Developpez.com
Club des développeurs

De Java à Cocoa

par Sylvain Gamel (Page d'accueil) (Blog)

Date de publication : 26/05/2008

Dernière mise à jour : 26/05/2008

Cet article, en plusieurs parties, permettra aux développeurs Java de pouvoir facilement passer à Cocoa et Objective-C.

Introduction ... 3
I - Classes d'objets et héritage ... 3
I-A - Le rôle d'une classe ... 3
I-B - Comment créer une classe ? ... 4
I-C - Étendre une classe ... 7
I-D - Pour finir ... 8
II - Définir les attributs d'une classe ... 8
II-A - Déclarer des variables d'instance ... 9
II-B - Et les variables de classe ? ... 10
II-C - Visibilité des attributs ... 12
II-D - Utilisation des attributs ... 12
II-E - Pour finir ... 13
III - Définir les méthodes d'une classe ... 13
III-A - Généralités ... 13
III-B - Déclaration des messages ... 15
III-C - Définition des méthodes ... 15
III-D - Appel de méthode ... 16
III-E - Pour finir ... 16
IV - Initialiser classes et objets ... 16
IV-A - Initialisation de la classe ... 17
IV-B - Construction d'instance ... 18
IV-C - Destruction d'une instance ... 20

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

Introduction

On présente souvent Objective-C comme un dérivé du C. C'est syntaxiquement exact, mais Objective-C tient beaucoup de son grand père Smalltalk, tout comme Java.

Java est aujourd'hui bien plus populaire que le C et attire certainement une audience un peu plus jeune.

Bien que je possède une longue expérience du C et du C++, il me semble plus approprié d'apprendre Objective-C en prenant Java comme point de référence. Tous deux sont de vrais langages à objets et tous deux offrent des comportements bien plus dynamiques que C++.

Autre différence primordiale entre C/C++ et Java/Objective-C : *les bibliothèques de développement*.

Alors que le C n'est généralement utilisé qu'avec la bibliothèque standard C. C++ fait de même avec la STL (1) . Mais aucune de ces deux bibliothèques ne permet de développer une application complète.

Les bibliothèques de développement ne sont pas aussi fortement liées au langage que dans Java et Objective-C. C'est tellement vrai pour ce dernier qu'il est souvent confondu avec Cocoa, la bibliothèque de composants du Mac héritée de NeXTStep/OpenStep.

Java et Objective-C partagent donc nombre de caractéristiques.

Il me semble approprié de présenter le couple Objective-C et Cocoa en mettant en avant les similitudes avec Java et en soulignant si besoin leur différences.

I - Classes d'objets et héritage

Java comme Objective-C sont des langages à objets. Comme beaucoup des langages de cette famille, ils dépendent d'un système de classe.

Cet article présente la syntaxe de création d'une classe en Objective-C, en la comparant avec celle de Java.

I-A - Le rôle d'une classe

La classe est un moule à partir duquel on peut construire des objets : les instances d'une classe.

La classe couvre différents objectifs dont les plus significatifs sont :

- définir les données manipulées par un objet ;
- définir les services fournis par un objet ;
- masquer les détails d'implémentation en *encapsulant* les données et les services techniques ;

Donnée	Service
attribut, propriété, variable	méthode, message

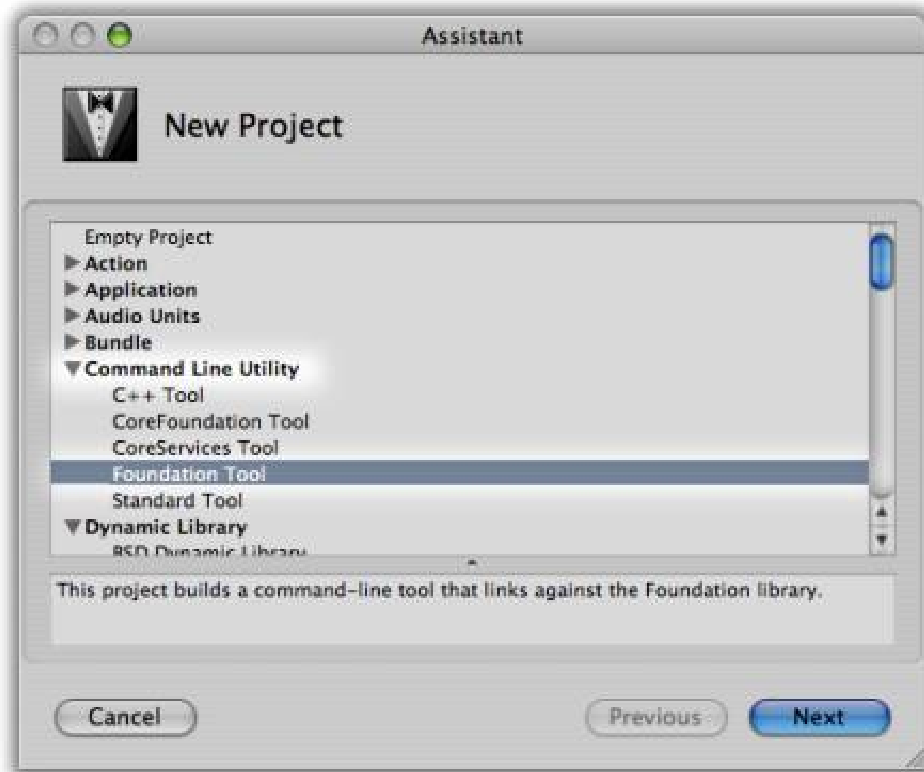
Une classe peut atteindre ses objectifs en s'appuyant sur des outils techniques.

