catégories ne redéfinissent pas les méthodes définie dans la classe, et une classe ne doit pas définir plusieurs fois une même méthode. Remarque: Quand une méthode de catégorie recouvre une méthode de classe il est possible de faire appel au méthode s définie dans la super classe (grâce à super) mais la méthode de la classe qui a été recouverte n'est plus accessible !

0.6.2 Comment utiliser les catégories

Les catégories peuvent être utilisées pour étendre des classes définies par d'autres programmeurs, comme par exemple celles d'une bibliothèque de classe. Les nouvelles méthodes seront héritées par les sous classes et ne seront pas distinguables des méthodes originales.

Mais les catégories peuvent aussi permettre de distribuer l'implémentation d'une nouvelle classe dans plusieurs fichiers, par exemple, vous groupez les méthodes d'une classe en plusieurs catégories et vous placez chaque catégorie dans un fichier différent. Utilisées ainsi les catégories peuvent aider au développement de plusieurs façons :

- Elles fournissent un moyen simple de regrouper des méthodes travaillant sur les mêmes fonctionnalités. Des méthodes similaires définies sur différentes classe peuvent ainsi être placées dans le même fichier.

- Elles simplifient la gestion d'une grande classe quand plusieurs développeurs contribues à la définition de la classe.

- Elles permettent d'utiliser les possiblitées de compilation incrémentale pour de très grandes classes.

- Elle permettent de simplifier la localisation des références pour des méthodes très fréquemment utilisées.

- Elles permettent de configurer une classe pour différentes applications en limitant la taille des sources. Les catégories permettent aussi de déclarer des protocoles informels, voir dessous.

0.7 Protocoles

Les classes et les catégories définissent des méthodes qui sont associées à une classe particulière, soit les méthodes que la classe implémente. Inversement les protocoles formels et informels déclarent des méthodes qui ne sont pas associées à une classe mais que n'importe quelles classes (et souvent de nombreuses) peuvent implémenter. Les protocoles permettent de déclarer des méthodes indépendamment de la hiérarchie de classe, elles peuvent être ainsi utilisées d'une façons,que classes et catégories ne peuvent fournir. Ils listent des méthodes qui sont (ou peuvent être) implémentées quelque part, mais la classe exacte qui les implémente est sans importance. Les protocoles peuvent jouer un rôle important en conception, spécialement quand un projet est réalisé par plusieurs programmeurs et qu'il utilise des objets définis dans d'autres projets, ils sont très souvent utilisés pour décrire les moyens de communication inter-processus sur le NeXT à travers des messages Objective-C.

Il n'est pas nécessaire d'utiliser des protocoles dans un programme en Objective-C, il sont optionels à l'inverse des définitions de classe et des messages. Leur utilisation depend de la tache à accomplir.

0.7.1 Comment utiliser les protocoles

Les protocoles sont utilisés dans au moins trois cituations particulières :

- Pour déclarer de méthodes que d'autre doivent implémenter.

- Pour déclarer l'interface d'un objet sans déclarer sa classe (on cache la classe).

- Pour utiliser des similarités entre classes non reliées dans la hiérarchie.

0.7.2 Méthodes implémentées par d'autres

Si vous connaissez la classe d'un objet vous pouvez regardez son interface et celle des classes dont elle hérite, pour connaître les messages auquel elle repond. Ces déclarations publient les messages qu'elles peuvent recevoir. Un protocole permet de publier les messages qu'elles envoient. La communication est une voie à double sens, les objets ne font pas que recevoir des messages ils en envoient. Par exemple, un objet peut déléguer la responsabilité d'une certaine opération à un autre objet, il il peut, plus simplement, avoir besoin de demander des informations à un autre objet. Dans d'autre cas un objet peut envoyer des messages de notification à d'autres objets pour indiquer ce qu'il fait, afin qu'ils prennent les dispositions nécessaires.

Si vous déclarer la classe de sender (pourvoyeur) et la classe de receiver (receveur) dans un même projet (ou si quelqun d'autre a fournit le receveur et son fichier d'interface), cette communication est facilement coordonné. Le pourvoyeur déclare les méthodes qu'il utilise en important le fichier d'interface du receveur.

