De C++ à Objective-C

version 1.5-fr

Pierre Chatelier
e-mail : pierre.chatelier@club-internet.fr

Remerciements : Pour leurs lectures attentives et leurs multiples remarques, je tiens avant tout à remercier Pascal Bleuyard, Jérôme Cornet, François Delobel et Jean-Daniel Dupas, en l'absence de qui l'écriture de ce document aurait été parfaitement possible, mais en aurait largement pâti.

Table des matières

Introduction

Ce document est un guide de passage de C++ à Objective-C. Il existe plusieurs documentations soucieuses d'enseigner le modèle objet *via* Objective-C, mais aucune à ma connaissance n'est destinée aux codeurs expérimentés en C++, désirant se renseigner sur les concepts du langage pour les comparer à ce qu'ils connaissent déjà. Le langage Objective-C m'avait semblé au premier abord un obstacle plutôt qu'un tremplin à la programmation avec **Cocoa** (cf. section 1 page suivante) : il est si peu répandu que je ne comprenais pas son intérêt face à un C++ puissant, efficace et maîtrisé. Il a donc fallu longtemps pour que je comprenne qu'il était au contraire un réel concurrent grâce à la richesse des concepts qu'il propose. Ce document ne se présente pas comme un didacticiel mais comme une référence de ces concepts. Il permettra ainsi, je l'espère, d'éviter qu'une mauvaise connaissance d'Objective-C conduise un développeur C++, soit à abandonner trop vite ce langage, soit à utiliser à mauvais escient ses outils habituels, produisant alors un code bâtard, inélégant et inefficace. Ce document ne se veut pas une référence *complète*, mais *rapide*. Pour une description approfondie d'un concept, mieux vaut consulter une documentation Objective-C spécifique [1].

1 Objective-C et Cocoa

Une première précision me semble indispensable : **Objective-C** est un langage, et **Cocoa** est un ensemble de classes permettant de programmer de façon native sous MacOS X. En théorie, on peut faire de l'Objective-C sans Cocoa : il existe un front-end pour GCC. Sous MacOS X, les deux sont presque indissociables, la plupart des classes fournies avec le langage faisant en réalité partie de Cocoa.

Pour être précis, Cocoa est l'implémentation par Apple, pour MacOS X, du standard OpenStep publié par NeXT Computer en 1994. Il s'agit d'une bibliothèque de développement d'applications basée sur Objective-C. Citons l'existence de GNUstep [3], une autre implémentation, libre, d'OpenStep. Cette implémentation se veut la plus portable possible pour fonctionner avec de nombreux systèmes Unix. Elle est toujours en développement au moment où j'écris ces lignes.

2 Bref historique d'Objective-C

Il est assez difficile de dater précisément les naissances des langages de programmation. Selon que l'on considère leurs balbutiements, leur développement, leur annonce officielle ou leur standardisation, la marge est importante. Malgré tout, un historique sommaire, présenté en figure 1, permet de situer Objective-C vis-à-vis de ses ancêtres et ses concurrents.

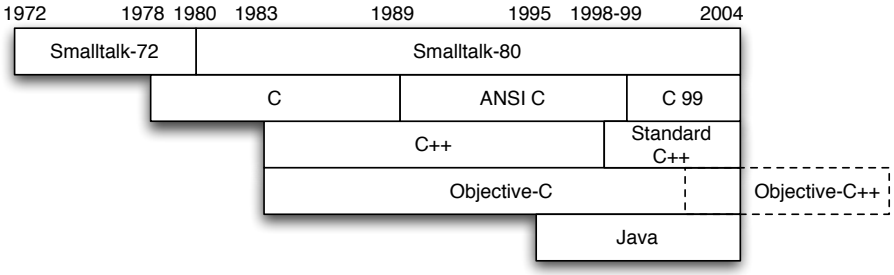

Fig. 1 – Historique sommaire de Smalltalk, C, C++ et Objective-C

Smalltalk-80 est un des tous premiers langages réellement objet. Le C++ et l'Objective-C sont deux branches différentes visant à créer un langage objet basé sur le C. Objective-C emprunte énormément à Smalltalk, pour le dynamisme et la syntaxe, tandis que C++ se tourne vers une forme de programmation très statique orientée vers la performance d'exécution. Java veut se placer en remplaçant du C++ mais il est également très inspiré de Smalltalk par son aspect objet plus pur. C'est pourquoi, malgré le titre de ce document, de nombreuses références sont également faites au Java.

Objective-C++ est une fusion de l'Objective-C et du C++. Il est déjà en grande partie opérationnel, mais certains comportements sont encore imparfaits à la date d'écriture de ces lignes. Objective-C++ doit permettre de mélanger des syntaxes Objective-C et C++ pour tirer parti des fonctionnalités des deux mondes (cf. section 14 page 47).

3 Généralités sur la syntaxe

3.1 Mots-clés

Objective-C est un sur-ensemble du langage C. Comme en C++, un programme correctement écrit en C devrait être compilable en Objective-C, sauf s'il utilise certaines mauvaises pratiques autorisées par le C. L'Objective-C n'a fait que rajouter des concepts et les mots-clés les accompagnant. Pour éviter tout conflit, ces mots-clés commencent par le caractère @ (at). En voici la (courte) liste exhaustive : `@class`, `@interface`, `@implementation`, `@public`, `@private`, `@protected`, `@try`, `@catch`, `@throw`, `@finally`, `@end`, `@protocol`, `@selector`, `@synchronized`, `@defs`, `@encode`. Notons également la présence des valeurs `nil` et `Nil`, du type `id`, du type `SEL` et du type `BOOL` avec ses

valeurs `YES` et `NO`. Enfin, il existe 6 mots-clés qui ne sont définis que dans un certain contexte bien particulier, et qui sont ne sont pas réservés hors de ce contexte : `in`, `out`, `inout`, `bycopy`, `byref`, `oneway`. Ils peuvent se rencontrer dans la définition de protocoles (cf. section 6.4.3 page 22).

Il y a une très forte confusion possible entre les mots-clés du langage et certaines méthodes héritées de la classe racine `NSObject` (la mère de toutes les classes, cf. section 5.1 page 9). Par exemple, les apparents « mots-clés » de la gestion mémoire que sont `alloc`, `retain`, `release` et `autorelease`, sont en fait des méthodes de `NSObject`. Les mots `super` et `self` (cf. section 5.3.1 page 12), pourraient également passer pour des mots-clés, mais ils sont en fait des données d'instances (des attributs) de `NSObject`. Il ne me paraît cependant pas préjudiciable de faire l'amalgame entre ces faux « mots-clés » et les vrais, vu leur indispensable présence.

Tous les mots-clés d'Objective-C ne sont pas présentés dans ce document. `@synchronized`, `@defs` et `@encode` ont été laissés de côté, car ils dépassent le cadre de la comparaison avec le C++ : ils n'y ont pas d'équivalent. Ils ne sont pas indispensables pour porter un code C++ standard en Objective-C.

3.2 Commentaires

Les commentaires `/* ... */` et `//` sont autorisés.

3.3 Mélange code/déclarations

Il est possible comme en C++ d'insérer des déclarations de variables au milieu d'un bloc d'instructions.

3.4 Nouveaux types et valeurs

3.4.1 BOOL, YES, NO

En C++, le type `bool` a été implémenté. En Objective-C, on dispose du type `BOOL` qui prend les valeurs `YES` ou `NO`.

3.4.2 `nil`, `Nil` et `id`

Ces trois mots-clés sont expliqués dans la suite de ce document. On peut toutefois les présenter brièvement :

- Tout objet est de type `id`. C'est un outil de typage faible ;
- `nil` est l'équivalent de `NULL` pour un pointeur d'objet. `nil` et `NULL` ne sont pas interchangeables ;
- `Nil` est l'équivalent de `nil` pour un pointeur de classe, car en Objective-C, les classes sont aussi des objets (instances de méta-classes).

3.4.3 `SEL`

`SEL` est le type utilisé pour stocker les sélecteurs (équivalent des pointeurs de méthodes), obtenus par l'utilisation de `@selector`. C'est l'implémentation Objective-C des pointeurs de méthodes membres. Voyez la section 5.3.4 page 13 pour des explications à ce sujet.

3.5 Nom des classes : pourquoi NS ?

Dans ce document, la plupart des classes sont précédées de *NS*, comme *NSObject*, *NSString*... La raison en est simple : ce sont des classes Cocoa, et toutes les classes Cocoa commencent par *NS* puisqu'elles ont été initiées sous NeXTStep.

3.6 Différenciation entre fonctions et méthodes

Objective-C n'est pas un langage « dont les appels de fonction s'écrivent avec des crochets ». C'est en effet ce que l'on pourrait penser en voyant écrit

```
[object doSomething];
```

au lieu de

```
object.doSomething();
```

D'une part, comme Objective-C est un sur-ensemble du C, les *fonctions* respectent la même syntaxe et la même logique que le C quant à la déclaration, l'implémentation et l'appel. En revanche, les *méthodes*, qui n'existent pas en C, ont une syntaxe spéciale avec des crochets. De plus, la différence ne se situe pas seulement au niveau de la syntaxe mais également de la sémantique. Cela est expliqué plus en détails dans la section 5.2 page suivante : ce n'est pas un appel de méthode, c'est l'envoi d'un message. Cette originalité n'est pas une préciosité gratuite, c'est un mécanisme beaucoup plus cohérente avec le langage. Et la syntaxe est aussi beaucoup plus claire, surtout en cas d'appels en cascade (cf. section 5.3.1 page 12).

4 Organisation du code source : fichiers *.h*, fichiers *.m* et inclusion

Comme en C++, il est bienvenu d'utiliser un couple de fichiers interface/implémentation par classe. En Objective-C, le fichier d'en-tête est un fichier *.h* et le code est dans un fichier *.m* ; on rencontre aussi l'extension *.mm* pour l'Objective-C++, qui fait l'objet d'une section spéciale en fin de document (section 14 page 47). Enfin, notons que l'Objective-C introduit la directive de compilation `#import` pour remplacer avantageusement `#include`. En effet, tout fichier d'en-tête en C doit posséder des gardes de compilation pour empêcher son inclusion multiple ; le `#import` gère cela automatiquement. Ci-après se trouve un exemple typique de couple interface/implémentation. La syntaxe Objective-C est expliquée dans la suite du document.

C++

```
//Dans le fichier Toto.h

#ifndef __TOTO_H__ //garde de
#define __TOTO_H__ //compilation

class Toto
{
...
};

#endif
```

```
//Dans le fichier Toto.cpp

#include "Toto.h"

...
```

Objective-C

```
//Dans le fichier .h

//déclaration de classe
//(différent du «interface» de Java)
@interface Toto : NSObject
{
...
}
@end
```

```
//Dans le fichier Toto.m

#import "Toto.h"

@implementation Toto
...
@end
```

5 Classes et objets

Objective-C est un langage objet : on y crée des classes et des objets. Il respecte cependant mieux le paradigme objet que C++, lequel présente des lacunes vis à vis du modèle idéal. Par exemple, en Objective-C, les classes sont elles-mêmes des objets manipulables dynamiquement : on peut, pendant l'exécution du programme, ajouter des classes, créer des instances d'une classe d'un certain nom, demander à une classe quelles méthodes elle implémente, etc. Tout cela est beaucoup plus puissant que les RTTI du C++ (cf. section 13 page 46), qui ont été greffées à un langage fondamentalement « statique ». On déconseille d'ailleurs souvent de les utiliser, car les résultats obtenus dépendent du compilateur lui-même, et ne garantissent aucune portabilité.

5.1 Classe racine, type `id`, valeurs `nil` et `Nil`

Dans un langage objet, on crée généralement un diagramme de classes pour chaque programme. Une des particularités d'Objective-C par rapport à C++ est l'existence d'une classe racine (*root*) dont devrait hériter toute nouvelle classe. En Cocoa, il s'agit de `NSObject`, qui fournit un nombre de services gigantesque, et assure la cohérence du système d'exécution d'Objective-C. Mais la notion de classe racine n'est pas une spécifité d'Objective-C, c'est une particularité du modèle objet en général ; Smalltalk, Java, utilisent aussi une classe racine. Le C++ quant à lui ne l'impose pas.

En toute rigueur, tout objet devrait donc être de type `NSObject` et tout pointeur vers un objet pourrait ainsi être déclaré comme un `NSObject*`. Cependant, ce n'est pas absolument obligatoire : pour une raison sans doute exceptionnelle, une classe peut ne pas dériver de la classe racine. En revanche, tout objet Objective-C, qu'il dérive de `NSObject` ou pas, est de type `id`. Il est donc plus sûr, et surtout plus pratique, d'utiliser le type `id` au lieu de `NSObject*`. Attention cependant : un pointeur d'objet nul ne doit pas être mis à `NULL` mais à `nil`. Ces valeurs n'ont pas vocation d'être interchangeables. Un pointeur C normal peut être à `NULL`, mais `nil` a été introduit en Objective-C pour les pointeurs d'objets. En Objective-C, les classes sont aussi des objets (instances de méta-classes), et il existe donc des pointeurs de classes, pour lesquels la valeur vide est `Nil`.

5.2 Déclaration de classes

Il est difficile de montrer par un seul exemple toutes les différences existant entre le C++ et l'Objective-C pour la déclaration des classes et l'implémentation des méthodes. En effet, syntaxe et concepts s'entremêlent et nécessitent des explications. Les différences sont donc exposées en plusieurs étapes très ciblées.

5.2.1 Attributs et méthodes

En Objective-C, les attributs sont appelés *données d'instance*, et les fonctions membres des *méthodes*.