- des règles de visibilité pour définir qui peut voir ses attributs et méthodes ;
- l'héritage défini la généalogie d'une classe ;
- le *polymorphisme* pour se restreindre à utiliser une interface plutôt qu'une implémentation spécifique.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

I-B - Comment créer une classe ?

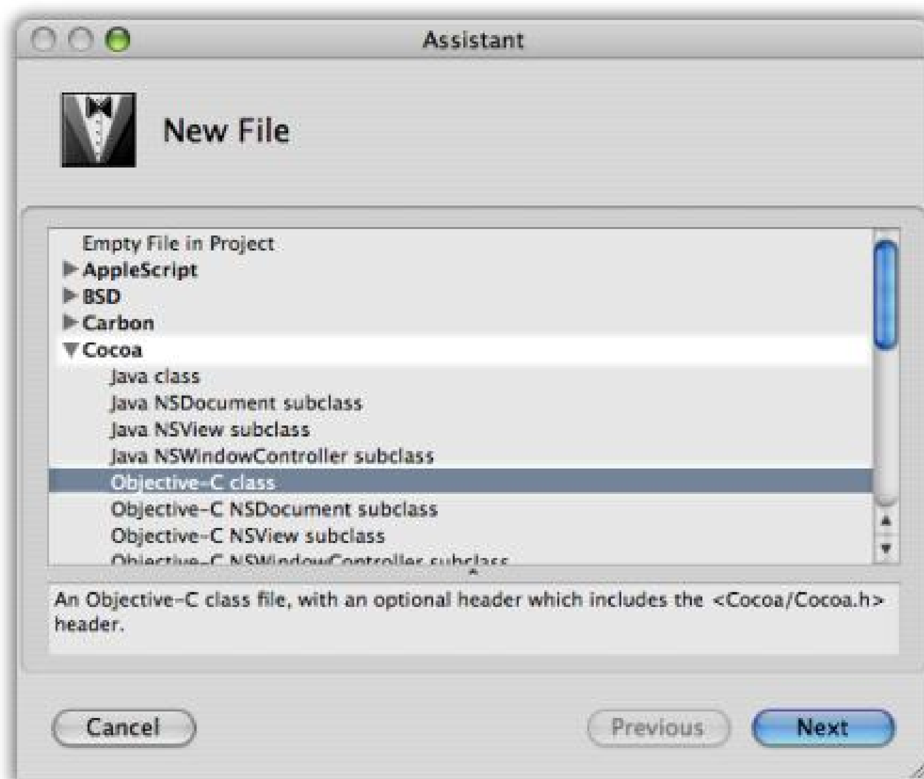
La méthode la plus simple pour commencer est de créer un projet de type *Foundation Tool* dans la famille *Command Line Utility*.



Pour créer une classe utilisez l'assistant de création :

- Allez dans le menu File, entrée New File...
- Choisissez alors dans le groupe *Cocoa* l'entrée *Objective-C Class*.
- Donnez un nom à votre classe.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com



XCode vous crée un squelette de classe vide sous la forme de deux fichiers. Dans mon cas j'ai créé la classe *SimpleClass* :

- *SimpleClass.h* défini l'interface de ma classe.
- *SimpleClass.m* contiendra l'implémentation de ma classe.

Les nouveau fichiers apparaissent dans le groupe du projet sur la gauche. Je les ai déplacé dans le groupe " Source " pour mieux organiser mon projet.

La première différence par rapport à Java saute aux yeux : *Là où Java mélange dans un fichier unique interface et implémentation, Objective-C sépare les deux dans des fichiers indépendants.*

Cette séparation est un héritage direct du langage C au dessus duquel est construit Objective-C. L'avantage de cette solution est de pouvoir utiliser une classe sans disposer de son code source. Le fichier en-tête est suffisant pour le compilateur et décrit l'ensemble de l'interface publique de la classe et de ses objets.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

L'interface d'une classe est définie dans le fichier d'en-tête *.h* (2)

```
#import <Cocoa/Cocoa.h>

@interface SimpleClass : NSObject {
// Déclaration des attributs associés à la classe
}
// Déclaration des méthodes associées à la classe
@end
```

L'implémentation de la classe est dans un fichier *.m* (3)

```
#import "SimpleClass.h"

@implementation SimpleClass

// Implémentation des différentes méthodes
// déclarées dans le fichier en-tête

@end
```

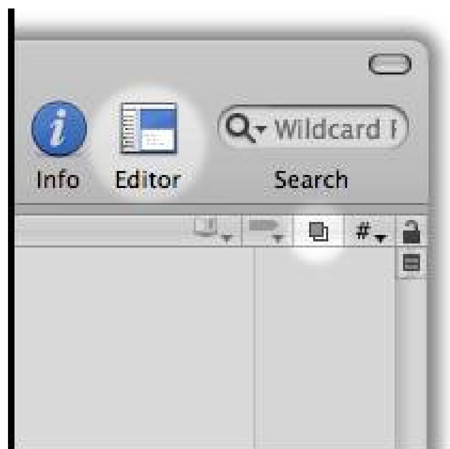
La directive #import indique au compilateur qu'il lui faut inclure le fichier en-tête indiqué. Ainsi, l'implémentation de notre classe importe le fichier en-tête qui lui est associé.

En Java le code aurait été réduit à un simple fichier *.java* :

```
public class SimpleClass {
// Définitions des attributs et méthodes
}
```

Deux outils indispensables à connaître :

1 dans la barre d'outils, le bouton pour passer dans l'éditeur pour le fichier sélectionné ;
2 Dans l'éditeur, en haut à gauche, un bouton en forme de double carré permet de passer de l'interface à l'implémentation et inversement.



Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

En résumé, une classe se déclare ainsi :

<table>
<tr><th>Java</th><th>Objective-C</th></tr>
<tr><td><pre>class UneClasse</pre></td><td><pre>@interface UneClasse
...
@end

@implementation UneClasse
...
@end</pre></td></tr>
</table>

I-C - Étendre une classe

Objective-C supporte le même modèle d'héritage que Java, à savoir une seule classe mère pour une classe donnée.

Contrairement à Java, il n'existe pas une hiérarchie unique pour la généalogie des classes. Au contraire, toute nouvelle classe peut former la racine d'une nouvelle arborescence de classes.

Le squelette de classe généré par XCode indique automatiquement que votre nouvelle classe dérive de la classe racine de Cocoa : NSObject. Hériter de NSObject n'est donc qu'une facilité et en aucun cas une obligation. Cependant, le type d'objet par défaut ne défini aucun comportement, ce qui fait que Objective-C dépend très fortement du framework applicatif utilisé.

```
@interface SimpleClass : NSObject {
 ...
```

Ce qui en Java aurait pu s'écrire :

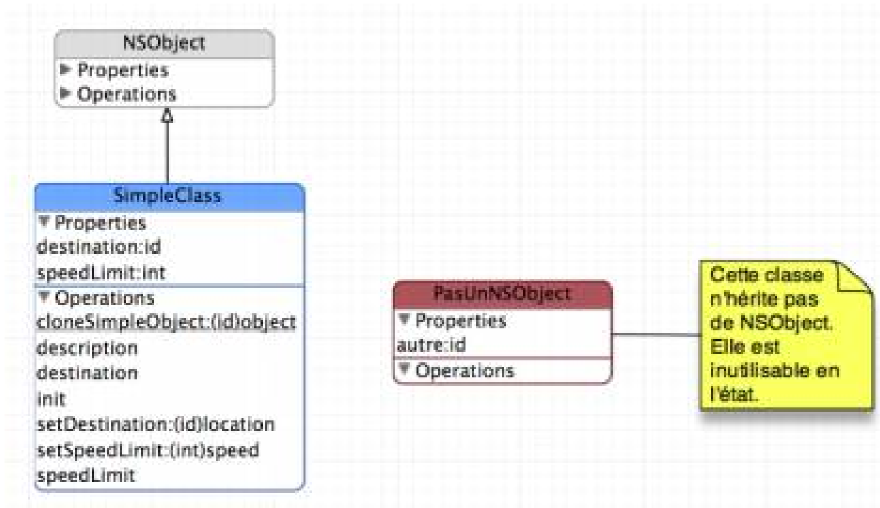
```
public class SimpleClass extends java.lang.Object {
 ...
```

Mais en Java, toute classe dérivant obligatoirement de la classe Object, cet héritage n'est généralement pas spécifié explicitement par les développeurs. Cette facilité devra être évitée à tout pris en Objective-C sous peine de voir son code ne pas réussir à fonctionner correctement.

Pour manipuler des objet le langage Objective-C utilise le type id qui est une référence sur un objet de type indéterminé.

Même si le langage permet des arbres de classes distincts de celui dont la racine est NSObject, il est fortement déconseillé de se construire sa propre hiérarchie de classes. Objective-C n'a que peu d'intérêt s'il n'est pas utilisé avec le framework Cocoa, et dériver ses classes à partir de NSObject et le meilleur moyen pour en profiter (4)

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com



Si l'héritage simple peut paraître trop limité par rapport à un modèle d'héritage multiple tel que celui proposé par C ++, nous verrons que comme en Java, Objective-C utilise la notion d'interface. Cela permet sans compromettre la simplicité de supporter des modèles complexes.

Héritage	Java	Objective-C
Classe Racine	java.lang.object	NSObject (par facilité, mais pas forcément)
Déclaration	class MaClasse **extends** ClasseMere	@interface MaClasse : ClasseMere
Héritage Implicite	Oui, de java.lang.Object	Non

I-D - Pour finir

Nous savons maintenant définir une classe.

Cette classe ne nous sert pas à grand chose pour l'instant et il nous reste encore quelques étapes avant de la rendre réellement utilisable :

- définir des attributs pour les instances ;
- définir les méthodes (services) proposés par les objets ;
- construire et initialiser des instances de cette classe.

II - Définir les attributs d'une classe

Les variables présentent l'état d'un objet. On les appelle propriétés, attributs, données membre. Cocoa retiendra plutôt le terme propriété alors que Java réserve ce terme aux attributs des Java Beans.

Un objet fourni un jeu de services, mais pour pouvoir être implémentés ces services dépendent de l'état courant de l'objet.

C'est le rôle des attributs d'un objet que de définir son état à un instant donné.

Nous allons donc voir ici comment nous allons définir les attributs de nos objets Objective-C.

