BecomeAnXcoder

Français

Page originale de la traduction française :

http://www.cocoalab.com/?q=BecomeAnXcoder-Francais

(pour tout renseignement concernant la version Anglaise de cet ouvrage, ainsi que sa traduction française, merci de visiter le lien ci-dessus)

Mise en page, export au format PDF :

Carmelo INGRAO

carmelo42@gmail.com

http://c.ingrao.free.fr/blog/

Téléchargement des futures versions :

http://c.ingrao.free.fr/Xcode/

Version de la mise en page : 1.0

Information

Un livre gratuit pour commencer avec Cocoa à l'aide d'Objectif-C

Donnez s'il vous plaît! Si vous appréciez notre travail, veuillez prendre une minute pour envoyer un don.

Bert Altenburg, auteur d'AppleScript for Absolute Starters (AppleScript pour parfaits débutants), en collaboration avec Alex Clarke et Philippe Mougin, a publié un livre pour les nouveaux venus sur la programmation Cocoa à l'aide d'Objectif-C et XCode.

Ce tutoriel est écrit pour non-programmeurs et vise à niveler, autant que possible, la courbe d'apprentissage. Dans la plus pure tradition, BecomeAnXcoder est publié sous la forme d'un livret au format pdf sous licence Creative Commons Attribution. Il est également disponible en ligne, suivez simplement les liens ci-dessous.

BecomeAnXcoder a été téléchargé plus de 110.000 fois depuis Mai 2006. Si vous souhaitez contribuer à la création d'une traduction en français, en espagnol, en italien ou en toute autre langue, veuillez me contacter: alex sur http://www.cocoalab.com/.

A propos

Ce livre a pour but de vous initier, de manière relativement indolore, aux concepts fondamentaux de la programmation sur Mac en utilisant Xcode et Objective-C.

Il ne requiert aucune connaissance préalable de la programmation.

Introduction

Apple fournit gratuitement tous les outils dont vous avez besoin pour créer de superbes applications en Cocoa. Cet ensemble d'outils, connu sous le nom de Xcode, est distribué avec Mac OS X. Vous pouvez également le télécharger depuis la section développeur (english : developer section) du site d'Apple.

Plusieurs bons livres existent pour la programmation sur Mac, mais ils exigent tous que vous ayez déjà une expérience en programmation. Pas le présent ouvrage. Celui-ci vous apprendra les bases de la programmation, en particulier avec Objective-C et en utilisant Xcode. Après 5 chapitres, vous serez déjà capable de créer un programme basique sans Interface Graphique Utilisateur (english : Graphical User Interface ou GUI). Après quelques chapitres de plus, vous saurez comment créer des programmes simples, avec GUI. Quand vous aurez terminé ce petit guide, vous serez prêt pour les livres plus avancés sus-mentionnés. Vous devrez les étudier également, car il y a beaucoup à apprendre. Pour l'instant, ne vous inquiétez pas: ce petit guide va faciliter vos premiers pas.

Comment utiliser ce livre

Comme vous le constaterez, certains paragraphes sont affichés dans une boîte comme celle-ci:

Un point technique
Nous vous suggérons de lire chaque chapitre (au moins) deux fois. La première fois, sautez les sections contenues dans ces boîtes. Quand vous relirez le chapitre, incluez cette fois le texte des boîtes. Vous réviserez ce que vous avez déjà appris, mais découvrirez également quelques points intéressants qui vous auraient distrait à la première lecture. En utilisant le livre de cette façon, vous transformerez votre courbe d'apprentissage en une jolie pente douce.

Ce livre contient des douzaines d'exemples qui consistent en une ou plusieurs lignes de code. Pour être sûr que vous associerez la bonne explication au bon exemple, chaque exemple est repéré par un numéro placé entre crochets, comme ceci: [1]. Beaucoup d'exemples ont plus de 2 lignes de code. Dans ce cas, un deuxième nombre est utilisé pour repérer une certaine ligne. Par exemple, [1.1] fait référence à la première ligne de l'exemple [1]. Dans ces longs passages de code, la référence est placée après la ligne de code, comme ceci:

```
//[1]
volume = tailleSurface * hauteur;  // [1.1]
```

Programmer n'est pas un boulot facile. Il vous faudra de la persévérance pour refaire tous les exercices enseignés ici. Vous ne pouvez pas espérer apprendre à conduire ou à jouer du piano simplement en lisant un livre. Le même principe s'applique concernant l'apprentissage de la programmation. Ce livre est au format électronique, vous n'avez donc aucune excuse pour ne pas effectuer des aller-retours fréquents entre celui-ci et Xcode. Ainsi, comme au chapitre 5, nous vous suggérons de parcourir chaque chapitre 3 fois. La deuxième fois, essayez réellement les exemples, et tentez quelques modifications du code pour explorer comment tout cela fonctionne.

Avant de commencer

Ce livre a été écrit pour vous. Comme il est gratuit, permettez moi de dire quelques mots sur la promotion du Mac en retour. Chaque utilisateur de Macintosh peut contribuer à promouvoir sa plate-forme favorite à peu de frais. Voici comment.

Plus vous serez efficace avec votre Mac, plus les autres seront amenés à considérer le Mac. Essayez de vous tenir à la page en visitant régulièrement des sites Mac et en lisant des magazines à propos du Mac. Evidemment, apprendre Objective-C ou AppleScript et inciter les autres à le faire est important également. Dans les affaires, l'usage d'AppleScript peut sauver des tonnes de temps et d'argent. Jetez un oeil au petit livre gratuit de Bert 'AppleScript pour grands débutants' (english: 'AppleScript for Absolute Starters'), disponible ici: http://www.macscripter.net/books

Montrez à la planète que tout le monde n'utilise pas un PC en rendant le Macintosh plus visible. Porter un tee-shirt Mac en public peut être un moyen, mais vous pouvez aussi améliorer la visibilité du Mac depuis chez vous. Si vous utilisez parfois l'application Moniteur d'Activité (english: Activity Monitor, dans le dossier Utilitaires de votre dossier Applications), vous noterez que votre Mac n'utilise sa pleine puissance qu'en de rares occasions. Des scientifiques ont eu l'initiative de plusieurs projets de mise en réseaux de cette puissance de calcul (english: 'Distributed Computing', ou 'DC projects'), comme Folding@home ou SETI@home. Ils utilisent cette puissance de calcul inexploitée, la plupart du temps pour le bien de tous.

Téléchargez un petit programme gratuit, un client DC, et mettez vous au travail comme d'habitude. Ces clients DC tournent avec le plus bas niveau de priorité. Si vous utilisez un programme sur votre Mac qui nécessite la pleine puissance de votre ordinateur, le client DC fait immédiatement une pause. Vous ne remarquerez même pas qu'il tourne. Comment cela aide t-il le Mac? La plupart des projets DC fournissent des rapports sur les plate-formes qui les supportent. Si

vous rejoignez une équipe Mac (vous les reconnaîtrez à leurs noms dans les rapports), vous aiderez l'équipe Mac de votre choix a mieux se positionner dans les rapports. Ainsi, les utilisateurs d'autres plate-formes verront ce que les Macs font. Il existe des clients DC pour de nombeux domaines comme les maths, la recherche médicale et bien d'autres. Pouir choisir un projet DC qui vous convient, rendez vous ici: http://distributedcomputing.info/projects.html

Il n'y a qu'un problème avec ma suggestion: ça peut devenir contagieux!

Assurez vous que la plate-forme Mac possède les meilleurs logiciels. Pas seulement en créant ces programmes vous même. Prenez l'habitude d'envoyer de petits retours (polis) aux développeurs des logiciels que vous utilisez. Même après avoir essayé un petit bout de programme que vous n'avez pas apprécié, signalez vos critiques au developpeur. Reportez les bugs avec une description aussi précise que possible des actions réalisées quand vous avez rencontré ce bug. Achetez les logiciels que vous utilisez. Aussi longtemps que le marché du logiciel pour Macintosh sera viable, les programmeurs continueront à produire d'excellents programmes.

Merci de contacter au moins 3 utilisateurs de Macintosh qui pourraient s'intéresser à la programmation, et parlez leur du présent guide et du site où ils peuvent le trouver. Ou parlez leur de tout ce que je viens d'évoquer.

OK, pendant que vous téléchargez votre client DC client en tache de fond, attaquons!

01: Un programme est une série d'instructions

Introduction

Si vous apprenez à conduire une voiture, vous devez apprendre à manier plusieurs choses en même temps. Vous devez saisir à la fois le fonctionnement des pédales d'embrayage, de l'accélérateur et des freins. La programmation exige également d'avoir à l'esprit un grand nombre de choses, sans quoi votre programme plantera. Si l'intérieur d'une voiture vous était déjà familier avant de commencer à apprendre à conduire, vous n'aurez pas cet avantage en apprenant comment programmer avec Xcode. Afin de ne pas vous submerger, nous garderons la question de l'environnement de programmation pour un chapitre ultérieur. Tout d'abord nous allons vous mettre à l'aise avec le code Objectif-C, en commençant par des maths de base qui vous sont familières.

A l'école primaire, vous deviez faire des calculs en remplissant les pointillés:

```
2 + 6 = ...?
... = 3 * 4 (l'étoile * est le moyen conventionnel de
représenter la multiplication sur les claviers d'ordinateur)
```

En secondaire les pointillés n'étaient plus à la mode, et des variables nommées x et y (ainsi qu'un nouveau nom sophistiqué, "algèbre") faisaient tout un foin. Avec le recul, on peut se demander pourquoi les gens furent si intimidés par ce tout petit changement de notation.

```
2 + 6 = x?
y = 3 * 4
```

Variables

Objectif-C utilise aussi des variables. Les variables ne sont rien de plus que des noms opportuns pour se référer à un morceau de

données spécifique, tel un nombre. Voici une déclaration en Objectif-C, c'est à dire une ligne de code dans laquelle une certaine valeur est donnée à une variable:

```
//[1]
x = 4;
```

Le point-virgule

La valeur 4 est donnée à la variable nommée x . Vous noterez qu'il y a un point-virgule à la fin de la déclaration. Ceci car le point-virgule est indispensable à la fin de chaque déclaration. Pourquoi? En fait, le fragment de code de l'exemple [1] peut vous paraître très simple, mais un ordinateur ne sait pas du tout qu'en faire. Un programme spécial, appelé compilateur, est nécessaire pour convertir le texte que vous avez tapé en 0 et 1 que, seuls, votre Mac comprend. Lire et comprendre le texte tapé par un humain est très difficile pour un compilateur, donc vous devez lui fournir un certain nombre d'indices, par exemple l'endroit où se termine une déclaration donnée. C'est ce que vous faites en utilisant le point-virgule.

Si vous oubliez un seul point-virgule dans votre code, le code ne pourra pas être compilé, ce qui signifie qu'il ne pourra être converti en un programme exécutable par votre Mac. Ne vous souciez pas trop de cela, car s'il ne peut compiler votre code, le compilateur vous le dira. Comme nous le verrons dans un prochain chapitre, il essaiera de vous aider à découvrir ce qui ne va pas.

Nommer les variables

Alors que les noms des variables en eux-même n'ont pas de signification précise pour le compilateur, des noms de variables descriptifs peuvent grandement faciliter la lecture d'un programme, et donc sa compréhension par les humains. C'est un grand plus si vous devez dépister une erreur dans votre code.

Les erreurs dans un programme sont traditionnellement appelées des bogues. Les trouver et les corriger se dit déboguer.

Donc, dans du vrai code, on évite d'utiliser des noms de variables non-descriptifs tels que x. Par exemple, le nom de la variable pour la largeur d'une image pourra s'appeler imageLargeur [2].

```
//[2]
imageLargeur = 8;
```

Du gros problème de ce que l'ordinateur fait de l'oubli d'un point-virgule, vous comprendrez que la programmation est toute affaire de détails. L'un de ces détails auquel faire très attention est le fait que le code est sensible à la casse : c'est à dire qu'il importe de savoir si vous utilisez ou non des majuscules. Le nom de variable imageLargeur n'est pas le même que imageLARGEUR, ou ImageLargeur. En accord avec les conventions générales, je crée mes noms de variables en fusionnant plusieurs mots, le premier sans majuscule, et tous les autres mots composants le nom de la variable commençant par une majuscule, tout comme vous pouvez le voir dans l'exemple [2]. Ce style est souvent appelé « camelCase » (casseDuChameau). En collant à ce modèle, je réduis fortement les risques d'erreurs de programmation dus à la sensibilité de casse.

Bien noter qu'un nom de variable n'est toujours composé que d'un seul mot (ou d'un seul caractère, à la rigueur).

Bien que vous ayez toute liberté pour choisir les noms de variables, il y a diverses règles auxquelles votre nom de variable doit se conformer. Bien que je puisse toutes les citer, ce serait ennuyeux à ce stade. La principale règle à suivre est que votre nom de variable ne doit pas être un mot réservé d'Objectif-C (c'est-à-dire un mot ayant une signification particulière en Objectif-C). En composant un nom de variable par la contraction de mots, comme imageLargeur, vous êtes tranquille. Pour que le nom de variable reste lisible, l'usage des

capitales à l'intérieur des noms de variables est recommandé. Si vous collez à ce modèle, vous aurez moins de bogues dans vos programmes.

Si vous tenez vraiment à apprendre deux ou trois règles, finissez ce paragraphe. En dehors des lettres, l'usage des chiffres est autorisé, mais un nom de variable ne doit pas commencer par un chiffre. Le caractère de soulignement « _ » est également autorisé. Voici quelques exemples de noms de variables.

Noms de variables corrects:

- porte8k
- po8rte
- po_rte

Défendu:

- porte 8 (contient un espace)
- 8porte (commence par un nombre)

Déconseillé:

- Porte8 (commence par une majuscule)

Utiliser des variables pour calculer

Maintenant que nous savons comment donner une valeur à une variable, nous pouvons réaliser des calculs. Jetons un œil au code du calcul de la surface d'une image. Voici le code [3] qui ne fait que cela.

```
//[3]
imageLargeur=8;
imageHauteur=6;
imageAire=imageLargeur*imageHauteur;
```

Curieusement, le compilateur ne pinaille pas sur la présence ou non d'espaces (excepté à l'intérieur des noms de variables, des mots-clés,

etc.!). Pour faciliter la lisibilité du code, nous pouvons utiliser des espaces.

```
//[4]
imageLargeur = 8;
imageHauteur = 6;
imageAire = imageLargeur * imageHauteur;
```

Entiers et flottants

Maintenant, examinons l'exemple [5], et en particulier les deux premières déclarations.

```
//[5]
imageLargeur = 8;
imageHauteur = 4.5;
imageAire = imageLargeur * imageHauteur;
```

Les nombres peuvent généralement être distingués en deux types: Les entiers (nombres entiers) et les nombres fractionnaires. Vous pouvez en voir un exemple de chaque, respectivement, dans les déclarations [5,1] et [5.2]. Les entiers sont utilisés pour compter, chose que nous ferons lorsque nous aurons à répéter un certain nombre de fois une série d'instructions (voir chapitre 7). par exemple, vous entendez parler de nombres fractionnaire ou à virgule flottante dans les moyennes des frappes au base-ball (en France, dans les chiffres de la natalité… :-).

Le code de l'exemple [5] ne fonctionnera pas. Le problème est que le compilateur veut que vous lui disiez à l'avance quels noms de variables vous allez utiliser dans votre programme, et à quel type de données elles se rapportent, c'est-à-dire des nombres entiers ou à

virgule flottante. En langage branché, cela s'appelle "déclarer une variable".

```
//[6]
int imageLargeur;
float imageHauteur, imageAire;
imageLargeur = 8;
imageHauteur = 4.5;
imageAire = imageLargeur * imageHauteur;
```

A la ligne [6.1], int indique que la variable imageLargeur est un entier. A la ligne suivante, nous déclarons deux variables d'un seul coup, en séparant les noms de variables d'une virgule. Plus précisément, la déclaration [6.2] dit que les deux variables sont de type float (flottant), c'est-à-dire des nombres qui contiennent des parties fractionnaires. Dans notre cas, il est un peu ridicule que imageLargeur soit d'un autre type que les deux autres variables. Mais ce que vous pourrez voir, c'est que si vous multipliez un int par un float, le résultat du calcul est un float, c'est la raison pour laquelle vous devez déclarer la variable imageAire comme un float [6,2].

Pourquoi le compilateur veut-il savoir si une variable représente un entier ou un nombre avec une partie fractionnaire? Eh bien, un programme informatique a besoin d'une partie de la mémoire de l'ordinateur. Le compilateur réserve de la mémoire (des octets) pour chaque variable qu'il rencontre. Puisque les différents types de données, dans ce cas int et float, exigent différentes quantités de mémoire et une représentation différente, le compilateur a besoin de réserver la bonne quantité de mémoire et d'utiliser la bonne représentation.

Et si nous travaillons avec de très grands nombres ou des nombres décimaux de très haute précision? Ils ne tiendront pas dans les quelques octets réservé par le compilateur, non? C'est exact. Il y a

deux réponses à cela: premièrement, int et float ont tous deux des homologues qui peuvent stocker de plus grand nombre (ou des nombre de plus haute précision). Sur la plupart des systèmes ce sont, respectivement, long long et double. Mais même ceux-ci peuvent faire le plein, ce qui nous amène à la seconde réponse: en tant que programmeur, il vous appartiendra d'être aux aguets des problèmes. En tout cas, ce n'est pas un problème à débattre dans le premier chapitre d'un livre d'introduction.

Par ailleurs, les entiers et les nombres décimaux peuvent tous deux être négatifs, ce que vous voyez par exemple sur votre relevé de banque. Si vous savez que la valeur d'une variable ne sera jamais négative, vous pouvez élargir la gamme des valeurs à tenir dans les octets disponibles.

```
//[7]
unsigned int chocolatBarresEnStock;
```

Il n'existe pas de nombre négatif de barres de chocolat en stock, donc ici un unsigned int pourra être utilisé. Le type unsigned int représente des nombres entiers supérieurs ou égaux à zéro.

Déclarer une variable

Il est possible de déclarer une variable et de lui assigner une valeur d'un seul coup [8].

```
//[8]
int x = 10;
float y= 3.5, z = 42;
```

Cela vous évite un peu de frappe.

Types de données

Comme nous l'avons vu, les données stockées dans une variable peuvent être d'un ou plusieurs types spécifiques, par exemple, un int ou un float.

En Objectif-C, d'aussi simples types de données que ceux-ci sont aussi appelés données scalaires. Voici une liste des types de données scalaires courants disponibles en Objectif-C:

Nom	Type	Exemple	
void	Vide	Rien	
int	Entier	...-1, 0, 1, 2...	
unsigned	Entier non signé	0, 1, 2...	
float	Nombre à virgule flottante	-0.333, 0.5, 1.223, 202.85556	
double	Nombre double précision à virgule flottante	0.525252523332340938903245927930 21	
char	Caractère	bonjour	
BOOL	Booléen	0, 1; TRUE, FALSE; YES, NO.	(0, 1; VRAI, FAUX; OUI, NON.)

Opérations mathématiques

Dans les exemples précédents, nous avons réalisé une multiplication. Utilisez les symboles suivants, officiellement connus sous le nom d'opérateurs, pour faire des calculs mathématiques de base.

```
+  pour addition
-  pour soustraction
/  pour division
```

```
*  pour multiplication
```

À l'aide des opérateurs, nous pouvons effectuer un large éventail de calculs. Si vous examinez le code des programmeurs professionnels en Objectif-C, vous rencontrerez quelques particularités, probablement parce que ce sont des flemmards du clavier.