C++	Objective-C

`class Toto`
`{`
`  double x;`
 	
`  public:`
`  int  f(int x);`
`  float g(int x, int y);`
`};`
 	
`int  Toto::f(int x) {...}`
`float Toto::g(int x, int y) {...}`
	`@interface Toto : NSObject`
	`{`
	`  double x;`
	`}`
	
	`-(int)  f:(int) x;`
	`-(float) g:(int)x :(int)y;`
	`@end`
	
	`@implementation Toto`
	`{`
	` -(int)  f:(int) x {...}`
	` -(float) g:(int)x :(int)y {...}`
	`}`
	`@end`

En C++, attributs et méthodes sont déclarés ensembles au sein de la classe. Pour implémenter les méthodes, dont la syntaxe est similaire au C, on utilise l'opérateur de résolution de portée `Toto::` .

En Objective-C, les attributs et les méthodes ne peuvent être mélangés. Les attributs sont déclarés entre accolades, et les méthodes leur succèdent. Le corps des méthodes est spécifié dans une partie `@implementation`.

Il existe une différence fondamentale avec le C++, dans la mesure où des méthodes peuvent être implémentées sans avoir été déclarées dans l'interface. Ce point est détaillé plus tard. Sommairement, cela permet d'alléger les fichiers *.h* en ne répétant pas dans une sous-classe les déclarations des méthodes virtuelles redéfinies qui ne sont appelées qu'automatiquement, comme un destructeur par exemple. Voyez la section 6.3.2 page 19 pour plus d'explications.

Les méthodes sont précédées du signe « - » ou parfois du signe « + », pour différencier méthodes d'instance et méthodes de classe (cf. section 5.3.9 page 17). Ce signe n'a rien à voir avec la notation UML signifiant *public* ou *private*. Les types des paramètres sont encadrés de parenthèses, et surtout les paramètres sont séparés par des « : ». Voyez la section 5.3.1 page 12 pour plus de détails sur la syntaxe des prototypes.

Notez aussi qu'il n'y a pas de point-virgule nécessaire à la fin d'une déclaration de classe en Objective-C. Remarquez également que c'est bien `@interface` qui est employé pour déclarer une classe, et non `@class` comme on aurait pu le croire. Le mot-clé `@class` est utilisé dans les déclarations anticipées (*forward*) (cf. section 5.2.2 de la présente page). Sachez enfin que si la classe ne déclare pas de données d'instance, le bloc d'accolades ouvrantes-fermantes (qui serait vide) n'est pas nécessaire.

5.2.2 Déclarations forward : `@class`, `@protocol`

Pour éviter les dépendances cycliques des fichiers d'en-tête, il est nécessaire de recourir à la *déclaration forward* des classes lorsqu'on a juste besoin de spécifier leur existence, et non leur définiton. En C++, le mot-clé `class` remplit aussi ce rôle ; en Objective-C, on utilise `@class`. Il existe aussi le mot-clé `@protocol` pour anticiper la déclaration des protocoles (cf. section 6.4 page 20).

C++

```
//Dans le fichier Toto.h

#ifndef __TOTO_H__
#define __TOTO_H__

class Titi; //déclaration forward
class Toto
{
  Titi* titi;

  public:
    void utiliserTiti(void);
};

#endif
```

```
//Dans le fichier Toto.cpp

#include "Toto.h"
#include "Titi.h"

void Toto::utiliserTiti(void)
{
 ...
}
```

Objective-C	

`//Dans le fichier .h`
 	
`@class Titi; //déclaration forward`
`@interface Toto : NSObject`
`{`
`  Titi* titi;`
`}`
 	
`-(void) utiliserTiti;`
 	
`@end`
	`//Dans le fichier Toto.m`
	
	`#import "Toto.h"`
	`#import "Titi.h"`
	
	`@implementation Toto`
	
	`-(void) utiliserTiti`
	`{`
	` ...`
	`}`
	
	`@end`

5.2.3 `public`, `private`, `protected`

Le modèle objet repose en partie sur l'encapsulation des données, qui permet de limiter leur visibilité dans différentes portions du code, pour offrir des garanties d'intégrité.

C++	Objective-C

`class Toto`
`{`
`  public:`
`  int x;`
`  int tata();`
 	
`  protected:`
`  int y;`
`  int titi();`
 	
`  private:`
`  int z;`
`  int tutu();`
`};`
	`@interface Toto : NSObject`
	`{`
	`  @public`
	`  int x;`
	
	`  @protected:`
	`  int y;`
	
	`  @private:`
	`  int z;`
	`}`
	
	`-(int) tata {...}`
	`-(int) titi {...}`
	`-(int) tutu {...}`
	`@end`

En C++, attributs et méthodes peuvent appartenir à un contexte de visibilité `public`, `protected` ou `private`. Si elle n'est pas spécifiée, la visibilité est `private`.

En Objective-C, seules les données d'instance peuvent être `public`, `protected` ou `private`, et la visibilité par défaut est ici `protected`. Les méthodes ne peuvent être que publiques. On peut cependant imiter le mode privé, en implémentant des méthodes uniquement dans la partie `@implementation`, ou en utilisant la notion de catégories de classe (cf. section 6.5 page 23). Notez que le fait de pouvoir implémenter des méthodes sans les avoir déclarées dans l'interface est une fonctionnalité propre à l'Objective-C, et qui sert en autre but, expliqué en section 6.3.2 page 19.

On ne peut pas en Objective-C spécifier un caractère `public`, `protected` ou `private` pour l'héritage. ll est forcément `public`. L'héritage en Objective-C est beaucoup plus semblable à celui du Java que du C++ (section 6 page 19).

5.2.4 Attributs `static`

Il n'est pas possible avec la version actuelle du compilateur Objective-C de déclarer des données de classe, (attributs `static` en C++). Il faut utiliser des variables globales et des méthodes de classes (qui, elles, sont possibles, cf. 5.3.1 de la présente page), pour en imiter le comportement.

5.3 Méthodes

La syntaxe des méthodes en Objective-C est bien différente des fonctions C classiques. Cette section explique l'intérêt de cette syntaxe particulière, et déborde un peu sur le principe de l'envoi de messages, sur lequel repose Objective-C.

5.3.1 Prototype et appel, méthodes d'instance, méthodes de classe

- Une méthode est précédée d'un « − » si c'est une méthode d'instance (le cas le plus courant), et d'un « + » si c'est une méthode de classe (`static` en C++). Ce symbole n'a rien à voir avec la signification *public* ou *private* de l'UML. Rappelons que les méthodes sont toujours publiques en Objective-C ;
- les types de retour et des paramètres sont entre parenthèses ;
- les paramètres sont séparés par le caractère « : » ;
- les paramètres peuvent recevoir une **étiquette**, un nom avant le « : », qui sera utilisable lors de *l'appel* de la fonction, rendant ce dernier extrêmement lisible. À l'usage, cette pratique est incontournable. À noter que le premier paramètre ne peut pas recevoir d'étiquette ; c'est le nom de la fonction lui-même qui lui donne une signification ;
- le nom d'une méthode peut être strictement le même que celui d'un attribut, sans provoquer de conflit. Cela est très utile pour écrire des accesseurs en lecture (cf. section 8.4.8 page 38).

C++

```
//prototype
void Array::insertObject(void *anObject, unsigned int atIndex)

//utilisation avec une instance "etagere" de la classe Array
//et un objet "livre"
etagere.insertObject(livre, 2);
```

Objective-C

Sans étiquette (transcription directe d'un prototype C++)

```
//prototype
- (void)insertObject:(id)anObject:(unsigned int)index

//utilisation avec une instance "etagere" de la classe Array
//et un objet "livre"
[etagere insertObject:livre:2];
```

Avec étiquette

```
//prototype. On assigne ici au paramètre "index" l'étiquette "atIndex"
//Cela permettra de lire cet appel comme une phrase
- (void)insertObject:(id)anObject atIndex:(unsigned int)index

//utilisation avec une instance "etagere" de la classe Array
//et un objet "livre"
[etagere insertObject:livre:2];         //Erreur !
[etagere insertObject:livre atIndex:2]; //Ok
```

Notez que la syntaxe à base de crochets ne se lit pas comme *appeler la méthode* `insertObject` de l'objet « etagere » mais plutôt comme *envoyer le message* `insertObject` à l'objet « etagere ». Cette

phrase traduit tout le dynamisme d'Objective-C. On peut envoyer les messages que l'on souhaite à une cible donnée. Si elle n'est pas capable de traiter ce message, elle l'ignorera (en pratique, une exception est levée, mais le programme ne s'interrompt pas). Si à un instant donné de l'exécution du programme la cible sait traiter le message, elle le fera avec la méthode correspondante. Il y a tout de même des avertissements de la part du compilateur si un message est envoyé à un objet de classe connue, pour laquelle on sait que le message est invalide ; mais ce n'est pas considéré comme une erreur (du fait du possible *forwarding* cf. section 5.4.2 page 17). Si la cible du message n'est connue que sous le type `id`, il n'y a que lors de l'exécution du programme qu'il pourra être déterminé si le message peut être traité. Aucun avertissement n'est donc levé lors de la compilation.

5.3.2 `this`, `self` et `super`

Il existe deux cibles particulières pour un message : `self` et `super`. `self` représente l'objet courant (équivalent de `this` en C++), `super` représente la classe mère, comme en Java. Le mot-clé `this` n'existe pas en Objective-C, `self` le remplace.

Remarque : `self` n'est cependant pas un véritable mot-clé, c'est une donnée d'instance de l'objet, dont on peut changer la valeur, contrairement au `this` du C++. Mais cette pratique n'est utilisée que dans les constructeurs (cf. section 7.1 page 25).

5.3.3 Accès aux données d'instance de son objet déclencheur

Tout comme en C++, une méthode en Objective-C peut accéder aux données d'instance de son objet déclencheur. L'éventuel `this->` du C++ est remplacé par `self->`.

C++	Objective-C

`class Toto`
`{`
`  int x;`
`  int y;`
`  void  f(void);`
`};`
 	
`void Toto::f(void)`
`{`
`  x = 1;`
`  int y; //crée une ambiguité avec`
`  //this->y`
`  y = 2; //utilise le y local`
`  this->y = 3; //résout l'ambiguité`
`}`
	`@interface Toto : NSObject`
	`{`
	`  int x;`
	`  int y;`
	`}`
	
	`-(void) f;`
	`@end`
	
	`@implementation Toto`
	`{`
	`  -(void) f`
	`  {`
	`  x = 1;`
	`  int y;  //crée une ambiguité avec`
	`  //self->y`
	`  y = 2; //utilise le y local`
	`  self->y = 3; //résout l'ambiguité`
	`  }`
	`}`
	`@end`

5.3.4 Identifiant et signature du prototype, surcharge

Une fonction est une partie de code qui doit pouvoir être référencée, par exemple pour utiliser des pointeurs de fonctions, ou des foncteurs. De plus, si le nom de la fonction est un bon candidat pour jouer le rôle d'identifiant, il ne doit toutefois pas poser problème en cas de surcharge de ce nom. Les langages C++ et Objective-C sont antagonistes dans leur façon de différencier les prototypes. En effet, le premier se focalise sur les types des paramètres, et le second sur les étiquettes des paramètres.

En C++, deux fonctions de même nom peuvent être différenciées par le type des paramètres. Dans le cas des méthodes, l'option `const` est également déterminante.

C++

```
int f(int);
int f(float); //OK, float est différencié de int

class Toto
{
  public:
    int g(int);
    int g(float); //OK, float est différencié de int
    int g(float) const; //OK, le const est discriminant
};

class Titi
{
  public:
    int g(int); //OK, on est dans Titi::, différenciable de Toto::
}
```

En Objective-C, les fonctions son celles du C : elles ne peuvent être surchargées. En revanche, les méthodes (qui ont une syntaxe différente) sont différenciables à travers les étiquettes des paramètres.

Objective-C

```
int f(int);
int f(float); //Erreur : fonctions C non surchargeables

@interface Toto : NSObject
{
}

-(int) g:(int) x;
-(int) g:(float) x; //Erreur : cette méthode n'est pas différenciée
                    // de la précédente (pas d'étiquette)
-(int) g:(int) x :(int) y;   //Ok : il y a deux étiquettes anonymes
-(int) g:(int) x :(float) y; //Erreur : indifférenciable de la méthode
                             //précédente
-(int) g:(int) x andY:(int) y;   //Ok : la deuxième étiquette est "andY"
-(int) g:(int) x andY:(float) y; //Erreur : indifférenciable de la
                                 //méthode précédente
-(int) g:(int) x andAlsoY:(int) y; //Ok : la deuxième étiquette est
                                   //"andAlsoY", différent de "andY"
@end
```

Cette méthode de différenciation par les noms permet d'exprimer simplement quel est le nom « exact » de la fonction, comme expliqué ci-après.

```
@interface Titi : NSObject {}

//le nom de cette méthode est "g"
-(int) g;

//le nom de cette méthode est "g:"
-(int) g:(float) x;

//le nom de cette méthode est "g::"
-(int) g:(float) x :(float) y;

//le nom de cette méthode est "g:andY:"
-(int) g:(float) x andY:(float) y;

//le nom de cette méthode est "g:andZ:"
-(int) g:(float) x andZ:(float) z
@end
```

Comme on le voit, deux méthodes Objective-C ne sont pas différenciées par les types mais par les noms. C'est donc grâce à ce nom qu'on dispose de l'équivalent des « pointeurs de fonction », appelés *sélecteurs*, détaillés dans la section 5.3.5 de la présente page.

5.3.5 Pointeur de méthode : Sélecteur

En Objective-C, seules les *méthodes* ont cette syntaxe particulière utilisant parenthèses et étiquettes. Il n'est pas possible de déclarer des *fonctions* avec cette syntaxe. La notion de pointeur de fonction est la même en C qu'en Objective-C. C'est en revanche pour les pointeurs de *méthodes* qu'un problème se pose. La notion de pointeur de méthode n'est pas implémentée en Objective-C comme en C++.