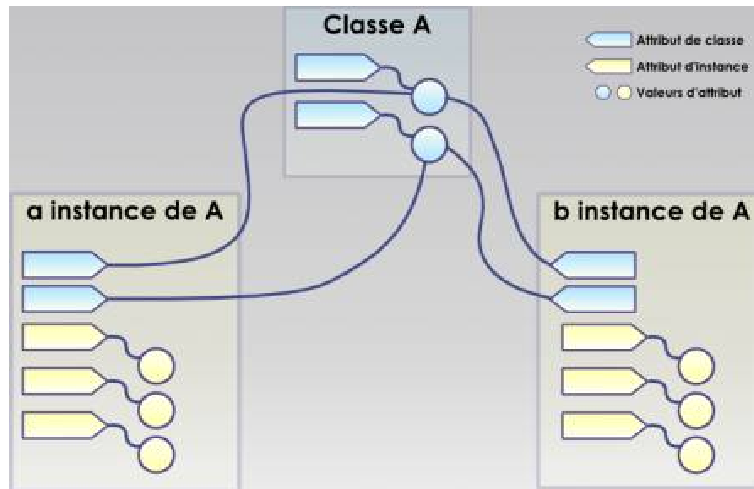
Pour rappel, il est important clarifier ce qu'est une variable. *Une variable n'est qu'une étiquette qui est rattachée à une valeur.*

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

Un objet peut manipuler des variables d'instances et de classes :

- les variables de classes ont une valeur définie dans la classe.
- les variables d'instance ont une valeur défini dans l'objet.

Le schéma suivant représente cette structure :



On y voit clairement que si chaque instance partage le même jeu de noms de variables, ces variables ont toutes des valeurs différentes.

Au contraire, les variables de classes ont une valeur unique partagée entre toutes les instances de la classe.

II-A - Déclarer des variables d'instance

Les variables sont déclarées dans l'interface de la classe.

```
@interface MaClasse {
 // Déclaration des variables d'instance entre
 // les accolades
}
...
```

La syntaxe est celle des déclarations C normales. Pour plus de détails je vous renvoie donc à la documentation de référence disponible sur le site Apple ou dans votre installation d'XCode.

Notre classe *SimpleClass* va définir deux attributs :

- un identifiant vers un objet destination ;
- un nombre indiquant une limite de vitesse.

```
@interface SimpleClass : NSObject {
   id destination;
   int speedLimit;
```


Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

```
}
```

En Objective-C, contrairement à Java, il est obligatoire d'utiliser explicitement la syntaxe C des pointeurs lorsqu'on déclare une variable de type objet :

```
(TYPE) * variable;
```

La seule exception étant l'utilisation du type id qui est implicitement un pointeur sur un type quelconque d'objet : un équivalent du void *, mais restreint aux objets.

En effet, comme nous le verrons en parlant du cycle de vie des objets, l'initialisation suivante n'est pas valide puisque les méthodes *alloc* et *init* retournent toutes deux un *id* qui est un type *pointeur* sur objet.

```
// Définition invalide
NSString chaine = [[NSString alloc] init];
```

Seule l'expression suivante permet de construire une chaîne :

```
// Définition correcte
NSString * chaine = [[NSString alloc] init];
```

II-B - Et les variables de classe ?

Pour un développeur Java, les attributs d'une classe se divisent en deux groupes :

- les *attributs des instances* de la classe : chaque instance de la classe partage le même ensemble de variables, mais chaque instance leur associe des valeurs distinctes.
- les *attributs de la classe* : les attributs de classe sont partagés entre toutes les instances de la classe.

C'est le modèle classique présenté au début de cet article.

En Objective-C les variables de classe n'existent pas. Elles sont en fait inutiles car le langage C permet d'obtenir la même fonction par l'utilisation de variables privées dans un module.

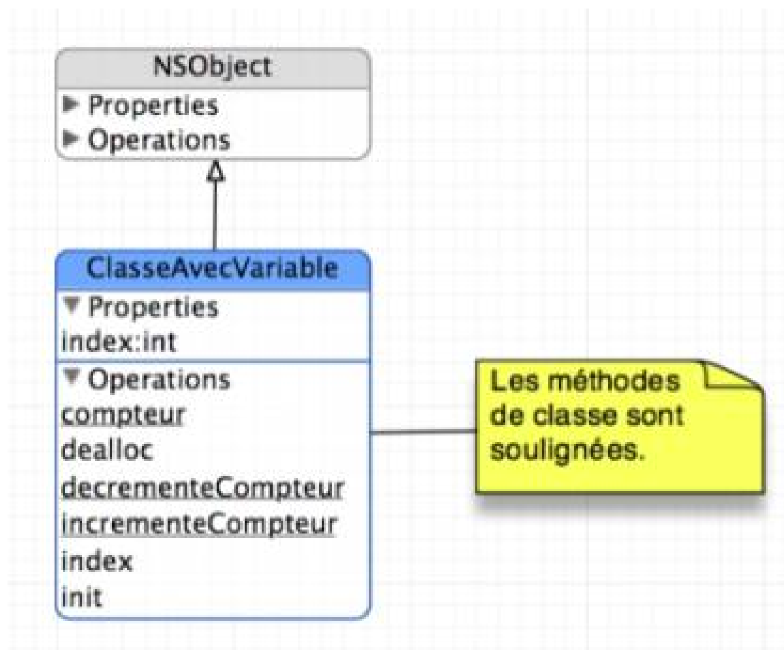
Capable de posséder ses propres variables et sa propre portée, le fichier module joue le rôle de conteneur que la classe joue en Java.

Pour illustrer ce concept nous allons créer une nouvelle classe ClasseAvecVariable dont l'interface est la suivante :

```
@interface ClasseAvecVariable : NSObject {
   @private
   int index;
}
/* ... */
@end
```

La variable de classe n'est pas définie dans l'interface. Elle ne le sera que par les méthodes de classes pour y accéder.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com



Dans l'implémentation nous déclarons la variable de classe comme une variable globale mais dont la portée est restreinte au fichier source courant.

```
#import "ClasseAvecVariable.h"

static int compteur = 100; // Variable privée au module

@implementation ClasseAvecVariable
/* ... */
@end
```

Nous remarquerons que la variable étant d'un type simple nous pouvons combiner en une seule ligne la déclaration et la définition de la variable.