Au lieu d'écrire x = x + 1; les programmeurs recourent souvent à quelque chose comme [9] ou [10]

```
//[9]
x++;
```

```
//[10]
++x;
```

Dans les deux cas, cela signifie: incrémenter x de un. Dans certaines circonstances, il est important que le ++ soit placé avant ou après le nom de la variable. Examinez les exemples [11] et [12] ci-dessous.

```
//[11]
x = 10;
y = 2 * (x++);
```

```
//[12]
x = 10;
y = 2 * (++x);
```

Dans l'exemple [11], en fin de compte, y est égal à 20 et x est égal à 11. En revanche, dans la déclaration [12.2], x est incrémenté de un avant que la multiplication par 2 n'ait lieu. Donc, au final, x est égal à 11 et y est égal à 22. Le code de l'exemple [12] est équivalent à celui de l'exemple [13].

```
//[13]
x = 10;
x++;
y = 2 * x;
```

Ainsi, le programmeur a en fait fusionné deux déclarations en une seule. Personnellement, je pense que cela complique la lecture d'un programme. Si vous prenez le raccourci c'est bien, mais soyez conscient qu'un bogue peut y traîner.

Parenthèses

Cela vous paraîtra ringard si vous avez réussi votre secondaire, mais les parenthèses peuvent être utilisées pour déterminer l'ordre dans lequel les opérations seront effectuées. Ordinairement, * et / ont la priorité sur + et -. Donc, 2 * 3 + 4 est égal à 10. En utilisant des parenthèses, vous pouvez forcer l'exécution de la modeste addition en premier lieu: 2 * (3 + 4) est égale à 14.

Division

La division mérite une attention particulière, parce qu'il y a une différence considérable selon qu'elle est utilisée avec des entiers ou des flottants. Jetez un oeil aux exemples ci-après [14, 15].

```
//[14]
```

```
int x = 5, y = 12, rapport;
rapport =  y / x;
```

```
//[15]
float x = 5, y = 12, rapport;
rapport =  y / x;
```

Dans le premier cas [14], le résultat est 2. Seul le second cas [15], donne le résultat que vous attendez probablement: 2.4.

Booléens

Un booléen est une simple valeur logique, vraie ou fausse. 1 et 0 qui signifient vrai et faux sont souvent utilisés à la place, et peuvent être considérés comme équivalents:

True (Vrai)	False (Faux)
1	0

Ils sont fréquemment utilisés pour évaluer l'opportunité d'effectuer des actions en fonction de la valeur booléenne d'une variable ou d'une fonction.

Modulo

% (modulo) est un opérateur qui ne vous est probablement pas familier. Il ne fonctionne pas comme vous pourriez vous y attendre: l'opérateur modulo n'est pas un calcul de pourcentage. Le résultat de l'opérateur % est le reste de la division entière de la première opérande par la seconde (si la valeur de la seconde est égale à zéro, le comportement de % est indéfini).

```
//[16]
int x = 13, y = 5, reste;
reste =  x % y;
```

Maintenant, le résultat est que reste est égal à 3, parce que x est égal à 2*y + 3.
Voici quelques autres exemples de modulo:

```
21 % 7 est égal à 0
22 % 7 est égal à 1
23 % 7 est égal à 2
24 % 7 est égal à 3
27 % 7 est égal à 6
30 % 2 est égal à 0
31 % 2 est égal à 1
32 % 2 est égal à 0
33 % 2 est égal à 1
34 % 2 est égal à 0
50 % 9 est égal à 5
60 % 29 est égal à 2
```

Cela peut parfois être pratique, mais notez qu'il ne fonctionne qu'avec des entiers.
Une utilisation courante de modulo est de déterminer si un entier est pair ou impair. S'il est pair, un modulo de deux sera égal à zéro.
Sinon, il sera égal à une autre valeur. Par exemple:

```
//[17]
int unInt;
//Du code qui définit la valeur de unInt
if ((unInt % 2) == 0)
{
NSLog(@"unInt est pair");
}
else
{
NSLog(@"unInt est impair");
}
```

02: Pas de commentaires ? C'est inacceptable !

Introduction

En utilisant des noms de variables sensiblement différents, et intelligibles, nous pouvons rendre notre code plus compréhensible, que ce ce soit pour une relecture (pour nous), ou bien une relecture par les autres (dans le cas de logiciels libres par exemple) [1].

```
//[1]
float imageLargeur, imageLongueur, imageAire;
imageLargeur = 8.0;
imageLongueur = 4.5;
imageAire = imageLargeur * imageLongueur;
```

Jusqu'ici nos bouts de codes ne font que quelques lignes, mais même des programmes assez simples peuvent comporter des centaines de lignes de code. Quand vous relisez votre code après quelques semaines, ou quelques mois, il peut être difficile de se rappeler comment vous avez décidé de programmer, et quels moyens vous utilisez pour y parvenir. C'est là que les commentaires rentrent en jeu. Les commentaires vous aident à savoir ce qu'une partie de votre code fait et pourquoi c'est cette solution que vous avez choisi. Certains programmeurs vont si loin qu'ils commentent même leur classes, ce qui les aident à organiser leur idée, et leur évite de tomber dans une impasse.

Il est conseillé de commenter son code. Nous pouvons vous assurer que c'est un investissement rentable, qui vous rendra service plus tard. Aussi, si vous partagez votre code avec d'autres personnes, vos commentaires les aideront à adapter à leurs propres besoins votre programme.

Réaliser un commentaire

Pour commencer un commentaires, alignez deux slashes à la suite, tout ce qui suivra dans la ligne de votre code, sera considéré comme un commentaire.

```
// Ceci est un commentaire.
```

Dans Xcode les commentaires sont en vert. Si un commentaire doit s'étaler sur plusieurs lignes (par exemple dans l'en-tête de présentation de votre code source), il vaut mieux utiliser cette méthode : `/* Votre commentaire */`.

```
/* Ceci est un commentaire
écrit sur plusieurs lignes. */
```

« Outcommenting »

Nous vous expliquerons comment débugger un programme facilement — grâce à Xcode — plus longuement dans un autre chapitre. Cependant, une façon à « l'ancienne » de débugger un programme est de commenter des parties de code, afin de savoir quelle est la partie qui fonctionne mal. Via les `/* */`, désactiver certaines parties du code, pour voir si le reste tourne comme prévu. Cela vous permet de faire la chasse aux bugs. Si la partie commentée devait, par exemple, retourner une valeur, vous pouvez inclure une ligne temporaire où vous affectez à la variable une autre valeur particulière pour tester la bonne marche de votre programme.

Pourquoi commenter ?

L'importance des commentaires ne doit pas être sous-estimée. C'est souvent utile d'apposer une explication en bon français sur ce que fait

une longue série de commandes. C'est parce que de cette façon, vous n'aurez pas à déduire du code, ce qu'il fait. C'est le commentaire lui même qui le rappellera immédiatement. Vous devriez aussi utiliser les commentaires pour souligner les parties difficiles, ou impossible à déduire à partir du code. Par exemple, si vous programmez une fonction mathémathique utilisant un modèle spécifique décrit en détail dans un livre, vous souhaiterez surement mettre une référence bibliographique.

Quelques fois il peut être utile d'écrire un commentaire avant de coder. Cela vous aidera à structure vos pensées et programmer sera plus simple.

Les lignes de code contenues dans ce livre ne contiennent pas beaucoup de commentaires, car elles sont longuement expliqué après chaque partie de code.

03: Fonctions

Introduction

Le plus long morceau de code que vous avez pu voir jusqu'ici comportait seulement cinq déclarations. Réaliser des programmes de plusieurs milliers de lignes doit vous sembler bien lointain, mais de par la nature d'Objective-C, nous devons aborder tout d'abord la façon dont les programmes sont organisés.

Si un programme devait consister en une longue et continue succession de déclarations, il serait difficile de trouver et de réparer les erreurs. De plus, une série particulière de déclarations peut apparaitre en différents endroits de votre programme. Si il y avait un bug, vous devriez le réparer à ces différents endroits. Un cauchemar, parce qu'il est facile d'en oublier une (ou deux)! C'est pourquoi un moyen d'organiser le code a été pensé, rendant la correction d'erreurs plus aisée.

La solution à ce problème est de regrouper les déclarations selon leurs fonctions. Par exemple, vous pouvez avoir tout un groupe de déclarations vous permettant de calculer la surface d'un cercle. Une fois le code de ce groupe de déclarations jugé fiable, vous n'aurez plus jamais à le revérifier lors de la recherche d'une erreur de programmation. Ce groupe de déclarations, appelé fonction, à un nom, et vous pouvez appeler ce groupe de déclarations par son nom pour exécuter son code. Ce concept d'utilisation de fonctions est tellement fondamental, qu'il y a toujours au moins une fonction dans un programme: La fonction `main()`. Cette fonction `main()` est ce que le compilateur recherche, ainsi il saura où doit commencer le code lors de son exécution.

La fonction main()

Etudions plus en détail cette fonction `main()`. [1]

```
//[1]
main()
{
    // Corps de la fonction main() . Placer votre code ici.
}
```

La déclaration [1.1] montre le nom de la fonction, par exemple "main", suivi par des parenthèses entrantes et sortantes. Alors que "main" est un mot réservé, et que la fonction `main()` est obligatoire, lorsque vous définissez vos propres fonctions, vous pouvez les nommer comme bon vous semble. Les parenthèses sont là pour une bonne raison, mais nous n'en parlerons que plus loin dans ce chapitre. Dans les lignes qui suivent [1.3, 1.5], il y a des accolades. Nous devons placer notre code à l'intérieur de ces accolades `{ }`. Tout ce qui est placé entre ces accolades est appelé le corps de la fonction. J'ai repris un peu du code issu du premier chapitre et l'ai placé où il doit l'être [2].

```
//[2]
main()
{
    // Les variables sont déclarées ci-dessous
    float pictureWidth, pictureHeight, pictureSurfaceArea;
    // Nous initialisons les variables (nous donnons une
valeur aux variables)
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    // C'est ici que le calcul est réalisé
    pictureSurfaceArea = pictureWidth * pictureHeight;
}
```

Notre première fonction

Si nous devions continuer à ajouter du code à l'intérieur de la fonction `main()`, nous finirions par rencontrer des difficultés pour corriger le code, or nous voulions éviter d'écrire du code non structuré. Écrivons un autre programme, mais cette fois-ci en le structurant. En dehors de l'indispensable fonction `main()`, nous allons créer une fonction `circleArea()` [3].

```
//[3]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea;
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    pictureSurfaceArea = pictureWidth * pictureHeight;
}
circleArea()   // [3.9]
{
}
```

Ce fut facile, mais notre fonction personnalisée débutant à la déclaration [3.9] ne fait pour le moment rien. Notez que la spécification de la fonction est en dehors du corps de la fonction `main()`. En d'autres termes, les fonctions ne sont pas imbriquées.

Notre nouvelle fonction `circleArea()` doit être appelée depuis la fonction `main()`. Voyons comment nous pouvons faire cela [4].

```
//[4]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea;  // [4.4]
    pictureWidth  = 8.0;
    pictureHeight = 4.5;
    circleRadius  = 5.0; // [4.7]
    pictureSurfaceArea =  pictureWidth * pictureHeight;
    // Ici nous appelons notre fonction!
    circleSurfaceArea = circleArea(circleRadius); // [4.10]
}
```

Note: le reste du programme n'est pas affiché ici (voir [3]).

Passer des arguments

Nous avons ajouté deux nom de variables de type `float` [4.4], et nous avons initialisé la variable `circleRadius`, par exemple en lui donnant une valeur [4.7]. Le plus intéressant est la ligne [4.10], ou la fonction `circleArea()` est appelée. Comme vous pouvez le constater, le nom de la variable `circleRadius` a été placé entre parenthèses. C'est un argument de la fonction `circleArea()`. La valeur de cette variable `circleRadius` va être passée à la fonction `circleArea()`. Quand la fonction `circleArea()` à fait son travail en effectuant le calcul, elle doit retourner le résultat. Modifions la fonction `circleArea()` de [3] pour refléter ceci [5].

Note: seule la fonction `circleArea()` est montrée.

```
//[5]
circleArea(float theRadius) // [5.1]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;  // pi
multiplié par r au carré   [5.4]
    return theArea;
}
```

En [5.1] nous définissons que pour la fonction `circleArea()` une valeur de type `float` est requise en entrée. Une fois reçue, cette valeur est stockée dans une variable nommée `theRadius`. Nous utilisons une seconde variable, par exemple `theArea` pour stocker le résultat du calcul en [5.4], nous devons alors la déclarer en [5.3], de la même façon que nous avons déclaré les variables dans la fonction `main()` [4.4]. Vous noterez que la déclaration de la variable `theRadius` est faite à l'intérieur de parenthèses [5.1]. La ligne [5.5] retourne le résultat à l'endroit du programme d'où la fonction a été appelée. Pour résultat, à la ligne [4.11], la variable `circleSurfaceArea` est définie avec cette valeur.

La fonction en exemple [5] est complète, excepté pour une chose. Nous n'avons pas spécifié le type de données que la fonction va retourner. Le compilateur attend que nous le fassions, nous n'avons alors d'autre choix que d'obéir et d'indiquer `float` comme type [6.1].

```
//[6]
float circleArea(float theRadius)  //[6.1]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

Comme l'indique le premier mot de la ligne [6.1], la valeur retournée par cette fonction (par exemple, la valeur de la variable `theArea`) est de type `float`. En tant que programmeur, vous devez vous assurer que la variable `circleSurfaceArea` dans la fonction `main()` [4.8] est de ce type également, afin que le compilateur ne nous harcèle pas à son propos.

Toutes les fonctions ne requièrent pas un argument. Si il n'y en a pas, les parenthèses () sont toujours requises, même si elles sont vides.

```
//[7]
int throwDice()
{
    int noOfEyes;
    // Code pour générer une valeur aléatoire entre 1 et 6
    return noOfEyes;
}
```

Retourner des valeurs

Toutes les fonctions ne retournent pas de valeur. Si une fonction ne retourne pas une valeur, elle est de type `void`. La déclaration `return` est alors optionnelle. Si vous l'utilisez, le mot clé `return` ne doit pas être suivi d'une valeur/nom de variable.

```
//[8]
void beepXTimes(int x);
{
    // Code pour biper x fois
    return;
}
```

Si une fonction a plus d'un argument, comme la fonction pictureSurfaceArea() ci-dessous, les arguments sont séparés par une virgule.

```
//[9]
float pictureSurfaceArea(float theWidth, float theHeight)
{
    // Code pour calculer la surface
}
```

La fonction main() devrait, par convention, retourner un entier, et si oui, elle doit avoir une déclaration return aussi. Elle devrait retourner 0 (zéro, [10.9]), pour indiquer que la fonction a été exécutée sans problème. Comme la fonction main() retourne un entier, nous devons écrire "int" avant main() [10.1]. Placons tout le code que nous avons dans une liste.

```
//[10]
int main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
       circleRadius, circleSurfaceArea;
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea =  pictureWidth * pictureHeight;
    circleSurfaceArea = circleArea(circleRadius);      //
[10.8]
    return 0;      // [10.9]
}
float circleArea(float theRadius)           // [10.12]
```

```
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

Faisons tout marcher

Comme vous pouvez le voir [10], nous avons une fonction `main()` [10.1] et une autre fonction que nous avons défini nous même [10.12]. Si nous devions compiler ce code, le compilateur bloquerait. A la ligne [10.8] il affirmerait ne connaitre aucune fonction nommée `circleArea()`. Pourquoi? Apparemment, le compilateur commence la lecture de la fonction `main()` et rencontre soudainement quelque chose qu'il ne connait pas. Il ne va pas voir plus loin et vous donne cet avertissement. Pour le satisfaire, ajoutez juste une déclaration de fonction avant la déclaration contenant `int main()` [11.1]. Il n'y a rien de dur, car c'est la même qu'à la ligne [10.12], sauf qu'elle se termine par un point-virgule. Maintenant le compilateur ne sera pas surpris quand il rencontrera cet appel de fonction.

```
//[11]
float circleArea(float theRadius);  // déclaration de
fonction
int main()
{
    // Code de la fonction main...
}
```

Note: le reste du programme n'est pas affiché (voir [10]).

Nous allons bientôt compiler ce programme pour de vrai. Mais voyons tout d'abord quelques détails pratiques.

Lors de l'écriture d'un programme, il est souhaitable d'avoir à l'esprit une future réutilisation du code. Notre programme pourrait avoir une fonction `rectangleArea()`, comme affichée ci-dessous [12], et cette fonction porrait être appelée dans notre fonction `main()`. Ceci est utile même si le code placé dans la fonction est seulement utilisé une fois. La fonction `main()` gagne en lisibilité. Si nous avons à débugger notre code, il sera plus facile de trouver où les erreurs se trouvent dans notre programme. Vous pourriez en découvrir dans une fonction. Au lieu d'avoir à parcourir une longue séquence de déclarations, vous avez alors juste à vérifier les déclarations de cette fonction, qui sont faciles à trouver, grâce aux accolades entrantes et sortantes.

```
//[12]
float rectangleArea(float length, float width)
{
    return (length * width); //[12.3]
}
```

Comme vous pouvez le voir, dans un cas simple comme celui ci, il est possible d'avoir une seule déclaration [12.3] pour et calculer et retourner le résultat. J'ai utilisé la variable superflue `theArea` en [10.14] simplement pour vous montrer comment déclarer une variable dans une fonction.

Alors que les fonctions que nous avons définies dans ce chaptire sont plutôt triviales, il est important de réaliser que vous pouvez modifier une fonction sans impact sur le code qui l'appelle aussi longtemps que vous ne changez pas la déclaration de cette fonction (c'est à dire, sa première ligne).

Par exemple, vous pouvez changer le nom des variables dans la fonction, et elle fonctionnera toujours (sans que cela n'affecte non plus le fonctionnement du reste du programme). Quelqu'un d'autre pourra écrire une fonction, et vous pourrez l'utiliser sans savoir ce

qu'elle contient. Tout ce que vous avez à savoir est comment utiliser la fonction. Ce qui signifie savoir:

- le nom de la fonction
- le nombre, l'ordre et le type des arguments de la fonction
- ce que la fonction retourne (la valeur de la surface du rectangle), et le type du résultat

Dans l'exemple [12], ces réponses sont, respectivement:

- rectangleArea
- Deux arguments, tous les deux de type `float`, où le premier représente la longueur, le second la largeur.
- La fonction retourne quelque chose, et le résultat est de type `float` (comme nous l'apprend le premier terme de la déclaration [12.1]).

Variables protégées

Le code à l'intérieur de la fonction est protégé du programme principal, et des autres fonctions, d'ailleurs.

Ce que cela signifie est que la valeur d'une variable au sein d'une fonction n'est par défaut pas affectée par une variable d'une autre function, même si elle a le même nom. C'est une des plus essentielles fonctionnalités d'Objective-C. Dans le Chapitre 5, nous étudierons à nouveau ce comportement. Mais tout d'abord, nous allons débuter avec Xcode lancer le programme ci-dessus [10].

04: Affichage à l'écran

Introduction

Nous avons bien progressé dans notre programme, mais n'avons pas discuté de la façon d'afficher les résultats de nos calculs. Le langage Objectif-C ne sait pas comment faire cela, mais heureusement des gens ont écrit des fonctions d'affichage, dont nous pouvons tirer profit. Il existe différentes façons d'afficher un résultat à l'écran. Dans ce livre, nous utiliserons une fonction fournie par l'environnement Cocoa d'Apple: La fonction NSLog(). C'est appréciable, car maintenant vous n'avez plus à vous soucier de "l'impression" de vos résultats à l'écran (ni rien à programmer).

Mais où donc NSLog() s'affiche t-elle? Xcode fournit la Console pour afficher des messages d'historique. Pour ouvrir la Console, choisir Console dans le menu Run (Exécuter) (Pomme-Maj-R). Lorsque vous créez et exécutez votre application, tous les messages NSLog() de cette application s'affichent ici.

La fonction NSLog() est originellement destinée à l'affichage des messages d'erreur, et non à la sortie des résultats d'application. Cependant, elle est tellement facile à utiliser que nous l'adopterons dans ce livre pour afficher les résultats. Une fois acquise une certaine maîtrise de Cocoa, vous pourrez utiliser des techniques plus sophistiquées.