Par contre si vous développer un objet qui envoi des messages à des objets que d'autre programmeur écrirons, vous avez besoin d'un moyen de définir les méthodes qu'ils doivents implémenter. C'est le rôle du protocole. Il informe le compilateur sur les méthodes exterieur que la classe utilise et informe les autre programmeurs des méthodes qu'ils doivent fournirent pour que leurs objets fonctionnent avec les votres.

Si un objet envoi des messages à des objets non encore définis il ne peut connaître leurs adresses. Il doit donc fournir un moyen aux autres objets de s'enregistrer comme des receveurs potentiels de ses messages. Supposons par exemple que votre objet utilise une variable d'instance assistant qui sert de cible à tous ses messages. Il doit exister une méthode qui acompagne cette variable d'instance :

- setAssistant:anObject assistant = anObject; return self;

En suite, à chaque fois qu'un message doit être envoyer à l'assistant, une vérification est faite pour vérifier que le receveur implémente la méthode :

- travail if (assistant && [assistant respondsTo:@selector(aider:)]) [assistant aider:self]; return self; return nil;

La méthode aider: doit être définie dans un protocole.

0.7.3 Objets Anonymes

Un protocole peut aussi être utilisé pour déclarer les methodes d'un objet anonyme d'une classe inconnue. An objet anonyme peut representer un service ou un accès à un groupe de fonctions, surtout quand un seul de objet de ce type est nécessaire. (Un objet que les utilisateurs initialisent, ou bien qui joue un rôle fondamental dans l'application ne sont pas des bons candidats à l'annonimité). Les objet ne sont pas anonymes pour leurs développeurs, mais peuvent l'être à leurs utilisateurs quand il sont produit par quelquun d'autre ou qu'il appartiennent à un autre processus!

Dans une bibliothèque d'objet par exemple certains objets ne sont pas nécessairement identifiés par une classe ni même un fichier d'interface. Sans ces informations l'utilisateur ne peut créer d'instances. Le fournisseur doit donc fournir une instance toute faite à la quelle il est possible d'accèder par une méthode définie dans une autre classe:

id formatter = [receiver formattingService];

L'objet rendu par la méthode est un objet sans classe (le fournisseur se réservant le secret). Par contre le fournisseur doit nous donner les messages auxquels cet objet répond pour qu'il nous soit d'une quelquonque utilité. Ceci est fait en associant à l'objet une liste de méthodes déclarées dans un protocole.

Il est possible d'envoyer des messages objetive C à des objet distants appartenant à d'autres applications. (voir "messages à distance" pour plus d'information).

Les autres applications ont leur propre structure, classes et logique interne. Vous n'avez pas besoin de savoir comment l'application marche ni même quels sont ses composants internes pour pouvoir communiquer avec. En tant qu'extérieur tout ce dont vous avez besoin est de savoir quels messages (le protocole) envoyer à qui (le receveur).

Une application qui publie une objet comme un receveur potentiel doit aussi publier le protocole qui déclare les méthodes que l'objet utilisera pour répondre aux messages. Il n'a pas besoin de publier autre chose sur l'objet. L'application cliente n'a pas besoin de connaître la classe de l'objet, ni d'utiliser la classe dans sa propre conception. Elle n'a besoin que du protocole!

Les protocoles permettent les objets anonymes. Sans protocoles il n'est pas possible de déclarer une interface avec un objet sans déclarer sa classe.

0.7.4 Similarités non hiérarchiques

Si plus d'une classe implémente un ensemble de méthodes, ces classes sont souvent regroupées sous une superclasse abstraite qui déclare les méthodes qu'elles ont en commun. Chaque sous classe réimplémente les méthodes à sa façon, mais la hiérarchie d'héritage et les déclarations communes dans la superclasse abstraite capture l'essentiel des similarités entre les sous classes.