En C++, il s'agit d'une syntaxe difficile, mais cohérente avec le C : un pointeur qui se focalise sur les types.

C++

```
class Toto
{
  public:
    int f(float x) {...}
};

Toto titi
int (Toto::*p_f)(float) = &Toto::f; //Pointeur vers Toto::f

(titi.*p_f)(1.2345); //appel de titi.f(1.2345);
```

En Objective-C, un nouveau type a été introduit. Un pointeur de méthode s'appelle un *sélecteur*. Il est de type `SEL` et sa valeur est calculée par un appel à `@selector` sur le nom exact de la fonction (comprenant les étiquettes des paramètres). Déclencher la méthode peut se faire grâce à la classe `NSInvocation`. Dans la plupart des cas, les méthodes utilitaires `performSelector` (avec plus ou moins de paramètres) seront plus pratiques, mais plus limitées. Les trois méthodes `performSelector` les plus simples sont les suivantes :

```
-(id) performSelector:(SEL) unSelecteur;
-(id) performSelector:(SEL) unSelecteur withObject:unObjetPourParametre;
-(id) performSelector:(SEL) unSelecteur withObject:unObjetPourParametre
                                        withObject:unAutreObjetPourParametre;
```

La valeur renvoyée est celle de la méthode déclenchée. Pour des méthodes dont les paramètres ne sont pas des objets, il faudra songer à utiliser des classes d'encapsulation comme `NSNumber`, pouvant fournir des `float`, des `int`, etc. implicitement. On peut aussi se tourner vers la classe `NSInvocation`, plus générale et plus puissante (voir la documentation).

D'après ce qui précède, rien n'empêche d'essayer de déclencher une méthode dans un objet dont la classe ne l'implémente pas. Mais le déclenchement ne découle que de l'acceptation du message `performSelector`. Une exception, interceptable, est levée dans le cas où la méthode à déclencher n'est pas connue par l'objet, mais le programme ne s'interrompt pas brutalement. En outre, on peut tester explicitement si un objet sait traiter une méthode *via* un appel à `respondsToSelector`.

Enfin, la valeur de `@selector()`) étant évaluée à la compilation, elle ne pénalise pas le temps d'exécution.

Objective-C

```
@interface Larbin : NSObject {}
  -(void) lireDossier:(Dossier*)dossier;
@end

//Supposons l'existence d'un tableau tab[] de 10 larbins
//et d'un dossier <dossier>

//utilisation simple
for(i=0 ; i<10 ; ++i)
  [tab[i] performSelector:@selector(lireDossier:) withObject:dossier];

//le type d'un sélecteur est SEL
//la version ci-dessous n'est pas plus efficace que la précédente,
//car @selector() est évalué à la compilation
SEL selecteur = @selector(lireDossier:);
for(i=0 ; i<10 ; ++i)
  [tab[i] performSelector:selecteur withObject:dossier];

//sur un objet "toto" de type inconnu (id)
//le test n'est pas obligatoire mais empêche la levée
//d'exception si l'objet n'a pas de méthode lireDossier:.
if ([toto respondsToSelector:@selector(lireDossier:)])
  [toto performSelector:@selector(lireDossier:) withObject:dossier];
```

5.3.6 Paramètres par défaut

L'Objective-C ne permet pas de spécifier des valeur par défaut pour les paramètres. Il faut donc créer autant de fonctions que nécessaire lorsque plusieurs nombres d'arguments sont envisageables. Dans le cas des constructeurs, il faut se référer à la notion d'*initialisateur désigné* (section 7.1.5 page 27) pour respecter les canons.

5.3.7 Nombre d'arguments variable

L'Objective-C permet de spécifier des méthodes au nombre d'arguments variable. Tout comme en C, il s'agit de rajouter « ... » en dernier argument. C'est une pratique fort rare pour l'utilisateur, même si de nombreuses méthodes implémentées dans Cocoa le font. Le lecteur curieux peut consulter la documentation Objective-C pour de plus amples détails sur ce point.

5.3.8 Arguments muets

En C++, il est possible de ne pas nommer tous les paramètres dans le prototype d'une fonction, puisque leur type suffit à caractériser la signature. En Objective-C, cela n'est pas possible.

5.3.9 Modificateurs de prototype (`const`, `static`, `virtual`, « `= 0` », `friend`, `throw`)

En C++, un certain nombre de modificateurs peuvent être ajoutés aux prototypes des fonctions. Aucun d'eux n'existe en Objective-C. En voici la liste :

- `const` : une méthode ne peut être spécifiée `const`. Le mot-clé `mutable` n'existe donc pas ;
- `virtual` : toutes les méthodes sont virtuelles en Objective-C, ce mot-clé est donc inutile. Les fonctions virtuelles **pures** s'implémentent par un protocole formel (cf. section 6.4 page 20) ;
- `static` : la différence entre méthode d'instance et méthode de classe se fait par l'utilisation du « - » ou du « + » devant le prototype ;
- `friend` : il n'y a pas de notion de classe ou méthode amie en Objective-C ;
- `throw` : en C++, on peut autoriser une méthode à ne transmettre que certaines exceptions. En Objective-C, ce n'est pas le cas.

5.4 Messages et transmission

5.4.1 Délégation d'un message vers un objet inconnu

La délégation est très présente avec les éléments d'interface graphiques de Cocoa (boutons, boîtes à cocher...). Il s'agit d'exploiter le fait qu'il est possible d'envoyer des messages à un objet inconnu. Un objet peut se décharger de certaines tâches en exploitant un objet assistant.

```
//Supposons l'existence d'une fonction d'attribution d'assistant
-(void) setStagiaire:(id)esclave
{
  [stagiaire autorelease]; //voir la section sur la gestion mémoire
  stagiaire = [esclave retain];
}
```

```
//la méthode faireUnTrucPenible peut implémenter une délégation
-(void) faireUnTrucPenible:(id) truc
{
  //le stagiaire est inconnu.
  //On vérifie qu'il peut traiter le message
  if ([stagiaire respondsToSelector:@(faireUnTrucPenible:)])
    [stagiaire faireUnTrucPenible:truc];
  else
    [self changerDeStagiaire];
}
```

5.4.2 Forwarding : gestion d'un message inconnu

En C++, on ne peut pas à la compilation forcer un objet à essayer d'exécuter une méthode qu'il n'implémente pas. En Objective-C, ce n'est pas pareil : on peut toujours envoyer un message à un objet. S'il ne peut pas le traiter, il l'ignorera (en levant une exception) ; ou mieux : plutôt que de l'ignorer, il peut le retransmettre à un tiers.

Notez que lorsque le compilateur détecte un envoi de message à un objet, et que d'après son type cet objet ne connaît pas le message, un avertissement est levé. Ce n'est cependant pas une erreur, car lorsqu'un objet reçoit un message qu'il ne sait pas traiter, une second chance lui est offerte. Cette seconde chance prend forme par un appel automatique à la méthode `forwardInvocation`, qui peut être surchargée pour re-router le message au dernier moment. C'est bien sûr une méthode de `NSObject` qui par défaut ne fait rien. C'est encore une façon de gérer des objets assistants.

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
  //si on est ici, c'est que l'objet ne sait pas traiter
  //le message de l'invocation
  //le selecteur fautif est accessible par l'envoi du message "selector"
  //à l'objet "anInvocation"
  if ([unAutreObjet respondsToSelector:[anInvocation selector]])
    [anInvocation invokeWithTarget: unAutreObjet];
  else  //ne pas oublier de tenter sa chance avec la superclasse
    [super forwardInvocation:anInvocation];
}
```

Remarquons toutefois que si un message peut être traité dans une `forwardInvocation`, et uniquement là, un test d'aptitude basé sur un `respondsToSelector` renverra `NO` malgré tout. En effet, par souci d'efficacité, le mécanisme de `respondsToSelector` ne teste pas l'envoi de message en conditions réelles, et ne peut donc soupçonner la présence de toutes les roues de secours.

5.4.3 Downcasting

Le *downcasting* est nécessaire en C++ pour appeller les méthodes d'une classe dérivée, lorsque l'on ne dispose que du pointeur d'une classe mère. Cette pratique n'est pas incorrecte, *via* l'appel à `dynamic_cast`. En Objective-C, elle n'est pas incontournable, puisqu'un message peut être envoyé même si le type semble ne pas indiquer que l'objet puisse y répondre.

Dans le cas d'un type explicite, pour éviter un avertissement à la compilation, on peut *caster* le type de l'objet ; il n'y a pas d'opérateur explicite de *downcasting* en Objective-C, on utilise le *cast* traditionnel du C.

6 Héritage

6.1 Héritage simple

Objective-C implémente bien sûr la notion d'héritage, mais ne supporte pas l'héritage multiple. Cette apparente limitation est en revanche complétée par d'autres concepts (protocoles, catégories de classe) qui sont expliqués plus loin dans ce document (sections 6.4 page suivante, 6.5 page 23).

C++	Objective-C

`class Toto : public Tutu,`
`  protected Tata`
`{`
`}`
	`@interface Toto : Tutu`
	` //il y a des techniques pour`
	` //"dériver" aussi de Tata`
	`{`
	`}`
	`@end`

En C++, une classe peut dériver d'une ou plusieurs autres classes, de façon `public`, `protected` ou `private`. Dans les méthodes, on peut faire référence à une super-classe grâce à l'opérateur de résolution de portée (`Tutu`::, `Tata`::).

En Objective-C, on ne peut dériver que d'une seule classe, de façon publique. Une méthode peut faire référence à la classe mère comme en Java, par le (faux) mot-clé `super`.

6.2 Héritage multiple

Objective-C n'implémente pas l'héritage multiple, et le compense par d'autres concepts, les *protocoles* (cf. 6.4 page suivante) et les *catégories* (cf. 6.5 page 23).

6.3 Virtualité

6.3.1 Méthodes virtuelles

Les méthodes sont automatiquement et obligatoirement virtuelles en Objective-C. On n'a donc pas besoin de le spécifier. De ce fait, le mot-clé `virtual` n'existe pas et n'a pas d'équivalent.

6.3.2 Redéfinition silencieuse des méthodes virtuelles

Il est possible en Objective-C d'implémenter une méthode sans l'avoir déclarée au préalable dans l'interface. Cette fonctionnalité n'est pas destinée à compenser l'absence du qualificateur `@private` pour les méthodes, mais à alléger considérablement les déclarations d'interface pour les méthodes héritées. En effet, dans le contexte d'une API « objet » comme Cocoa, de nombreuses classes sont fournies pour être dérivées tout en redéfinissant certains comportements. Par exemple, la classe `NSCell`, représentant la cellule d'un tableau, contient une méthode-prédicat `isBordered` renvoyant un `BOOL` qui indique si la cellule doit être dessinée avec une bordure ou non. On redéfinit souvent de telles méthodes dans les sous-classes, pour enrichir leurs comportements. Grâce au dynamisme d'Objective-C, il n'est pas nécessaire de les redéclarer dans l'interface des sous-classes.

Ce n'est pas une mauvaise pratique : les méthodes concernées sont plutôt les méthodes « bien connues » déclarées dans les super-classes. De nombreuses méthodes de la classe racine `NSObject` sont ainsi redéfinies silencieusement la plupart du temps. On peut citer par exemple le constructeur `init` (cf. section 7.1 page 25), les méthodes de clonage `copy`, `mutableCopy` (cf. section 7.3 page 30), le destructeur `dealloc` (cf. section 7.2 page 29), etc.

À l'usage, les interfaces sont plus simples, même si l'on perd de vue ce que l'on *peut* redéfinir, ce qui rend obligatoire la consultation régulière des documentations des classes mères.

Notez enfin que la notion de méthode virtuelle pure, dont la redéfinition est obligatoire dans les sous-classes, est résolue par le concept de *protocoles formels* (cf. section 6.4.1 page suivante page suivante).

6.3.3 Héritage virtuel

L'héritage virtuel n'a pas de raison d'être en Objective-C, puisque l'héritage y est simple et ne soulève aucun des problèmes de l'héritage multiple.

6.4 Protocoles

Java compense l'absence d'héritage multiple par la notion d'*interface*. En Objective-C, la même notion est utilisée, mais est appelée *protocole*. En C++, on utiliserait une classe abstraite. Un protocole n'est pas une classe à proprement parler, car il ne peut proposer que des méthodes, et ne peut contenir de données. Il existe deux types de protocoles : les protocoles *formels* et les protocoles *informels*.

6.4.1 Protocole formel

Un protocole formel est un ensemble de méthodes qui doivent être implémentées par toute classe adhérente. Cela peut aussi être vu comme une certification accordée à une classe lorsqu'elle implémente tout ce qui est nécessaire à un service donné. Une classe peut adhérer à un nombre quelconque de protocoles.

C++

```
class MouseListener
{
  public:
    virtual bool mousePressed(void) = 0; //méthode virtuelle pure
    virtual bool mouseClicked(void) = 0; //méthode virtuelle pure
}

class KeyboardListener
{
  public:
    virtual bool keyPressed(void) = 0; //méthode virtuelle pure
}

class Toto : public MouseListener,  public KeyboardListener {...}

//Toto DOIT implémenter mousePressed, mouseClicked et keyPressed
//On pourra donc l'utiliser comme auditeur d'événements
```

Objective-C

