

project : Ω

Project:Omega's Objective-C Crash-Course

Pejvan BEIGUI , pejvan@projectOmega.org

Project:Omega

Version 0.2 (draft)

Copyright © 2002 Pejvan BEIGUI for [Project:Omega](http://www.ProjectOmega.org) (<http://www.ProjectOmega.org>)

Table of Contents

1. Disclaimer	2
2. Pourquoi ce document ?	2
3. Introduction	3
4. Avant de commencer	3
5. Notions de base de la POO	4
5.1. Les Objets	4
5.2. Les Classes	4
5.3. La notion d'héritage	4
5.4. Un premier exemple	4
6. De plein pied dans Objective-C	5
6.1. Les Objets	5
6.2. Les Messages	5
6.3. Les Methodes	6
6.4. Les Classes	8
6.5. Les Objets Class	9
6.6. Création d'instances	10
6.7. Les Variables de Classe	10
7. Définir les classes en Objective-C	11
7.1. La directive import	11
7.2. Définir l'interface	11
7.3. L'implémentation	12
7.4. Accès et portée pour les variables d'instances	14
7.5. Sur la propagation des messages	15
7.6. La directive @selector	15

Abstract

Ce document à pour but d'introduire le plus rapidement possible les concepts de base d'Objective-C afin de permettre une entrée en matière rapide dans le monde de ce langage. Comme tout crash-course, il ne se veut pas complet et n'est pas du tout une référence de l'objective-C. Je vous invite à consulter la courte bibliographie si vous cherchez un document de référence.

Les prérequis sont assez nombreux mais rien d'exceptionnel pour quelqu'un qui à un minimum d'expérience de la programmation, surtout de la *POO (Programmation Orientée Objets)*. Bon alors ? Quels sont les prérequis ? Le seul prérequis essentiel, sans lequel il est inutile de continuer la lecture de ce document, c'est la *connaissance du C*. Si le lecteur n'a pas de connaissance sur ce langage, nous lui conseillons vive-

project : Ω

ment de lire l'[introduction au langage C](http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html) (http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html), de Bernard Cassagne. Connaître les bases de la programmation orientée objets, et avoir de l'expérience avec des langages objets tels le *C++* ou le *Java* est bien sûr un grand avantage.

De plus, nous considérerons que le lecteur possède au moins une idée intuitive des concepts orientés objets tels que objets, les classes d'objets ou l'héritage. Nous donnerons les éventuels détails que nous jugerons nécessaires à la bonne compréhension du document, mais nous ne détaillerons pas en détail les concepts de la POO.

1. Disclaimer

Ce document est encore à l'état de 'draft' (brouillon). Il n'est pas encore 'finalisé' et il lui manque encore beaucoup de choses, qu'il va me falloir du temps à ajouter. Je le mets toutefois déjà à disposition en espérant que : 1- des personnes compétentes en Objective-C le liront et me feront des commentaires sur les éventuels erreurs, typos, etc. qui se trouvent dedans ; 2- les personnes qui ne connaissent pas encore Objective-C me feront part de leur besoin, de ce qui va et de ce qui va moins dans le texte (qu'est-ce qui est clair ? c'est quoi ce truc que je ne comprends pas ???) et qu'ils apprendront quand même pas mal de choses sur Objective-C, même si leur connaissance de sera pas suffisante pour connaître les aspects les plus complexes et exotiques du langage.

De plus, n'étant pas un grand spécialiste mondial de l'Objective-C, il y aura forcément des imprécisions et des erreurs dans le texte. Je fais tout mon possible pour que cela ne se produise pas, mais dans le cas contraire, j'espère ne pas en être tenu rigueur.

2. Pourquoi ce document ?

La principale raison pour laquelle j'ai entrepris d'écrire ce document fut mon incapacité à trouver des documents et exemples utiles sur le net, au moment où j'ai entrepris d'apprendre ce langage. En fait, avec un minimum de recherche plus pointues et plus dirigées, on peut trouver des choses assez sympathiques, telles que cette URL : [GNU Objective-C runtime features](http://gcc.gnu.org/onlinedocs/gcc-2.95.3/objc-features.html) (<http://gcc.gnu.org/onlinedocs/gcc-2.95.3/objc-features.html>).

Après avoir cherché en vain un document de référence sur Objective-C, je me suis résigné à lire l'"Object-Oriented Programming and the Objective-C Language" disponible sur le site d'Apple [ici](http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/) (<http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/>).