Utiliser NSLog

Voyons comment la fonction NSLog() est utilisée. Dans le fichier main.m, entrez le code suivant:

```
//[1]
int main()
{
    NSLog(@"Julia est mon actrice favorite.");
    return 0;
}
```

A l'exécution, la déclaration de l'exemple [1] se traduira par l'affichage du texte "Julia est mon actrice préférée.". Ce texte entre @" et " est appelé une chaîne.

En plus de la chaîne elle-même, la fonction NSLog() affiche diverses informations supplémentaires, comme la date et le nom de l'application. Par exemple, la sortie complète du programme [1] sur mon système est:

```
2005-12-22 17:39:23.084 test[399] Julia est mon actrice favorite.
```

Une chaîne peut avoir une longueur de zéro ou plusieurs caractères.

Note: Dans les exemples suivants, seules les déclarations intéressantes de la fonction main() sont affichées.

```
//[2]
NSLog(@"");
NSLog(@" ");
```

La déclaration [2.1] contient zéro caractère et est appelée une chaîne vide (c'est-à-dire, qu'elle a une longueur égale à zéro). La déclaration [2.2] n'est pas une chaîne vide, en dépit de son apparence. Elle contient un espace, de sorte que la longueur de cette chaîne est de 1.

Plusieurs séquences de caractères spéciaux ont une signification particulière dans une chaîne. Ces séquences de caractères spéciaux sont dites séquences d'échappement.

Par exemple, pour forcer le dernier mot de notre phrase à commencer à s'afficher sur une nouvelle ligne, un code spécial doit être inclus dans la déclaration [3,1]. Ce code est \n, raccourci du caractère de retour à la ligne.

```
//[3]
NSLog(@"Julia est mon actrice \nfavorite.");
```

Désormais, la sortie ressemble à ceci (seule la sortie concernée est affichée):

```
Julia est mon actrice
favorite.
```

Le slash inversé de [3.1] est un caractère d'échappement, qui indique à la fonction NSLog() que le caractère suivant n'est pas un banal caractère à afficher à l'écran, mais un caractère qui a une signification spéciale: dans ce cas, le "n" signifie "commencer une nouvelle ligne".

Dans les rares cas où vous souhaitez afficher un slash inversé à l'écran, cela peut vous sembler un problème. Si un caractère suivant un slash inversé a un sens particulier, comment est-il possible d'afficher un slash inversé? Eh bien, il suffit de mettre un autre slash inversé avant (ou bien sûr après) le slash inversé. Cela indique à la fonction NSLog() que le (second) slash inversé, c'est-à-dire celui le plus à droite, doit être affiché, et que toute signification spéciale doit être ignorée). Voici un exemple:

```
//[4]
NSLog(@"Julia est mon actrice favorite.\\n");
```

La déclaration [4.1] donnerait, lors de l'exécution

```
Julia est mon actrice favorite.\n
```

Afficher les variables

Jusqu'ici, nous n'avons affiché que des chaînes statiques. Affichons à l'écran la valeur obtenue par un calcul.

```
//[5]
int x, entierAffichable;
x = 1;
entierAffichable = 5 + x;
NSLog(@"La valeur de l'entier est %d.", entierAffichable);
```

Bien noter que, entre les parenthèses, nous avons une chaîne, une virgule et un nom de variable. La chaîne contient quelque chose de bizarre: %d. A l'instar du slash inversé, le caractère pourcentage % a une signification spéciale. S'il est suivi d'un d (raccourci pour nombre décimal), après exécution, la valeur de sortie de ce qui se trouve après la virgule, c'est-à-dire la valeur actuelle de la variable entierAffichable, sera insérée à l'emplacement de %d. L'exécution de l'exemple [5] donne pour résultat

```
La valeur de l'entier est 6.
```

Pour afficher un flottant, vous devez utiliser %f au lieu de %d.

```
//[6]
float x, flottantAffichable;
x = 12345.09876;
flottantAffichable = x/3.1416;
NSLog(@"La valeur du flottant est %f.", flottantAffichable);
```

Le nombre de chiffres significatifs affichés (ceux après le point) dépend de vous. Pour afficher deux chiffres significatifs, mettre .2 entre % et f, comme ceci:

```
//[7]
float x, flottantAffichable;
x = 12345.09876;
flottantAffichable = x/3.1416;
NSLog(@"La valeur du flottant est %.2f.",
flottantAffichable);
```

Ensuite, si vous savez comment répéter les calculs, vous pourrez souhaiter créer une table de valeurs. Imaginez une table de conversion de degrés Fahrenheit en degrés Celsius. Si vous souhaitez un bel affichage des valeurs, vous voudrez que les valeurs des deux colonnes de données aient une largeur donnée. Vous pouvez spécifier cette largeur par une valeur entre % et f (ou % et d, dans ce cas). Toutefois, si la largeur que vous spécifiez est inférieure à la largeur du nombre, la largeur du nombre prévaut.

```
//[8]
int x = 123456;
NSLog(@"%2d", x);
NSLog(@"%4d", x);
NSLog(@"%6d", x);
NSLog(@"%8d", x);
```

L'exemple [8] donne la sortie suivante:

```
123456
123456
123456
  123456
```

Dans les deux premières déclarations [8.2, 8.3], nous réclamons trop peu de place pour l'affichage du nombre dans son intégralité, mais l'espace est quand même pris. Seule la déclaration [8.5] spécifie une largeur supérieure à la valeur, aussi nous voyons maintenant l'apparition d'espaces additionnels, ce qui est indicatif de la largeur de l'espace réservé pour le nombre.

Il est également possible de combiner les spécifications de largeur et le nombre de chiffres décimaux à afficher.

```
//[9]
float x=1234.5678;
NSLog(@"Réserver un espace de 10, et afficher 2 chiffres
significatifs.");
NSLog(@"%10.2f", x);
```

Afficher plusieurs valeurs

Bien sûr, il est possible d'afficher plus d'une valeur, ou toute combinaison de valeurs [10,3]. Vous devez vous assurer que vous avez bien indiqué le type de données (int, float), en utilisant %d et %f.

```
//[10]
int x = 8;
float pi = 3.1416;
NSLog(@"La valeur de l'entier est %d, alors que la valeur du flottant est %f.", x, pi);
```

Faire correspondre les symboles aux valeurs

Une des erreurs la plus courante des débutants, est de mal spécifier le type de données dans NSLog() et autres fonctions. Si vos résultats sont bizarres, ou tout simplement que le programme plante sans raison, examinez les indicateurs de votre frappe de données!

Par exemple, si vous vous trompez pour le premier, le deuxième peut ne pas être affiché correctement non plus! Par exemple,

```
//[10b]
int x = 8;
float pi = 3.1416;
NSLog(@"La valeur de l'entier est %f, alors que la valeur du flottant est %f.", x, pi);
// Ceci lira: NSLog(@"La valeur de l'entier est %d, alors que la valeur du flottant est %f.", x, pi);
```

donnera la sortie suivante:

```
La valeur de l'entier est 0.000000, alors que la valeur du flottant est 0.000000.
```

Liaison avec Fondation

Nous ne sommes qu'à une question et une réponse de l'exécution de notre premier programme.

Alors, comment notre programme a t-il connaissance de cette si utile fonction NSLog()? Eh bien, il ne l'a pas, à moins que nous ne lui demandions. Pour ce faire, notre programme doit dire au compilateur d'importer une bibliothèque de choix (qui heureusement est fournie gratuitement avec tout Mac), incluant la fonction NSLog(), en utilisant la déclaration:

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

Cette déclaration doit être la première déclaration de notre programme. Lorsque l'on réunit tout ce que nous avons appris dans ce chapitre, on obtient le code suivant, que nous exécuterons dans le prochain chapitre.

```
//[11]
#import <Foundation/Foundation.h>
float cercleAire(float leRayon);
float rectangleAire(float largeur, float hauteur);
int main()
{
    float imageLargeur, imageHauteur, imageAire,
    cercleRayon, cercleSurfaceAire;
    imageLargeur  = 8.0;
    imageHauteur = 4.5;
    cercleRayon  = 5.0;
    imageAire = rectangleAire(imageLargeur, imageHauteur);
    cercleSurfaceAire = cercleAire(cercleRayon);
```

```
    NSLog(@"Aire du cercle: %10.2f.", cercleSurfaceAire);
    NSLog(@"Area d'image: %f. ", imageAire);
    return 0;
}



float cercleAire(float leRayon)                // première
fonction personnelle
{
    float lAire;
    lAire = 3.1416 * leRayon * leRayon;
    return lAire;
}
float rectangleAire(float largeur, float hauteur) // seconde
fonction personnelle
{
    return largeur*hauteur;
}
```

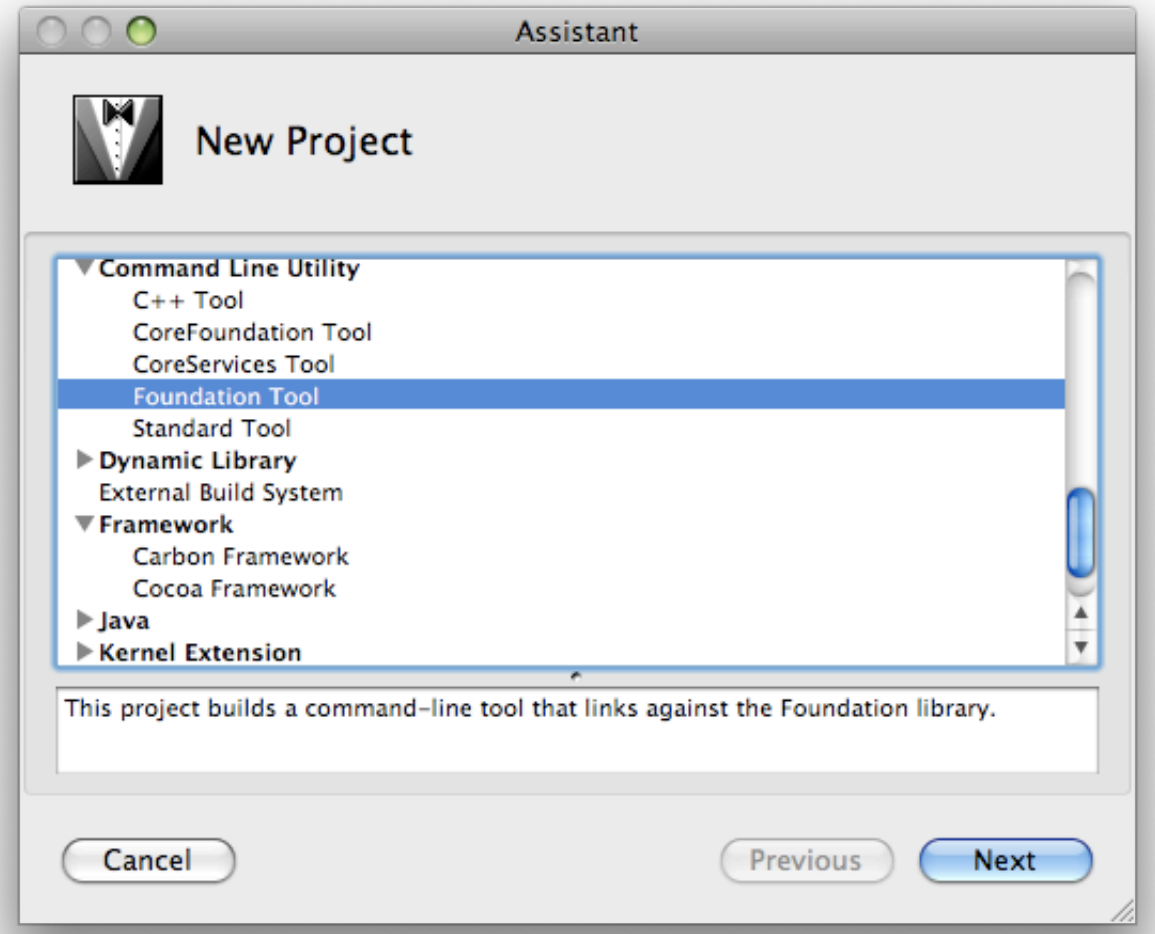
05: Compiler et exécuter un programme

Introduction

Le code que nous avons écrit jusqu'à présent n'est rien de plus que du texte que nous autres, humains, pouvons lire. Bien que ce ne soit pas vraiment de la prose pour nous, c'est encore pire pour votre Mac. Il ne peut rien en faire du tout! Un programme spécial, appelé compilateur, est nécessaire pour convertir votre code de programmation en un code d'exécution qui pourra être exécuté par votre Mac. C'est le rôle de l'environnement de programmation gratuit Xcode d'Apple. Vous devrez avoir installé Xcode depuis le disque fourni avec votre exemplaire de Mac OS X. En tout cas, vérifiez que vous avez la dernière version, que vous pouvez télécharger depuis la section développeur du site d'Apple à http://developer.apple.com (enregistrement gratuit requis).

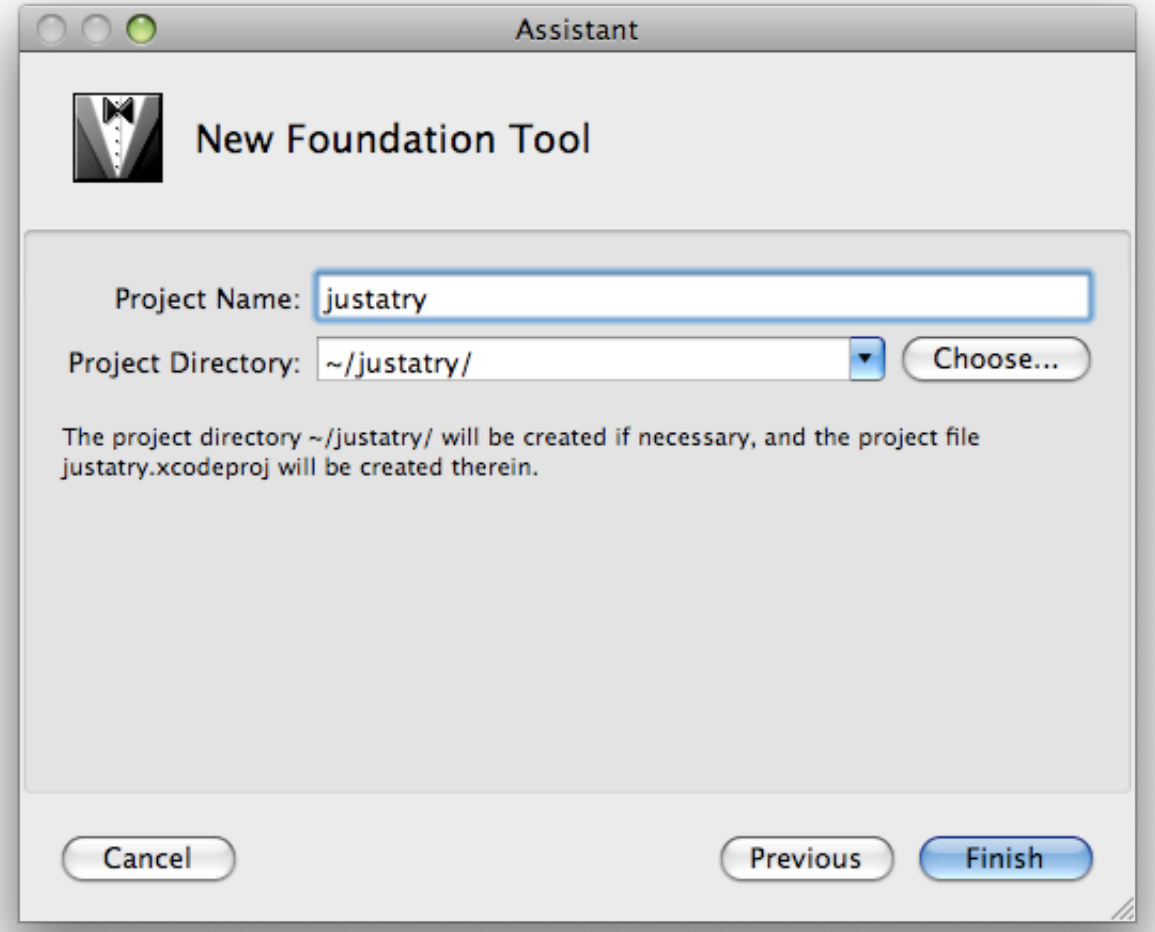
Créer un projet

Maintenant, lancez Xcode, que vous trouverez dans le dossier Applications du dossier Developer (Développeur). La première fois, il vous posera quelques questions. Acceptez les suggestions par défaut, elles sont excellentes, et vous pourrez toujours les changer ultérieurement, si vous le souhaitez, au moyen des Preferences (Préférences). Pour vraiment bien démarrer, choisissez New Project (Nouveau projet) dans le menu Fichier. Une fenêtre de dialogue s'affiche, contenant une liste des types de projets possibles.



L'assistant Xcode vous permet de créer de nouveaux projets.

Nous voulons créer un programme très simple en Objectif-C, sans GUI (Interface Graphique Utilisateur), donc faites défilez vers le bas jusqu'à sélectionner Foundation Tool (Outil de création) dans la section Command Line Utility (Utilitaire de ligne de commande).



Définir le nom et l'emplacement du nouveau projet.

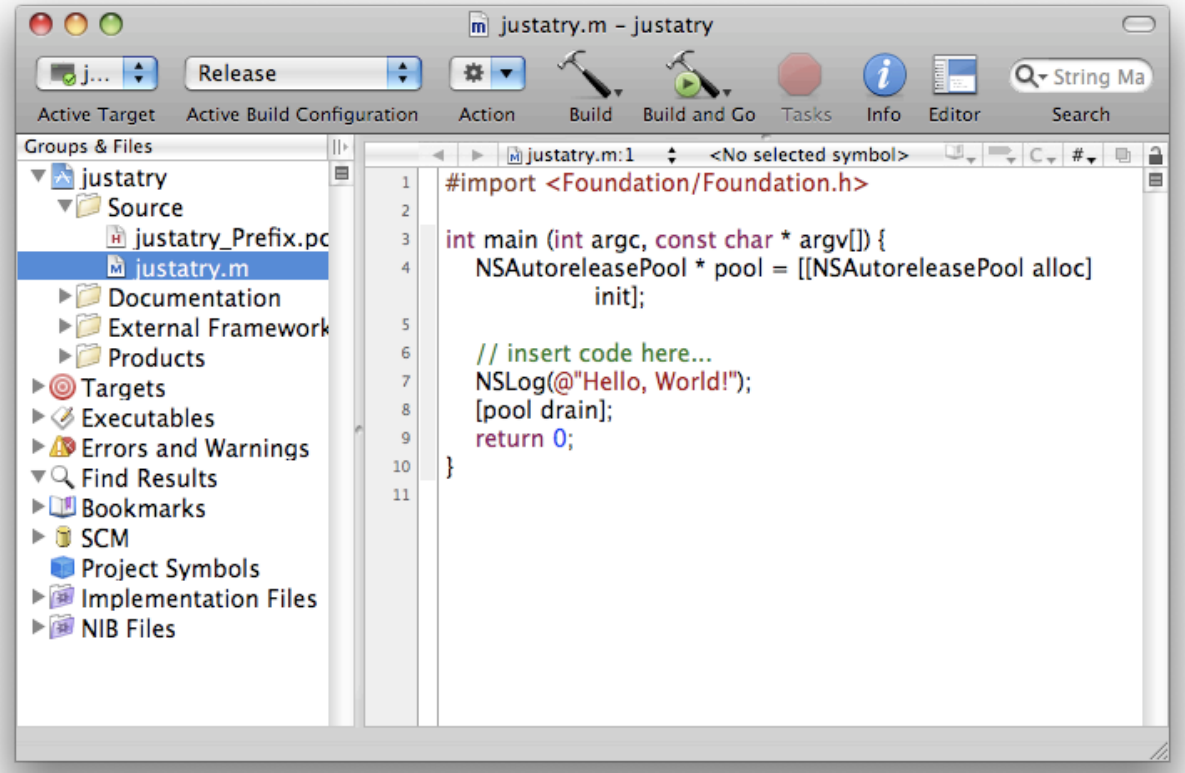
Entrez un nom pour votre application, tel que "justeunessai". Choisissez un emplacement où enregistrer votre projet, puis cliquez sur Terminer.