```
@protocol MouseListener
  -(bool) mousePressed;
  -(bool) mouseClicked;
@end

@protocol KeyboardListener
  -(bool) keyPressed;
@end

@interface Toto : NSObject  <MouseListener,  KeyboardListener>
{
...
}
@end

//Toto DOIT implémenter mousePressed, mouseClicked et keyPressed
//On pourra donc l'utiliser comme auditeur d'événements
```

En C++, un protocole s'implémente par une classe abstraite et des méthodes virtuelles pures. La classe abstraite du C++ est cependant plus puissante que le protocole d'Objective-C, car elle peut contenir des attributs.

En Objective-C, le protocole est un concept spécifique. La syntaxe à base de chevrons $<\ldots>$ n'a rien à voir avec les templates C++, qui n'existent pas en Objective-C.

Notez que l'on peut implémenter toutes les méthodes d'un protocole dans une classe, sans pour autant indiquer explicitement dans le code qu'elle y adhère. L'inconvénient est que la méthode `conformsToProtocol`: renvoit alors `NO`. Pour des raisons d'efficacité, cette fonction ne teste pas la conformité à un protocole méthode par méthode, mais se base sur la conformité explicite donnée par le développeur. Dans un tel cas, la réponse négative de `conformsToProtocol` n'empêche pas le programme de se comporter correctement par ailleurs. Voici le prototype de la méthode `conformsToProtocol` :

```
-(BOOL) conformsToProtocol:(Protocol*)
```

6.4.2 Protocole informel

Parfois, il est souhaitable qu'une classe adhère à un protocole, pour montrer qu'elle est candidate à une certaine tâche, mais sans pour autant l'obliger à implémenter *toutes* les méthodes du protocole. Par exemple, en Cocoa, la notion d'objet *délégué* est très présente : un objet peut se voir attribuer un assistant, auquel il peut déléguer certains travaux, mais pas forcément tous.

Une solution immédiate consiste à scinder un protocole formel en plusieurs protocoles formels, puis à n'adhérer qu'à un sous-ensemble, mais cela devient vite laborieux. Cocoa apporte une solution sous forme de *protocole informel*. Étonnamment, ce n'est pas avec une relaxation de protocole formel que l'on spécifie un protocole informel. On utilise un autre concept : la *catégorie de classe* (cf. section 6.5 page 23).

Supposons un service « traiter des dossiers ». Le problème est la présence de dossiers verts, bleus et rouges. Après tout, si une classe `Larbin` ne peut traiter que les dossiers bleus, elle peut quand même rendre service. Il ne serait pas pratique d'utiliser trois protocoles formels `TraiterDossierVert`, `TraiterDossierBleu` et `TraiterDossierRouge`. On ajoute plutôt à la classe `Larbin` une *catégorie* `TraiterDossier`, contenant les méthodes de traitement de dossier qu'elle est capable d'accomplir. Cela s'écrit alors ainsi, en spécifiant entre parenthèses le nom que l'on choisit pour la catégorie (expliquée plus précisément en section 6.5 page 23) :

```
@interface Larbin (TraiterDossier)
-(void) lireDossierBleu:(DossierBleu*) dossier;
-(void) jeterDossierBleu:(DossierBleu*) dossier;
@end
```

On peut imaginer d'autres classes utilisant la catégorie `TraiterDossier`, et proposant d'autres méthodes en rapport avec le service.

```
@interface LarbinConsciencieux (TraiterDossier)
-(void) traiterDossierBleu:(DossierBleu*) dossier;
-(void) traiterDossierRouge:(DossierRouge*) dossier;
@end
```

Un développeur tiers parcourant le code peut constater facilement que la classe a une catégorie `TraiterDossier`, il peut donc immédiatement la supposer candidate à certaines tâches, et n'a plus qu'à vérifier lesquelles exactement. S'il ne vérifie pas dans le code source, il peut toujours le faire à l'exécution :

```
if ([monLarbin respondsToSelector:@selector(traiterDossierBleu)])
  [monLarbin traiterDossierBleu:dossier];
```

Strictement parlant, le protocole informel n'a pas d'utilité pour le compilateur. En revanche, il est précieux comme auto-documentation du code, rendant plus aisée l'utilisation d'une bibliothèque développée par un tiers.

6.4.3 Qualificateurs pour messages entre objets distants

Le dynamisme d'Objective-C permet à des objets distants de communiquer entre eux. Ils peuvent appartenir à des programmes distincts, sur des machines différentes, mais sont capables de se déléguer des tâches et de s'échanger des informations. Or, les protocoles formels sont un moyen idéal pour certifier des objets conformes à un service donné, quelle que soit leur origine. Le concept de protocole formel a donc été enrichi de mots-clés supplémentaires pour permettre une implémentation plus efficace des envois de message à distance.

Ces mots-clés sont `in`, `out`, `inout`, `bycopy`, `byref` et `oneway`. Hors de la définition d'un protocole, ils ne sont pas réservés par le langage et peuvent être utilisés librement.

Ces mots-clés sont insérés dans les prototypes des méthodes déclarées par un protocole formel pour en préciser le comportement. On peut ainsi savoir si les paramètres correspondent à des données d'entrée, de sortie ; on connaît leur mode de passage (par copie ou par référence) ; on sait si la méthode est synchrone ou non.

Les significations sont les suivantes :
- un paramètre spécifié `in` est une variable d'entrée ;
- un paramètre spécifié `out` est une variable de sortie ;
- un paramètre spécifié `inout` peut être utilisé comme entrée et sortie ;
- un paramètre spécifié `bycopy` est passé par copie ;
- un paramètre spécifié `byref` est passé par référence (sans copie) ;
- une méthode spécifiée `oneway` est asynchrone (on n'attend pas de retour immédiat), et a forcément `void` comme type de retour.

Voici par exemple une méthode asynchrone qui demande un objet :

```
-(oneway void) donneMoiUnObjet:(bycopy out id *) unObjet;
```

Par défaut, les paramètres sont considérés `inout`, sauf les pointeurs `const`, qui sont considérés `in`. Réduire le sens à `in` ou `out` est une optimisation. Le mode de passage par défaut des paramètres est `byref`, et le comportement par défaut des méthodes est synchrone (sans `oneway`).

Pour les arguments passés par valeur, comme des variables non-pointeurs, `out` et `inout` ne signifient rien, seul `in` est correct.

6.5 Catégories de classe

Implémenter des *catégories* pour une classe permet de morceler sa définition et son implémentation. Chaque *catégorie* est un élément constituant de la classe. Une classe peut être implémentées par un nombre quelconque de catégories, mais elles ne peuvent déclarer de nouvelles données d'instance. On bénéficie alors des avantages suivants :

- Pour le développeur pointilleux, cela permet de regrouper des méthodes. Pour les classes très riches, cela permet d'isoler les différents rôles ;
- En conséquence immédiate, on bénéficie à la fois d'une compilation séparée, et de possibilités de travailler sur la même classe à plusieurs ;
- Si une interface et une implémentation de catégorie sont dans un fichier d'implémentation quelconque (fichier *.m*), cela revient à définir des méthodes *privées*, visibles uniquement dans ce fichier. Dans ce cas, le nom de la catégorie est souvent choisi comme *TotoPrivateAPI* ;
- Cela permet également de personnaliser une classe différemment pour plusieurs applications, sans avoir à dupliquer le code source commun. N'importe quelle classe peut ainsi être étendue, même celles de Cocoa.

Le dernier point est important : en effet, pour nos besoins particuliers, de petites méthodes supplémentaires dans les classes standards seraient parfois agréables. Ce n'est pas un problème en soi, puisqu'il suffit d'une dérivation pour étendre les fonctionnalités d'une classe. Cependant, dans un contexte d'héritage simple, cela pourrait générer une arborescence de classe un peu pénible. De plus, il peut être laborieux de créer, par exemple, une nouvelle classe `MyString` juste pour une méthode, et de devoir l'utiliser ensuite dans tout le programme. Les catégories de classe permettent de résoudre élégamment ce problème.

C++

```
class MyString : public string
{
  public:
    int compterVoyelles(void); //compte les voyelles
};

int MyString::compterVoyelles(void)
{
...
}
```

Objective-C

```
@interface NSString (QuiCompteLesVoyelles)
//Notez l'absence de {}
-(int) compterVoyelles; //compter les voyelles
@end

@implementation NSString (QuiCompteLesVoyelles)
-(int) compterVoyelles
{
...
}
@end
```

En C++, la nouvelle sous-classe est utilisable sans restrictions.

En Objective-C, la classe NSString (c'est une classe de Cocoa) se voit attribuer une extension valable dans l'intégralité du programme. Aucune nouvelle classe n'est créée. Tout objet NSString bénéficie de l'extension. Attention : lors de la déclaration d'une catégorie de classe, aucune variable d'instance ne peut être ajoutée. Il n'y a donc pas de bloc *{...}*

6.6 Utilisation conjointe de protocoles, catégories, dérivation :

La seule restriction dans l'utilisation conjointe de protocoles, catégories et dérivation et de ne pas pouvoir en même temps déclarer une sous-classe et implémenter une catégorie ; cela nécessite deux étapes.

```
@interface Toto1 : SuperClasse <Protocole1, Protocole2, ... > //ok
@end

@interface Toto2 (Categorie)  <Protocole1, Protocole2, ... > //ok
@end

 //ci-dessous : erreur de compilation
@interface Toto3 (Categorie)  : SuperClasse <Protocole1, Protocole2, ... >
@end

//pour bien faire :
@interface Toto3 : SuperClasse  //étape 1
@end
@interface Toto3 (Categorie) <Protocole1, Protocole2, ... >  //étape 2
@end
```

7 Instanciation

L'instanciation d'une classe soulève deux problèmes : comment est implémentée la notion de constructeur/destructeur/opérateur de copie, et comment est gérée la mémoire ?

D'abord une remarque très importante : en C et en C++, les variables sont dites « automatiques » par défaut : à moins d'être déclarées `static`, elles n'existent que dans leur bloc de définition. Seule la mémoire allouée dynamiquement persiste jusqu'au `free()` ou au `delete` adapté. Les objets n'échappent pas à cette règle en C++.

Cependant, en Objective-C, **tout objet est créé dynamiquement**. Cet état de fait est somme toute logique, le C++ étant un langage typé statiquement, et l'Objective-C étant dynamique. Le dynamisme serait entravé si les objets n'étaient pas créés lors de l'exécution du programme.

Voyez la section 8 page 32 pour des explications plus poussées sur la façon de maintenir ou libérer les objets.

7.1 Constructeurs, initialisateurs

7.1.1 Distinction entre *allocation* et *initialisation*

En C++, l'allocation et l'initialisation d'un objet sont confondues dans l'appel au constructeur. En Objective-C, ce sont deux méthodes différentes.

L'allocation est assurée par la **méthode de classe** `alloc`, qui a également pour effet d'initialiser toutes les données d'instance. Les données d'instance sont initialisées à 0, excepté le pointeur `isa` (est-un) de `NSObject`, initialisé de telle sorte qu'il décrive le type exact de l'objet lors de l'exécution du code. Si les données d'instance doivent être initialisées à des valeurs particulières, dépendantes des paramètres de construction, le code correspondant est déporté dans une **méthode d'instance**, dont le nom commence traditionnellement par *init*. La construction est ainsi clairement séparée en deux étapes : l'allocation et l'initialisation. Notez que le message `alloc` est envoyée à la *classe*, et le message `init...` est envoyé à l'*objet* instancié par `alloc`.

La phase d'initialisation n'est pas optionnelle, et un `alloc` devrait toujours être suivi d'un `init`, lequel, par le jeu des appels aux constructeurs des super-classes, doit finir par déclencher le `init` de `NSObject` assurant différentes tâches importantes.

En C++, le nom du constructeur est imposé par le langage. En Objective-C, ce n'est pas le cas, car à part le préfixe `init`, qui est traditionnel sans être obligatoire, le nom de la méthode est libre. Il est cependant très déconseillé de ne pas respecter ce canon, à tel point qu'il vaut mieux l'énoncer comme une loi : **le nom d'une méthode d'initialisation doit commencer par « init »**.

7.1.2 Utilisation de `alloc` et `init`

L'appel à `alloc` renvoie l'objet nouvellement créé sur lequel on effectue le `init`. L'appel à `init` renvoie aussi un objet. Dans la plupart des cas, ce sera l'objet initial. Mais parfois, comme par exemple si on utilise un singleton (objet dont on ne veut pas plus d'une instance à la fois), le `init` peut se permettre de substituer une autre valeur de retour. Il ne faut donc pas ignorer la valeur de retour d'un `init` ! Généralement, on enchaîne donc les appels à `alloc` et `init` sur la même ligne d'instructions.

C++

```
Toto* toto = new Toto;
```

Objective-C

```
Toto* toto1 = [Toto alloc];
[toto1 init]; //mauvais, on doit se soucier de la valeur de retour

Toto* toto2 = [Toto alloc];
toto2 = [toto2 init]; //ok, mais pas pratique

Toto* toto3 = [[Toto alloc] init]; //ok, c'est ainsi que l'on fait
```

Notez que pour savoir si un objet a été effectivement créé, C++ nécessite soit une interception d'exception, soit un test avec un 0 si `new(nothrow)` a été utilisé. Avec Objective-C, il suffit de tester si l'objet est à `nil`.

7.1.3 Exemple d'initialisateur correct

Les contraintes d'un initialisateur sont donc :
- d'avoir un nom commençant par *init* ;
- de renvoyer l'objet initialisé à utiliser ;
- d'appeler un `init` de la classe mère, de telle sorte qu'au moins le `init` de `NSObject` soit appelé.

Ci-après est donné un code d'exemple de construction d'objet en C++ et en Objective-C.

C++

```
class Point2D
{
  public:
    Point2D(int x, int y);
  private:
    int x;
    int y;
};
Point2D::Point2D(int unX, int unY) {x = unX; y = unY;}
...
Point2D  p1(3,4);
Point2D* p2 = new Point2D(5, 6);
```

Objective-C