Pour les objets, il est possible de les initialiser avec une méthode d'initialisation de la classe. Ce sujet sera traité avec le cycle de vie des objets.

Cette même classe pourrait être définie de façon plus concise en Java comme suit :

```
public class ClasseAvecVariable {
 private static int compteur = 100; // Variable de classe
 private int index; // Variable d'instance
 // ...
}
```

Il est tout de même important de remarquer que si l'utilisation de variables privée à un module semble similaire à une variable de classe, nous n'avons pas vraiment une stricte équivalence avec Java :

- Comme en Java un variable de classe si elle est surchargée, elle n'en est pas pour autant capable de polymorphisme.
- Contrairement au Java, une variable de classe est systématiquement privée. Les variables de classes ne sont ni protégées, ni publiques. Une classe peut cependant être associée à des variables globales publiques déclarées dans son fichier en-tête et définies dans le module d'implémentation.

Il est possible de restreindre la visibilité d'une variable externe à un module, mais cela implique de définir un fichier en-tête spécifique et complique peut-être inutilement le code.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

Variable de classe	Java	Objective-C
Déclaration	Source Java	Fichier en-tête
Visibilité	Publique, protégée, ou privée	Privée
Syntaxe	static TYPE var ;	static TYPE var ;

II-C - Visibilité des attributs

En Java, mais aussi en Objective-C, les variables de classes peuvent profiter de trois niveaux de visibilité :

1. *Privée*, la variable n'est visible que dans la classe qui la déclare.
2. *Protégée*, la variable est visible dans la classe qui la déclare ainsi que dans ses classes dérivées.
3. *Publique*, la variable est visible à partir de toute partie de code.

Java et Objective-C définissent la visibilité d'une variable membre différemment :

- Java défini la visibilité pour *chaque variable* en ajoutant un qualificatif de portée à la déclaration.
- Objective-C défini des *sections* dans la déclaration des variables, chaque section définie le niveau de portée des variables qui y seront déclarées.

En cela, Objective-C ressemble d'avantage à C++ qu'à Java.

Les mots clefs utilisés sont similaires.

Portée	Java	Objective-C
Privée	private	@private
Protégée	protected	@protected
Publique	public	@public

II-D - Utilisation des attributs

Dans une méthode de la classe, les attributs sont accessibles comme des variables normales. Elles sont dans la portée de la classe et à se titre sont directement accessibles.

Nous pourrions donc écrire la méthode suivante pour la classe SimpleClass :

```
(id) init {
 [super init];
 destination = location;
 speedLimit = limit;
 return self;
}
```

Si l'on n'est pas dans la portée de la classe, par exemple lorsqu'on accède à une variable publique, il faut utiliser la notation classique du C pour accéder à une valeur d'un type structuré. Les objets étant déclarée par des pointeurs sur le type, c'est l'opérateur flèche d'indirection de pointeur qui sera utilisé :

```
MaClasse * objet;
// ...
```


Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

```
objet->variable = valeur;
```

II-E - Pour finir

Nous sommes maintenant capable d'ajouter des données à nos objet en leur associant des variables d'instance ou de classe.

Mais nos classes resteront inertes tant qu'elles ne proposeront pas des services. Ces services sont fournis par l'intermédiaire de méthodes répondant à des messages. C'est le sujet de la troisième partie de cette introduction à Objective-C.

III - Définir les méthodes d'une classe

Les méthodes permettent de masquer les variables et exposent l'interface publique d'un objet ou d'une classe.

Comme Java, Objective-C connaît les méthodes d'instance et de classes, mais son interprétation du modèle objet diffère de Java.

III-A - Généralités

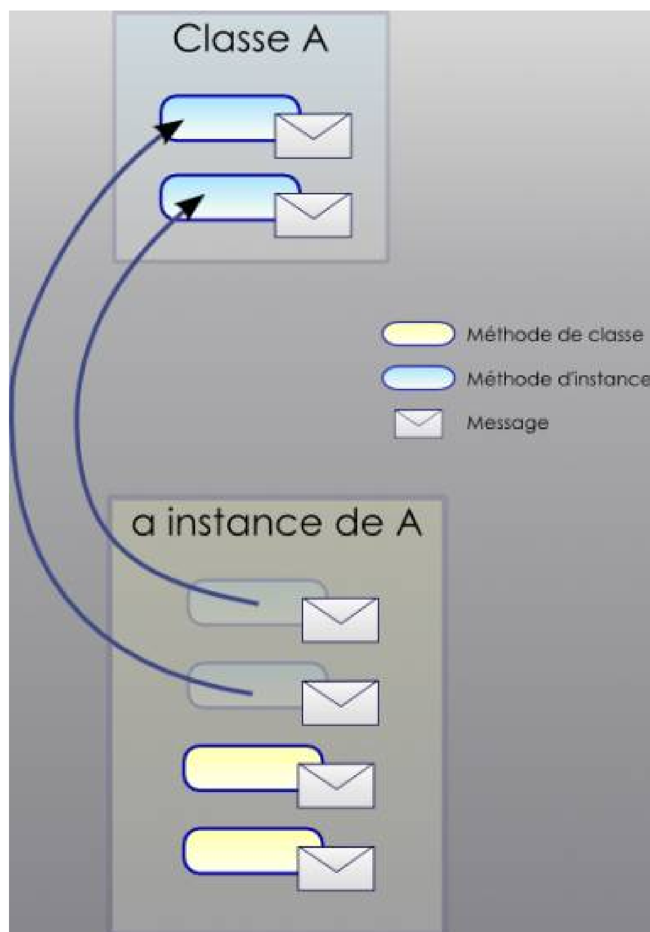
Java parle volontiers de méthode et Objective-C adopte plutôt la terminologie de message.