Malheureusement il n'est pas toujours possible de regrouper les méthodes communes dans une superclasse abstraite. Des classes sans points communs dans la hiérarchie d'héritage peuvent néanmoins avoir des méthodes très similaires. Ces méthodes similaires peuvent ne pas justifier une relation hiérarchique. Par exemple, de nombreuse classes peuvent utiliser des méthodes qui permettent la gestion du nombre de références:

- setRefCount:(int)count; - (int)refCount; - incrementCount; - decrementCount;

Ces méthodes peuvent être regroupées dans un protocole et les similarités entre classes sera perceptible par le fait qu'elles adoptent toutes les même protocole.

Les objets peuvent ainsi êtres typés par les protocoles auxquels il se conforment plutôt que par leur classes. Par exemple qu'un robinet doivent ce connecté à un autre objet qui peut lui fournir de l'eau. Au lieu de ce connecter à un objet de type tuyaux (un type basé sur une classe), il peut permettre une connection avec tout objet qui implémente les méthodes qui répondent à ses messages (un type basé sur un protocole).

0.7.5 Protocoles informels

Le moyen le plus simple de déclarer un protocole est de regrouper des méthodes dans une déclaration de catégorie de la classe Object:

@interface Object (RefCounting) - setRefCount:(int)count; - (int)refCount; - incrementCount; - decrementCount; @end

Les protocoles informels sont typiquement déclarés comme des catégories de la classe racine Object, comme ceci associe largement les méthodes à toutes les classes qui hérites d'Object. Comme toutes les classes hérites d'Object les méthodes ne sont pas limitées à une partie de la hiérarchie. Cette catégorie n'a pas d'implémentation qui lui corresponde. Ce sont les classes qui implémentent le protocole en déclarant les méthodes dans leur propre interface et en définissant leur implémentation avec leur autres méthodes dans leur propre fichier d'implémentation.

Un protocole informel détourne la règle des déclarations de catégorie pour lister un groupe de méthodes sans les associées avec une classe ou une implémentation particulière.

0.7.6 Protocoles formels

Objective-C permet de définir de façons formel un protocole:

@protocol ProtocolName Déclarations de méthodes @end

Par exemple, le protocole de décompte peut être déclaré:

@protocol ReferenceCounting - setRefCount:(int)count; - (int)refCount; - incrementCount; - decrementCount; @end

Les classes adoptent des protocoles en listant leurs noms entre chevrons après le nom de la super classe:

@interface ClassName : ItsSuperclass ¡protocol list ¿

ainsi que les catégories :

@interface ClassName (Category) ¡protocol list ¿

Les noms de protocoles sont séparées par des virgules. Les méthodes déclarées dans des protocoles adoptés ne sont pas déclarées ailleur dans l'interface de la classe ou de la catégorie.

Par exemple la classe suivante:

@interface Formatter : Object ¡Archives,Enjoliver ¿

adopte les deux protocoles Archives et Enjoliver.

Une classe qui adopte un protocole est obligée d'implémenter toutes les méthodes que le protocole déclare. Le compilateur engendre un Warning sinon. La classe Formatter au dessus doit définir toutes les méthodes déclarées dans les deux protocoles en plus de ces propres méthodes.

0.7.7 Inclusion de protocoles

Un protocole peut lui même utilisé d'autres protocoles en utilisant la même syntaxe.

@protocol UnProtocole ¡protocol list ¿

Tout les protocoles de la liste sont considérés comme une partie de UnProtocole. Quand une classe adopte un protocole elle doit implémenter toutes les méthodes définies dans le protocole et dans les protocoles qu'il contient.

0.7.8 Les Objets protocole.

Sur beaucoup de points les protocoles sont similaires aux classes. Ils contiennent des déclarations de méthodes et à l'exécutions sont représenter par des objets, pour les classes, par des objets de type Class et pour les protocoles, par des instances de type Protocol. Comme les objets classe, les objets protocole sont créés à partir des définitions et déclarations trouvées dans le source et sont utilisés à l'exécution. Il ne peuvent être alloués ou initialisés autrement que par le compilateur.

Une ligne de source peut faire référence à un objet protocole grâce à la directive @protocol() avec le nom du protocole entre parenthèses:

Protocol *counter = @protocol(ReferenceCounting);

C'est l'unique moyen d'accèder à un objet protocole dans le source. Le nom du protocol n'est utilisable que dans cette directive et uniquement par le compilateur.