Le projet que nous nous apprêtons à créer peut être exécuté depuis le Terminal. Si vous vous sentez en mesure de le faire, et souhaitez éviter des tracas, assurez-vous que le nom de votre projet ne se compose que d'un mot. De même, il est de coutume de ne pas faire commencer par une majuscule les noms de programmes lancés par le terminal. D'autre part, les noms de programmes ayant une interface utilisateur graphique devront commencer par une majuscule.

Explorer Xcode

Maintenant, vous vous trouvez face à une fenêtre que vous verrez très souvent en tant que programmeur. La fenêtre comporte deux cadres. A gauche il y a le cadre "Groups & Files (Groupes & Fichiers)" permettant d'accéder à tous les fichiers composant votre programme. Actuellement il n'y a en pas trop, mais ultérieurement, lorsque vous créerez des programmes multilingues avec GUI, c'est là que les fichiers de votre GUI et les différentes langues pourront être trouvés. Les fichiers sont regroupés et conservés dans des dossiers, mais ne recherchez pas ces dossiers sur votre disque dur. Xcode fournit ces dossiers virtuels ("Groups") à seules fins d'organiser votre bazar.

Dans le cadre de gauche nommé Groupes et Fichiers, ouvrez le groupe justeunessai pour atteindre le groupe intitulé Source. Dans celui-ci, il y a un fichier nommé justeunessai.m [1]. Vous vous souvenez que chaque programme doit contenir une fonction appelée `main()`? Eh bien, c'est le fichier qui contient cette fonction `main()`. Plus loin dans ce chapitre, nous le modifierons pour y inclure le code de notre programme. Si vous ouvrez justeunessai.m en double-cliquant sur son icône, vous aurez une agréable surprise. Apple vous a déjà créé la fonction `main()`.



Xcode affichant la fonction `main()`.

```
//[1]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) //[1.2]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]
init]; //[1.4]
  // insérez votre code ici...
  NSLog(@"Bonjour, tout le monde!");
  [pool drain]; //[1.7]
  return 0;
}
```

Jetez un coup d'oeil au programme et cherchez-y des choses connues. Vous verrez:

- La déclaration d'importation requise pour des fonctions telles que `NSLog()`, commençant par un dièse.
- La fonction `main()`.
- Les accolades qui contiendront le corps de notre programme.
- Un commentaire, qui nous invite à mettre notre code là.
- Une déclaration `NSLog()` pour afficher une chaîne à l'écran.
- La déclaration `return 0;`.

Il y a aussi deux ou trois choses que vous ne connaîtrez pas:

- Deux curieux arguments entre les parenthèses de la fonction main() [1.2]
- Une déclaration commençant par NSAutoreleasePool [1.4]
- Une autre déclaration contenants les mots pool et drain [1.7]

Personnellement, je ne suis pas très ravi quand des auteurs de livre me présentent du code plein de déclarations inconnues en me promettant que tout s'éclaircira plus tard. Pour sûr. C'est la raison pour laquelle j'ai changé mes habitudes en vue de vous familiariser avec la notion de «fonctions» afin que vous ne soyez pas confronté à de trop nombreux nouveaux concepts.

Vous savez déjà que les fonctions sont une moyen d'organiser un programme, que chaque programme a une fonction `main()`, et à quoi ressemble une fonction. Cependant, je dois admettre que, pour l'instant, je ne peux pleinement expliquer tout ce que vous voyez dans l'exemple [1]. Je suis vraiment désolé de devoir vous demander d'ignorer, pour le moment, ces déclarations ([1.2, 1.4 et 1.7]). Il y a d'autres choses du langage Objectif-C avec lesquelles vous devez d'abord vous familiariser, afin de pouvoir écrire de petits programmes. La bonne nouvelle est que vous avez déjà passé deux chapitres difficiles, et que les trois chapitres à venir sont assez faciles. Ensuite, nous devrons à nouveau faire face à des choses plus difficiles.

Si vous ne souhaitez vraiment pas rester sans explication, en voici le résumé.

Les arguments de la fonction main() sont nécessaires pour exécuter le programme depuis le Terminal. Votre programme occupe de la mémoire. De la mémoire que d'autres programmes souhaiteront utiliser lorsque vous en aurez fini avec lui. En tant que programmeur, c'est à vous de réserver la mémoire dont vous avez besoin. Tout aussi important, vous devez restituer la mémoire quand vous en avez fini. C'est ce à quoi servent les deux déclarations contenant le mot "pool"

.

Build and Go (Compiler et exécuter)

Exécutons le programme fourni par Apple [1]. Tout d'abord nous devons ouvrir la fenêtre Console, se trouvant dans le menu Run (Exécuter), pour voir les résultats. Puis pressez sur l'icône du deuxième marteau étiquetée "Build and Go" pour construire (compiler) et exécuter l'application.

Le bouton Build and Go.

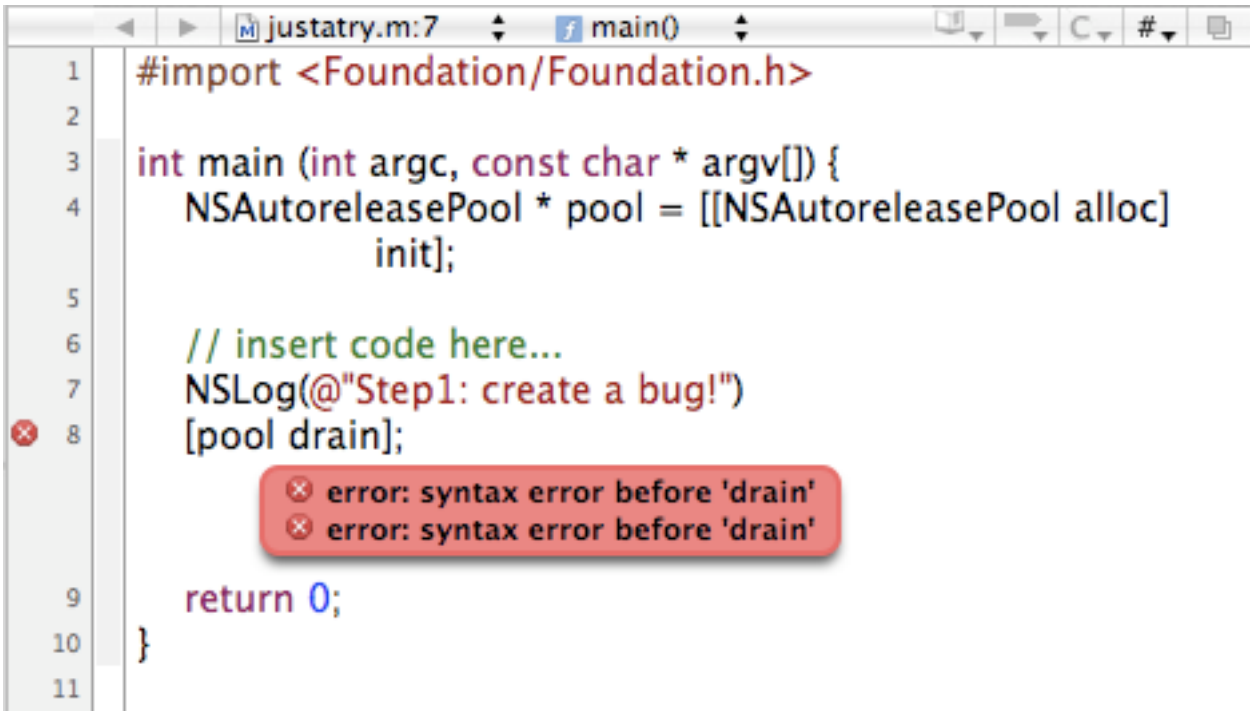
Le programme est exécuté et les résultats sont affichés dans la fenêtre Console, avec quelques informations supplémentaires. La dernière phrase indique que le programme a quitté (s'est arrêté) sur return 0. Là, vous voyez la valeur zéro qui est retournée par la fonction `main()`, comme indiqué au chapitre 3 [7-9]. Donc, notre programme est parvenu à la dernière ligne et ne s'est pas arrêté prématurément. jusqu'ici, tout va bien!

Boguer

Revenons à l'exemple [1], et voyons ce qui se passe s'il y a un bogue dans le programme. Par exemple, j'ai remplacé la déclaration `NSLog()` par une autre, mais ai "oublié" le point-virgule indiquant la fin de la déclaration.

```
//[2]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]
init];
  // insérez le code ici...
  NSLog(@"Julia est mon actrice favorite")   //Oups, j'ai
oublié le point-virgule!
  [pool drain];  //[2.9]
  return 0;
}
```

Pour construire l'application, pressez sur l'icône de construction dans la barre d'outils. Un cercle rouge apparaît devant la déclaration [2,9].



Xcode signale une erreur de compilation.

Si vous cliquez dessus, la ligne en dessous de la barre d'outils donne une brève description du grief:

```
error: parse error before "drain" (erreur: erreur de syntaxe avant "drain")
```

L'analyse syntaxique est l'une des premières choses que fait un compilateur: Il parcourt le code et vérifie s'il peut comprendre chaque ligne. Pour l'aider à comprendre la signification des différentes parties, c'est à vous de fournir des indices. Ainsi, pour la déclaration d'importation [2.1], vous devez mettre un signe dièse (#). Pour indiquer la fin d'une instruction [2.8], vous devez mettre un point-virgule. Quand le compilateur arrive à la ligne [2.9], il remarque que quelque chose ne va pas. Toutefois, il ne se rend pas compte que le problème ne s'est pas produit dans cette ligne, mais dans la ligne où le point-virgule est manquant. La grande leçon ici est que si le compilateur essaie de donner un commentaire judicieux, le commentaire n'est pas nécessairement une description précise du vrai problème, et que la position dans le programme n'est pas nécessairement le véritable emplacement de l'erreur (mais elle sera probablement très proche).

Corrigez le programme en rajoutant le point-virgule et exécutez de nouveau le programme pour être sûr qu'il marche bien.

Notre première application

Prenons maintenant le code du dernier chapitre, et intriquons-le dans le code fourni par Apple [1], ce qui donne l'exemple [3].

```
//[3]
#import <Foundation/Foundation.h>
float cercleAire(float leRayon);    // [3.3]
float rectangleAire(float largeur, float hauteur);    //
[3.4]
int main (int argc, const char * argv[])    // [3.6]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]
init];
  int imageLargeur;
  float imageHauteur, imageAire,
        cercleRayon, cercleSurfaceAire;
  imageLargeur = 8;
  imageHauteur = 4.5;
  cercleRayon = 5.0;
  imageAire =  imageLargeur * imageHauteur;
  cercleSurfaceAire = cercleAire(cercleRayon);
  NSLog(@"Aire de l'image: %f.  Aire du cercle: %10.2f.",
        imageAire, cercleSurfaceAire);
  [pool drain];
  return 0;
}



float cercleAire(float leRayon)           // [3.22]
{
```

```
  float lAire;
  lAire = 3.1416 * leRayon * leRayon;
  return lAire;
}
float rectangleAire(float largeur, float hauteur)  // [3.29]
{
  return largeur*hauteur;
}
```

Prenez votre temps pour être sûr de bien comprendre la structure du programme. Nous avons les en-têtes de fonction [3.3, 3.4] de nos propres fonctions `cercleAire()` [3.22] et `rectangleAire()` [3.29] avant la fonction `main()` [3.6], comme il se doit. Nos propres fonctions se trouvent en dehors des accolades de la fonction `main()` [3.5]. Nous avons mis le code du corps de la fonction `main()` là où Apple nous a dit de le mettre.

Lorsque le code est exécuté, nous obtenons la sortie suivante:

```
Aire de l'image: 36.000000.  Aire du cercle:      78.54.
justeunessai a quitté avec l'état 0.
```

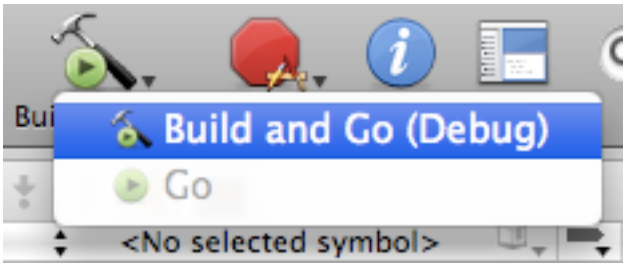
Déboguer

Pour un programme plus compliqué, il devient plus difficile de le déboguer. Si jamais vous voulez découvrir ce qui se passe à l'intérieur du programme pendant qu'il tourne, Xcode rend cela facile à faire. Il suffit de cliquer dans la marge grise à gauche des déclarations dont vous souhaitez connaître les valeurs des variables. Xcode va insérer un "point d'arrêt" représenté par une flèche bleue.

```
8
9   #import <Foundation/Foundation.h>
10
11  float circleArea(float theRadius);    // [3.3]
12  float rectangleArea(float width, float height);    // [3.4]
13  int main (int argc, const char * argv[])    // [3.6]
14  {
15          NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
16          int pictureWidth;
17          float pictureHeight, pictureSurfaceArea,
18          circleRadius, circleSurfaceArea;
19          pictureWidth = 8;
20          pictureHeight = 4.5;
21          circleRadius = 5.0;
22          pictureSurfaceArea =  pictureWidth * pictureHeight;
23          circleSurfaceArea = circleArea(circleRadius);
24          NSLog(@"Area of picture: %f.  Area of circle: %10.2f.",
25                  pictureSurfaceArea, circleSurfaceArea);
26          [pool drain];
27          return 0;
28  }
```

Définir un point d'arrêt dans votre code

Bien noter que vous verrez les valeurs des variables d'avant que cette déclaration donnée ne soit exécutée; donc, souvent, vous devrez mettre le point d'arrêt sur la déclaration suivant celle qui vous intéresse.
Maintenant, maintenez la souris enfoncée tout en cliquant sur le deuxième bouton de marteau de la barre d'outils, et un menu se déroulera.



Le menu déroulant Build and Go (Debug).

Qui peut se traduire par:

Pour suivre ce qui se passe, vous devrez ouvrir les fenêtres Console et Debugger (Débogueur) du menu Run (Exécuter). La console affiche un texte de ce genre:

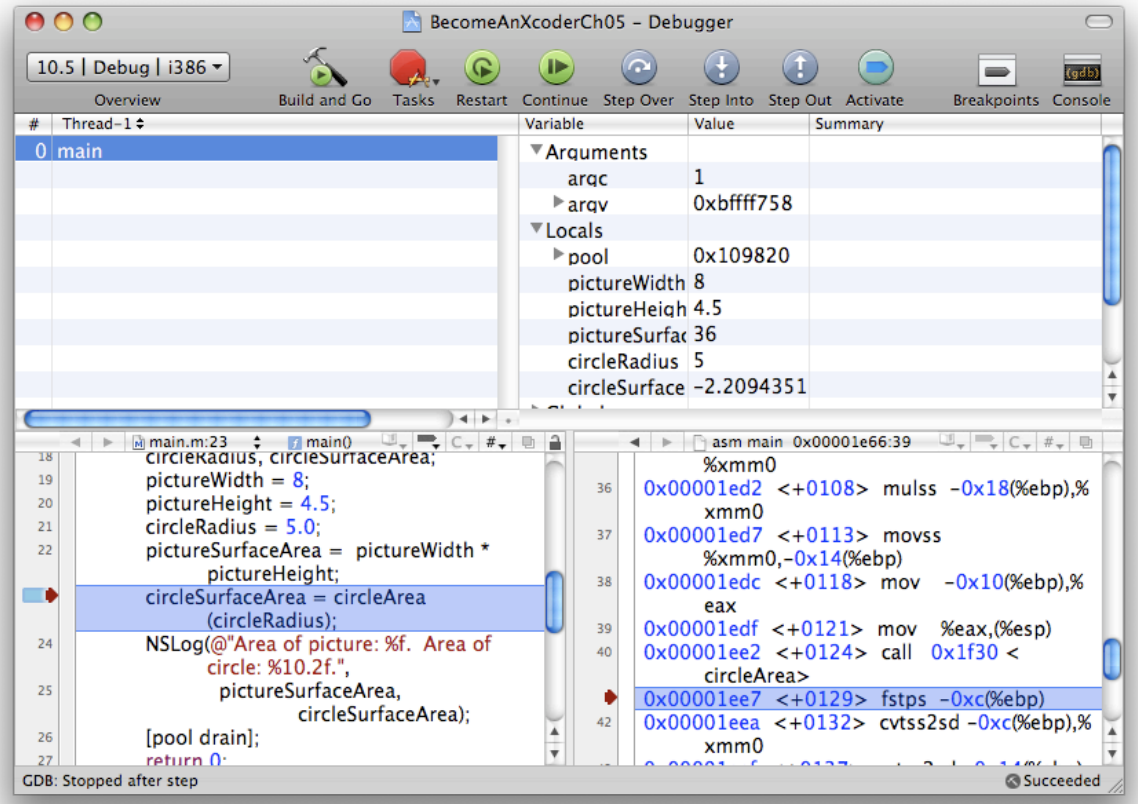
```
[Session started at 2009-06-03 15:48:02 +1000.]
Loading program into debugger…
GNU gdb 6.3.50-20050815 (Apple version gdb-956) (Wed Apr 30 05:08:47 UTC 2008)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-apple-darwin".tty /dev/ttys000
Program loaded.
sharedlibrary apply-load-rules all
run
[Switching to process 86746 local thread 0x2d03]
Running…
```

Ce qui peut se traduire par:

```
[Session commencée le 03/06/2009 à 15:48:02 +1000.]
Chargement du programme dans le débogueur ...
GNU gdb 6.3.50-20050815 (version gdb-956 d'Apple) (Mer 30 Avril 05:08:47 UTC 2008)
```

```
Copyright 2004 Free Software Foundation, Inc
GDB est un logiciel libre, couvert par la Licence publique
générale GNU, et vous êtes invité à en modifier et / ou à en
distribuer des exemplaires sous certaines conditions.
Taper "show copying" pour voir les conditions.
Il n'y a absolument aucune garantie concernant GDB. Taper
"show warranty" pour plus de détails.
Ce GDB a été configuré en tant que "i386-apple-darwin".tty /
dev/ttys000
Programme chargé.
bibliothèque partagée apply-load-rules complète
Exécuter
[Bascul sur le processus 86746 fil local 0x2d03]
Exécution…
```

Ceci nous montre que notre application a été compilée et lancée, et que le débogueur s'est chargé dedans.
La fenêtre Debugger (Débogueur) devrait ressembler à cela:



Le débogueur de Xcode vous permet d'exécuter le programme pas à pas et d'examiner les variables.

Le programme s'exécutera jusqu'à ce qu'il atteigne le premier point d'arrêt. Si vous examinez le panneau supérieur droit de la fenêtre, vous pourrez voir les valeurs des différentes variables. Toutes les valeurs définies ou modifiées depuis le dernier point d'arrêt sont affichées en rouge. Pour poursuivre l'exécution, utilisez le bouton Continue / Continuer. Le débogueur est un puissant outil. Jouez dessus un moment afin de vous familiariser avec.

Conclusion

Nous possédons maintenant tout ce qu'il faut pour écrire, déboguer et exécuter de simples programmes pour Mac OS X.
Si vous ne souhaitez pas créer de programmes avec interface utilisateur graphique, la seule chose qu'il vous reste à faire est d'accroître vos connaissances en Objectif-C afin de vous permettre de développer des programmes non-graphiques plus sophistiqués. C'est

précisément ce que nous ferons dans les prochains chapitres. Après cela, nous plongerons dans les applications avec GUI. Lisez la suite!