Cette différence dépasse le vocabulaire et reflète une réalité concrète. Les services des objets ne sont effectivement pas des appels vers des fonctions, mais des envois de messages qui sont ensuite traités par une fonction. La différence est subtile, mais peut aider à comprendre l'esprit du langage.

Là ou Java se rapproche de C++, Objective-C est vraiment dans l'esprit de SmallTalk par cette idée de message. Le coût est limité au moteur du langage (5) .

Le diagramme ci-dessous illustre comment un message peut-être géré par un objet.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com



- La classe déclare les messages qu'elle gère.
- La classe déclare les messages gérés par ses instances.
- Les instance ou la classe reçoivent des messages correspondant à ce que la classe à déclaré.
- Les messages de classes sont gérés par l'implémentation fournie par la classe.
- Les messages d'instance sont gérés par l'implémentation fournie par l'objet.

C'est en partie cette dernière caractéristique qui explique que les méthodes, contrairement aux variables, ne bénéficie pas de règles de visibilité. Les méthodes sont soit des méthodes d'instances, soit des méthodes de la classe.

Mais comme pour les variables de classes, le module peut servir à implémenter des messages privés. Il suffit de ne définir une méthode répondant à un message dans l'implémentation, sans déclarer ce message au niveau de l'interface de l'objet. Mais même si le message n'est pas visible il peut toujours être envoyé par un objet qui n'a pas connaissance de l'implémentation privée. Je ne pense pas que cette pratique soit à conseiller.

1. Java défini des méthodes sur les objets ou les classes. Une méthode Java est une fonction qui utilise la portée (l'espace de nommage) de la classe qui la défini.
2. Objective-C défini les services d'une classe par un ensemble de messages auxquels elle est capable de répondre. La classe répond à un message en appelant une méthode.

Le mode de fonctionnement d'Objective-C permet d'introduire une forte dynamique dans le langage sans imposer le surcoût important d'un système d'introspection.

Une classe peut ainsi répondre à une message sans forcément implémenter la méthode qui y répondra. Il est très facile de déléguer la gestion du message à un objet tiers en lui faisant simplement suivre le message.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

III-B - Déclaration des messages

Comme toutes les déclarations d'une classe les messages gérés par une classe sont déclarés dans le fichier en-tête de la classe.

```
@interface SimpleClass : NSObject {
        // ...
}

// Constructeur
- (id) init;

// Affichage
- (NSString *) description;

// Accès et modification sur 'destination'
- (id) destination;
- (void) setDestination: (id) location;

// Accès et modification sur 'speedLimit'
- (int) speedLimit;
- (void) setSpeedLimit: (int) speed;

// Une méthode de classe
+ (id) cloneSimpleObject: (id) object;

@end
```

Une déclaration se décompose en trois types d'éléments :

1. Un préfixe
   1. Un signe moins - pour les messages gérés par les instances.
   2. Un signe plus + pour les messages gérés par la classe.
2. Le type de retour de chaque message.
3. Le nom du message
4. Si le message accepte un paramètre il est déclaré en indiquant son type et un identifiant

Un message peut accepter plusieurs paramètres, dans ce cas la syntaxe varie largement du Java. Par exemple, pour déclarer un constructeur qui permette de définir les valeurs des variables d'instance on peut utiliser la déclaration suivante :

```
- (id) initWithDestination: (id) location andSpeedLimit: (int) limit;
```

Dans cet exemple, le nom du message est initWithDestination:andSpeedLimit : et les deux paramètres sont location et limit. L'avantage de cette syntaxe est simple : le nom des méthodes, s'il est correctement choisit, permet d'expliciter le rôle de chaque paramètre.

En Java, le rôle de chaque paramètre ne peut être déduit que si votre éditeur offre une complétion automatique et un accès à la Javadoc au fils de l'écriture de votre code. Cela est utile lorsque vous écrivez votre code, mais absent lorsque vous le relisez.

En Objective-C, le langage permet d'exposer directement le rôle des paramètres, que vous soyez en pleine écriture du code ou en train de relire un source vieux de plusieurs mois. *Le code source est auto-suffisant pour assurer sa compréhension.*

III-C - Définition des méthodes

Les méthodes qui répondent aux message déclarés dans l'interface de la classe sont définies dans le module d'implémentation de la classe.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

Pour chaque message on défini une méthode :

```
@implementation SimpleClass
//...

- (int) speedLimit
{
   return speedLimit;
}


- (void) setSpeedLimit: (int) speed
{
   speedLimit =  speed;
}
//...
@end
```

Si vous de fournissez pas de méthode pour chaque message déclaré votre classe est abstraite.

III-D - Appel de méthode

Pour invoquer un service sur un objet ou une classe il suffit de lui envoyer le message correspondant :

```
// Message sans paramètre
[object message];
/* Appel le message messageD:situeA:
* avec les paramètres expediteur et lieu
*/
[objet messageDe: expediteur situeA: lieu]
```

Voici par exemple l'implémentation de la méthode description pour notre classe simple :

```
- (NSString *) description;
{
   NSMutableString * descr =
       [[NSMutableString alloc] initWithString: @"SimpleClass"];
   if ( nil != [self destination] ) {
       [descr appendFormat: @" destination %@", [self destination]];
   }
   [descr appendFormat: @" speedLimit %d", [self speedLimit]];
   return descr;
}
```

III-E - Pour finir

Nos objets viennent d'acquérir la capacité de proposer des services. Ils sont donc largement plus utiles.