Le compilateur cré un objet protocole pour chaque protocole qu'il rencontre pourvu que :

- soit le protocole ai été adopté par une classe
- soit que l'on y réfère quelque part dans le source (en utilisant @protocol()).

0.7.9 Se conformer à un protocole

On dit d'une classe qu'elle se conforme à un protocole formel si elle adopte ce protocole ou hérite d'une classe qui a adopté le protocole. Une instance se conforme au mêmes protocoles que sa classe.

Comme une classe doit implémenter toutes les méthodes déclarées dans un protocole qu'elle adopte, ces méthodes sont héritées par les sous classes. D'ou l'on peut dire que une classe se conforme à un protocole si elle implémente toutes les méthodes demandées.

Il est possible de vérifier qu'un objet se conforme à un protocole en lui envoyant un message conformsTo:

if ([receiver conformsTo:@protocol(ReferenceCounting)]) [receiver incrementCount];

Le test conformsTo: est très proche du test respondsTo: pour une seule méthode, excepté qu'il vérifie que le protocole a bien été adopté (et à priori toutes les méthodes qu'il demande) plutôt que de tester si une méthode est bien implémenté ou non. Comme conformsTo: vérifie toute la liste des méthodes ce peut être plus pratique a utiliser. Ce test est devient très utile avec les objets anonimes. Comme la classe de l'objet est inconnue, les protocoles sont le seul moyen de publier (ou découvrir) les messages que l'objet anonyme peut réalisé. C'est une bonne pratique de tester si l'objet est conforme au protocole avant de lui envoyer des messages.

0.7.10 Vérification de type

Les protocoles peuvent servire de type pour les objets et permettent ainsi une autre forme de vérification de types par le compilateur.

Dans une déclaration de type les protocoles sont indiqué après le nom de type entre chevrons :

- (id ¡Formatting¿)formattingService; id ¡ReferenceCounting, AutoFreeing¿anObject;

Come le typage statique permet de tester la concordance des types avec la hiérarchie des classes cette syntaxe permet de vérifier qu'un type se conforme à un protocole.

Par exemple, si Formatter est une super classe abstraite, la déclaration

Formatter *anObject;

regroupe tous les objets qui héritent de Formatter dans un même type et le compilateur vérifie sur les affectation la concordance du type.

De même la déclaration,

id ¡Formatting¿anObject;

regroupe tous les objets qui se conforment au protocole Formatting en un type, indépendamment de leur possition dans la hiérarchie de classe. Le compilateur vérifie par rapport à ce type toutes les affectations.

Dans les deux cas le type regroupe des objets par leurs similarités soit par un ancêtre commun ou par une similarité de service.

0.8 Options Statiques

Les Objets sont des entités dynamiques. Un maximum de décisions sont reportées à l'exécution:

- Il sont dynamiquement alloués à l'exécution par des méthodes de classe qui crées de nouvelles instances.

- Les objets sont dynamiquement typés. Dans le source (a la compilation), n'importe quel objet peut être typé id indépendamment de sa classe. La classe exacte d'une variable de type id (et donc sa structure) n'est pas déterminée avant l'exécution du programme (et peut changer au cours du temps)

- Les messages sont associés aux implémentations au moment de l'appel, voir "la gestion des messages" plus haut. C'est une procédure qui associe sélecteur et implémentation.

Tous ces points donne à Objective-C une grande flexibilité et une grande puissance, mais il y a un prix. Les messages sont plus lent que les appels de fonctions, la vérification de typage est plus laxiste sur les variables de type id. Comme dans certaines occasions, ces points sont moins importants que la vitesse et la précision, Objective-C vous permet de ne pas utiliser ces options orientées objet et de transferrer les decisions de l'exécution vers la compilation.

0.8.1 Le typage Statique

Si un nom de classe est utilisé à la place de id pour une déclaration d'objet:

Véhicule *lObjet;

cela restreind la variable aux instances de la classe et de ces sous classes. Dans l'exemple lObjet doit être un Véhicule quelconque.