```
@interface Point2D : NSObject
{
    int x;
    int y;
}
//Remarque : id est un peu l'équivalent de void* en Objective-C.
//(id) est le type "général" d'un objet.
-(id) initWithX:(int)unX andY:(int) unY;

@implementation Point2D

-(id) initWithX:(int)unX andY:(int)unY
{
  if (![super init]) //il faut penser à appeler un constructeur...
     return nil;     //...de la superclasse. Si la superclasse est...
                     //...NSObject, c'est simplement init.
  //si l'initialisation a marché...
  x = unX;
  y = unY;
  return self;
}
@end
...
Point2D* p1 = [[Point2D alloc] initWithX:3 andY:4];
```

Polémique : Beaucoup de gens utilisent la forme `self = [super init] ; if (self) {...}` dans l'initialisateur. Cette pratique est justifiée par le fait que dans certains cas, `[super init]` pourrait renvoyer un objet différent. Cependant, Will Shipley montre dans un document très intéressant [6] que c'est une mauvaise pratique. Il est beaucoup plus cohérent d'utiliser la forme de l'exemple ci-dessus, où l'on teste que `[super init]` renvoie `nil` ou non.

Le principe de l'application successive de `alloc` puis `init` peut sembler laborieux dans certains cas. Il peut fort heureusement être court-circuité par ce qu'on appelle un *constructeur de commodité*. Expliquer ce qu'est un tel constructeur nécessite des connaissances sur la gestion mémoire en Objective-C. Les véritables explications sont donc déportées plus loin en 8.4.6 page 35. Brièvement, un tel constructeur, dont le nom devrait toujours être préfixé par celui de la classe, a le même rôle qu'une méthode d'`init`, mais elle fait le `alloc` elle-même. Cependant, l'objet renvoyé est inscrit au bassin de libération programmée (*autorelease pool*, cf. section 8.4 page 33) et sera donc temporaire s'il ne lui est pas envoyé un `retain`. Un exemple d'utilisation est donné ci-après :

```
//laborieux
NSNumber* tmp1 = [[NSNumber alloc] initWithFloat:0.0f];
...
[tmp1 release];

//plus sympathique
NSNumber* tmp2 = [NSNumber numberWithFloat:0.0f];
...
//pas besoin de release
```

7.1.4 Échec de l'initialisation

Nous avons vu que l'initialisateur débute par un appel à celui de la superclasse. Si ce dernier échoue, la valeur `nil` est renvoyée. Mais dans ce cas, que devient l'objet en cours de construction ? En renvoyant nil pour indiquer l'échec, ne génère-t-on pas une fuite mémoire ? Ne devrait-on pas demander à l'objet de se désallouer, puisque le `alloc`, lui, a bien réservé la mémoire ?

En fait, ce processus est automatique. Si la superclasse est bien faite, elle provoque bien un `dealloc` (voire section 7.2 page 29) en cas d'échec. Mais si vous, pour une raison quelconque, décidez d'interrompre votre initialisateur, et de renvoyer `nil`, vous devez bien vous désallouez vous-même, avec un `[self autorelease]` (la gestion de la mémoire est expliquée en section 8 page 32).

7.1.5 Constructeur par défaut : initialisateur désigné

La notion de constructeur par défaut n'a pas vraiment de sens en Objective-C. Les objets étant tous alloués dynamiquement, leur construction est toujours explicite. Cependant, on retrouve la problématique d'un initialisateur privilégié pour éviter une redondance de code malvenue. En effet, un initialisateur correct contient la plupart du temps un code similaire à :

```
if (![super init]) //"init" ou un autre initialisateur
  return nil;      // plus approprié de la superclasse

//en cas de succès...
//...ajouter du code...
return self;
```

Toute redondance de code étant à proscrire, il semble inopportun de reproduire ce schéma dans chaque initialisateur. La meilleure solution consiste effectivement à privilégier l'initialisateur le plus essentiel pour y placer ce code. Les autres initialisateurs ne feront qu'appeler cet initialisateur « de

base », nommé *initialisateur désigné* (*designated initializer*). Logiquement, l'initialisateur désigné est celui qui a le plus de paramètres, puisqu'il est impossible en Objective-C de donner des valeurs par défaut aux paramètres.

```
-(id) initWithX:(int)x
{
  return [self initWithX:x andY:0 andZ:0];
}

-(id) initWithX:(int)x andY:(int)y
{
  return [self initWithX:x andY:y andZ:0];
}

//initialisateur désigné
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (![super init])
    return nil;
  self->x = x;
  self->y = y;
  self->z = z;
  return self;
}
```

Si l'initialisateur désigné n'est pas celui qui a le plus de paramètres, ce n'est pas très pratique :

```
//Le code suivant n'est pas bien pratique.
-(id) initWithX:(int)x //initialisateur désigné
{
  if (![super init])
    return nil;
  self->x = x;
  return self;
}
-(id) initWithX:(int)x andY:(int)y
{
  if (![self initWithX:x])
    return nil;
  self->y = y;
  return self;
}
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (![self initWithX:x])
    return nil;
  self->y = y;
  self->z = z;
  return self;
}
```

7.1.6 Listes d'initialisation et valeur par défaut des données d'instance

Les *listes d'initialisation* des constructeur C++ n'existent pas en Objective-C. Mais il est important de noter que contrairement au C++, en Objective-C, **les bits des données d'instance sont tous initialisés à 0** par `alloc`. Tous les types de base prennent donc la valeur 0 par défaut, ainsi que les pointeurs (qui sont donc à `nil`). Rappelons que cela ne pose pas de problèmes pour les éventuelles données d'instance qui seraient des objets, car elles ne pourraient être en réalité que des pointeurs d'objets : en Objective-C, les objets ne peuvent être alloués que dynamiquement.

7.1.7 Constructeur virtuel

Il est possible en Objective-C d'obtenir de véritables constructeurs virtuels. Voyez pour cela la section 8.4.6 page 35, présentée après l'introduction à la gestion de la mémoire (section 8 page 32).

7.1.8 Constructeur de classe

Les classes étant elles-mêmes des objets manipulables en Objective-C, elles bénéficient également d'un constructeur que l'on peut redéfinir. Il s'agit bien sûr d'une méthode de classe (héritée de NSObject), de prototype `+(void) initialize ;`

7.2 Destructeurs

En C++, le destructeur, tout comme le constructeur, est une méthode particulière que l'on peut redéfinir. En Objective-C, c'est une méthode d'instance appelée `dealloc`.

En C++, le destructeur est appelé automatiquement quand on demande la libération d'un objet ; en Objective-C, c'est la même chose. Seule la façon de libérer l'objet change (cf. section 8 page 32).

Le destructeur ne devrait jamais être appelé explicitement. En réalité, il existe en C++ un seul cas où un destructeur peut être appelé explicitement : cela survient si le développeur gère lui-même le bassin de mémoire utilisé pour l'allocation. Mais en Objective-C, aucun cas ne justifie un appel explicite à `dealloc`. On peut utiliser des zones mémoires personnalisées grâce à Cocoa, mais leur utilisation ne perturbe pas les pratiques d'allocation/désallocation usuelles (cf. section 7.3 page suivante).

C++

```
class Point2D
{
  public:
    ~Point2D();
};

Point2D::~Point2D() {}
```

Objective-C

```
@interface Point2D : NSObject
-(void) dealloc; //on peut redéfinir cette méthode
@end

@implementation Point2D
//dans cet exemple, ce n'était pas la peine de redéfinir dealloc
-(void) dealloc
{
  [super dealloc]; //penser à la super classe
}
```

7.3 Opérateurs de copie

7.3.1 Clonage classique, `copy` et `copyWithZone`

En C++, il est important de définir une implémentation cohérente du constructeur par recopie et de l'opérateur d'affectation. En Objective-C, la surcharge d'opérateur étant impossible, on se contente du concept de clonage d'objet.

Le clonage est associé en Cocoa à un protocole (cf. section 6.4 page 20), du nom de `NSCopying` demandant l'implémentation de la méthode

```
-(id) copyWithZone:(NSZone*)zone;
```

dont l'argument, inattendu à première vue, est une zone mémoire dans laquelle allouer le clone. Cocoa permet en effet de gérer des réserves de mémoire personnalisées : les méthodes concernées peuvent prendre une zone mémoire en argument. Dans la plupart des cas, la zone mémoire par défaut est idéale, et il serait lourd de perpétuellement devoir la spécifier en argument. Heureusement, `NSObject` fournit une méthode

```
-(id) copy;
```

qui encapsule un simple appel à `copyWithZone`, avec en paramètre la zone par défaut. Mais c'est bien `copyWithZone` qui est déclarée dans `NSCopying`. L'implémentation de `copyWithZone` pour une classe quelconque Toto aura sans doute la forme suivante :

```
//si la superclasse n'implémente pas copyWithZone
-(id) copyWithZone:(NSZone*)zone
{
  Toto* clone = [[Toto allocWithZone:zone] init];
  //Certaines données d'instance doivent sûrement être recopiées "à la main"
  clone->age = self->age; //exemple...
  return clone;
}
```

Il ne faut cependant pas oublier de tenir compte des implémentations éventuelles de `copyWithZone` dans une superclasse.

```
//si la superclasse implémente copyWithZone
-(id) copyWithZone:(NSZone*)zone
{
  Toto* clone = [super copyWithZone:zone];
  //recopies de données d'instances spécifiques à la sous-classe courante
  clone->age = self->age; //exemple...
  return clone;
}
```

Remarquez dans le premier exemple (où la superclasse n'implémente pas `copyWithZone`) l'utilisation de `allocWithZone` à la place d'un simple `alloc`, pour prendre en compte la zone passée en argument. En réalité, `alloc` encapsule un simple appel à `allocWithZone` avec la zone par défaut. Pour en savoir plus sur la gestion de zones mémoires personnalisées, consultez plutôt une documentation Cocoa.

7.3.2 Clonage optimal, mutabilité, `mutableCopy` et `mutableCopyWithZone`

Si l'on clone un objet qui ne peut changer, une optimisation fondamentale consiste à ne pas réellement le dupliquer, mais à renvoyer une simple référence vers lui. Partant de ce principe, on distingue la notion d'objet *non modifiable* et d'objet *modifiable* (*mutable*).

Un objet non modifiable n'a aucune méthode permettant de changer la valeur de ses données d'instance ; seul le constructeur lui a donné un état. Dans ce cas, on peut sans risques le « pseudo-cloner » en ne renvoyant comme clone qu'une simple référence vers lui-même. Comme ni l'original ni son clone ne peuvent être modifiés, aucun des deux ne risque d'être silencieusement affecté par une perturbation de l'autre. Une implémentation très efficace de `copyWithZone` peut donc être proposée :

```
-(id) copyWithZone:(NSZone*)zone
{
  //renvoie l'objet lui-même (référencé une fois de plus)
  return [self retain]; //voyez la section sur la gestion mémoire
}
```

L'utilisation de `retain` découle de la gestion de la mémoire en Objective-C (cf. section 8 page suivante). On incrémente de 1 le compteur de références de l'objet pour officialiser l'existence du clone, de telle sorte qu'une demande de suppression de ce dernier n'entraîne pas la suppression de l'original.

Ce « pseudo-clonage » n'est pas une optimisation marginale. La création d'un objet demande une allocation de mémoire, ce qui est un processus « long » qu'il vaut mieux éviter si c'est possible. Il est donc intéressant de classer les objets en deux familles : les objets non modifiables, pour lesquels on est sûr que le clonage est optimal, et les autres. Pour réaliser la distinction, il suffit de créer des classes « non-mutables », et d'éventuellement les dériver en versions « mutables », par adjonction de méthodes permettant de modifier les données d'instance. Par exemple, en Cocoa, `NSMutableString` dérive de `NSString`, `NSMutableArray` dérive de `NSArray`, `NSMutableData` dérive de `NSData`, etc.

Cependant, avec les techniques présentées jusqu'ici, il semble impossible d'obtenir un véritable clone, modifiable, d'un objet non modifiable qui ne saurait que se « pseudo-cloner ». Une telle lacune limiterait grandement l'intérêt des objets non modifiables en les isolant trop du monde extérieur.

En plus du protocole `NSCopying`, il existe donc un protocole (cf. section 6.4 page 20) du nom de `NSMutableCopying`, demandant l'implémentation de

```
-(id) mutableCopyWithZone:(NSZone*)zone;
```

La méthode `mutableCopyWithZone` doit renvoyer un clone modifiable, dont les perturbations n'affectent pas l'original. De façon similaire à la méthode `copy`, il existe une méthode `mutableCopy` qui simplifie l'appel à `mutableCopyWithZone` en lui passant automatiquement la zone par défaut en argument. L'implémentation de `mutableCopyWithZone` aura sans doute la forme suivante, similaire à celle présentée pour la copie classique dans la section précédente :

```
//si la superclasse n'implémente pas mutableCopyWithZone
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Toto* clone = [[Toto allocWithZone:zone] init];
  //Certaines données d'instance doivent sûrement être recopiées "à la main"
  clone->age = self->age; //exemple...
  return clone;
}
```

Ne toujours pas oublier de tenir compte des implémentations éventuelles de `mutableCopyWithZone` dans une superclasse :