Ils nous manque un dernier élément : comprendre le cycle de vie des objets pour pouvoir les animer correctement. C'est le sujet de la quatrième et dernière partie de cette introduction.

IV - Initialiser classes et objets

Nous avons vu comment déclarer une classe et comment préparer l'implémentation des méthodes qui seront associées aux messages acceptés par la classe.

Pour aller plus loin il faut être capable de construire proprement des instances d'une classe, et si besoin initialiser les variables globales utilisées par une classe.

Cette partie a donc pour but de détailler :

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

- l'initialisation d'une classe ;
- le mécanisme de construction d'un objet ;
- la fin de vie d'un objet.

IV-A - Initialisation de la classe

Les classes ont parfois besoin d'une phase d'initialisation qui leur soit propre. Ceci est particulièrement vrai si l'on utilise des variables de classes.

Java propose un mécanisme simple pour initialiser les membres d'une classe : *l'initialisation au moment de la déclaration* :

```
class UneClasse {
 // Une constante
 static public final String LABEL = "unLabel";

 // Un objet
 static private java.net.URL baseURL =
   new java.net.URL ( "http://www.google.com/" ) ;
}
```

Cette méthode, si elle fonctionne très bien pose un problème pour la création d'objets plus complexes. Dans le cas de la construction de la variable baseURL on se trouve dans un cas où une alternative doit être trouvée pour pouvoir gérer les cas d'erreurs.

En effet, la construction de l'URL peut générer une exception qui ne sera pas traitée par la classe, mais qui sera interceptée par le chargeur de classe (ClassLoader). Pour peu que la classe ne soit pas initialisée explicitement, une telle erreur peut être ensuite plus délicate à identifier correctement.

Une bien meilleure solution est proposée en Java : l'initialisation dans le bloc static.

```
class UneClasse {
 // Une constante
 static public final String LABEL = "unLabel";

 // Un objet non initialisé
 static private java.net.URL baseURL;

 // Initialisation
 static {
   try {
     baseURL = new java.net.URL ( "http://www.google.com/" ) ;
   }
   catch ( final Throwable t ) {
     // ... Je peux gérer les erreurs
   }
 }
}
```

Objective-C propose un système tout a fait comparable, mais sans introduire de syntaxe particulière.

Comme dans Objective-C les classes sont des objets comme les autres, l'objet associé à une classe peut être initialisé en implémentant une méthode initialize.

La déclaration se fait dans le fichier d'interface :

```
@interface UneClasse : NSObject {
}

+ (void) initialize; // Initialise les membres de classe
```


Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

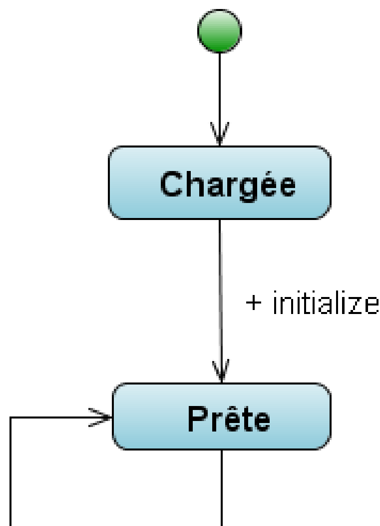
```
@end
```

L'implémentation est localisé avec celle des autres méthodes :

```
static NSURL baseURL;

@implementation UneClasse

+ (void) initialize
{
 baseURL = [NSURL URLWithString: "http://www.google.com/"];
}

@end
```

Il est important de remarquer que dans un cas comme dans l'autre l'initialisation des membres de classe est appelée pour chaque classe chargée en mémoire. Il ne faut donc pas appeler la méthode initialize de la classe parente.

Les états que peut prendre une classe sont présentés dans le diagramme d'états ci-dessous :



Pour résumer :

	Java	Objective-C
Principe	Construction Syntaxique	Méthode de classe
Initialisation	static { ... }	+initialize ;

IV-B - Construction d'instance

En Java comme en Objective-C les instances d'une classe sont construites en deux étapes :

1. allocation de la zone mémoire contenant l'objet ;
2. initialisation des valeurs des membres de l'objet.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

La syntaxe Java est la suivante :

```
String chaine = new String("une chaine");
```

L'opérateur new masque les opérations d'allocation et d'initialisation en une seule étape. C'est la syntaxe de Java qui oblige simplement indiquer quel constructeur va être utilisé pour initialiser l'objet. L'opérateur new alloue automatiquement la zone mémoire avant d'appeler le constructeur.

Dans le respect de la philosophie d'Objective-C, le langage n'ajoute aucune structure syntaxique pour cette opération. Les deux opérations sont explicitement visibles dans le code source en deux parties :

1. l'appel d'allocation, envoyé à la classe ;
2. l'appel d'initialisation, envoyé à la nouvelle instance.