Bien que lObjet soit typé statiquement, il est alloué dynamiquement par la même méthode de classe que pour une variable typée id. Comme id est un pointeur sur objet lObjet doit être déclaré comme un pointeur sur un véhicule.

Les objets typés statiquement ont la même structure interne que les objets déclaré id. Le type n'a pas d'influence sur l'objet. Mais le compilateur accède à plus d'informations sur l'objet. Et cela n'affecte en rien la façon dont l'objet est traité à l'exécution. Les messages sont toujours envoyés de façons dynamique (avec calcul).

En donnant plus d'information au compilateur sur un objet, le typage statique ouvre des possibilitées qui sont absentes pour les objets typé id:

- Dans certaines situations, cela permet de réaliser des vérifications dés la compilation.

- Cela permet de s'affranchir de la contrainte : les méthodes de même nom doivent avoir des arguments et des valeurs de retour identiques.

– Cela vous permet d'accèder directement aux champs de la structure de l'objet.

Ce troisième cas a été vue dans "accès aux variables d'instance", les deux premiers sont vu ci-dessous.

0.8.2 Vérifications de typage

Avec les informations suplémentaires fournies par le typage statique, le compilateur peut fournir une meilleur vérification de type dans deux situations :

- Quand un message est envoyé à un receveur typé, le compilateur peut vérifié que ce receveur peut répondre. Un Warning est émis si le receveur ne peut accèdé à la méthode nommé dans le message.
- Quand un objet typé est affecté à une variable typé, le compilateur vérifie leur compatibilité. Un Warning est émis quand il ne le sont pas.

Le typage reste cohérent avec la hiérarchie de classes. Comme c'est illustré dans l'exemple suivant :
View *uneView; Matrix *uneMatrix;
uneMatrix = [[Matrix alloc] init]; uneView = uneMatrix;

Ici uneMatrix peut être assigné à la variable uneView car Matrix est une sorte de View, la classe Matrix hérite de View. Par contre les rôles ne peuvent être inversés car une View n'est pas une Matrix. Dans ce cas le compileur émet un message d'erreur (toutes les view ne sont pas des Matrix).

Il n'y a pas de vérification quand une des expressions de l'affectation est typé id. Un objet typé peut être librement affecté à une variable id, est un id à une variable statiquement typé. Du fait que des méthodes comme alloc et init retourne des id, le compilateur ne vérifie pas que l'objet retourné placé dans une variable typée est compatible. Le code suivant dangereux est néant en moins correct :

Matrix *aMatrix; aMatrix = [[Window alloc] init];

Remarque: Ceci est en harmonie avec l'implémentation de void * (pointeur sur void) du C ANSI. De la même façons que void * est générique et permet d'éviter la coercicion dans les affectations entre pointeurs, id est un type générique qui permet d'éviter l'opération de coercition dans une class spéciale en cas d'affectation entre objets.

0.8.3 Type des arguments et valeurs de retour

En général les méthodes qui partage le même sélecteur (le même nom) doivent aussi partager les mêmes arguments et les même valeurs de retour. Cette contrainte est imposé par l'édition de lien dynamique. Le fait que la classe du receveur est inconnu à la compilation et donc les réactions spécifique de la classe à la méthode ne sont pas connues, le compilateur est obligé de traiter toutes ces méthodes de la même façon. Quand le compilateur préparer les informations sur le type des arguments et de la valeur de retour pour le noyau d'exécution, il crée une seule description de méthode pour chaque sélecteur.

Par contre, quand le message est envoyé à un objet statiquement typé, la classe du receveur étant connue par le compilateur, la restriction est levée.

0.8.4 Le typage statique avec une classe héritée

Une instance peut être statiquement typé avec sa propre classe ou par une des classes dont elle hérite. Toutes les instances peuvent être typé comme Object. Le compilateur ne fait de vérification de typage que sur le nom de class et fait ces vérifications en fonction de cela. Typer une instance dans une classe héritée peut donc aboutir à des différences entre ce que le compilateur pense sur se qui va arriver à l'exécution et ce qui se passera réelement.