Finalement, à force de chercher, j'ai fini par trouver dans [les pages du manuel de gcc 3.02](http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc_2.html) (http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc_2.html) :

“

There is no formal written standard for Objective-C. The most authoritative manual is "Object-Oriented Programming and the Objective-C Language", available at a number of web sites; <http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/> has a recent version, while <http://www.toodarkpark.org/computers/objc/> is an older example. <http://www.gnustep.org> includes useful information as well.

”

Ce qui signifie en fait il n'existe pas de standard pour Objective-C et qu'aucune référence écrite ne lui est consacrée.

Enfin... Étant donné que j'ai eu beaucoup de mal à trouver le peu d'information dont je dispose, et que j'ai perdu beaucoup de temps en recherches inutiles, j'ai décidé d'écrire le document dont j'avais rêvé. J'espère un jour pouvoir atteindre ce but. En attendant, j'espère que ce document vous sera utile.

3. Introduction

Objective-C est un sur-ensemble du *C*. Il a été créé pour fournir des capacités orientées objets au *C*. Mais, contrairement au *C++*, ces possibilités ont été ajoutées de manières simples et directes. Ansi, ces extensions sont très peu nombreux et leur syntaxe est basée sur *SmallTalk*, l'un des premiers langages objets.

Sans vouloir rentrer dans des détails trop complexes, *Objective-C* peut aussi être considéré comme un sur-ensemble du *C++*, on parle alors d'*Objective-C++*. En effet, *C++* n'est pas un pur langage objets, et il lui manque l'*aspect dynamique*.

Objective-C essaie de rendre les programmes aussi dynamique et fluide que possible et de supprimer les éléments limitants que sont les contraintes de compilation et d'édition de liens : par exemple, lors de la compilation, l'ensemble des types utilisable est défini, et il est alors impossible d'introduire de nouveaux types. De même, il serait impossible d'inclure de nouveaux modules pendant que le programme tourne.

Trois types de dynamisme sont particulièrement importants en programmation orientée objets (POO) :

- Le *typage dynamique (dynamic typing)* qui permet d'attendre l'exécution du programme pour déterminer le type d'un objet.
- Le *liage dynamique (dynamic binding)* qui permet de déterminer pendant l'exécution quelle méthode (ou fonction membre) invoquer.
- Le *chargement dynamique (dynamic loading)* qui permet de charger de nouveaux composants pendant l'exécution du programme.

Nous ne rentrerons pas plus en détail dans ces concepts, dans l'état actuel de ce document.

Cet aspect dynamique qui va caractériser les langages objets, nécessite un *environnement d'exécution* (communément appelé *runtime*) dans lesquels les programmes vont s'exécuter. Nous aborderons rapidement (brièvement) quelques aspects de base du runtime, des possibilités de dynamisme d'*Objective-C* au cours de notre lecture.

4. Avant de commencer

Le compilateur que nous utiliserons est gcc sur MacOS X 10.1. La version actuelle est la 2.95.2 :

```
pejvan% cc --version  
2.95.2
```

Sauf erreur de ma part, il existe deux implémentations distinctes d'Objective-C dans gcc. L'une faite par la FSF, dont la classe de base est `Object`, et qui possède son propre runtime. L'autre faite par Apple Computer Inc. dont la classe de base est `NSObject` et qui possède un autre runtime aussi. Mais nous reviendrons sur ces sujets un peu plus loin

Les fichiers sources en Objective-C sont décomposés de la même manière que les fichiers C ou C++ : un fichier entête, ou header avec une extension `.h` et contenant les déclarations et un fichier source, avec une extension `.m` contenant les définitions.

Voilà, on est prêt à entrer dans le vif du sujet ;-)

5. Notions de base de la POO

5.1. Les Objets

Un objet, en terme de POO, représente une abstraction, dont une entité est appelée *instance*. Chaque instance contient différentes variables locales à l'objet, appelés *variables d'instances* et des fonctions, locales à l'objet aussi, appelés *méthodes* ou *fonctions membres* suivant le langage (la dénomination *méthode* provenant du Java il me semble, tandis que *fonction membre* provient du C++). Nous utiliserons indifféremment les deux ici.