06: Déclarations Conditionnelles

if() (si)

De temps en temps, vous voudrez que votre code exécute une série d'actions seulement si une condition particulière est remplie. On fournit des mots-clés spéciaux pour cet réaliser [1.2].

```
//[1]
// age is an integer variable that stores the user's age
if (age > 30)    // The > symbol means "greater than"
{
    NSLog(@"age is older than thirty.");  //[1.4]
}
NSLog(@"Finished."); //[1.6
```

La ligne [1.2], L'utilisation de l'instruction `if()`, aussi connue comme une instruction conditionnelle. Où Vous reconnaîtrez les accolades des bouclées, qui contiendront tout le code à exécuter, qui sera fourni par l'expression logique entre des parenthèses si elle est égale à vrai. Ici, si l'âge de condition > 30 est remplie alors la chaine de caractère [1.4] sera imprimée. Si la condition est rencontrée ou pas, la chaine de caractère de ligne [1.6] sera imprimée, parce qu'il est à l'extérieur des accolades bouclées du `if()`.

if() else() (si, sinon)

Nous pouvons aussi fournir un jeu alternatif d'instructions si la condition n'est pas rencontrée, utilisant un if ... else à la déclaration [2].

```
//[2]
// age is an integer variable that stores the user's age
if (age > 30)
{
    NSLog(@"age is older than thirty.");  //[2.4]
}
else
{
    NSLog(@"age is younger than thirty.");  //[2.7]
}
NSLog(@"Finished.");
```

Comparaisons

A part le signe plus grand que lors de la déclaration [2.2], les opérateurs de comparaison suivants pour des nombres est à votre disposition.

```
==   equal to
>    greater than
<    less than
>=   greater than or equal to
<=   less than or equal to
!=    not equal to
```

Prenez en note, la particularité de l'opérateur d'égalité - il a deux signes égale. Il est facile de faire une erreur en utilisation un simple signe égal. Malheureusement c'est l'opérateur d'attribution qui fixera la variable à une valeur particulière. C'est une cause commune de confusion et d'eurreur de code , pour des débutants. Dites-le

maintenant à haute voix : je n'oublierai pas d'utiliser deux signes égaux en évaluant l'égalité!

Les opérateurs de comparaison sont tout à fait utiles quand vous voulez répéter une série de déclarations plusieurs fois. Ce sera le sujet du chapitre suivant. D'abord, nous discuterons quelques autres aspects de si les déclarations qui peut entrer pratique.

Exercice

Jetons un œil sur l'exécution d'une comparaison. Une opération de comparaison aboutit à un de seulement deux résultats possibles : le résultat est vrai ou faux.

Dans l'Objectif-C, vrai et faux sont représenté comme 1 ou 0 respectivement. Il y a même un type de données spécial, nommé BOOL que vous pouvez utiliser pour représenter de telles valeurs. Pour dénoter "la vraie" valeur, vous pouvez écrire 1 ou OUI. Pour dénoter la valeur "fausse", vous pouvez écrire 0 ou NON.

```
//[3]
int x = 3;
BOOL y;
y = (x == 4);  // y will be 0.
```

Il est possible de vérifier pour plus d'une conditions. Si vous avez plus qu'une condition qui doive être remplie , l'utilisation un opérateur logique ET, représentée par deux esperluettes : &&. Si au moins une des conditions doive être remplie, l'utilisation un logique OU représentée par deux pipes : ||.

```
//[4]
if ( (age >= 18) && (age < 65) )
{
   NSLog(@"Probably has to work for a living.");
}
```

Il est aussi possible d'emboîter des déclarations conditionnelles. C'est simplement une question de mise une déclaration conditionnelle à l'intérieur des accolades d'une autre déclaration conditionnelle. D'abord la condition la plus éloignée sera évaluée, alors, s'il est rencontré, la déclaration suivante à l'intérieur, et cetera :

```
//[5]
if (age >= 18)
{
    if (age < 65)
    {
        NSLog(@"Probably has to work for a living.");
    }
}
```

07: Répétition des déclarations

Introduction

Dans tout le code dont nous avons parlé jusqu'à présent, chaque déclaration n'a été exécuté qu'une seule fois. On peut toujours reproduire le code dans des fonctions en les appelant à plusieurs reprises [1].

```
//[1]
NSLog(@"Julia est mon actrice préférée.");
NSLog(@"Julia est mon actrice préférée.");
NSLog(@"Julia est mon actrice préférée.");
```

Mais il faudra quand même répéter l'appel. Parfois, vous devrez exécuter une ou plusieurs des déclarations à plusieurs reprises. Comme tous les langages de programmation Objectif-C offre plusieurs moyens d'y parvenir.

for() (pour())

Si vous connaissez le nombre de fois que la déclaration (ou un groupe de déclarations) doit être répétée, vous pouvez le préciser en mettant ce nombre dans la déclaration for de l'exemple [2]. Le nombre doit être un entier, car vous ne pouvez répéter une opération 2,7 fois.

```
//[2]
int x;
for (x = 1; x <= 10; x++)
{
    NSLog(@"Julia est mon actrice préférée.");    //[2.4]
}
NSLog(@"La valeur de x est %d", x);
```

Dans l'exemple [2], la chaîne de caractères [2.4] sera affichée 10 fois. Tout d'abord, on attribue la valeur 1 à x. L'ordinateur évalue alors la condition de la formule que nous avons mise en place: x <= 10. Cette condition est remplie (puisque x est égal à 1), de sorte que la ou les déclaration(s) entre les accolades sont effectuées. Puis la valeur de x est incrémentée, ici de un, en raison de l'expression x++. Ensuite, la valeur résultante de x, maintenant 2, est comparée à 10. Comme elle est toujours inférieure et non égale à 10, les déclarations entre les accolades sont à nouveau exécutées. Dès que x vaut 11, la condition x <= 10 n'est plus remplie. La dernière déclaration [2.6] a été mise pour vous démontrer que x est égal à 11, et non 10, une fois la boucle terminée.

Parfois, vous devrez faire plus de choses qu'une simple incrémentation à l'aide de x++. Tout ce que vous avez à faire, c'est de la remplacer par l'expression dont vous avez besoin. L'exemple[3] suivant convertit des degrés Fahrenheit en degrés Celsius.

```
//[3]
float celsius, tempEnFahrenheit;
for (tempEnFahrenheit = 0; tempEnFahrenheit <= 200;
tempEnFahrenheit = tempEnFahrenheit + 20)
{
    celsius = (tempEnFahrenheit - 32.0) * 5.0 / 9.0;
    NSLog(@"%10.2f -> %10.2f", tempEnFahrenheit, celsius);
}
```

La sortie de ce programme est:

```
0.00 -> -17.78
20.00 -> -6.67
40.00 ->  4.44
```

```
60.00 -> 15.56
80.00 -> 26.67
100.00 -> 37.78
120.00 -> 48.89
140.00 -> 60.00
160.00 -> 71.11
180.00 -> 82.22
200.00 -> 93.33
```

while() (tant que())

Objectif-C dispose de deux autres moyens pour répéter une suite de déclarations:

```
while () { }
```

et

```
do {} while ()
```

Le premier est essentiellement identique à la boucle-for débattue ci-dessus. Il y a d'abord une évaluation de condition. Si le résultat de cette évaluation est faux, les déclarations de la boucle ne sont pas exécutées.

```
//[4]
int compteur = 1;
while (compteur <= 10)
{
    NSLog(@"Julia est mon actrice préférée.\n");
    compteur = compteur + 1;
}
NSLog(@"La valeur de compteur est %d", compteur);
```

Dans ce cas, la valeur de compteur est de 11, si vous en avez besoin plus tard dans votre programme!
Avec l'instruction `do {} while ()`, les déclarations entre les accolades sont exécutées au moins une fois.

```
//[5]
int compteur = 1;
do
{
    NSLog(@"Julia est mon actrice préférée.\n");
    compteur = compteur + 1;
}
while (compteur <= 10);
NSLog(@"La valeur de compteur est %d", compteur);
```

La valeur finale de compteur est de 11.

Vous avez maintenant acquis un peu plus de compétences en programmation, pour attaquer à un sujet difficile. Dans le prochain chapitre, nous allons construire notre premier programme avec interface utilisateur graphique (GUI).

08: Un programme avec une interface graphique

Introduction

Ayant accru notre connaissance de l'Objectif-C, nous sommes prêts à discuter de la façon de créer un programme avec une interface utilisateur graphique (GUI). Je dois ici avouer une chose. Objectif-C est en fait une extension d'un langage de programmation appelé C. La plupart de ce dont nous avons parlé jusqu'ici n'est purement que du C. En quoi Objectif-C diffère t-il du simple C? Cela se situe au niveau de la partie "Objective". Objectif-C traite des notions abstraites appelées objets.

Jusqu'à maintenant, nous avons surtout traité des nombres. Comme vous l'avez appris, Objectif-C prend naturellement en charge la notion de nombre. C'est à dire qu'il vous autorise à créer des nombres en mémoire et à les manipuler à l'aide d'opérateurs et de fonctions mathématiques. C'est très bien quand votre application traite des nombres (par exemple une calculatrice). Mais qu'en est-il si votre application est, disons, un juke-box traitant de choses comme des chansons, des listes de lecture, des artistes etc? Ou si votre application est un système de contrôle de trafic aérien qui s'occupe d'avions, de vols, d'aéroports etc? Ne serait-il pas appréciable de pouvoir manipuler de telles choses avec Objectif-C, aussi facilement que vous manipulez les nombres?

C'est là que les objets interviennent. Avec Objectif-C, vous pouvez définir le type d'objets que vous souhaitez traiter, puis écrire des applications qui les manipulent.

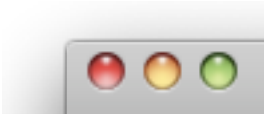
Objets en action

A titre d'exemple, examinons la façon dont les fenêtres sont traitées dans un programme écrit en Objectif-C, tel Safari. Examinez une

fenêtre Safari sur votre Mac. En haut à gauche, il y a trois boutons. Le rouge est le bouton de fermeture. Que se passe t-il si vous fermez une fenêtre en cliquant sur ce bouton rouge? Un message est envoyé à la fenêtre. En réponse à ce message, la fenêtre exécute du code afin de se fermer elle-même.

Un message de fermeture est envoyé à la fenêtre

La fenêtre est un objet. Vous pouvez la déplacer. Les trois boutons sont des objets. Vous pouvez cliquer dessus. Ces objets ont une représentation visuelle à l'écran, mais ce n'est pas vrai pour tous les objets. Par exemple, l'objet qui représente une connexion entre Safari et un site web donné n'a pas de représentation visuelle à l'écran.



Un objet (par exemple, la fenêtre) peut contenir d'autres objets (par exemple, les boutons)

Classes

Vous pouvez avoir autant de fenêtres Safari que vous le souhaitez. Pensez-vous que les programmeurs d'Apple:

- a. ont programmé à l'avance chacune de ces fenêtres, en utilisant leurs énormes ressources intellectuelles pour anticiper le nombre de fenêtres que vous pourrez vouloir, ou
- b. fait une sorte de modèle et laissent Safari créer des objets fenêtre à partir de lui à la volée?

Bien évidemment, la réponse est b. Ils ont créé du code, appelé classe, qui définit ce qu'est une fenêtre, à quoi elle devra ressembler et comment elle devra se comporter. Lorsque vous créez une nouvelle fenêtre, c'est en fait cette classe qui crée la fenêtre pour vous. Cette classe représente le concept d'une fenêtre, et toute fenêtre donnée est en fait une instance de ce concept (de la même façon que 76 est une instance de la notion d'un nombre).

Note du traducteur : En programmation orientée objet, une instance d'une classe est un objet avec un comportement correspondant à cette classe et un état initial.

Variables d'instance

La fenêtre que vous avez créée est présente à un certain endroit de l'écran de votre Mac. Si vous placez la fenêtre dans le Dock, et la faites réapparaître, elle reprendra exactement la même position à l'écran qu'elle avait avant. Comment cela marche t-il? La classe définit les variables appropriées pour se souvenir de la position de la fenêtre à l'écran. L'instance de la classe, c'est-à-dire l'objet, contient les valeurs de ces variables. Ainsi, chaque objet fenêtre contient les valeurs de certaines variables, et différents objets fenêtre contiendront en général des valeurs différentes pour ces variables.

Méthodes

La classe, n'a pas seulement créé l'objet fenêtre, mais lui a aussi donné accès à une série d'actions qu'il peut effectuer. Une de ces

actions est de fermer. Lorsque vous cliquez sur le bouton "Fermer" de la fenêtre, le bouton envoie le message fermer à cet objet fenêtre. Les actions pouvant être réalisées par un objet sont appelées méthodes. Comme vous le verrez, elles ressemblent très étroitement aux fonctions, de sorte que vous n'aurez pas beaucoup de difficulté à apprendre comment les utiliser si vous nous avez suivi jusqu'ici.

Objets en mémoire

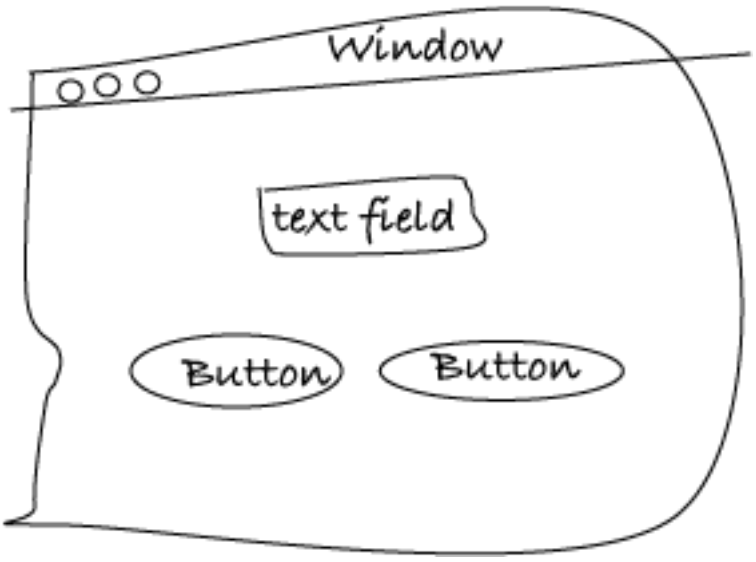
Lorsque la classe crée un objet fenêtre pour vous, il se réserve de la mémoire (RAM) pour y stocker la position de la fenêtre et d'autres informations. Toutefois, il ne fait pas une copie du code pour fermer la fenêtre. Ce serait un gaspillage de la mémoire puisque ce code est le même pour chaque fenêtre. Le code pour fermer une fenêtre ne doit être présent qu'une seule fois, mais chaque objet fenêtre a accès à ce code appartenant à la classe fenêtre.

Comme avant, le code que vous verrez dans ce chapitre contient des lignes pour réserver de la mémoire puis la restituer au système. Comme indiqué précédemment, nous ne débattrons de ce sujet avancé que beaucoup plus tard. Désolé.

Exercice

Notre application

Nous allons créer une application avec deux boutons et un champ de texte. Si vous pressez un bouton, une valeur est entrée dans le champ de texte. Si vous pressez l'autre bouton, une autre valeur est mise dans le champ de texte. Voyez cela comme une calculatrice à deux boutons qui ne peut faire de calculs. Bien sûr, une fois que vous en aurez appris davantage, vous pourrez comprendre comment créer une vraie calculatrice, mais j'aime les petites étapes.



Un croquis de l'application que nous voulons créer

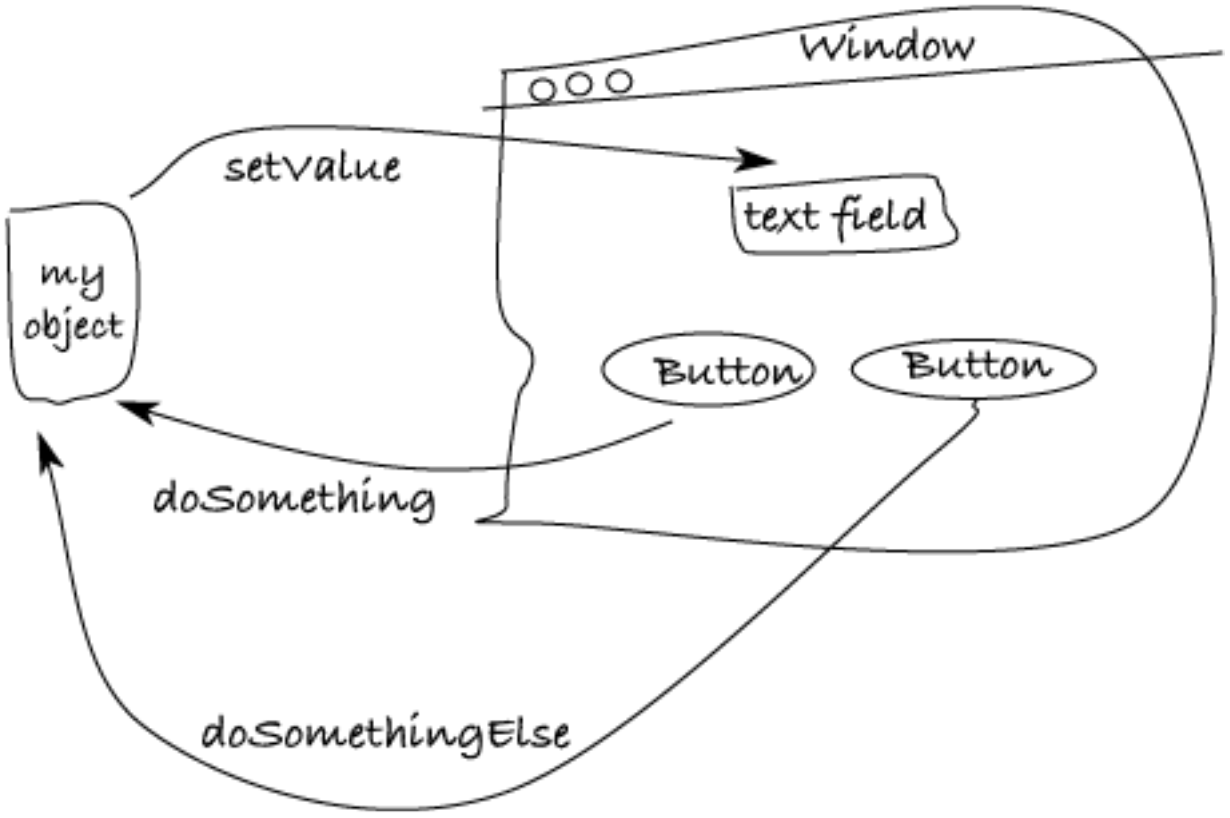
Si l'un des boutons de notre application est pressé, il enverra un message. Ce message contient le nom d'une méthode à exécuter. Bien, ce message est envoyé à quoi? Dans le cas de la fenêtre, le message de fermeture sera envoyé à cet objet fenêtre, qui est une instance de la classe fenêtre. Ce dont nous avons besoin maintenant, c'est d'un objet qui soit capable de recevoir un message de chacun des deux boutons et puisse dire à l'objet champ de texte d'afficher une valeur.

Notre première classe

Nous devons donc créer notre propre classe, puis créer une instance de celle-ci. Cet objet sera le destinataire du message des boutons (référez-vous au croquis ci-dessous). Tout comme un objet fenêtre, notre instance est un objet, mais à la différence d'un objet fenêtre, nous ne pouvons pas voir notre instance à l'écran quand on exécute le programme. C'est juste quelque chose dans la mémoire du Mac.

Lorsque notre instance reçoit un message envoyé par l'un des (deux) boutons, la méthode appropriée est exécutée. Le code de cette méthode est stocké dans la classe (et non dans l'instance elle-même). Lors de l'exécution, cette méthode réinitialisera le texte dans l'objet champ de texte.