Par exemple, si vous typez statiquement une instance de Matrix en View,
View *maMatrix = [[Matrix alloc] init];

le compilateur va traiter cette instance comme une View. Si vous envoyez à cette Matrix un message spécifique aux Matrix,

id cell = [maMatrix selectedCell];

le compilateur va emettre un message comme quoi cet objet ne peut répondre à ce sélecteur ce qui est faux! Ce qui est vrai c'est que le sélecteur selectedCell est défini dans la classe Matrix mais pas dans la classe View.

Par contre, si vous envoyer un message avec un sélecteur défini dans View il n'y aura pas de message bien que Matrix a sa propre méthode.

[maMatrix display]; et à l'exécution c'est la méthode de Matrix qui seras réalisée.

Similairement, suspposons que notre classe Haut définisse une méthode inquiet qui retourne un float,

- (float)inquiet;

et que la classe Milieu sous classe de Haut redéfinisse la méthode avec un nouveau type de valeur de retour:

- (int) inquiet;

Si une instance est typé statiquement comme instance de la classe Haut, le compilateur va supposer que la méthode inquiet retourne une float, si l'instance est de type Milieu il pensera que inquiet retourne un int. Des erreurs vont évidemment apparaître si une instance de Milieu est typé en Haut. Le compilateur va informer le noyau d'exécution que la valeur retournée est un float, mais à l'exécution se sera un int et une erreur.

Le typage statique peut lever cette contraite sur les méthodes homonymes uniquement si elle sont définie dans des branches diférente de la hiérarchie de classes.

0.8.5 Optenir l'adresse d'une méthode

Le seul moyen de circonvenir l'édition de lien dynamique est d'optenir l'adresse d'une méthode et de l'apeller directement comme une fonction. Ceci peut être utile si la méthode doit être appellée de façons répétitive un grand nombre de fois, et que vous voulez éviter le coût de la gestion de message à chaque fois. En utilisant la méthode définie sur la classe Object methodFor: vous pouvez demander un pointeur sur la procédure qui implémente la méthode, et d'utiliser en suite ce pointeur pour appeler la procédure. Le pointeur que retourne methodFor: doit être proprement casté sur le bon type de fonction, le type des arguments et de la valeur de retour doivent être explicités.

L'exemple suivant montre comment la méthode setTag: peut être appellée :

id (*setter)(id, SEL, int); int i;

setter = (id (*)(id, SEL, int))[target methodFor:@selector(setTag:)];

for (i = 0; i ¡1000, i++) setter(targetList[i], @selector(setTag::), i);

Les deux premier arguments passés à la procédure sont le receveur (self) et le sélecteur (_cmd) . Ces argument qui sont cachés dans la syntaxe des methodes doivent être utilisé explicitement quand la méthode est appellée comme une fonction.

L'utilisation de la méthode methodFor: pour circonvenir l'édition de lien dynamique sauve presque tous le temps perdu en gestion de message, mais le gain ne devient sensible uniquement quand un message se vois répété un grand nombre de fois comme dans la boucle précédente.

0.8.6 Manipuler la structure de donnée d'un objet

Un des points de fort de la programmation orienté Objets est que les structure de données d'un objet lui sont privées. Les informations stockés dans l'objet ne sont accessible que par l'envois de messages. Malgré cela il existe un moyen de dépouiller la structure de son coté objet et de la manipuler comme une structure C. Ceci rend évidement public tous les variables d'instance.

Quand on donne à la directive @defs() un nom de classe elle produit la liste des déclarations pour une instance de la classe. Cette liste n'est utile que pour déclarer des structure et ne peut donc apparaître que dans le corps d'une déclaration de structure. Le code suivant, déclare une structure qui est identique la structure des instances de la classe Robinets:

struct robinetDef @defs(Robinet) *publique;

Ici publique est déclaré comme un pointeur sur une structure indistinguable d'une instance de Robinet. Avec un peu de coercition une instance de Robinet peut être affectée au pointeur et manipulée par ce biais:

id unRobinet; unRobinet = [[Robinet alloc] init];

publique = (struct robinetDef *)unRobinet; publique-¿eauOuverte = OUI;