5.2. Les Classes

Une classe est un ensemble regroupant tous les mêmes objets qui sont du même type. Un objet est alors une instance d'une certaine classe. De même qu'il existe des variables d'instances, il existe des *variables de classe* et des *fonctions de classes* ou *méthodes de classes*. Tous les objets d'une même classe se partagent alors *une seule et même variable et/ou fonction*.

5.3. La notion d'héritage

Une dernière chose à connaître est le concept d'*héritage*. Pour aller le plus rapidement possible, le fait d'hériter d'une certaine classe permet de *transmettre* l'ensemble des attributs d'une classe (appelée *super-classe*) à une autre (appelée *sous-classe*). Cette notion très intuitive vous sera plus claire au fur et à mesure des exemples et de vos propres expériences.

5.4. Un premier exemple

Si vous n'avez pas tout compris ici, ce n'est pas grave. Les exemples qui suivront de-

project : Ω

vront rendre les choses plus claires. Expliquer en détail les concepts de la POO sort du cadre de ce document. Je vous conseille donc vivement de consulter un livre de POO si vous voulez bien maîtriser le sujet.

Un exemple pour mieux comprendre : unRectangle peut être une *instance* de la classe rectangle. Longueur et largeur seront les *variables d'instance* de unRectangle et getLongueur et getLargeur seront ses *fonctions membres*. De plus, la classe rectangle *dérive* (ou *hérite*) d'une classe figure.

6. De plein pied dans Objective-C

6.1. Les Objets

En objective-C, tous les identifiants d'objets possède un type distinct des variables du C : le type `id`. Ce type est défini comme un pointeur vers un objet. Pour déclarer un objet sans spécifier à l'avance la classe, il suffit alors de taper :

```
id unObjetQuelconque ;
```

Il est à noter que le type `id` est le type par défaut dans les constructions objets, alors que le type `int` reste le type par défaut des constructions C. Par exemple, le type de retour par défaut des méthodes est `id` et celui des retours de fonction C est `int`. De même, le type `nil` vient se substituer au type `null`. Ainsi, `nil` est défini comme un pointeur l'objet nul, ou encore un `id` de valeur zéro.

Si vous avez bien suivi la petite note sur le dynamisme, au début de ce document, vous avez peut être déjà compris le grand avantage de cette façon de faire : un objet déclaré comme `id` pourra alors contenir n'importe quel objet. On obtient donc le concept de *typage dynamique*. A cette fin, tout objet possède une variable d'instance `isa` ('est un' en français) permettant d'identifier la classe à laquelle l'objet appartient.

Bien sûr, il est aussi possible (et même vivement conseillé, pour des raisons de rapidité d'exécution, et d'erreur pendant l'exécution) de spécifier les classes à la compilation, pour les variables qui ne vont pas contenir des objets de différentes classes pendant l'exécution. C'est ce que l'on appelle le *typage statique* (*statically typing*). Exemple :

```
rectangle unRectangle ;
```



Bon à savoir : Contrairement à C++ et Java qui permettent d'accéder directement aux variables de classes, sans passer forcément par des méthodes, Objective-C nécessite forcément l'utilisation de fonctions membres pour y accéder, que cela soit pour la modification des valeurs ou simplement la lecture. Cela permet, entre autres, de coder plus proprement, et d'avoir un code plus facilement modifiable.

6.2. Les Messages

project : Ω

On présente souvent et FAUSSEMENT les messages comme étant la même chose que l'appel une fonction membre. Bienque la différence soit subtile, il convient de bien faire la différence pour comprendre correctement les notions de dynamisme. Je vais essayer de d'expliquer les messages dans le contexte de l'objective-C et de faire comprendre EN MÊME TEMPS la subtilité dont je parlais.

Pour qu'un objet fasse quelquechose, il faut que vous lui envoyiez un *message* lui disant d'exécuter une méthode. On voit déjà que *message* et *méthode* ne sont pas la même chose. L'envoi de message se fait grâce à une expression entre crochets :

```
[recepteur message]
```

Nous verrons un peu loin les détails de la syntaxe pour l'appel de méthodes et l'envoi de messages.

Le recepteur est un objet, et le message lui dit quoi faire. Dans le source code, le message est simplement le nom d'une méthode et l'ensemble de ses paramètres. Quand un message est envoyé, c'est le runtime qui choisi la méthode appropriée et l'invoque. Le nom de la méthode ne sert qu'a *sélectionner* l'implémentation d'une méthode. C'est pourquoi les noms de méthodes dans les messages sont souvent appelés *sélecteurs* (*selectors*).