Comment la méthode dans notre propre classe sait-elle comment réinitialiser le texte dans un champ de texte? En fait, elle ne le sait pas. Mais le champ de texte lui-même sait comment réinitialiser son propre texte. Donc, nous envoyons un message à l'objet champ de texte, en lui demandant de ne faire que ça. Quel genre de message ce doit-être? Bien sûr, nous devons spécifier le nom du destinataire (c'est-à-dire l'objet champ de texte dans notre fenêtre). Nous devons aussi dire, dans le message, ce que nous voulons que le destinataire fasse. Nous spécifions cela à l'aide du nom de la méthode que le champ de texte devra exécuter à la réception du message. (Bien sûr, nous devons savoir quelles méthodes les champs de texte peuvent exécuter, et comment elles se nomment). Nous devons aussi indiquer à l'objet champ de texte quelle valeur afficher (en fonction du bouton cliqué). Donc, l'expression du message d'envoi, ne contient pas seulement le nom de l'objet et le nom de la méthode, mais aussi un argument (valeur) qui sera utilisé par la méthode de l'objet champ de texte.



Un croquis des échanges de message entre les objets dans notre application

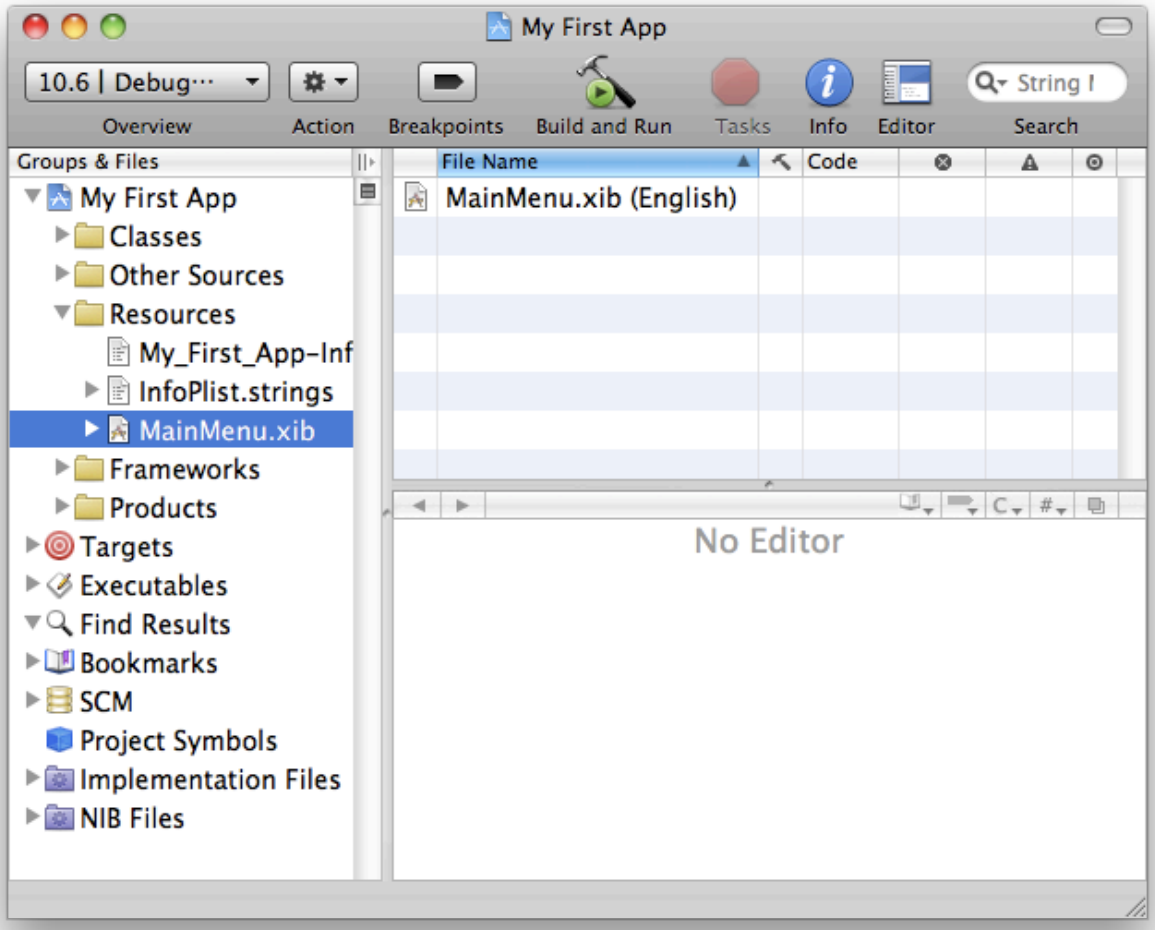
Voici le format global de la façon d'envoyer des messages en Objectif-C, à la fois sans [1.1] et avec [1-2] un argument:

```
//[1]
[destinataire message];
[destinataire messageAvecArgument:l'Argument];
```

Comme vous pouvez le voir, dans chacune de ces déclarations l'ensemble de ce que vous décrivez est placé entre crochets, et l'éternel point-virgule est présent en touche finale. Entre les crochets, l'objet destinataire est mentionné en premier, suivi du nom de l'une de ses méthodes. Si la méthode appelée exige une ou plusieurs valeurs, elles doivent aussi être fournies [1,2].

Création du projet

Voyons comment cela fonctionne réellement. Lancez Xcode pour créer un nouveau projet. Sélectionnez Cocoa Application dans la rubrique Application. Donnez un nom à votre projet, (par convention, le nom de votre application graphique devra commencer par une majuscule). Dans le cadre Groupes & Fichiers de la fenêtre Xcode qui apparaît, ouvrez le dossier Ressources. Double-cliquez sur MainMenu.xib.

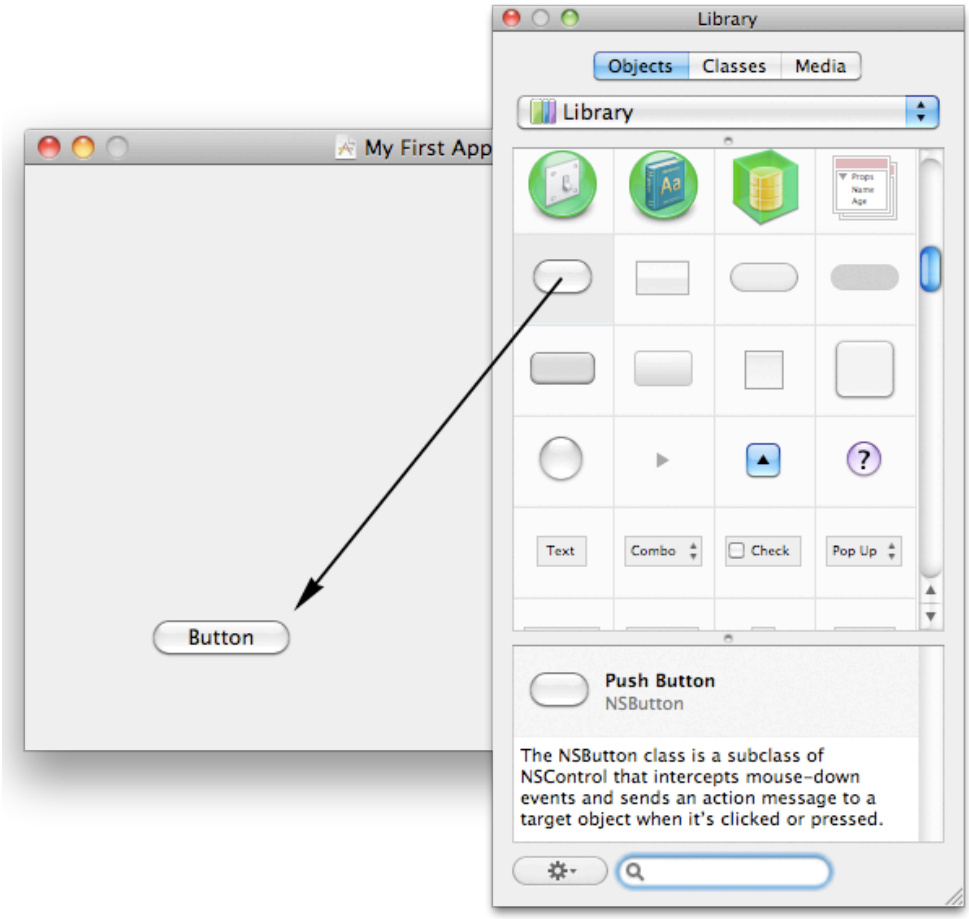


Double-clic sur le fichier MainMenu.xib dans Xcode

Avant Xcode 3, les fichiers **xib** s'appelaient les fichiers nib, pour NeXT Interface Builder. Si vous utilisez une ancienne version d'Xcode, vous pourrez donc voir un fichier MainMenu.nib. Ce détail n'est pas important; ils sont identiques en tout point de vue.

Création de l'interface graphique

Un autre programme, Interface Builder, se lancera. Comme beaucoup de fenêtres apparaissent, vous pourrez choisir de Masquer les autres (Hide Others) dans le menu Interface Builder. Vous verrez trois fenêtres. Celle nommée "Window" est la fenêtre que verront les utilisateurs de votre application. Elle est un peu grande, donc vous pourrez la redimensionner. À la droite de la fenêtre "Fenêtre", il y a une fenêtre intitulée "Library (Bibliothèque)". C'est une sorte de dépositoire de tous les types d'objets que vous pouvez avoir dans votre interface graphique, connu aussi sous le nom de "Library palette (palette de Bibliothèque)". Sélectionnez l'item "Views & Cells" en haut de la liste de cette fenêtre et glissez deux boutons, un par un, dans la fenêtre "Fenêtre" de l'interface graphique. De même, glissez un Label de texte ayant le texte "Label" dans la fenêtre de l'interface graphique.



Glissage d'objets d'interface graphique dans la fenêtre de votre application depuis la fenêtre de palettes.

En coulisses, l'action de glisser un bouton de la fenêtre de palettes dans la fenêtre de votre application crée un nouvel objet bouton et le met dans votre fenêtre. Il en va de même pour le champ de texte et de tout autre objet que vous pourriez glisser dans votre fenêtre depuis la fenêtre de palettes.

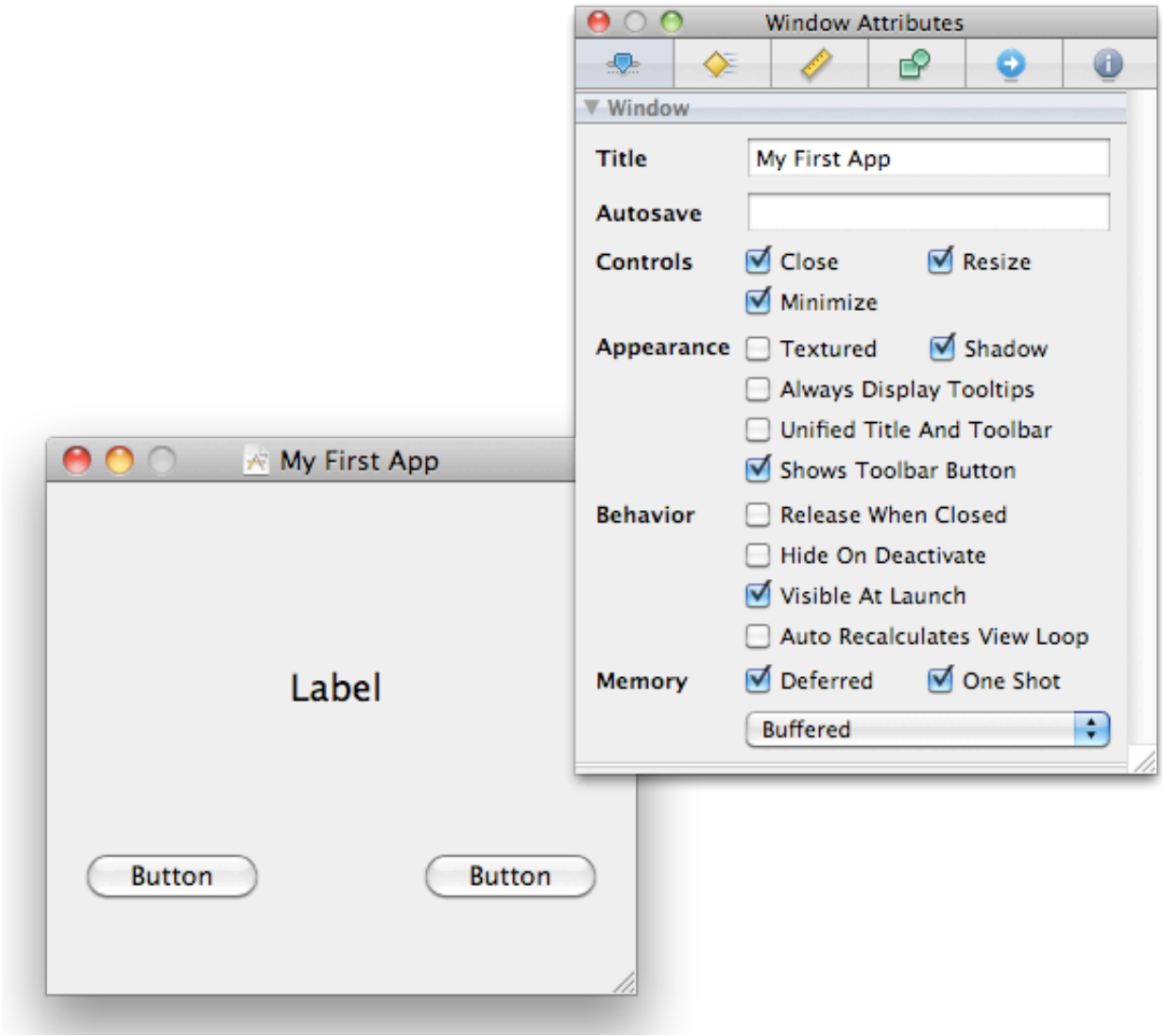
Notez que si vous maintenez votre curseur sur une icône de la fenêtre de palettes, un nom s'affichera, comme `NSButton` ou `NSTextView`. Ce sont les noms de classes fournies par Apple. Plus loin dans ce chapitre, nous verrons comment nous pouvons trouver les méthodes de ces classes, dont nous avons besoin pour réaliser les actions nécessaires de notre programme.

N'oubliez pas de sauvegarder régulièrement votre nib, afin qu'Xcode et Interface Builder puissent rester synchrones.

Disposez correctement les objets glissés dans la fenêtre "Fenêtre". Redimensionnez-les comme bon vous semble. Modifiez le texte des objets bouton en double-cliquant dessus, un par un. Je vous invite à explorer la fenêtre de palettes une fois ce projet fini afin de maîtriser la façon d'ajouter d'autres objets dans votre fenêtre.

Exploration d'Interface Builder

Pour modifier les propriétés d'un objet, sélectionnez-le et appuyez simultanément sur Pomme-Maj-I. Explorez cela aussi. Par exemple, sélectionnez la fenêtre "Fenêtre" (vous pouvez voir qu'elle est sélectionnée dans la fenêtre xib) et appuyez sur Pomme-Maj-I. Si la barre de titre du haut est Window Attributes (Attributs de fenêtre), vous pouvez cocher la case Textured (Texturé), ce qui donne un aspect métal à votre fenêtre. Vous constaterez que vous pouvez, dans une large mesure, personnaliser l'aspect de votre application sans écrire une seule ligne de code!



Notre fenêtre dans Interface Builder, avec son inspecteur

Contexte de Classe

Comme promis ci-dessus, nous allons créer une classe. Mais avant de le faire, voyons un peu plus en profondeur la façon dont les classes fonctionnent.

Pour économiser un gros effort de programmation, ce serait bien si vous pouviez construire à partir de ce que d'autres ont déjà construit, au lieu d'écrire tout en partant de zéro. Si, par exemple, vous souhaitez créer une fenêtre avec des propriétés particulières, vous n'auriez qu'à ajouter le code de ces propriétés. Vous n'auriez pas besoin d'écrire du code pour tous les autres comportements, tels la réduction ou la fermeture d'une fenêtre. En construisant à partir de ce

que d'autres programmeurs ont fait, vous héritez gratuitement de tous ces types de comportement. Et c'est ce qui rend Objectif-C si différent du simple C.

Comment faire cela? Eh bien, il y a une classe fenêtre (NSWindow), et vous pouvez écrire une classe qui hérite de cette classe. Supposons que vous ajoutiez des comportements à votre propre classe fenêtre. Que se passe t-il si votre fenêtre spéciale reçoit un message "fermer"? Vous n'avez écrit aucun code pour cela, ni copié de tel code dans votre classe. C'est simple; si la classe de la fenêtre spéciale ne contient pas le code d'une méthode donnée, le message est automatiquement transféré à la classe dont la classe fenêtre spéciale hérite (sa "superclasse"). Et si nécessaire, cela continue jusqu'à ce que la méthode soit trouvée (ou l'atteinte du sommet de la hiérarchie de l'héritage).

Si la méthode ne peut être trouvée, vous avez envoyé un message qui ne peut être traité. C'est comme demander à un garage de changer les pneus de votre traîneau. Même le patron du garage ne peut vous aider. Dans de tels cas, Objectif-C, signalera une erreur.

Classes personnelles

Que faire si vous voulez mettre en place votre propre comportement pour une méthode déjà héritée de votre superclasse? C'est facile, vous pouvez supplanter des méthodes particulières. Par exemple, vous pouvez écrire du code qui, en cliquant sur le bouton de fermeture, fera disparaître la fenêtre avant de la fermer. Votre classe fenêtre spéciale utilisera, pour la fermeture d'une fenêtre, le même nom de méthode que celui défini par Apple. Ainsi, lorsque votre fenêtre spéciale reçoit un message de proximité, la méthode d'exécution est la vôtre, et non celle d'Apple. Donc, maintenant, la fenêtre disparaîtra avant d'être réellement fermée.

Hé, fermer réellement une fenêtre a déjà été programmé par Apple. De l'intérieur de notre propre méthode de fermeture, on peut toujours invoquer la méthode de fermeture mise en œuvre par notre super-

classe, quoique cela requiert un appel légèrement différent pour s'assurer que notre propre méthode de fermeture n'est pas appelée récursivement.

```
//[2]
// Ici le code pour faire disparaître la fenêtre.
[super fermeture]; // Utiliser la méthode de fermeture de la superclasse.
```

Tout cela est bien trop avancé pour ce livret d'introduction et nous ne nous attendons pas à vous le faire "capter" avec ces quelques lignes.

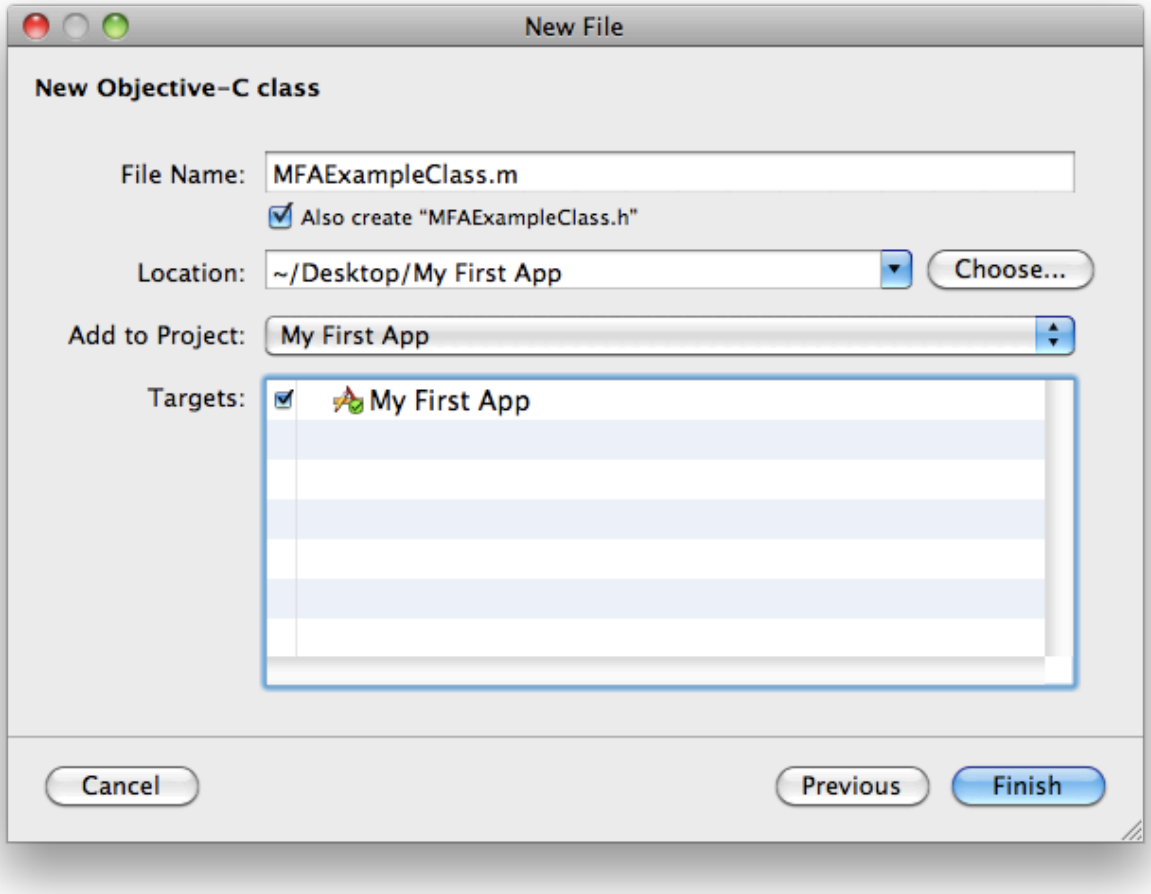
Une classe pour les gouverner toutes

La classe intitulée `NSObject` est la reine des classes. Pratiquement toutes les classes que vous créerez ou utiliserez seront des sous-classes de `NSObject`, directement ou indirectement. Par exemple, la classe NSWindow est une sous-classe de la classe `NSResponder`, qui est elle-même une sous-classe de `NSObject`. La classe `NSObject` définit les méthodes communes à tous les objets (par exemple, générer une description textuelle de l'objet, demander à l'objet s'il est capable de comprendre un message, etc.).