```
//si la superclasse implémente mutableCopyWithZone
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Toto* clone = [super mutableCopyWithZone:zone];
  //recopies de données d'instances spécifiques à la sous-classe courante
  clone->age = self->age; //exemple...
  return clone;
}
```

8 Gestion mémoire

8.1 `new` et `delete`

Les mots-clés `new` et `delete` du C++ n'existent pas en Objective-C. Ils sont respectivements remplacés par un appel à `alloc` (cf. section 7.1 page 25) et à `release` (cf. section 8.2 de la présente page).

8.2 Compteur de références

La gestion mémoire en Objective-C est un des points les plus importants du langage. En langage C ou C++, une zone mémoire est allouée une fois et détruite une fois. On peut y faire de multiples références en utilisant le nombre de pointeurs approprié. Sur un seul des pointeurs sera effectué `delete`.

L'Objective-C implémente quant à lui un système de comptage de références. L'objet est conscient du nombre de liens dirigés vers lui. On peut expliquer ce principe par la métaphore du chien et des laisses (exemple directement tiré de *Cocoa Programming for MacOS X* [4]). Si l'objet est un chien, chacun peut réclamer une laisse pour le maintenir en place. Si quelqu'un se désintéresse du chien, il peut lâcher la laisse. Tant que le chien a au moins une laisse, il est contraint de rester en place. Mais dès que le nombre de laisses tombe à 0, le chien est... libéré !

Plus techniquement, le compteur de référence d'un objet nouvellement créé est fixé à 1. Si une portion du code nécessite de référencer cet objet durablement, elle lui envoie un message `retain`, qui incrémente de 1 le compteur. Si une portion de code référençant l'objet n'en a plus l'utilité, elle lui envoie un message `release`, qui décrémente le compteur de 1.

Un objet peut recevoir un nombre quelconque de `retain` et de `release`, tant que le compteur de références a une valeur strictement positive. En effet dès qu'il tombe à 0, le destructeur `dealloc` est automatiquement appelé. Envoyer de nouveaux `release` à l'adresse, devenue invalide, de l'objet, provoque une faute mémoire.

Notez que cette technique n'est pas équivalente aux `auto_ptr` de la STL. En revanche, la bibliothèque Boost [2] propose une encapsulation des pointeurs nommée `shared_ptr`, qui implémente le comptage de références. Elle ne fait cependant pas partie du standard.

8.3 `alloc`, `copy`, `mutableCopy`, `retain`, `release`

Savoir comment fonctionne le gestionnaire mémoire n'explique pas complètement comment il s'utilise. Le but de cette section est de donner quelques règles fondamentales d'utilisation. Le mot-clé `autorelease` est mis à part à dessein, car il est assez subtil à comprendre.

La règle de base à appliquer est **Tout responsable de l'incrémentation du compteur de référence, *via* `alloc`, `[mutable]copy` ou `retain` est chargé d'appliquer le ou les `[auto]release` correspondants**. Ce sont effectivement les trois façons d'incrémenter le compteur de références. Cela signifie qu'il ne faut se préoccuper de faire un `release` que :

- si on instancie explicitement un objet avec `alloc` ;
- si on crée explicitement une copie de cet objet avec `copy` ou `mutableCopy` (que la copie soit techniquement une duplication ou, si elle est non mutable, que ce soit un simple pointeur vers l'objet initial, n'a aucune importance, cela doit rester transparent) ;
- si on effectue un `retain` explicite.

L'usage de `retain` est assez restreint. Mais si son emploi est restreint, cela ne signifie pas pour autant qu'il est rare. Dans la section suivante, le concept de `autorelease` met en avant l'utilisation la plus courante de `retain`.

Notez enfin qu'il est légal d'envoyer `release` à `nil`. Cela n'a simplement aucun effet et simplifie le code en diminuant le nombre de tests à implémenter.

8.4 autorelease

8.4.1 Indispensable autorelease

La règle citée dans la section précédente est si importante qu'il n'est pas inutile de la répéter : **Tout responsable de l'incrémentation du compteur de référence, *via* `alloc`, `[mutable]copy` ou `retain` est chargé d'appliquer le ou les `[auto]release` correspondants**.

Assurément, avec les seuls `alloc`, `retain` et `release`, cette règle serait absolument inapplicable. En effet, il existe des fonctions qui ne sont pas des constructeurs, mais qui ont pour but de construire des objets : par exemple un opérateur binaire d'addition en C++. Dans le cas du C++, l'objet renvoyé par la fonction est déposé sur la pile, et sera détruit automatiquement lorsqu'il deviendra caduc. Or, en Objective-C, les objets automatiques n'existent pas. Une telle fonction a donc forcément utilisé un `alloc`, mais ne peut détruire l'objet avant de le déposer sur la pile ! Ci-après son illustrés quelques exemples problématiques d'utilisation.

```
(Point2D*) add:(Point2D*) p1 and:(Point2D*) p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  return result;
}
//ERREUR : la fonction effectue un "alloc", donc elle produit
//un objet dont le compteur de référence vaut 1. Pourtant,
//d'après la règle, elle est chargée de le supprimer.

//Exemple de fuite mémoire induite en cas d'addition de trois Point2D:
[calculator add:[calculator add:p1 and:p2] and:p3];

//le résultat de la première addition est anonyme, donc personne
//ne peut le libérer. C'est une fuite mémoire.
```

```
(Point2D*) add:(Point2D*) p1 and:(Point2D*) p2
{
  return [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                               andY:([p1 getY] + [p2 getY])];
}
//ERREUR : c'est exactement le même code que précédemment.
//Ne pas utiliser de variable intermédiaire ne change absolument
//rien au problème.
```

```
(Point2D*) add:(Point2D*) p1 and:(Point2D*) p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result release];
  return result;
}
//ERREUR : bien sûr, il ne rime à rien de créer l'objet puis de le
//détruire.
```

Le problème semble insoluble ; il le serait si `autorelease` n'existait pas. Pour faire simple, disons qu'envoyer `autorelease` à un objet revient à lui envoyer un `release` qui sera appliqué « un peu plus tard ». Attention, « un peu plus tard » ne veut pas dire « n'importe quand ». Mais dans un premier temps, montrons son utilisation dans la seule solution possible :

```
(Point2D*) add:(Point2D*) p1 and:i(Point2D*) p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                        andY:([p1 getY] + [p2 getY])];
  [result autorelease];
  return result;
}
//CORRECT : "result" sera automatiquement libéré plus tard,
//après son éventuelle utilisation dans toute fonction appelante.
```

8.4.2 Bassin d'autorelease

D'après la section précédente, le `autorelease` est en quelque sorte un `release` magique qui sera appliqué au bon moment. Il ne serait cependant ni raisonnable ni efficace de laisser deviner au compilateur quel est ce bon moment. Autant utiliser un ramasse-miettes dans ce cas. Pour y remédier, il convient d'expliquer comment fonctionne `autorelease`.

A chaque fois qu'un objet reçoit un `autorelease`, il est inscrit dans un bassin de désallocation automatique (*autorelease pool*). Quand ce bassin est vidé, l'objet reçoit un `release` effectif. Le problème s'est déplacé : comment gère-t-on ce bassin ?

La réponse est double : si vous utilisez Cocoa pour une application avec interface utilisateur, vous n'avez rien besoin de faire. Sinon, il vous faut créer le bassin et le vider vous même.

Une application avec interface utilisateur utilise une *boucle événementielle*. C'est une boucle qui attend une action de l'utilisateur pour réveiller le programme et lui transmettre cette action. En Cocoa, lorsque l'on programme une telle application, un bassin d'autorelease est automatiquement créé à chaque réveil du programme, et vidé à chaque remise en attente. C'est tout à fait logique : généralement, une action de l'utilisateur provoque une succession de tâches. Des objets temporaires sont créés, puis détruits, car ils n'ont pas besoin d'être conservés pour la suite des événements. Si certains d'entre eux doivent l'être, c'est au développeur de prévoir les `retain` nécessaires.

En revanche, s'il n'y a pas d'interface utilisateur, il faut créer un bassin d'autorelease en amont du code le nécessitant. Lorsqu'un objet reçoit `autorelease`, il sait automatiquement trouver le bassin d'allocation actuellement déclaré, il n'y a pas besoin de le lui spécifier. Ensuite, quand il est opportun de vider le bassin, il faut en fait le détruire, avec un simple `release`. Typiquement, un programme Cocoa destiné à la ligne de commande contient :

```
int main(int argc, char* argv[])
{
  NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
  //...
  [pool release];
  return 0;
}
```

8.4.3 Utilisation de plusieurs bassins d'autorelease

Il est possible, et parfois utile, d'avoir plusieurs bassins d'autorelease dans le même programme. Un objet recevant `autorelease` s'inscrira dans le dernier bassin créé. De ce fait, si une fonction crée et utilise un grand nombre d'objets temporaires, un gain de performance peut être obtenu en créant un bassin d'autorelease local. De cette manière, la foule d'objets temporaires sera détruite au plus vite, et n'encombrera pas la mémoire inutilement. Même si elle mérite d'être connue, cette technique n'est cependant utilisée que dans des cas assez rares.

8.4.4 Prudence avec `autorelease`

Ce n'est pas parce qu'`autorelease` est pratique qu'il doit être mal exploité.

- D'abord, envoyer plus de `autorelease` que nécessaire est aussi néfaste que d'envoyer trop de `release` : cela provoque une faute mémoire lors du vidage du bassin ;

– Ensuite, même s'il est vrai que dans un programme Objective-C, n'importe quel message `release` peut être remplacé par un message `autorelease`, cela serait au prix d'une baisse de performance, car le bassin d'autorelease est plus lent à gérer qu'un `release` normal. De plus, remettre à plus tard toute libération tendrait à provoquer des pics d'utilisation mémoire inutiles et malvenus.

8.4.5 `autorelease` et `retain`

Grâce à `autorelease`, une méthode qui crée un objet est capable de planifier elle-même la libération. Pourtant, il est courant que l'objet construit doive être conservé. Dans ce cas, il suffit d'appliquer un `retain` sur le retour de la fonction. Mais il faut alors planifier un `release` au moment opportun. C'est le cas d'utilisation le plus courant de `retain`, typiquement illustré en sortie d'un constructeur de commodité (cf. section 8.4.6 de la présente page) ou dans un mutateur (cf. section 8.4.7 page suivante).

8.4.6 Constructeurs de commodité, constructeur virtuel

Le principe de l'application successive de `alloc` puis de `init` lors de l'instanciation d'une classe est laborieux dans certains cas. Il peut fort heureusement être court-circuité par ce qu'on appelle un *constructeur de commodité*. Un tel constructeur, dont le nom devrait toujours être préfixé par celui de la classe, a le même rôle qu'une méthode d'`init`, mais il fait le `alloc` lui-même. Cependant, l'objet renvoyé est inscrit au bassin d'autorelease, et sera donc temporaire s'il ne lui est pas envoyé un `retain`. Exemple d'utilisation :

```
//laborieux
NSNumber* zero_a = [[NSNumber alloc] initWithFloat:0.0f];
...
[zero_a release];

...

//plus sympathique
NSNumber* zero_b = [NSNumber numberWithFloat:0.0f];
...
//pas besoin de release
```

Si vous avez lu la section consacrée à la gestion de la mémoire (section 8 page 32), vous devinez qu'un tel constructeur repose sur `autorelease`. Le code correspondant n'est cependant pas évident, car il nécessite d'utiliser `self` correctement. En effet, un constructeur de commodité est une *méthode de classe*, donc `self` se réfère à un objet `Class`, qui est une instance de *méta-classe*. Dans un *initialisateur*, qui est une méthode *d'instance*, `self` est une *instance* de la classe, et donc se réfère à un objet « normal ».

Il est facile d'écrire un mauvais constructeur de commodité. Supposons pour l'exemple l'existence d'une classe `Vehicule` dotée d'une couleur et d'un constructeur de commodité.

```
//La classe Vehicule
@interface Vehicule : NSObject
{
  NSColor* couleur;
}
-(void) setCouleur:(NSColor*) couleur;

//constructeur de commodité
+ (id) vehiculeDeCouleur:(NSColor *) couleur;
@end
```

L'implémentation du constructeur de commodité est assez particulière.

```
//Mauvais constructeur de commodité
+ (Vehicule *)vehiculeDeCouleur:(NSColor *) couleur
{
  // la valeur de "self" ne devrait pas changer !
  self = [[self alloc] init]; // ERREUR !
  [self setCouleur:couleur];
  return [self autorelease];
}
```

`self` dans cette méthode de classe se réfère à la *classe*. Il ne faut donc pas lui attribuer une *instance*.

```
//Constructeur presque parfait
+ (id)vehiculeDeCouleur:(NSColor *) couleur
{
  id nouvelleInstance = [[self alloc] init]; // OK
  [nouvelleInstance setCouleur:couleur];
  return [nouvelleInstance autorelease];
}
```

Une dernière subtilité subsiste : il est possible en Objective-C d'avoir un constructeur virtuel. Il suffit que le constructeur de commodité fasse une introspection pour savoir quel est le type de classe réel de l'objet exécutant la méthode. Dans ce cas, il produit directement un objet du bon type dérivé. On a besoin pour cela du faux mot-clé `class`, qui renvoie l'objet de classe (l'instance de la méta-classe). `class` est en fait une méthode de `NSObject`.

```
@implementation Vehicule
 + (id)vehiculeDeCouleur:(NSColor *)couleur
 {
   id nouvelleInstance = [[[self class] alloc] init]; // PARFAIT
   [nouvelleInstance setCouleur:couleur];
   return [nouvelleInstance autorelease];
}
@end

@interface Voiture : Vehicule {...}
@end

...