C'est là que nous allons parler de *liaison dynamique* (*dynamic binding*). Une différence cruciale entre les appels de fonctions et les messages est le fait qu'une fonction et ses paramètres sont codés ensembles dans le fichier source et réunis dans le code compilé alors qu'un message et le récepteur associé ne sont pas unis jusqu'à l'exécution du programme et l'envoi du message. Ainsi la méthode qui va être invoquée en réponse à un message est déterminée à l'exécution et non lors de la compilation.

La méthode exacte qui va être invoquée dépend du récepteur. Différents récepteurs peuvent avoir différentes implémentations pour une méthode portant le même nom (c'est ce que l'on appelle le *polymorphisme*). La sélection de la bonne implémentation se fait lors de l'exécution. C'est l'*environnement d'exécution* (*runtime*) qui fait ce travail.

Nous reviendrons un peu plus en détail sur le travail du *runtime* et sur ses conséquences sur l'écriture de vos programmes. Nous ne rentrerons pas en plus dans les détails de la gestion des messages, mais j'espère que j'ai réussi à faire comprendre ce qu'était la *liaison dynamique* (*dynamic binding*).

Pour les méthodes qui retourne des valeurs, on peut imbriquer les messages. Ça donne quelquechose de plus ou moins lisible, mais ça peut être pratique dans certaines situations. Voici quelques exemples :

```
[[unRectangle alloc] init];
int lg = [[unRectangle SetLongueur:8] GetLongueur];
// SetLongueur change la longueur du rectangle a 8, et retourne l'objet unRectangle
// GetLongueur recupere la longueur du rectangle unRectangle, sous forme d'entier,
// et la retourne. L'entier lg est alors affecte de la valeur de longueur (ici 8).
```

6.3. Les Methodes

Nous avons déjà vu brièvement comme fonctionnaient les messages, dont voici un exemple :

project : Ω

[recepteur message]

Nous allons voir maintenant comment définir les méthodes (les déclarer) et les appeler. Ceci est un peu prématuré dans la progression de ce document, mais je trouvais cela important de faire une mise au point la dessus avant d'aborder les classes. Faisons donc vite ET bien ;-) Nous allons faire des analogies avec le C++ et le Java dans la section sur les Classes.

Comme toute fonction, les méthodes peuvent avoir une valeur de retour. Il faut donc spécifier ce type. Par contre, conformément à ce que nous avons vu précédemment, si aucun type n'est spécifié, c'est le type `id` par défaut, donc `id` qui est implicitement utilisé. Le type de retour d'une méthode est spécifié entre parenthèses, avant le nom de celle-ci. Exemples (non compilables car les déclarations sont incomplètes – cf section Classe) :

```
(id) init; // type de retour id et sans parametres en entree
init;     // la meme mais avec la valeur de retour par default
(int) GetLongueur; //methode retournant un entier (int) et
              //sans parametres en entree
```

Nous allons voir maintenant comment spécifier les paramètres en entrée des méthodes. Une fonction avec un argument se déclare avec le signe `:` après le nom, on spécifie ensuite le type du paramètre d'entrée, de la même manière que nous avons spécifié celle de retour de la méthode. Enfin, on donne le nom du paramètre. Exemples :

```
(int) SetLargeur:(int) valeur; // methode prenant un unique parametre:
                               // l'entier valeur
                               // et retournant un entier aussi.
```



A noter que dans les méthodes, les `:` font partie intégrante du nom. Par exemple, le nom de la méthode dans l'exemple précédent n'est pas `SetLargeur` mais plutôt `SetLargeur:`. De même, nous verrons des noms tels que `uneMethode:::`.

Là où ça se complique un tout petit peu, c'est quand les méthodes ont plusieurs paramètres. La spécification veut que l'on commence la déclaration comme une fonction à un seul paramètre, puis pour chaque paramètres supplémentaire, il faut ajouter un caractère d'espace, une étiquette (pas obligatoire), `:`, type du paramètres (ex : `(int)`), nom du paramètre. Cela peut paraître compliqué ainsi dit, mais voici un exemple qui devrait rendre les choses plus claires :

```
(id) init:(int)longueurInit largeurInitLabel:(int)largeurInit
// methode retournant un type id, prenant deux parametres :
// le premier est un int : longueurInit,
// le second est aussi un int : largeurInit
// l'etiquette du second parametre est largeurInitLabel

(id) init:(int)longueurInit :(int)largeurInit
// la meme chose, sans l'etiquette
```

Les messages correspondant aux méthodes à plusieurs paramètres s'écrivent alors naturellement :

```
[unRectangle init:100 largeurInitLabel:50]; //on peut laisser l'etiquette
[unRectangle init:100 :50]; //ou non ;-)
```



Ici le nom de la méthode est `init::` et non `init`.