Avant de vous ennuyer avec trop de théorie, voyons comment créer une classe.

Création de notre classe

Allez dans votre projet Xcode et sélectionnez New File (Nouveau fichier) dans le menu File (Fichier). Choisissez une classe Objectif-C dans la liste, puis cliquez sur Suivant. J'ai nommé la mienne "MFAExampleClass". Cliquez sur Finish (Terminer).



Création de la classe MFAExampleClass

Les deux premières majuscules de MFAExampleClass signifient Mon Application. Vous pouvez inventer tous les noms de classes que vous voulez. Une fois que vous commencez à écrire vos propres applications, nous vous recommandons de choisir un préfixe de deux ou trois lettres, que vous utiliserez pour toutes vos classes afin d'éviter toute confusion avec des noms de classes existantes. Cependant, ne pas utiliser NS, car cela pourrait vous embrouiller plus tard. NS est utilisé pour les classes d'Apple. Il signifie NeXTStep, NeXTstep étant le système d'exploitation sur lequel Mac OS X était

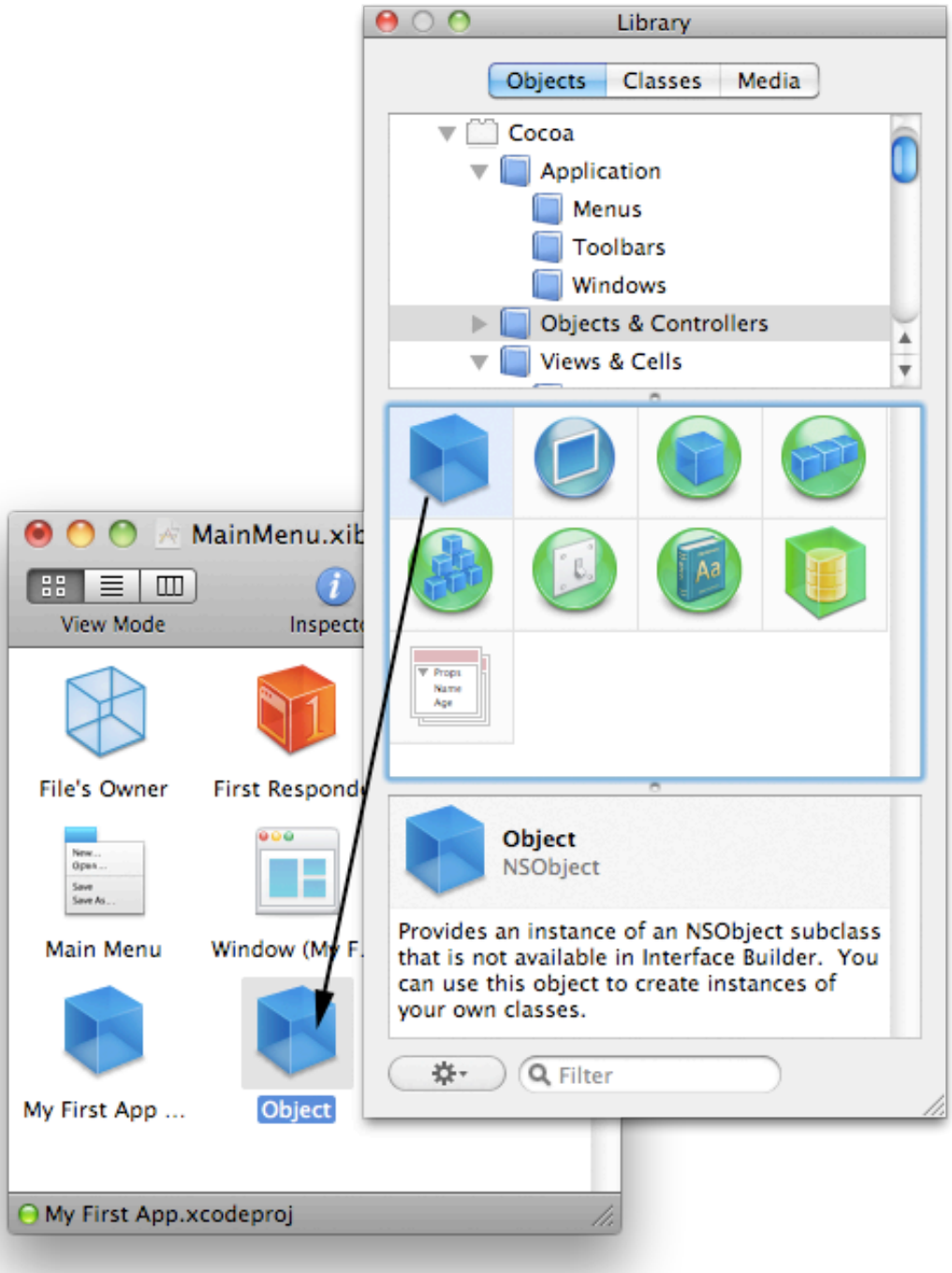
basé lorsqu'Apple a acheté NeXT, Inc., avec le retour de Steve Jobs en bonus.

Le wiki de CocoaDev contient une liste d'autres préfixes à éviter. Vous devriez le consulter au moment de choisir votre propre préfixe: http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix

Lors de la création d'une nouvelle classe, pensez à lui donner un nom exprimant des informations utiles sur cette classe. Par exemple, nous avons déjà vu que, dans Cocoa, la classe utilisée pour représenter les fenêtres se nomme NSWindow. Un autre exemple est la classe utilisée pour représenter les couleurs, qui se nomme NSColor. Dans notre cas, la classe MFAExampleClass que nous créons n'est là que pour illustrer la façon dont les objets communiquent ensemble dans une application. C'est pourquoi nous lui avons donné un nom générique, sans signification particulière.

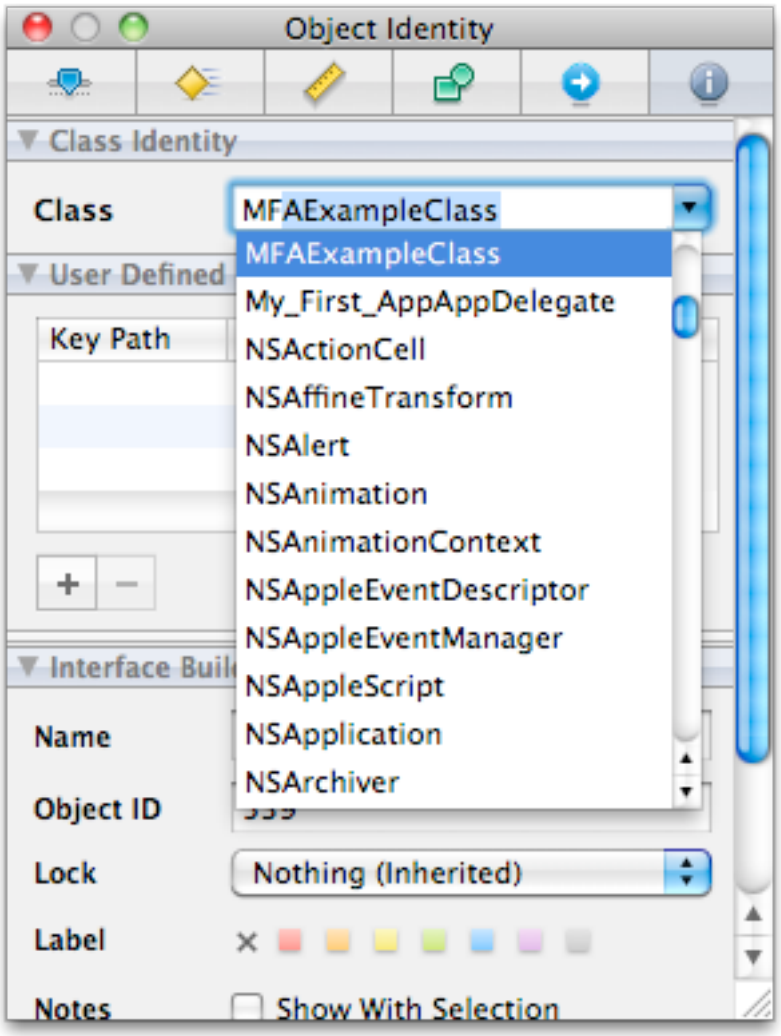
Création d'une instance dans Interface Builder

Retournez dans Interface Builder, allez à la palette Library (bibliothèque) et choisissez Objects & Controllers (Objets et Contrôleurs) dans le menu du haut (Cocoa). Puis glissez un Objet (blue cube / cube bleu) de la palette dans la classe MainMenu.xib.



Instanciation d'un nouvel objet

Ensuite, sélectionnez le bouton identité dans la palette Inspecteur (Pomme-6), puis choisissez MFAExampleClass dans le menu déroulant Classe. Nous avons maintenant instancié notre classe MFAExampleClass de Xcode en un objet de notre fichier xib. Cela permettra à notre code et notre interface de communiquer.



Réglage de l'identité de l'instance de notre objet

Création des connexions

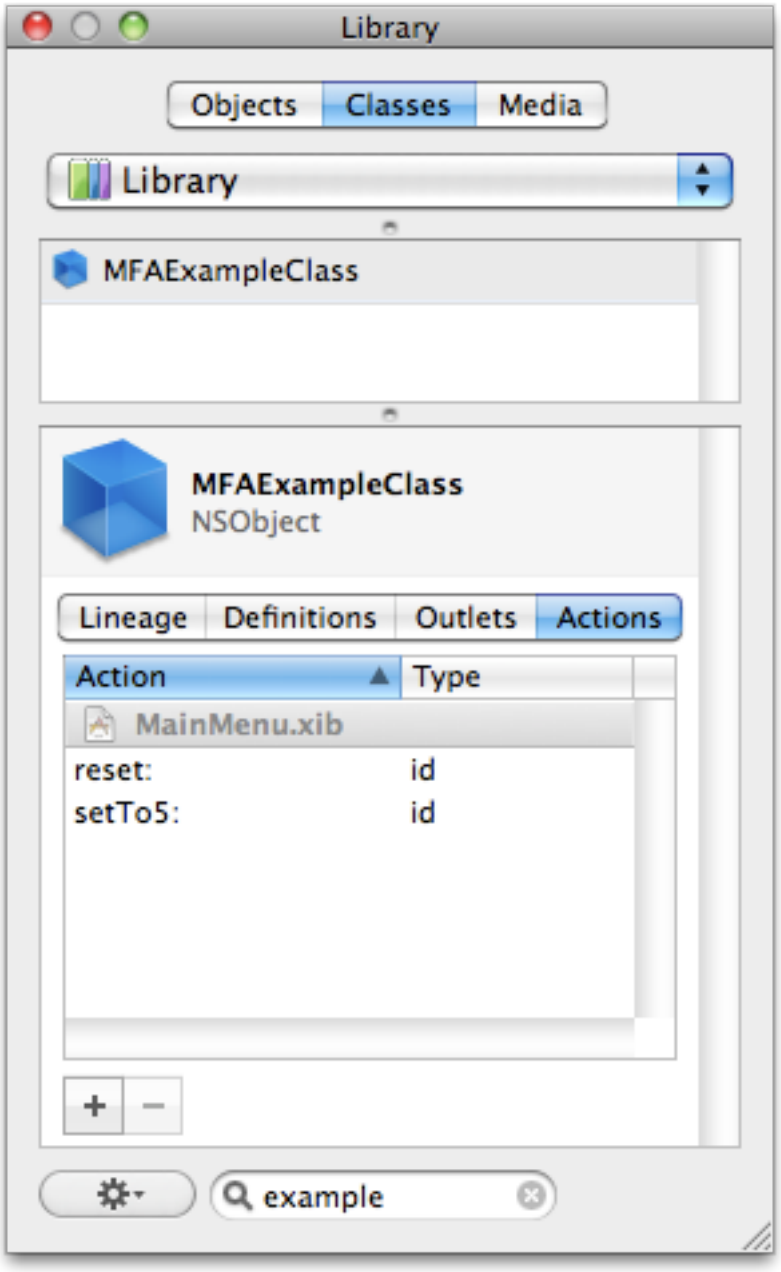
Notre prochaine étape est de créer des connexions entre les boutons (à partir desquels les messages sont envoyés) et notre objet MFAExampleClass. De plus, nous allons créer une connexion en retour de l'objet MFAExampleClass au champ de texte, car un message sera envoyé à l'objet champ de texte. Un objet n'a pas moyen d'envoyer un message à un autre objet s'il ne dispose pas d'une référence pour l'autre objet. En fabriquant une connexion entre un bouton et notre objet MFAExampleClass, nous fournissons à ce bouton une référence à notre objet MFAExampleClass. A 'aide de cette référence, le bouton sera en mesure d'envoyer des messages à notre objet MFAExampleClass. De même, établir une connexion de notre objet au champ de texte permettra au premier d'informer le dernier.

Examinons de nouveau ce que la application doit faire. Chacun des boutons peut envoyer, une fois cliqué, un message correspondant à une action donnée. Ce message contient le nom de la méthode de la classe MFAExampleClass qui doit être exécutée. Le message est envoyé à l'instance de la classe MFAExampleClass que nous venons tout juste de créer, l'objet MFAExampleClass. (Rappel: les instances d'objet elles-mêmes ne contiennent pas le code pour réaliser l'action, mais les classes oui). Alors, ce message envoyé à l'objet MFAExampleClass déclenche une méthode de la classe MFAExampleClass pour faire quelque chose: dans ce cas, envoyer un message à l'objet champ de texte. Comme chaque message, celui-ci se compose du nom d'une méthode (que l'objet champ de texte devra exécuter). Dans ce cas, la méthode de l'objet champ de texte a pour mission d'afficher une valeur, et cette valeur doit être envoyée dans le cadre du message (appelé un "argument", vous vous rappelez?), ainsi que le nom de la méthode à invoquer sur le champ de texte.

Notre classe nécessite deux actions (méthodes), qui seront appelées par les (deux) objets bouton. Notre classe nécessite un outlet (point de connexion), une variable pour se souvenir à quel objet (ici, l'objet champ de texte) doit être envoyé un message.

Assurez-vous que MFAExampleClass est sélectionné dans la fenêtre MainMenu.xib. Sur votre clavier, appuyez sur Pomme-6 afin d'afficher à l'écran l'inspecteur d'identité. Dans la fenêtre Inspecteur, à la section Action, cliquez sur le bouton Ajouter (+) pour ajouter une action (c'est-à-dire, une méthode d'action) à la classe MFAExampleClass. Remplacer le nom donné par défaut par Interface Builder par un nom plus significatif (par exemple, vous pouvez taper "mettreA5:" parce que nous allons programmer cette méthode pour afficher le chiffre 5 dans le champ de texte). Ajouter une autre méthode, et lui donner un nom (par exemple "réinitialiser:", parce que nous allons la programmer pour afficher le chiffre 0 dans le champ de texte). Notez que nos deux noms de méthode se terminent par un deux-points (":"). En en dira plus là-dessus plus loin.

Maintenant, dans la fenêtre Inspecteur, sélectionnez l'onglet Outlet (point de connexion), ajoutez un Outlet et donnez-lui un nom, par exemple "textField" (ChampTexte).



Ajout de méthodes d'action et d'Outlet à la classe MFAExampleClass

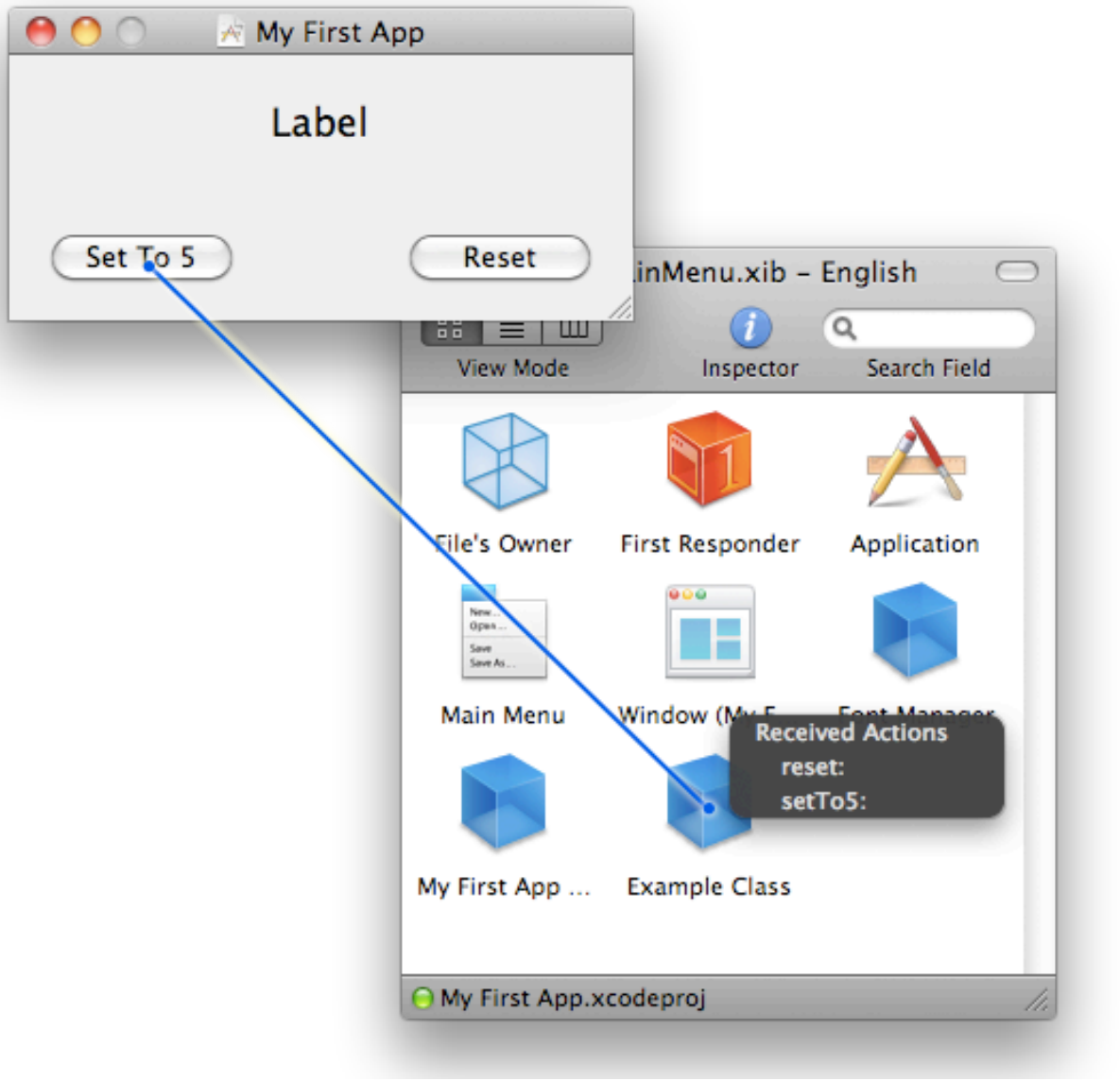
Avant d'établir des connexions entre les objets, nous allons donner des noms significatifs à nos deux boutons. Puisque le premier va demander à notre instance MFAExampleClass d'afficher le chiffre 5 dans le champ de texte, nous le nommons «Mettre à 5" (nous avons déjà appris comment changer le nom d'un bouton: double-cliquez sur son nom à l'écran , puis tapez le nouveau nom). De même, nous

nommons le deuxième "Réinitialiser". Notez que cette étape consistant à donner un nom particulier à ce bouton n'est pas nécessaire au bon fonctionnement de notre programme. C'est juste que nous voulons que notre interface utilisateur soit aussi descriptive que possible pour l'utilisateur final.

