Développement sous MacOS X

Laurent Chéno, colloque de Luminy

un printemps noir : avril-mai 2002

sommaire

— Objective C ;

— programmation en Cocoa.

Première partie

Le langage Objective C

Références bibliographiques

L'ouvrage *"Object oriented Programming and the Objective C Language"* est disponible en ligne à l'adresse : `http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/` ou en version imprimée chez Vervanté pour $ 17.25 (voir à `http://catalog.vervante.com/viewProduct.cfm?item_id=590489`).

La référence usuelle pour la définition du langage C est :

"The C Programming Language", par Kernighan et Ritchie, chez Prentice Hall, traduit en français chez Dunod.

1 Définition d'une classe

En général, on regroupe la définition de l'**interface** dans un fichier `MaClasse.h` et l'**implantation** dans un fichier `MaClasse.m`, l'habitude étant de nommer le fichier d'après le nom de la classe — qui commence par une majuscule alors que les méthodes et variables d'instance commencent par une minuscule (ce n'est qu'une convention d'usage).

Toutes les directives spécifiques à Objective-C commencent par le caractère `@`, et ici on utilisera donc `@interface`, `@implementation` et `@end`.

L'interface est ainsi de la forme suivante :

```
@interface MaClasse : ClasseMère
{
        déclarations des variables d'instance
}

déclarations des méthodes

@end
```

On aura déjà remarqué qu'une classe ne peut avoir qu'une classe père : Objective C ne prévoit pas l'héritage multiple. Il offre en revanche la notion de **protocole** qui peut répondre à certains usages de l'héritage multiple.

On importe quasi-systématiquement la déclaration de l'interface dans l'implantation, qui n'a plus qu'à définir les méthodes.

```
#import "MaClasse.h"

@implementation MaClasse : ClasseMère

définitions des méthodes

@end
```

La syntaxe pour l'invocation des méthodes est directement issue de SmallTalk et peut dérouter au début.

`[monObjet saMéthode]` est l'appel le plus simple, quand il n'y a pas d'argument à la méthode. Le nom de la méthode est ici `saMéthode`.

`[monObjet transformeAvec: x etAussi: y]` invoque une méthode qui s'appelle `transformeAvec:etAussi:` avec deux arguments.

Le nom (on dit plutôt le **sélecteur**) d'une méthode comporte donc autant de fois le caractère `:` qu'elle a d'arguments.

2 Un exemple

Interface :

```
@interface Point  : NSObject
{
        int x ;
}
- (int) position;
- (void) deplaceDe: (int) d;
```

L'implantation s'écrit alors :

```
#import "Point.h"
@implementation Point  : NSObject
- (int) position
{ return x;}
- (void) deplaceDe: (int) d
{ x = x + d; }
```

`NSObject` est la classe-racine de Cocoa. Elle définit les méthodes de base d'allocation des objets : `alloc` et `init`.

On peut bien entendu surcharger la méthode `init`, et ici, par exemple, on ajoutera à l'implantation (pas la peine de rien modifier à l'interface) la définition suivante :

```
- (id) init
{
        if (self = [super init]) {
                x = 0;
        }
        return self;
}
```

La convention est que `init` retourne toujours `nil` quand elle échoue.

`self` est le nom de l'objet lui-même ; l'invocation `[super init]` appelle la méthode `init` de la classe mère (ici `NSObject`). `id` est le type générique de tout objet.

Notons qu'une instance d'une classe `MaClasse` (un objet) est du type `(MaClasse *)`.

On peut écrire de nouvelles fonctions d'initialisation en ajoutant à l'interface :

```
- (id) initAtPos: (int) p;
- (id) initWithPoint: (Point *) p;
```

et à l'implantation :

```
- (id) initAtPos: (int) p
{        if (self = [super init]) {
                x = p;
         }
         return self; }

- (id) initWithPoint: (Point *) p
{        if (self = [super init]) {
                x = [p position];
         }
         return self; }
```

Pour créer une instance de la classe `Point` ainsi définie, on pourra écrire :

```
Point *p;
...
p = [[Point alloc] initAtPos: 12];
...
[p deplaceDe: 5];
if ([p position] != 17) {
        ...
```

Une autre solution consiste à créer une méthode qui ne sera plus une *méthode d'instance*, mais une *méthode de classe* : elle est invoquée non sur un objet particulier (c'est-à-dire une instance de la classe), mais sur la classe elle-même.