On peut aussi avoir des méthodes avec un nombre variable d'arguments (même si ces fonctions sont plutôt rares). Leur déclaration est alors similaires aux fonctions à nombre variables d'arguments du C. On utilise une virgule et une ellipse `...`. Voici un exemple avec l'envoi de message correspondant :

```
faireGroupe:groupe, ... ; //declaration. un argument minimum
[recepteur faireGroupe:groupe, premierMembre, secondMembre];
```

6.4. Les Classes

6.4.1. Un petit retour sur la notion d'héritage

[Explications sur l'héritage à venir (je sais pas trop quand encore :-/)]

6.4.2. La classe racine

Le système d'héritage lie toutes les classes hiérarchiquement. En Objective-C, contrairement au C++ et au Java, toutes les classes dérivent d'une seule et même classe, la classe située tout en haut de la hiérarchie, appelée *classe racine* (*root class*).

Dans l'implémentation GNU d'Objective-C, cette classe racine est la classe `Object` (`<objc/Object.h>`) alors que dans l'implémentation d'Apple, basée sur la `Foundation Framework` a pour classe racine `NSObject` (`<Foundation/NSObject.h>`).

Comme nous l'avons dit précédemment, la classe racine n'a pas de superclasse. Par contre, c'est la classe dont TOUTES les autres classes héritent (et dérivent). C'est elle qui définit toutes les interactions de base pour les objets Objective-C et les interactions entre objets. C'est elle qui fournit la capacité aux classes qui en héritent de se comporter comme des objets et de coopérer avec l'environnement d'exécution.

Récapitulons : *Toute classe possède comme superclasse la classe racine. Si une classe n'a besoin d'hériter d'aucune autre classe, elle doit quand même hériter de la classe racine*, en effet, tout objet de toute classe doit avoir au moins la capacité d'agir comme un objet Objective-C pendant l'exécution.



Une autre solution consisterait à réécrire une classe racine. On ne va pas réinventer la roue, surtout si on risque de commettre des erreurs. Écrire une classe racine est une tâche délicate et dangereuse à plus titre. De plus, toutes les fonctions de base de la classe racine telle que l'allocation des instances, leur connexion à leur classe et leur identification auprès de l'environnement d'exécution devraient être réécrites. Sans parler des pro-

project : Ω

tocoles à implémenter. Bref, il est fortement déconseillé de tenter d'écrire une classe racine, et à l'état actuel de mes connaissances, je ne sais pas le faire.

6.4.3. Introspection

Les instances de classes peuvent fournir leur type pendant l'exécution. Les méthodes `isMemberOfClass:` et `isKindOfClass:` sont définies par la classe racine à cette fin. `isMemberOfClass:` est utilisé pour savoir si l'instance appartient à une certaine classe :

```
if ( [unRectangle isMemberOfClass:Rectangle] )
    return [unRectangle GetLargeur];
else
    return 0;
```

`isKindOfClass:` est utilisé pour savoir si l'instance appartient à une certaine classe ou appartient à une classe qui dérive de cette classe :

```
if ( [unRectangle isKindOfClass:Rectangle] )
    [...]
```



Il existe d'autres fonctions d'introspection que nous verrons plus loin dans le document.

6.5. Les Objets Class

Une définition de classe va contenir diverses informations nécessaires au runtime et au compilateur. La plupart des ces informations sont sur les instances de cette classe. Par exemple :

- le nom de la classe et de sa superclasse
- une description de ses variables d'instance
- les prototypes de ses méthodes
- l'implémentation des méthodes

Toute cette information est compilée et stockée dans des structures de donnée que le runtime va utiliser. *Le compilateur crée un objet pour représenter la classe.* Cet objet, appelé *Objet Classe* a accès à toutes les informations sur la classe et a à la capacité de créer de les nouvelles instances conformes à la description fournit dans la définition de la classe.