Les méthodes appelées sont :

1. +alloc pour allouer la zone mémoire ;
2. Par convention, l'objet doit répondre à un message init pour s'initialiser. Mais plusieurs variantes sont possibles (6)

La syntaxe normale pour créer une instance est la suivante : (7)

```
NSString * chaine = [NSString alloc];
[chaine initWithCString:"une chaine"
 encoding: NSISOLatin1StringEncoding];
```

Il est bien évident que cette syntaxe est un peu trop longue. La construction est généralement compactée comme suit :

```
NSString * chaine =
 [NSString alloc] initWithCString:"une chaine"
   encoding: NSISOLatin1StringEncoding]];
```

La déclaration des constructeurs se fait dans le fichier d'en-tête .h :

```
@interface UneClasse : NSObject {
}

- (id) init;

@end
```

Pour sa part l'implémentation est logiquement dans le fichier .m :

```
@implementation UneClasse

- (id) init
{
 [super init]; // Il faut initialiser l'objet parent!
 // Mes initialisations spécifiques ci-dessous
 // ...
}

@end
```

La solution proposée par Objective-C a l'avantage de la simplicité syntaxique en limitant les extensions imposées au C. C'est aussi un moyen très simple pour une classe de suivre les créations d'instances puisqu'il est tout à fait possible de surcharger la méthode alloc.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

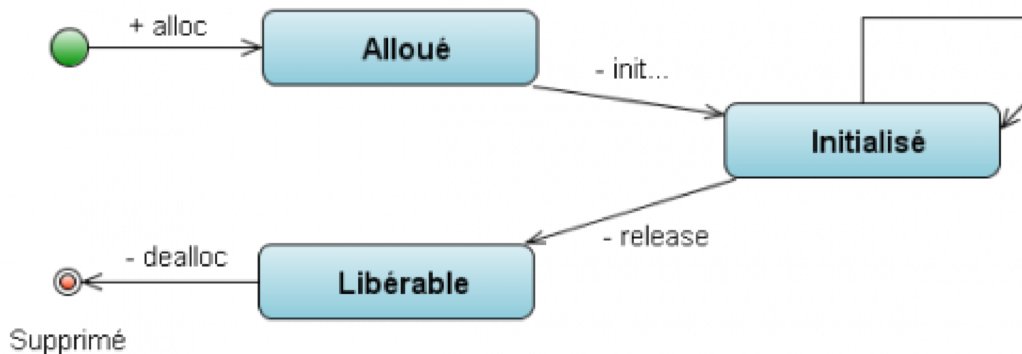
Si la chose est tout a fait possible en Java, elle doit cependant être réalisée dans le constructeur, ce qui peut malheureusement être surchargé. S'ajoute également l'inconvénient de mélanger un code spécifiquement lié au fonctionnement de la classe avec du code spécifique aux instances.

	Java	Objective-C
Principe	Opérateur et méthode d'instance spéciale.	Méthode de classe et méthodes d'instance.
Allocation	new UneClasse(...) ;	[UneClasse alloc]
Initialisation	UneClasse()	- init
Initialisation avec options	UneClasse(p1, p2)	- initFromParam : p1 andParam : p2

Un objet suit les évolutions suivantes :

1 allocation
2 initialisation
3 utilisation
4 libération

Ce qui est résumé dans le diagramme d'état suivant :



IV-C - Destruction d'une instance

Lorsqu'un objet deviens inutile il est souhaitable de le supprimer pour récupérer l'espace mémoire qui est lui associé.

En Java la gestion de la mémoire est quasi transparente du fait de la machine virtuelle qui gère tout avec un système de ramasse miettes (garbage collection).

La situation est très similaire avec Cocoa. Dans sa version actuelle le ramasse-miettes n'est pas implémenté mais il le sera dans la prochaine version disponible avec Léopard (Mac OS X.5). C'est donc au développeur de gérer la mémoire dans son application. Bien que simple, cette gestion de la mémoire fera l'objet d'un article spécifique.

Mais même avec un ramasse-miettes Java permet au développeur de faire le ménage dans ses objets. C'est la méthode finalize qui s'en charge :

```
protected void finalize() throws Throwable
```

Cette méthode est appelée par le ramasse miette et doit donc se contenter de vider ses collections ou remettre les références vers des objets tiers à null.

```
protected void finalize() throws Throwable {
```


Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

```
// Mon ménage ci-dessous
// ...
// Le ménage pour la partie de la classe parente.
super.finalize();
}
```

C'est exactement le même principe en Objective-C.

Pour faire le ménage derrière elle, une classe doit proposer une implémentation de la méthode dealloc. Et comme en Java, la méthode doit faire suivre l'appel vers sa classe parente.

```
- (void) dealloc {
 // Mon ménage ci-dessous
 // ... release des references d'objet
 // ... mise à nil des références facultative, mais utile

 // Le ménage pour la partie de la classe parente.
 [super dealloc];
}
```

Il est important de noter que cette méthode peut ne pas être appelée lorsqu'on quitte une application. Dans ce cas, il est plus efficace de laisser le système d'exploitation libérer la mémoire allouée par l'application en seul bloc, et c'est ce que fait le runtime Objective-C.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com

1 : Standard_Template_Library
2 : h pour header
3 : m pour module
4 : Si vous poursuivez l'exploration de Cocoa vous verrez certainement que *NSObject* n'est pas la seule racine. Cocoa fournit également la classe *NSProxy*. En parler plus en détails sort du cadre de cet article d'introduction.
5 : Objective-C est implémenté au dessus de C mais utilise un petit moteur, le runtime, pour implémenter les fonctionnalités dynamiques du langage.
6 : La littérature présente souvent cela sous la forme init..., pour indiquer " toutes les méthodes commençant par init ".
7 : Dans cet exemple on utilise un constructeur avec paramètres.

Le contenu de cet article a été rédigé par Sylvain Gamel et est disponible sous la licence Creative Commons "Paternité Pas d'Utilisation Commerciale" Par contre, les logos developpez.com, en-tête, pied de page, css et look & feel de l'article sont eux sous copyright 2008 developpez.com