Dans l'interface on écrira :

```
+ (id) nouveauPointEn: (int) p;
```

et dans l'implantation :

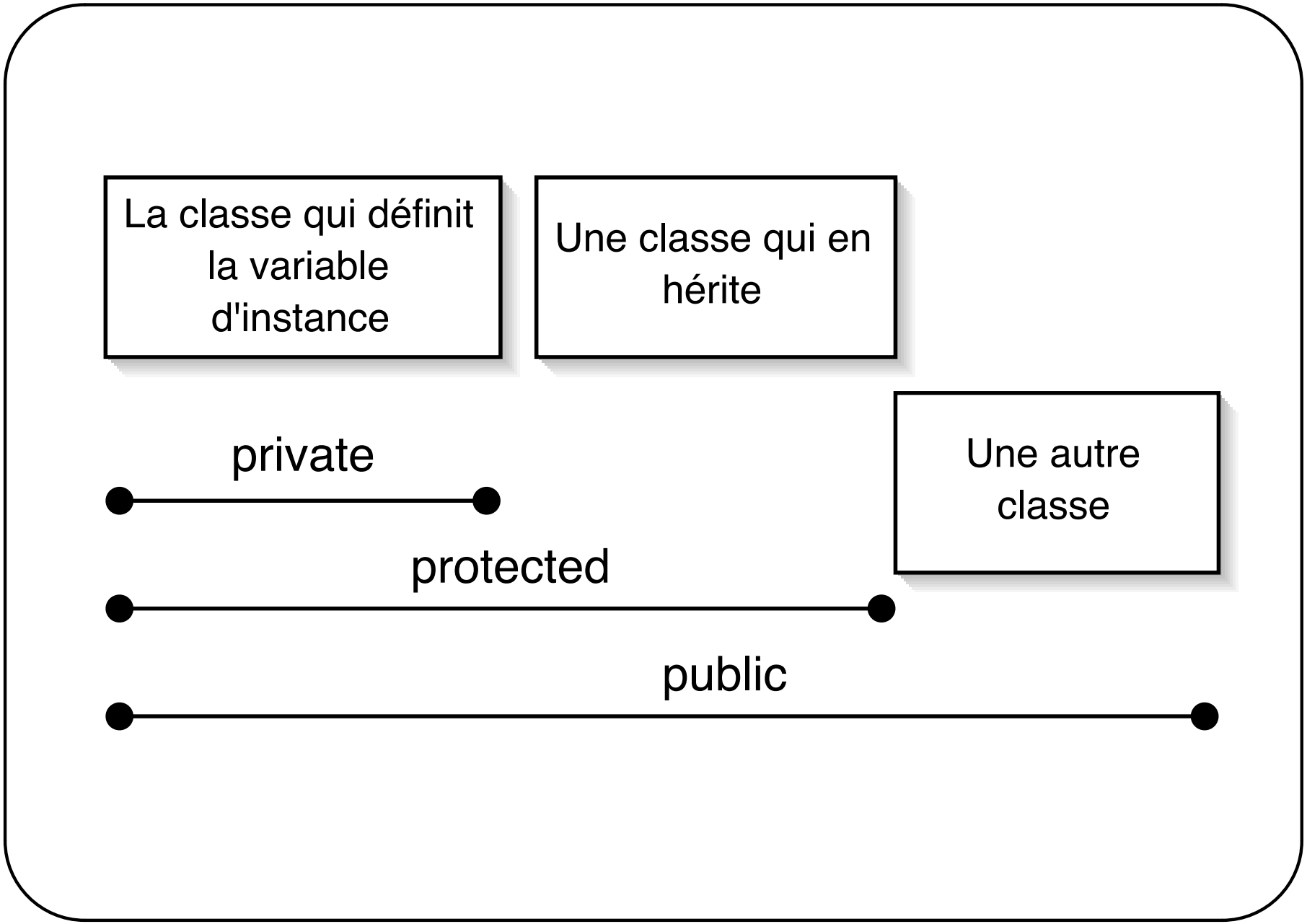
```
+ (id) nouveauPointEn: (int) p
{ Point *resultat;
        resultat = [[Point alloc] initAtPos: p];
        return resultat; }
```

Le lecteur attentif aura noté que `alloc` est elle-même une méthode de classe, définie dans NSObject.