Cet objet classe (qui n'est pas une instance de la classe) contient les variables et les méthodes de classe. Attention à bien faire cette distinction. Comme tout autre objet, l'objet classe hérite des variables et méthodes de classe des classes dont il dérive. Dans ce cas, c'est le nom de la classe qui est utilisé comme receveur lors des envois de messages à l'objet classe. De même, il est possible d'utiliser le type `id` pour

project : Ω

désigner une classe objet. Exemples :

```
int nbRect = [Rectangle howMany]; //howMany retourne la variable de
//qui compte le nb d'objets instanciés

id MaClasse = [Rectangle class]; //class retourne la classe de l'objet
//pour tout objet (meme les objets classe)
```



Attention : le nom de classe ne peut être utilisé que dans les messages où la classe objet est receveur du message. Par exemple, il ne peut être désigné comme argument d'un autre message. Mais le nom de classe peut être employé comme type dans les déclarations, comme vous pourrez le voir dans le prochain exemple.

6.6. Création d'instances

L'intérêt des classes est leur capacité à créer des instances. On crée une instance en envoyant le message `alloc` à l'objet classe. La méthode `alloc` alloue dynamiquement de la mémoire pour les variables d'instance du nouvel objet et les initialise à zéro (excepté la variable `isa` dont nous avons déjà parlé et qui sert à connecter l'instance à sa classe). S'il est besoin de faire plus d'initialisations que la mise à zéro des données, il faut utiliser la méthode `init`. Exemples :

```
Rectangle * rectangle1 = [Rectangle alloc] ; //typage statique
id rectangle2          = [Rectangle alloc] ; //typage dynamique

Rectangle * rectangle3 ;
rectangle3 = [[rectangle alloc] init] ; //allocation et initialisation
```



Un point qui est un peu hors sujet mais que je voulais toutefois aborder concerne les conventions de nommage. La convention adoptée en Objective-C veut que les noms de classes commencent par une majuscule, alors que les noms d'instances commenceront par une minuscule. Pour les noms correspondant à la concaténation de plusieurs autres noms, on mettra une majuscule à la première lettre de chaque début de mot (sauf à la première lettre du nom bien sûr)

6.7. Les Variables de Classe

Pour que l'ensemble des instances d'une classe puisse partager une seule et même variable, il faut que cette variable soit définie comme une variable de classe, et en Objective-C, on réalise ceci grâce au mot clef `static` du C (les variables de classe de C++ sont aussi introduites par le mot clef `static`). Il faut sans doute aussi définir les méthodes qui peuvent les manipuler. Déclarer une variable comme étant `static` dans le même fichier de définition que la classe limite sa portée à la classe et à ses instances. Attention toutefois que la variable n'est pas héritée lors des dérivations sauf si ces dérivations sont définies dans le même fichier ; ce qui peut être parfois

problématique.

[à détailler, et fournir des exemples, ce qui manque dans la doc d'Apple...]

7. Définir les classes en Objective-C

Le plus gros travail du développeur qui fait de la POO, c'est la conception de ses classes. Il faut beaucoup de modélisation et de conception avant de commencer à réellement coder. Mais une fois cette phase de modélisation passée, la plus grande partie du code est la partie où le programmeur déclare puis définit ses classes. En Objective-C comme en C++, les classes sont déclarées puis définies dans deux fichiers distincts : un fichier d'entête (le fichier header, .h) qui contient la déclaration des variables, des méthodes, la superclasse etc. et un fichier de source (le fichier .m) qui contient la définition des classes (soit en fait la source des méthodes).

Cette décomposition en différents fichiers n'est pas imposée, mais on la fait pour permettre la séparation de l'interface de l'implémentation. Ainsi, il est classique de fournir uniquement les sources des fichiers d'interface au personne qui ont besoin n'ont pas besoin de l'implémentation pour : 1- laisser les détails d'implémentation privées, afin de permettre des mises à jour du code plus simple ; 2- de ne pas avoir à distribuer le code source si celui-ci n'est pas libre. De même, l'usage veut que chaque fichier ne déclare ou n'implémente qu'une seule classe, et que le nom du fichier corresponde au nom de la classe déclarée ou implémentée suivit du suffixe du fichier.

7.1. La directive import

Chaque fichier d'interface doit importer les fichiers dont il dépend. Cela se réalise grâce à la directive `import` (`#import`). Les habitués de Java reconnaîtront cette directive alors que les habitués du C/C++ sont plutôt utilisateurs de la directive `include` (`#include`). `#import` est normalement équivalent à `#include` sauf qu'elle fait en sorte que chaque fichier soit inclus au plus une seule fois. C'est pourquoi `#import` est plus pratique qu' `#include`.