//produit une voiture (rouge) !
id voiture = [Voiture vehiculeDeCouleur:[NSColor redColor]];
```

Tout comme pour la règle du préfixe **init** sur les initialisateurs, il est très déconseillé de ne pas préfixer un constructeur de commodité par le nom de la classe. Il n'y a que peu de situations pour lesquelles cette règle est contournée, comme par exemple le `[NSColor redColor]` du code précédent, qui aurait dû s'écrire `[NSColor colorRed]`.

Enfin, rappelons la loi : **Tout responsable de l'incrémentation du compteur de référence, *via* `alloc`, `[mutable]copy` ou `retain` est chargé d'appliquer le ou les `[auto]release` correspondants**. En appelant un constructeur de commodité, on n'effectue pas soi-même le `alloc`, donc on n'est pas chargé du `release`. Par contre, lorsqu'on écrit un tel constructeur, on écrit `alloc`, donc on doit bien mettre `autorelease`.

8.4.7 Accesseurs en écriture : Mutateurs

Un accesseur en écriture, ou *mutateur*, est l'exemple typique de ce que l'on ne sait pas écrire quand on ne connaît pas la gestion de la mémoire en Objective-C. Supposons une classe encapsulant

une `NSString` appelée *chaine*, et supposons que l'on veuille changer la valeur de cette chaîne de caractères. Cet exemple anodin suffit à soulever le problème principal de la création des mutateurs. Il est important de faire dès maintenant un choix définitif de *style* de programmation, qui doit être réfléchi. En effet, lors de l'appel à un mutateur, l'objet passé en paramètre peut avoir différentes significations :

- S'il a pour vocation de n'être qu'un pointeur vers un objet extérieur à l'objet encapsulateur, alors il ne faut pas le recopier, il faut juste retenir son adresse. Ainsi, toute modification extérieure sera visible depuis l'objet encapsulateur (cas n°1) ;
- s'il a pour vocation de n'être qu'une copie d'un objet existant à l'extérieur, alors il faut le recopier complètement (cas n°2) ;
- dans le cas n°2, si l'objet est temporaire et n'a été créé que pour cet appel, une excellente optimisation consiste à ne pas le recopier, mais à simplement le référencer (avec `retain`). Grâce au compteur de référence, il peut donc être « virtuellement détruit » pour l'extérieur (qui lui envoie `release`), mais continuer d'exister dans l'objet encapsulateur (cas n°3).

En C++, l'optimisation du cas n°3 n'est pas possible, mais pour les cas n°1 et 2, on peut faire automatiquement la différence selon que l'objet encapsulé est un pointeur ou un objet « automatique » (non alloué par `new`). Ce n'est pas possible en Objective-C, où tous les objets sont alloués dynamiquement : on ne dispose que de pointeurs. Cette contrainte impose donc un choix difficile à première vue : qui est responsable de la recopie (cas n°2) ou de la non-recopie (cas n°1) de l'objet passé en paramètre ? Est-ce l'implémenteur du mutateur, ou son utilisateur ? Si c'est l'utilisateur, cela suppose de lui qu'il a lu la documentation relative au mutateur et sait ce qu'il fait. Si c'est l'implémenteur, alors l'optimisation (cas n°3) n'est plus accessible, car on ne peut deviner si l'objet passé en paramètre est temporaire ou non.

Le choix qui a été fait, et qui doit donc être respecté, est de faire confiance à l'utilisateur. Il n'est en effet pas irrationnel de supposer de lui qu'il lit la documentation (que nul n'est censé ignorer), plutôt que de sacrifier une optimisation. **Le mutateur n'est donc généralement pas chargé de la copie, il ne fait qu'un `retain` du paramètre**. Partant de ce principe, on peut écrire un mutateur pour notre exemple de chaîne de caractères, mais il faut se méfier de trois erreurs potentielles :

```
-(void) setString:(NSString*) nouvelleChaine
{
  chaine = [nouvelleChaine retain];
  //ERREUR : fuite mémoire : l'ancien "chaine" n'est plus référencé
}
```

```
-(void) setString:(NSString*) nouvelleChaine
{
  [release chaine];
  chaine = [nouvelleChaine retain];
  //ERREUR : si nouvelleChaine == chaine,  (cela peut arriver),
  //et que le compteur de références de nouvelleChaine était à 1
  //cela générera une faute mémoire lorsque chaine ou
  //nouvelleChaine sera libérée.
}
```

```
-(void) setString:(NSString*) nouvelleChaine
{
  if (chaine != nouvelleChaine)
    [release chaine]; //il est autorisé d'envoyer release à nil
  chaine = [nouvelleChaine retain];  //ERREUR : devrait être dans le "if"
  //sinon l'objet est référencé une fois de trop et ne sera donc
  //probablement jamais désalloué (fuite mémoire)
}
```

Trois erreurs potentielles, mais aussi trois façons d'implémenter correctement cette méthode. Elles sont notamment présentées dans le *Cocoa Programming for MacOS X* [4].

```
//Méthode "Vérifier avant de modifier"
//la méthode la plus intuitive pour un développeur C++
-(void) setString:(NSString*) nouvelleChaine
{
  if (chaine != nouvelleChaine) //éviter le cas dégénéré
  {
    [chaine release]; //libérer l'ancien
    chaine = [nouvelleChaine retain]; //retenir le nouveau
   }
}
```

```
//Méthode "Autorelease de l'ancienne valeur"
-(void) setString:(NSString*) nouvelleChaine
{
  [chaine autorelease]; //même si chaine == nouvelleChaine,
                        //c'est correct, puisque le release est retardé...
  chaine = [nouvelleChaine retain];
   //... et que ce retain intervient donc avant
}
```

```
//Méthode "retenir avant de libérer"
-(void) setString:(NSString*) nouvelleChaine
{
  [nouvelleChaine retain]; //on incrémente le compteur de 1...
  [chaine release]; //...pour qu'il ne risque pas de tomber à zero ici
  chaine = nouvelleChaine; //mais on ne réexécute pas de "retain" ici !
}
```

8.4.8 Accesseurs en lecture

Les objets étant toujours alloués dynamiquement en Objective-C, ils sont référencés et encapsulés sous forme de pointeurs. Typiquement, les accesseurs en lecture des objets Objective-C ne font que renvoyer la valeur du pointeur, et ne recopient pas l'objet à la volée. Notez que le nom d'un accesseur est usuellement celui de la donnée concernée, cela est possible en Objective-C et ne provoque pas de conflit. Dans le cas d'un booléen, le nom peut être précédé de `is` comme pour répondre à un prédicat.

```
@interface Button
{
  NSString* label;
  BOOL       pressed;
}
-(NSString*) label;
-(void) setLabel:(NSString*)newLabel;
-(BOOL) isPressed;
@end

@implementation Button
-(NSString*) label
{
  return label;
}

-(BOOL) isPressed
{
  return pressed;
}

-(void) setLabel:(NSString*)newLabel
{...}
@end
```

9 Exceptions

La gestion des exceptions en Objective-C est plus proche de celle du Java que de celle du C++ par l'adjonction du `@finally`. Le `finally` est connu en Java mais pas en C++. Il s'agit d'une clause supplémentaire (mais optionnelle) à un bloc `try()...catch()`, contenant du code qui sera exécuté dans tous les cas, que le `try` lève une exception ou non. Cela est généralement très utile pour une libération concise et propre des ressources.

Le comportement du `@try...@catch...@finally` en Objective-C est par ailleurs très classique ; en revanche, on ne peut lancer que des objets (contrairement au C++). Un exemple d'utilisation avec et sans `@finally` est présenté ci-après.

Sans finally	Avec finally

`bool probleme = true;`
`@try{`
`  actionRisquee();`
`  probleme = false;`
`}`
`@catch (MonException* e){`
`  faireUnTruc();`
`  nettoyage();`
`}`
`@catch (NSException* e){`
`  faireUnAutreTruc();`
`  nettoyage();`
`  //ici on relance l'exception`
`  @throw`
`}`
`if (!probleme)`
`  nettoyage();`
	`@try{`
	`  actionRisquee();`
	`}`
	`@catch (MonException* e){`
	`  faireUnTruc();`
	`}`
	`@catch (NSException* e){`
	`  faireUnAutreTruc();`
	`  @throw //relancer l'exception`
	`}`
	`@finally{`
	`  nettoyage();`
	`}`

On peut se passer du `@finally`, mais il est un bon atout pour une gestion propre des exceptions. Comme le montre l'exemple ci-dessus, il gère même le cas où une exception est relancée dans un `@catch`. En fait, le `@finally` est exécuté dès que l'on sort de la portée du `@try`. Ci-après s'en trouve une illustration.

```
int f(void)
{
  printf("f: 1-on me voit\n");
  //Voyez la section sur les chaînes de caractères pour comprendre
  //la syntaxe avec le "@"
  @throw [NSException exceptionWithName:@"kaput"
                                reason:@"c'est la faute a Rousseau"
                                userInfo:nil];
  printf("f: 2-on ne me voit pas\n");
}

int g(void)
{
  printf("g: 1-on me voit\n");
  @try {
    f();
    printf("g: 2-on ne me voit pas (dans cet exemple)\n");
  }
  @catch(NSException* e) {
    printf("g: 3-on me voit\n");
    @throw;
    printf("g: 4-on ne me voit pas\n");
  }
  @finally {
    printf("g: 5-on me voit\n");
  }
  printf("g: 6-on ne me voit pas (dans cet exemple)\n");
}
```

Enfin, le `catch(...)` du C++, chargé de tout intercepter, n'existe pas en Objective-C. Il est en effet inutile, puisque comme seuls des objets peuvent être lancés, on peut toujours les intercepter avec le type `id` (cf. section 5.1 page 9).

Notez qu'une classe `NSException` existe en Cocoa, et qu'il est fortement conseillé d'en faire dériver tout objet que l'on désire lancer. Aussi, un `catch(NSException* e)` devrait être l'équivalent idéal d'un `catch(...)`.

10 Chaînes de caractères en Objective-C

10.1 Seuls objets statiques possibles d'Objective-C

En langage C, les chaînes de caractères sont des tableaux de `char`, ou des pointeurs `char*`. La gestion de ces chaînes est difficile et source de bien d'erreurs. La classe `string` de C++ est un véritable soulagement. En Objective-C, il a été expliqué dans la section 7 page 25 que les objets ne peuvent être automatiques, et doivent être alloués à l'exécution du programme. Cela est incompatible avec l'utilisation de chaînes statiques. On en serait réduit à avoir des chaînes C statiques passées en paramètre de construction d'objets `NSString`, ce qui ferait double emploi, ajoutant lourdeur et allocations mémoires indésirables.

Heureusement, il existe des chaînes Objective-C statiques. Ce sont de simples chaînes C entre guillemets, mais précédées de `@`.

```
NSString* pasPratique = [[NSString alloc] initWithUTF8String:"helloWorld"];
NSString* toujoursPasPratique = //initWithFormat est une sorte de sprintf()
   [[NSString alloc] initWithFormat:@"%s", "helloWorld"];
NSString* pratique = @"hello world";
```

10.2 `NSString` et les encodages

Les chaînes Objective-C sont précieuses, car en plus d'implémenter un grand nombre de méthodes, elles permettent également de supporter de multiples encodages (ASCII, Unicode, ISO Latin 1...). Cela permet notamment une traduction (localisation) des applications grandement simplifiée.

10.3 Description d'un objet, extension de format `%@`

En Java, tout objet hérite de la classe `Object` et bénéficie d'une méthode `toString`, destinée à décrire cet objet par une chaîne de caractères bien pratique pour le débogage. En Objective-C, cette méthode s'appelle `description` et renvoie une `NSString`.

Le `printf` du langage C n'a pas été étendu pour supporter les `NSString`. On lui préférera `NSLog` ou toute fonction acceptant une chaîne de format du style de `printf`. Pour une `NSString`, le format à passer n'est pas « %s » mais « %@ ».

En outre, une `NSString` peut facilement être convertie en chaîne C par la méthode `UTF8String` (anciennement `cString`).