3 Portée des variables d'instance

On peut modifier la portée des variables d'instance grâce aux directives `@private`, `@protected` et `@public` (par défaut, Objective-C utilise `@protected`).

On peut aussi, en les définissant dans le fichier d'implantation d'une classe, mais avant la directive `@implementation`, des variables de classe.


La classe qui définit la variable d'instance
Une classe qui en hérite
Une autre classe
private
protected
public

4 Aspects dynamiques

Objective-C propose le type `SEL` des sélecteurs, et la directive `@selector()` correspondante. On écrira par exemple :

```
SEL deplace ;
deplace = @selector(deplaceDe:) ;
```

Le calcul du sélecteur a lieu ici pendant la compilation. On peut le repousser à l'exécution grâce aux fonctions Cocoa `NSSelectorFromString` et `NSStringFromSelector`.

Par exemple :

```
SEL deplace;
NSString *nomDuSelecteur = donneNom(...);
deplace = NSSelectorFromString(nomDuSelecteur);
```

Il existe de même des fonctions `NSStringFromClass` et `NSClassFromString`.

Tout ceci permet de modifier dynamiquement le comportement d'un bouton, par exemple :

```
[monBouton setAction: @selector(travailleBienAvec:)];
[monBouton setTarget: objetResponsable];
```

La méthode invoquée quand on clique sur le bouton va utiliser la méthode `performSelector:withObject:` dont la déclaration est :

```
- (id) performSelector: (SEL) aSelector
            withObject: (id) anObject;
```

(elle fait partie du protocole de NSObject).

Citons encore les méthodes `isKindOfClass:` et `respondsToSelector:` qui permettent des tests à l'exécution :

```
if ([monObjet isKindOfClass: [Rectangle class]])
        ...
if ([monObjet respondsToSelector: @selector(agitViteEtBien)])
        ...
```

On n'aurait pas pu écrire `[monObjet isKindOfClass: Rectangle]` car un nom de classe peut être utilisé seulement pour typer une variable ou pour recevoir un message.

5 Catégories

Les catégories permettent d'ajouter de nouvelles méthodes à une classe, qui seront normalement héritées par ses sous-classes, mais sans créer de nouvelle sous-classe.

Cela permet en quelque sorte, de programmer de façon plus modulaire, de partager en plusieurs fichiers une définition de classe complexe, et de bénéficier de la compilation incrémentielle.

```
#import "MaClasse.h"
@interface MaClasse (MaCategorie)
déclarations de méthodes supplémentaires
@end

@implementation MaClasse (MaCategorie)
définitions des méthodes
@end
```

6 Protocoles

Les protocoles pallient dans une certaine mesure l'absence d'héritage multiple, puisqu'une classe peut se conformer à un ou plusieurs protocoles, alors qu'elle ne peut hériter que d'une seule classe mère.

Un protocole ne contient que des déclarations que de méthodes, pas de variables d'instances, et aucune définition desdites méthodes. Par exemple (fictif) :

```
@protocole MouseResponder
- (void) mouseDown: (NSEvent *) event;
- (void) mouseUp: (NSEvent *) event;
- (void) mouseDragged: (NSEvent *) event;
@end
```

En indiquant qu'une classe se conforme à tel ou tel protocole, on s'impose de fournir l'implémentation des méthodes prévues par ces protocoles.

On écrira ainsi :

```
@interface MaClasse : ClasseMere < MouseResponder, Formatter >
...
@end
```

Et un contrôle pourra être effectué par le compilateur si l'on utilise des types qualifiés par un ou plusieurs protocoles :

```
id < MouseResponder, Printable > monObjet ;
...
- (id < Formatter >) serviceDeFormat;
...
```

Deuxième partie

Programmation en Cocoa

Références bibliographiques

Le site Apple fournit la documentation en ligne, mais aussi le lien sur un site de déchargement gratuit.