Les version non modifiées de gcc n'aiment apparemment pas la directive `#import` (pour la petite histoire, il paraîtrait que c'est monsieur RMS qui ne les aime pas) et donc il génère un warning à chaque fois qu'il en rencontre un. Pour les supprimer, il suffit d'utiliser l'option `-Wno-import` de gcc.

7.2. Définir l'interface

La définition de l'interface commence par `@interface` et se termine par `@end`. La première ligne déclare la nouvelle classe et celle dont elle dérive. Sans cette super-classe, il y a déclaration d'une classe racine (avec tous les problèmes qu'il y a gérer). Ensuite il faut déclarer les variables d'instance entre accolades et enfin on déclare les méthodes. Un exemple va clarifier tout ça :

```
#import <Foundation/Foundation.h> //on importe les entetes necessaires

@interface Rectangle : NSObject //@interface NomClasse ':' SaSuperClasse
{
    // entre accolades, les variables d'instance
    int longueur;
    int largeur;
    int perimeter;
}

//puis on declare les methodes :
// - pour les methodes d'instance
// + pour les methodes de classe

- (id) init;
- (id) init: (int)longueurInit :(int)largeurInit;
- (int) getLongueur;
- (int) getLargeur;
- (int) getPerimeter;
- (int) getLargeur:(int) valeur;
- (int) getLongueur:(int) valeur;
- (id) printSelf;
@end
```

7.2.1. le role de l'interface

[détails à venir bientôt]

1. [héritage -- à rédiger]
2. [besoins du compilateur -- à rédiger]
3. Grâce à la liste des méthodes, déclarées dans le fichier interface, chaque module sait quel message peut être envoyé à quel objet. De même, le développeur qui lit ce fichier connaît toutes les méthodes auxquelles il a accès. Delà, il y a une déduction tout à fait logique à faire : toutes les méthodes qui sont de type publique sont dans ce fichier, et les méthodes privées ne doivent donc pas y être. *Donc les méthodes privées ne doivent pas être déclarées dans le fichier interface. C'est la seule façon de déclarer les méthodes privées.*

7.3. L'implémentation

L'implémentation n'est pas très compliquée non plus. Il faut d'abord importer l'entête correspondant à l'interface (`#import NomDeLaClasse.h`) puis il faut mettre `@implementation`, exactement comme précédemment on avait mis `@interface`, suivi du nom de la classe. Il faut ensuite définir toutes les méthodes. Enfin, on fini par `@end`. Exemple incomplet (ne pas oublier que `-` et `+` désignent respectivement méthode d'instance et méthode de classe):

```
#import "rectangle.h"
@implementation rectangle

- (id) init
{
    [...]
}
```

project : Ω

```
}  
- (id) init:(int)longueurInit :(int)largeurInit  
{  
  [...]  
}  
  
- (int) getLongueur  
{  
  [...]  
}  
  
- (int) getLargeur  
{  
  [...]  
}  
  
- (int) getPerimeter  
{  
  [...]  
}  
  
- (int) setLongueur:(int) valeur  
{  
  [...]  
}  
- (int) setLargeur:(int) valeur  
{  
  [...]  
}  
  
- (id) printSelf  
{  
  [...]  
}  
@end
```

Il ne reste donc plus qu'à voir les détails de la définition des méthodes, et c'est bon pour l'implémentation :-). Les méthodes sont définies un peu comme les fonctions C : dans un bloc délimité par des accolades (les accolades étant précédées de la signature de la méthode. Encore une fois, j'espère que l'exemple suivant va rendre les choses plus compréhensibles :

```
#import "rectangle.h"  
@implementation rectangle  
  
- (id) init  
{  
  self=[super init];  
  largeur = 6;  
  longueur = 8;  
  perimeter = 2*largeur + 2*longueur;  
  return self;  
}  
  
- (id) init:(int)longueurInit :(int)largeurInit  
{  
  self=[super init];  
  largeur = largeurInit;  
}
```

```

    longueur = longueurInit;
    perimeter = 2*largeurInit + 2*longueurInit;
    return self;
}

- (int) getLongueur
{return (longueur);}

- (int) getLargeur
{return (largeur);}

- (int) getPerimeter
{return (perimeter);}

- (int) setLongueur:(int) valeur
{
    longueur = valeur;
    perimeter = 2*largeur + 2*longueur;
    return largeur;
}

- (int) setLargeur:(int) valeur
{
    largeur = valeur;
    perimeter = 2*largeur + 2*longueur;
    return largeur;
}

- (id) printSelf
{
    printf("longueur : %d \t largeur : %d\n", [self getLongueur] , [self getLargeur]);
    return self;
}
@end