```
char* nom = "Perrine";
NSString* parent = @"maman";
printf("Je m'appelle %s, j'aime ma %s, elle est dans la %s...\n",
       nom, [parent UTF8String], [@"marine" UTF8String]);
NSLog(@"Je m'appelle %s, j'aime ma %@, elle est pas dans la %@
        en ce moment.\n", nom, parent, @"marine");
```

11 STL et Cocoa

La bibliothèque standard du C++ est une autre de ses grandes forces. Même si elle a quelques lacunes, notamment dans les foncteurs (lacunes souvent comblées dans l'implémentation SGI de la STL [5]), elle est malgré tout très riche et vite indispensable. Ce n'est pas à proprement parler une partie du langage, puisqu'elle est construite dessus sans faire partie de sa grammaire, mais on a tôt fait d'en chercher un équivalent dans tout nouveau langage étudié. En l'occurence, en Objective-C, il faut bien sûr se tourner vers Cocoa pour trouver conteneurs, itérateurs et autres codes fournis clé en main.

11.1 Conteneurs

Bien sûr, l'approche Cocoa est purement objet : un conteneur n'est pas template, il ne peut contenir que des objets. Au moment où j'écris ces lignes, les conteneurs disponibles sont les suivants :

- `NSArray` et `NSMutableArray` pour les ensembles ordonnés ;
- `NSSet` et `NSMutableSet` pour les ensembles non ordonnés ;
- `NSDictionary` et `NSMutableDictionary` comme conteneurs associatifs.

Remarquez l'absence notable de classes `NSList` et `NSQueue`. En effet, il semblerait que ces deux dernières classes n'aient intérêt à être implémentées que comme des `NSArray`. Cela peut sembler une absurdité du point de vue des performances, mais la réalité est autre.

En effet, contrairement à un `vector<T>` du C++, le `NSArray` de l'Objective-C encapsule réellement son contenu et le rend inaccessible autrement que par des appels de méthodes. Ainsi, rien n'oblige le `NSArray` à maintenir son contenu comme des cases mémoires contiguës. Les implémenteurs du `NSArray` ont vraisemblablement choisi une technique représentant un bon compromis pour l'utilisation du `NSArray` à la fois comme tableau et comme liste. N'oublions pas qu'en Objective-C, un conteneur ne contient que des pointeurs d'objets, ce qui permet d'assurer une manipulation très efficace des « cases ».

11.2 Itérateurs

L'approche objet pure rend la notion d'itérateur plus souple qu'en C++. La classe `NSEnumerator` joue ce rôle. Exemple :

```
NSArray* tab = [NSArray arrayWithObjects:objet1, objet2, objet3];
NSEnumerator* enumerateur = [tab objectEnumerator];
id anObject = nil;
NSString* chaine = @"poufpouf";
while (anObject = [enumerateur nextObject]) {
  [anObject doSomethingWithString:chaine];
}
```

Une méthode du conteneur (`objectEnumerator`) renvoie un itérateur, lequel est alors capable de se déplacer de lui-même (`nextObject`). Le comportement est plus proche de celui du Java que du C++. Lorsque l'itérateur a parcouru tout le conteneur, `nextObject` renvoie `nil`.

11.3 Foncteurs (objets-fonctions)

La puissance du `selector` en Objective-C permet de se passer d'une notion spécifique de foncteur. En effet, le typage faible permet d'envoyer un message sans vraiment se soucier du récepteur. Par exemple, voici un code équivalent à celui présenté dans la section précédente sur les itérateurs :

```
NSArray* tab = [NSArray arrayWithObjects:objet1, objet2, objet3];
NSString* chaine = @"poufpouf";
[tab makeObjectsPerformSelector:@selector(doSomethingWithString:)
                     withObject:chaine];
```

Dans ce cas, rien n'oblige les différents objets à être de la même classe, ni même d'implémenter la méthode `doSomethingWithString` (au risque de faire lever une exception de « sélecteur non reconnu ») !

11.4 Algorithmes

Les très nombreux algorithmes généraux de la STL n'ont pas de réel équivalent dans la version actuelle de Cocoa. Il faut plutôt se tourner directement vers les méthodes offertes par chaque conteneur.

12 Fonctionnalités propres au C++

Jusque là, il a seulement été montré que les concepts objets du C++ étaient tous implémentés en Objective-C. Cependant, certaines particularités sont quant à elles purement absentes. Il ne s'agit cependant pas de concepts objets, mais bien de facilités de programmation.

12.1 Références

Les références (&) ne sont pas présentes en Objective-C. La gestion mémoire par compteur de référence et l'`autorelease` permettent de s'en passer. En fait, elles n'auraient pas grande utilité. Dans la mesure où les objets doivent toujours être alloués dynamiquement, on n'y fait référence que par pointeurs.

12.2 Inlining

L'inlining est également absent. Pour les méthodes, cela peut se comprendre, à cause du dynamisme d'Objective-C, qui ne peut se permettre de trop figer le code dans l'exécutable. En revanche, l'inlining peut faire défaut pour des fonctions C utilitaires, telles que `max()`, `min()`... C'est là un problème que peut résoudre Objective-C++.

Notez cependant que le compilateur GCC propose le mot-clef non standard `__inline` ou `__inline__` pour utiliser l'inlining en langage C, et donc en Objective-C. De plus, GCC sait également compiler le C99, une révision du langage C qui implémente l'inlining avec le mot-clef (standard cette fois) `inline`. L'Objective-C basé sur du C99 bénéficie donc également de l'inlining.

12.3 Templates

Les templates (ou *modèles*) sont un substitut au principe d'héritage et de méthodes virtuelles, conçu pour l'efficacité, mais intrinsèquement incompatible avec un modèle objet pur (Saviez-vous qu'un template permet de gagner un accès `public` sur des membres initialement `private` ?). Les templates ne sont pas implémentés en Objective-C, et ne pourraient l'être que difficilement compte tenu des règles de surcharge de fonctions.

12.4 Surcharge d'opérateurs

La surcharge d'opérateurs est absente d'Objective-C.

12.5 Friends

Il n'y pas de notion de *friend* en Objective-C. En effet, cette notion est surtout utile en C++ pour l'efficacité des opérateurs surchargés, qui ne sont pas disponibles en Objective-C.

12.6 Méthodes `const`

Les méthodes ne peuvent être déclarées `const` en Objective-C. Par conséquent, le mot-clé `mutable` n'a pas de raison d'être.

12.7 Liste d'initialisation dans le constructeur

Les listes d'initialisation à la construction n'existent pas en Objective-C.

13 RTTI (Run-Time Type Information)

Le langage C++ est en quelque sorte un faux langage objet ; comparé à Objective-C, il est extrêmement statique. C'est un choix tourné délibérément vers la recherche de performance d'exécution maximale. Les informations que l'on peut obtenir pendant l'exécution du programme, en C++, *via* la bibliothèque *typeinfo*, ne sont pas très fiables, car elles dépendent du compilateur. Connaître la classe d'un objet est une question que l'on se pose rarement, à cause du typage fortement statique. Mais elle peut être soulevée lors de l'exploration d'un conteneur. Le `dynamic_cast` et, plus rare, le `typeid`, peuvent être utilisés, mais ne permettent pas une grande interaction avec l'utilisateur du programme. Comment tester efficacement si un objet est une instance d'une classe donnée par l'utilisateur sous forme de chaîne de caractères ?

Objective-C est armé, par nature, pour ce genre de questions. Les classes étant des objets, elles en héritent des comportements.

Remarquez que le *downcasting* a été traité en section 5.4.3 page 18 car il s'agit d'un usage particulier du `dynamic_cast`.

13.1 `class`, `superClass`, `isMemberOfClass`, `isKindOfClass`

L'habilité d'un objet à connaître son type lors de l'exécution du programme est appelée *introspection*, et s'appréhende par différentes méthodes.

`isMemberOfClass` est une méthode qui répond à la question : « mon objet est-il une instance d'une classe donnée (sans considérer l'héritage) ? », tandis que `isKindOfClass` répond à « mon objet est-il une instance d'une classe donnée ou d'une dérivation de cette classe ? ».

L'utilisation de ces méthodes nécessite d'utiliser le faux mot-clé : `class` (et non `@class`, utilisé en déclaration forward). En effet, `class` est une méthode de NSObject, et renvoie un objet `Class`, qui est l'*objet de classe* (l'instance de la méta-classe). Notez au passage que la valeur « vide » d'un pointeur de classe n'est pas `nil` mais `Nil`.

```
BOOL test = [self isKindOfClass:[Toto class]];
if (test)
  printf("Je suis dans la classe de Toto\n");
```

Notez qu'il existe un moyen rapide d'obtenir l'objet de classe d'une classe mère. Plutôt que de faire `[[self super] class]`, on peut utiliser à la place `[self superClass]`.

13.2 `conformsToProtocol`

Cette méthode a été introduite dans la section consacrée aux protocoles (section 6.4 page 20). Elle permet de savoir si un objet adhère à un protocole ou non. Ce n'est pas extrêmement dynamique, car le compilateur se base sur ce qui a été déterminé explicitement dans le code source. Si un objet implémente toutes les méthodes d'un protocole, mais sans le déclarer officiellement, le programme est correct, mais `conformsToProtocol` renvoie `NO`.

13.3 `respondsToSelector`, `instancesRespondToSelector`

`respondsToSelector` est une méthode d'instance, héritée de `NSObject`. Elle est capable de déterminer si un objet implémente une méthode donnée, qu'elle soit héritée ou non. On utilise pour cela la notion de *sélecteur* (cf. section 5.3.4 page 13). Exemple :

```
if ( [self respondsToSelector:@selector(travailler)] )
{
  printf("Je ne suis pas Toto.\n");
  [self travailler];
}
```

Pour savoir si une classe donnée implémente une méthode, sans considérer l'héritage, on dispose de la méthode *de classe* `instancesRespondToSelector`.

```
if ([[self class] instancesRespondToSelector:@selector(trouverBoulot)])
{
    printf("Je suis capable de trouver un boulot"
           " sans l'aide de ma maman.\n");
}
```

Notez que pour des raisons de performance, un appel à `respondsToSelector` ne peut déterminer si une classe reconnaît un message par *forwarding* (cf. section 5.4.2 page 17).

13.4 Typage fort ou typage faible *via* `id`

Le C++ est un langage au typage fort : on ne peut utiliser un objet que si l'on connaît son type exact. En Objective-C, la contrainte est moins forte : si un objet **au type explicite** est la cible d'un message qu'il ne sait pas traiter *a priori*, le compilateur génère un avertissement (*warning*) mais le programme fonctionnera. Le message sera simplement perdu (en levant une exception), sauf si par *forwarding* (cf. section 5.4.2 page 17) il peut être récupéré. Si c'est bien ce qui a été prévu par le développeur, l'avertissement est une fausse alerte ; dans ce cas on l'évite en donnant à la cible du message le type `id` plutôt que son type réel. En fait, tout objet est de type `id`, et peut sous ce typage être la cible de n'importe quel message.

Cette fonctionnalité de typage faible est indispensable lorsque l'on utilise des processus de délégation : il n'est pas besoin de connaître le type exact d'un objet délégué pour pouvoir l'utiliser.En voici un exemple :

```
-(void) setAssistant:(id) unObjet
{
  [assistant autorelease];
  assistant = [unObjet copy];
}

-(void) traiterLeDossier:(Dossier*) dossier
{
  if ([assistant respondToSelector:@(traiterleDossier:)])
    [assistant traiterLeDossier:dossier];
  else
    printf("Vous avez le formulaire bleu ?\n");
}
```

Précisons enfin que les délégations sont monnaie courante en Cocoa dès que l'on utilise une interface graphique, dans laquelle tous les contrôles sont chargés de transmettre des actions de la part de l'utilisateur.

14 Objective-C++

Le langage Objective-C++ est en cours de développement. Il est déjà fonctionnel, mais présente encore quelques lacunes. Objective-C++ permet d'utiliser conjointement de l'Objective-C et du C++, pour tirer parti des fonctionnalités de chacun.

Au moment où j'écris ces lignes, le mélange des deux langages pose encore problème au niveau des exceptions, et de l'utilisation d'objets C++ comme données d'instance de classes Objective-C. Des programmes peuvent cependant d'ores et déjà être écrits en Objective-C++. Les fichiers d'implémentation portent l'extension *.mm*

Conclusion

Ce document n'est qu'un aperçu rapide des différents aspects d'Objective-C comparé au C++. Il est cependant utile en tant que référence rapide pour les développeurs confirmés désirant comprendre ce langage. En espérant que le but soit atteint au mieux, j'invite cependant le lecteur avisé à me faire part de toute remarque, critique ou correction, pour améliorer tout ce qui peut l'être.

Références

[1] Apple Computer, Inc., Developer documentation. The Objective-C Programming Language for MacOS X version 10.3, February 2004. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[2] Boost. `http://www.boost.org/`.

[3] GNUstep. `http://www.gnustep.org/`.

[4] Aaron Hillegass. *Cocoa Programming for MacOS X, 2nd edition.* Addison-Wesley, 2004.

[5] SGI. Standard template library programmer's guide. `http://www.sgi.com/tech/stl`.

[6] Will Shipley. `self = [supid init]`. `http://wilshipley.com/blog/2005/07/self-stupid-init.html`.

Révisions du document

version 1.5

- un grand merci à Jean-Daniel Dupas pour ses corrections !
- correction : (partie 5.2.3) en Objective-C, l'attribut par défaut n'est pas `private` mais `protected` ;
- correction : (partie 5.3.5) `@selector()` est évalué à la compilation et ne pénalise pas les performances ;
- correction : (partie 5.4.1) un `retain` est plus logique qu'un `copy` ;
- correction : (parties 6.4.2 et 6.6) on ne peut pas utiliser catégorie et super-classe en même temps ;
- correction : (partie 7.1.1) la donnée d'instance `isa` est initialisée dans `alloc` et non dans `init` ;
- précisions sur *l'inlining* dans la partie 12.2 ;
- les méthodes cString et initWithCString sont devenues obsolètes avec MacOS 10.4, il faut maintenant utiliser les équivalents UTF8 ;
- précisions sur self = [super init] dans la partie 7.1.3 ; les différents code d'exemple utilisant `self = [super init] ; if (self){...} ;` ont été corrigés en conséquence ;
- ajout de la partie 7.1.4 intitulée « Échec de l'initialisation » ;
- légère remise en page (séparation des sections par un saut de page).

version 1.4

- Précisions sur l'absence de NSList, NSQueue ;
- précisions sur les itérateurs ;
- corrections typographiques.

version 1.3

- Ajout d'une partie sur les arguments anonymes (ou muets) ;
- ajout d'une partie sur les données de classe ;
- ajout d'une partie sur les accesseurs en lecture ;
- légère modification du nom de la section traitant des mutateurs ;
- précision sur la possibilité de nommer une méthode comme une donnée d'instance ;
- ajout de « méthodes virtuelles pures » dans l'index.

version 1.2

- La partie sur les mutateurs était incorrecte ! Elle est maintenant conforme au *Cocoa programming for MacOS X* [4] ;
- modification de la section « Appel de méthodes » en « Différenciation entre fonctions et méthodes » ; contenu légèrement modifié également ;
- correction de quelques erreurs typographiques et de fautes d'orthographe.

version 1.1

- Ajout de la présente section ;
- les renvois du document sont maintenant des hyperliens pour naviguer plus rapidement ;
- meilleure gestion des accents dans les lecteurs PDF ;
- explications plus complètes sur la copie (clonage) ;
- plus de détails sur les objets mutables ;
- référence au « Cocoa Programming for MacOS X » ;
- référence à SGI ;
- quelques petits remaniements de phrases.

Index