0.9 Codage des types

Pour simplifier les outils du noyau d'exécution le compilateur encode le type de la valeur de retour et des arguments de chaque méthode dans une chaîne associée au sélecteur de la méthode. Ce système de codage des types peut être utilisé dans d'autres contextes et est rendu publique par la directive de compilation @encode(). Cette directive transforme une spécification de type en une chaîne représentant ce type. Le type peut-être un type de base, un pointeur une structure une union ou un nom de classe tous ce qui peut être tulisé comme argument de l'opérateur sizeof() du C. `char *buf1 = @encode(int **);`
`char *buf2 = @encode(struct key);`
`char *buf3 = @encode(Matrix);`
La Table sur la page suivante les codes des types. Notez que beaucoup de ces codes sont identique a ceux utilisés pour les entrées sorties formattées. Attention ces codes ne peuvent être utilisés pour les entrée/sorties formattées.

Code Meaning
c A char, caractères
i An int, entiers
s A short, entier cours
l A long, entier long
C An unsigned char, caractère non signé
I An unsigned int, entier non signé
S An unsigned short, entier court non signé
L An unsigned long, enteir long non signé
f A float, réel
d A double, réel double précision
v A void
* A character string (char *), pointeur sur char
@ An id
# A class object (Class), un objet de typoe class
: A method selector (SEL), un sélecteur de méthode

...

An array, un tableau
{...} A structure, une structure
(...) A union, une union
bnum A bitfield of num bits, un champs de bit de num bits
^type A pointer to type, un pointeur sur type
? An unknown type, un type inconnu

Pour une tableau le nombre d'élements du tableau est indiqué juste après le crochet ouvrant et le type des élménts en suite. Un tableau de douze pointeurs sur float est encodé par :

$$12^\wedge f$$

Les structures sont indiquées par des accollades, on indique le nom de la structure après l'accolade puis les types des champs dans l'ordre. Par exemple la structure :

```
typedef struct exemple {
    id unObjet;
    char *unechaine;
    int unEntier;
} Exemple;
```

sera encodé par :
`{example=@*i}`
Le même encodage est optenu en utilisant comme paramètre de encode le nom du type (Exemple) ou le nom de la structure (exemple). Un pointeur sur une structure contient autant d'informations:

`^ {example=@*i}`
mais un niveau d'indirection de plus fait disparaitre les spécification internes :

`^^{example}`
Les objets sont traités comme des structures. Par exemple l'objet class Object est encodé :

`{Object=#}`
c'est à dire que c'est une instance de la classe Class de nom Objet.

0.10 Langage Synopsis

Objective C apporte au C quelques constructions et une poigné de conventions pour interragir effectivement avec le noyau d'execution. Cette section liste toutes les additions au langauge C sans rentrer dans les détails. Pour plus d'information voir les sections précédentes.

0.10.1 Messages

Les messages sont écrits entre crochets.
[receiver message]
Le receveur peut-être

- Une variable ou expression qui a pour valeur un Objet.
- Un nom de classe indiquant un objet de type Class
- super (indiquant une autre forme de recherche de l'implémentation)

Le message est le nom de la méthode plus les arguments.

0.10.2 Les types prédéfinis

Les types principaux utiliés dans ObjectiveC sont définis dans objc/objc.h . Ils sont :

id Un Object (un pointeur sur sa structure) Class Un objet Class (un pointeur sur la structure de type Class) SEL Un sélecteur, un code définis par le compilateur qui identifie la méthode. IMP Un pointeur sur une implémentation qui retourne un id. BOOL Une Valeur Booléen, YES ou NO

De plus il est possible d'utiliser les noms de classes comme des types ce qui permet de typer statiquement les instances de la classe. Une instance peut être typé comme un pointeur sur sa classe ou sur une classe dont elle hérite.