`http://developer.apple.com/tools/`

Les acheteurs du système MacOS X trouvent aussi le CD Developer dans la boîte. Les mises à jour (ou le téléchargement complet pour ceux qui auraient *perdu* leur CD d'origine) s'obtiennent gratuitement par téléchargement (après inscription) à l'adresse :

`http://connect.apple.com/`

Parmi les outils développeurs, on trouve les outils habituels de programmation (compilateur gcc, débuggeur gdb, débuggeur pour le moteur graphique Quartz, merge avec interface graphique, éditeurs d'icônes, etc) et surtout les deux applications au cœur du processus de programmation pour MacOS X : à savoir ProjectBuilder et InterfaceBuilder. On trouve également une abondante documentation des outils et des interfaces.

Le programmeur a le choix entre trois environnements de programmation qui lui permettent d'accéder à l'interface graphique Aqua du système, à savoir : Carbon, Cocoa sous Objective-C et Cocoa sous Java. (Je n'évoque pas ici la possibilité d'utiliser Swing ou AWT pour Java, ni de travailler sous X11, par exemple en Caml avec Tk.)

Nous nous intéresserons ici à Cocoa, qui repose sur deux ensembles d'API : **Foundations** et **AppKit**. L'ancêtre de Cocoa est NextStep, qui avait été imaginé par Jean-Marie Hullot (parmi d'autres) pour la station Next. C'est la raison pour laquelle les noms des classes définies dans Cocoa commencent toutes par NS.

Foundations regroupe des classes d'usage général (dictionnaires, tableaux...), le traitement des chaînes de caractères au standard Unicode (classe NSString), mais aussi tout ce qui, dans NSObject en particulier, concerne la gestion de la mémoire.

AppKit regroupe les classes qui permettent de construire les éléments de l'interface Aqua : boutons, fenêtres, dialogues, menus...

Références bibliographiques

Les deux ouvrages disponibles en France que je recommande sur la programmation Cocoa sont :

"Learning Cocoa", chez O'Reilly, qui est l'ouvrage officiel de Apple ; et *"Cocoa programming for MacOS X"*, par Aaron Hillegass, chez Addison-Wesley.

7 La difficile question de la gestion de la mémoire

À la différence de Caml, il n'y a pas, quand on programme sous Cocoa en Objective-C, de gestion automatique de la mémoire (pas de GC). Et c'est sans doute là la question la plus difficile à traiter, et la première source de plantage d'un programme.

Considérons l'exemple d'une classe possédant une variable d'instance `maChose` qui est un objet de la classe `maClasse`, et écrivons la méthode

```
- (void) setMaChose: (maClasse *) uneChose
```

Nous proposons trois solutions.

```
- (void) setMaChose: (maClasse *) uneChose {
        [maChose autorelease];
        maChose = [uneChose retain];
}
```

Ici, le message `autorelease` envoie l'ancienne valeur de `maChose` dans le *"autorelease pool"* de l'application, et à la fin de la boucle d'événement, elle sera libérée automatiquement. Puis, avant d'affecter `uneChose` à `maChose`, on lui envoie le message `retain`, qui incrémente un compteur d'allocation : à chaque appel de `release` sur un objet, on décrémente ce compteur, et s'il tombe à zéro, on libère effectivement la mémoire par un appel à `dealloc`.

Une autre façon d'opérer aurait consister à écrire :

```
- (void) setMaChose: (maClasse *) uneChose {
        [maChose autorelease];
        maChose = [uneChose copy];
}
```

Si on ne souhaite pas utiliser `autorelease`, la situation est plus compliquée :

```
- (void) setMaChose: (maClasse *) uneChose {
        maClasse *vieilleVersion = maChose;
        maChose = [uneChose retain];
        [vieilleVersion release];
}
```

Là encore, on aurait pu préférer `copy` à `retain`.

Mais il est temps de passer à la pratique !...