```

7.4. Accès et portée pour les variables d'instances

La POO utilise comme l'un de ses principes de base l'*encapsulation des données*. Cette encapsulation a pour but de permettre à une classe (et donc aux objects, instances de cette classe) de cacher ses données, ceci pour éviter que lors des révisions et mises à jour de la classe, les modifications internes n'entraînent pas de problèmes aux personnes qui auraient utilisé cette classe.

Le compilateur va donc se charger de limiter la portée de certaines variables et méthodes. Mais pour des raisons de flexibilité, l'utilisateur (c'est à dire la personne qui définit la classe) va pouvoir définir explicitement cette portée. Comme en Java et en C++, il existe trois niveaux de portée, que le programmeur choisit par des directives au compilateur :

@private	la variable n'est accessible qu'à l'intérieur de la classe qui la déclare
@protected	la variable est accessible dans la classe qui la déclare, et les sous-classes de cette classe. C'est la portée par défaut pour les variables en Objective-C
@public	la variable est accessible partout.

Ces directives s'appliquent à toutes les variables qui la suivent, jusqu'à ce qu'une nouvelle directive soit définie.

project : Ω



Des variables d'instances marquées comme `@public` empêche l'objet de cacher ses données. Cela va à l'encontre des principes de bases de la POO. C'est pourquoi les variables publiques devraient être évité autant que faire se peut, et ne les utiliser que dans des cas exceptionnels.

7.5. Sur la propagation des messages

Comme nous l'avons déjà vu avant, les messages ne sont pas relié à une implémentation de méthode avant l'exécution. C'est ce que l'on a appelé le *dynamic binding*. A cette fin, le compilateur va convertir les messages [recepteur message] en une appelle de fonction : `objc_msgSend()` qui va s'occuper s'occuper de faire le *dynamic binding*. Anssi, un appel tel que `objc_msgSend(recepteur, selecteur, arg1, arg2, ...)` va :

les 3 étapes pour la fonction `objc_msgSend()`

1. Trouver la bonne procedure à appeler (l'implémentation de la méthode) à laquelle se réfère le selecteur. En effet, il peut y avoir plusieurs méthodes portant le même nom (polymorphisme).
2. Appeler la procedure, lui passer un pointeur vers l'objet recepteur, de même que les arguments de la méthode.
3. Enfin, il passe la valeur de retour de la procédure comme sa propre valeur de retour



C'est au compilateur de générer ces appels de fonction. Vous ne devez jamais le faire directement dans votre code.

Nous ne rentrerons pas plus en détail sur la compilation en structures pour les classe et les objets, et pour la manière dont se crée la *dispatch table*, etc.

7.6. La directive `@selector`

Pour des raisons d'efficacité, ce ne sont pas les noms en ASCII qui sont utilisés pour les sélecteurs de méthode dans le code compilé. A la place de ça, le compilateur range le nom de chaque méthode dans une table, et le relie avec un idenficateur unique qui va le représenter pendant l'exécution. L'environnement d'exécution s'assure que chaque identificateur est unique. Les sélecteurs compilés ont un type particulier : SEL, pour les distinguer des autres types de données. Un selecteur valide est différent de 0. Il est bien entendu inutile d'assigner vous-même des selecteurs aux méthodes, cela ne fonctionnera pas.

La directive `@selector` permet de se référer directement au selecteur compilé et non au nom de la méthode. Nous allons voir sur les exemples suivants comment cela fonctionne. Ici nous allons assigner le selecteur `setLongueur:` est assigné a la vari-

able setLongueur :

```
SEL setLongueurSelecteur ;  
setLongueurSelecteur = @selector(setLongueur:);
```



Les selecteurs identifiés toujours les noms des méthodes, et non pas leurs implémentations. Sinon, il n'y aurait plus de différence entre un appel de fonction et un envoi de message. L'utilité principale de @selector étant la possibilité de pouvoir créer des messages au cours d'exécution ou bien pour choisir le message à envoyer en cours d'exécution.