0.10.3 Les directives du préprocesseur

Le préprocesseur comprend les notations suivantes :
#import importe un fichier d'interface une et une seule fois.
// Commentaire jusqu'a la fin de la ligne

0.10.4 Directives de compilation

Toutes les directives commence par "@":
@interface Débute une déclaration d'interface de classe ou de catégorie @implémentation Débute la définition d'une classe ou d'une catégorie @protocol Débute une déclaration de protocole @end Termine la déclaration/définition d'une Classe/catégorie/protocole @class Declares Le nom de classes définies ailleurs. @selector(method) Retourne le sélecteur compilé de la méthode. @protocol(name) Retourne le nm du protocole (une instance de la classe protocole). @private Limite la porté des variables à la classe qui les déclarent. @protected Limite la porté des variables à la classes et aux classes qui en hérite (par défaut). @public Retire les limitations de porté des variables. @encode(spec) Fournit une chaîne de caractères qui encode le type. @defs(classname) Fournit le structure de donnée interne des instances de la class classname.

0.10.5 Classes

Une nouvelle classe est déclarée avec la directive @interface. Elle importe le fichier d'interface de sa superclass:
#import "saSuperclasse.h"
@interface maClass : saSuperclass ¡une liste de protocole ¿ déclarations des variables d'instance déclarations des méthodes @end
La liste de protocoles est optionnelle. Si les deux point et le nom de la super classe sont oubliés, la classe est une nouvelle classe racine.
Une définition de classe importe sa propre interface
#import "maClasse.h"
@implémentation maClasse définitions des méthodes @end

0.10.6 Catégories

Une catégorie est déclarée comme une classe. En important la classe sur laquelle elle est déclarée:
#import "maClasse.h"
@interface maClasse (maCategory) ¡une liste de protocole ¿déclarations des méthodes @end
Comme pour une classe, une définition de catégorie importe son propre fichier d'interface:
#import "maCategory.h"
@implémentation maClasse (maCategory) définitions des méthodes @end

0.10.7 Protocoles formels

Les Protocoles formels sont déclarés en utilisant la directive @protocol
@protocol unProtocol ¡une liste de protocole ¿déclarations des méthodes @end
Dans le code, on fait références aux protocoles avec la même directive @protocol() avec le nom du protocole entre parenthèses.
Les listes de protocole entre chevrons (¡. . . ¿) peuvent être utilisé pour trois choses différentes:

- Dans une classe ou une catégorie pour adopter le(s) protocole(s)
- Dans un déclaration de protocole pour inclure d'autres protocoles
- Dans une spécification de type, pour limité le type aux objets qui se conforment au protocole
-

0.10.8 Déclarations de Méthodes

Les conventions suivantes sont utilisées dans les déclarations de méthodes

- Un signe "+" précède les déclarations de méthodes de Classe.
- Un signe "-" précède les déclarations de méthodes d'instance.
- Les arguments sont indiqués après un deux-points (:). En général, un nom descriptif précède le deux-points. Les noms et les deux-points sont considérés comme faisant parti du nom de la méthode.
- Le type des arguments et des valeurs de retour sont donnés avec la syntaxe de coercition du C.
- Le type par défaut pour les paramètres et la valeurs de retour est le type id, il reste int pour les fonctions comme en C.

0.10.9 Implémentation des Méthodes

A chaque méthode est passé deux paramètres cachés :

- L'objet receveur (self)
- Le sélecteur de la méthode (_cmd)

Dans l'implémentation, self et super se réfère à l'objet receveur, super remplacent self quand seul les méthodes hérités sont à utiliser pour répondre au message.

0.10.10 Conventions sur les noms

Les fichiers qui contiennent de l'ObjectiveC ont une extention".m "
Les noms de classes commencent par une majuscule, les instances par une minuscule. Les noms de classe sont globaux (des externes en C).
Dans une classe peu de contrainte sur les noms:

- Une classe peut déclarer des noms de méthodes déjà utilisés dans d'autres classes
- Une méthode de classe peut avoir le même nom qu'une méthode d'instance
- Une méthode peut avoir le même nom qu'une variable d'instance.

0.10.11 D'autres Mots Clefs

Le fichier ¡objc/objc.h¿défini aussi deux termes utiles:
nil Un pointeur sur objet nul, (id)0 Nil Un pointeur sur Classe nul, (Class)0