Maintenant, nous sommes prêts à créer les vraies connexions entre

- 1. le bouton "Réinitialiser" et l'instance MFAExampleClass
- 2. le bouton "Mettre à 5" et l'instance MFAExampleClass
- 3. l'instance MFAExampleClass et le champ de texte.

Pour créer les connexions, appuyez sur la touche Contrôle de votre clavier et utilisez la souris pour glisser le bouton "Mettre à 5" dans l'instance MFAExampleClass de la fenêtre MainMenu.xib (ne pas faire le contraire!). Une ligne représentant la connexion apparaît à l'écran, et un menu déroulant apparaîtra sur l'icône d'instance de l'objet. Choisissez "mettreA5:" dans la liste.



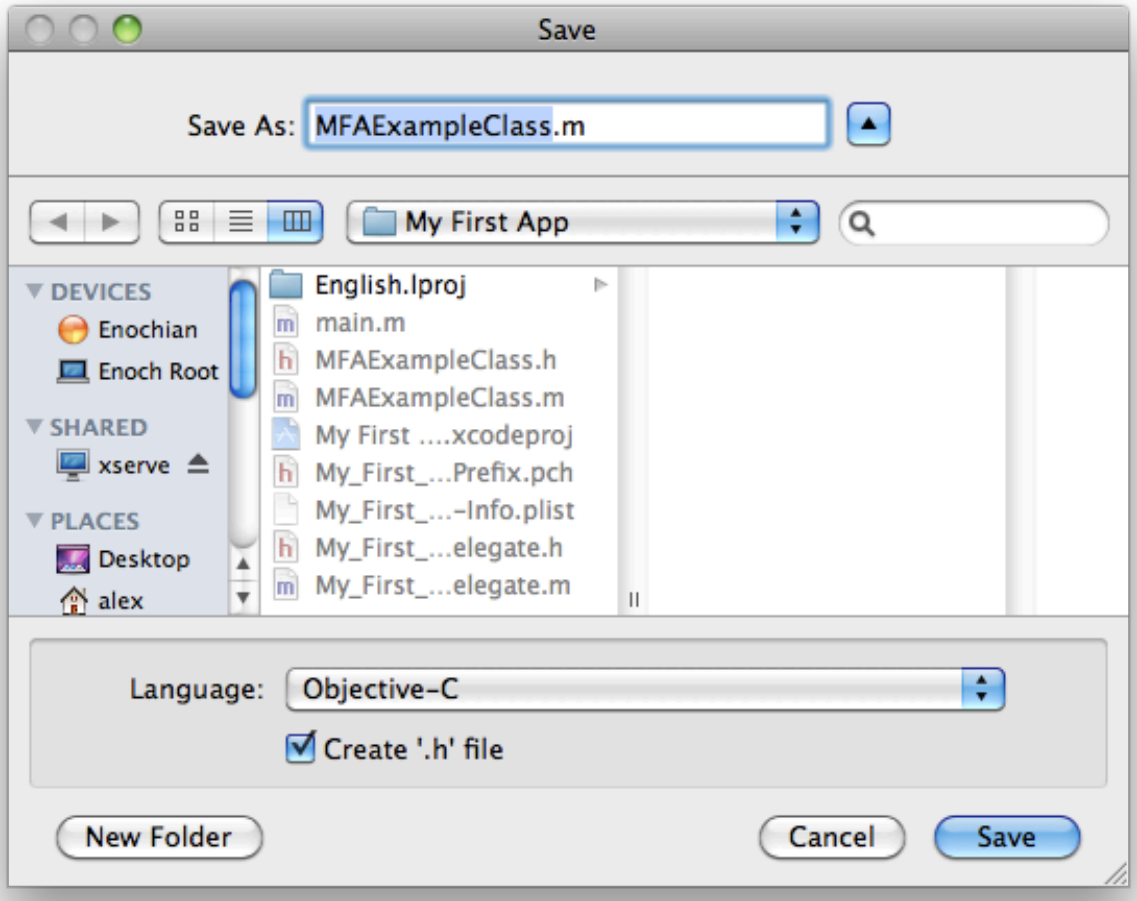
Etablissement de la connexion

Maintenant, le bouton détient une référence à notre objet MFAExampleClass, et lui enverra le message mettreA5 chaque fois qu'il sera pressé.
Vous pouvez maintenant connecter le bouton "Réinitialiser" à l'objet MFAExampleClass en appliquant le même processus.
Pour créer la connexion entre l'objet MFAExampleClass et le champ de texte, partez de l'objet MFAExampleClass et contrôle-glissez-le dans l'objet champ de texte. Cliquez sur "textField" dans le menu pour assigner la connexion.

Qu'en est-il au final? Eh bien, comme vous pourrez le voir dans une minute, vous avez en fait créé du code sans écrire une seule ligne.

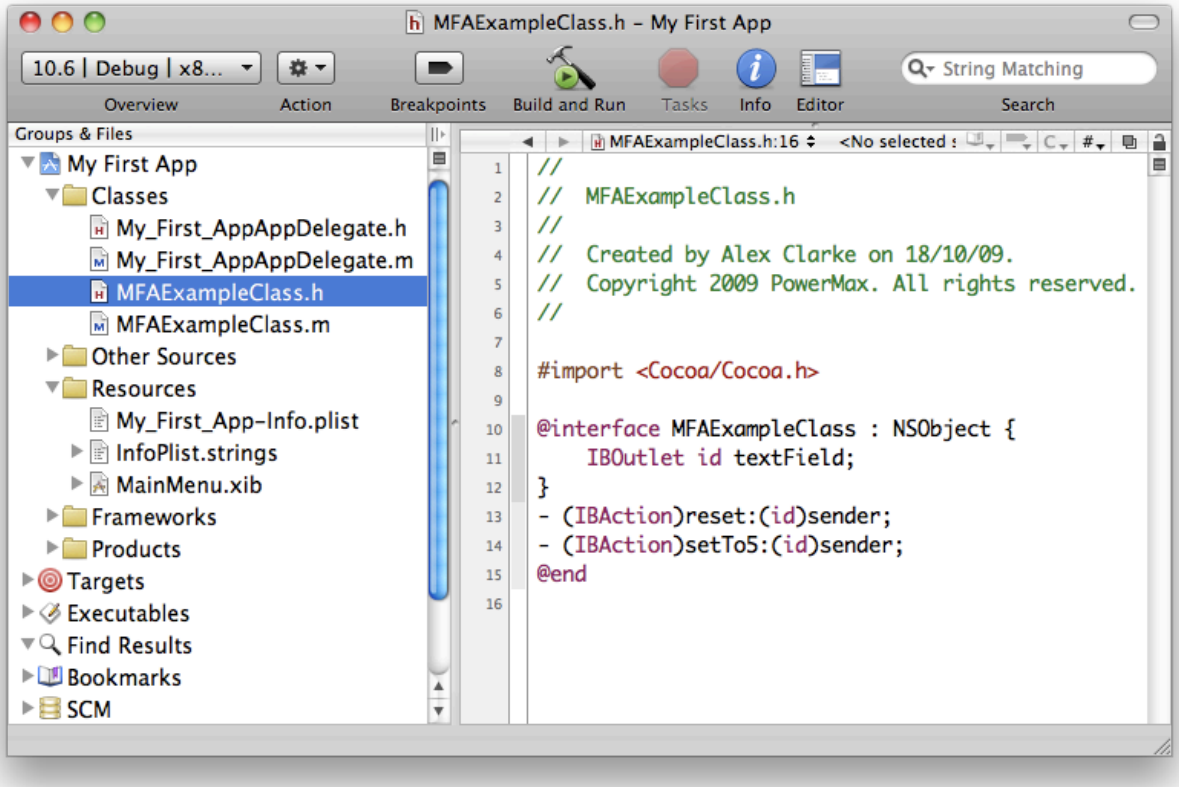
Générer le code

Tout d'abord, assurez-vous que MFAExampleClass est sélectionné dans la fenêtre MainMenu.xib. Aller dans le menu File (Fichier) d'Interface Builder et choisissez Write Class Files (Ecrire des fichiers classe). Interface Builder vous demande alors où mettre votre fichier généré sur le disque. Naviguez jusqu'au dossier projet de notre application et écraser la classe MFAExampleClass qui s'y trouve.



Maintenant, si vous revenez à Xcode, vous verrez les fichiers générés dans votre fenêtre de projet, dans le groupe Classes. Si elles apparaissent dans le groupe Ressources, ou un autre groupe, il suffit de les sélectionner, puis de les faire glisser dans le groupe Classes.

Cliquez sur le bouton Editor (Éditeur) de la barre d'outils, puis choisissez `MFAExampleClass.h`.



Les fichiers générés apparaissent dans notre projet Xcode

Revenons un instant au chapitre 3, où nous parlions de fonctions. Vous souvenez-vous de la fonction header [11.1]? C'était une sorte d'avertissement au compilateur pour lui indiquer à quoi il pouvait s'attendre. L'un des deux dossiers que nous venons de créer se nomme `MFAExampleClass.h`, et c'est un fichier header (d'en-tête): il contient des informations sur notre classe. Par exemple, vous reconnaîtrez qu'il y a une ligne [3,5] contenant NSObject, laquelle ligne signifie que notre classe hérite de la classe NSObject.

```
//[3]
/* MFAExampleClass */
#import <cocoa/cocoa.h>  // [3.2]
@interface MFAExampleClass : NSObject
{
    IBOutlet id textField;  // [3.7]
}
- (IBAction)réinitialiser:(id)expéditeur;
- (IBAction)mettreA5:(id)expéditeur;
@end
```

Vous verrez qu'il y a un outlet [3,7] pour l'objet champ de texte. id désigne l'objet. "IB" signifie Interface Builder, le programme que vous avez utilisés pour créer ce code.

IBAction [3.9, 3.10] est équivalent à void. Rien n'est retourné à l'objet qui envoie le message: les boutons de notre programme n'obtiennent pas de réponse de l'objet MFAExampleClass en réponse à leur message.

Vous pouvez également voir qu'il y a deux Actions Interface Builder. Ce sont deux méthodes de notre classe. Les méthodes sont très semblables aux fonctions, que nous connaissons déjà, mais il y a des différences. Plus de détails à ce sujet plus loin.

Précédemment, nous avons vu #import <Foundation/Foundation.h> au lieu de la ligne [3.2]. Le premier est pour les applications non-graphiques, le dernier pour les applications graphiques.

Maintenant, examinons le second fichier, MFAExampleClass.m. A nouveau, nous obtenons une grande quantité de code gratuitement.

```
//[4]
#import "MFAExampleClass.h"
@implementation MFAExampleClass
- (IBAction)réinitialiser:(id)expéditeur  // [4.5]
{
}
- (IBAction)mettreA5:(id)expéditeur
{
}
@end
```

Tout d'abord, le fichier d'en-tête `MFAExampleClass.h` est importé, de sorte que le compilateur sait à quoi s'attendre. Deux méthodes peuvent être reconnues: `réinitialiser:` et `mettreA5:`. Ce sont les méthodes de notre classe. Elles sont similaires aux fonctions en ce sens que vous devez placer votre code entre les accolades. Dans notre application, lorsqu'un bouton est pressé, il envoie un message à votre objet MFAExampleClass, en demandant l'exécution de l'une des méthodes. Nous n'avons aucun code à écrire pour cela. Faire les connexions entre les boutons et l'objet MFAExampleClass dans Interface Builder est tout ce qui est nécessaire. Toutefois, nous devons mettre en œuvre les deux méthodes, c'est-à-dire que nous devons écrire le code qui effectue leur fonction. Dans ce cas, ces méthodes ne font chacune qu'envoyer un message de notre objet MFAExampleClass à l'objet `textField`, donc nous fournissons les déclarations [5.7, 5.12].

```
//[5]
#import "MFAExampleClass.h"
@implementation MFAExampleClass
- (IBAction)réinitialiser:(id)expéditeur
{
    [textField setIntValue:0];  // [5.7]
}
- (IBAction)mettreA5:(id)expéditeur
{
    [textField setIntValue:5];  // [5.12]
}
@end
```

Comme vous pouvez le voir, nous envoyons un message à l'objet référencé par l'outlet `textField`. Puisque nous avons relié cet outlet au vrai champ de texte, à l'aide d'Interface Builder, notre message sera envoyé au bon objet. Le message est le nom d'une méthode, `setIntValue:`, ainsi qu'une valeur. La méthode `setIntValue:` est capable d'afficher une valeur entière dans un objet champ de texte. Dans le prochain chapitre, nous vous dirons comment nous avons découvert cette méthode.

Prêt à balancer

Vous êtes maintenant prêt à compiler votre application et à la lancer. Comme d'habitude, appuyez sur le bouton Build and Go de la barre d'outils Xcode. Xcode prendra quelques secondes pour construire l'application et la lancer. Finalement, l'application apparaîtra à l'écran et vous serez en mesure de la tester.

Notre application en exécution

En bref, vous venez de créer une (très basique) application, pour laquelle vous n'avez eu à écrire vous-même que deux lignes de code!

MFAExampleClass

09: Méthodes de recherche

Introduction

Au chapitre précédent, nous avons appris les méthodes. Nous avons écrit nous-mêmes (le corps de) deux méthodes, mais nous en avons également utilisé une fournie par Apple. setIntValue: était la méthode pour afficher la valeur d'un entier dans l'objet champ de texte. Comment avez-vous découvert les méthodes appropriées?

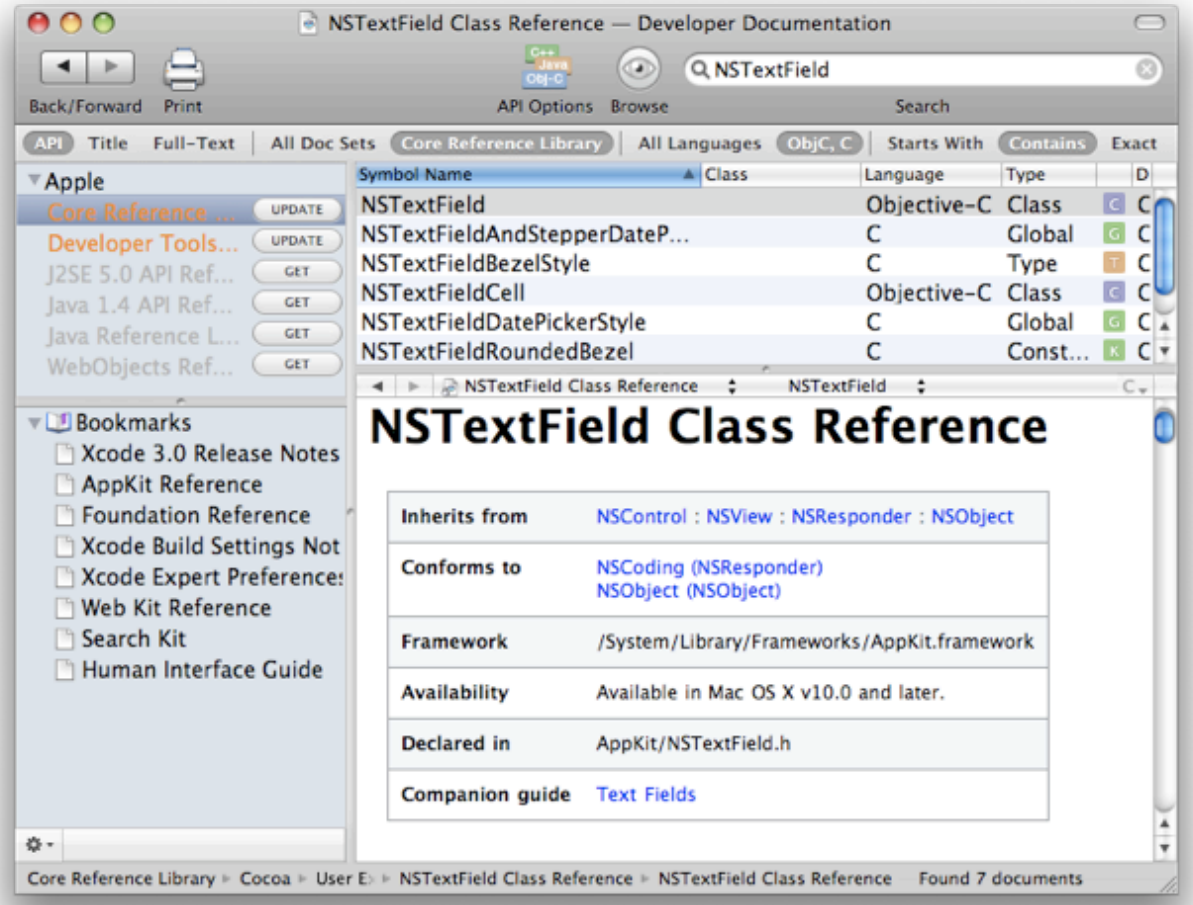
Souvenez-vous, pour toute méthode qui a été créée par Apple, vous n'avez aucun code à écrire vous-même. Mieux, il a plus de chance d'être exempt de bogue. Aussi, ça vaut toujours la peine de prendre le temps de vérifier si les méthodes appropriées sont disponibles avant de programmer les vôtres.

Exercice

Dans Interface Builder, si vous laissez le curseur sur un objet de la fenêtre de palettes, un petit label apparaît. Si vous laissez le curseur au-dessus de l'icône du bouton, vous verrez "NSButton". Si vous le laissez au-dessus du texte indiquant "System Font Text", vous verrez "NSTextField". Chacun de ces noms est un nom de classe. Examinons la classe `NSTextField` pour voir quelles méthodes sont appropriées.

Allez dans Xcode, et, dans le menu, sélectionnez Help (Aide) ? Documentation. Dans le cadre de gauche, sélectionnez Cocoa, puis tapez "NSTextField" dans le champ de recherche (assurez-vous que le mode de recherche API (Application Programming Interface = Interface de programmation) est sélectionné; voir la capture d'écran ci-dessous). Au fur et à mesure de votre frappe, la liste des résultats se réduit de façon significative et vous verrez bientôt apparaître NSTextField au sommet.

Cliquez sur la ligne `NSTextField` (du type Class) pour obtenir des informations sur la classe NSTextField, affichées dans le cadre du dessous.



Naviguer dans la documentation Cocoa avec Xcode

La première chose que vous devrez remarquer est que cette classe hérite d'une série d'autres classes. La dernière de la liste est celle qui commande, la reine NSObject. Un peu plus bas (faites défiler), il y a la rubrique:

Method Types (Types de méthodes)

C'est là que nous allons commencer notre recherche. Un rapide coup d'œil aux sous-rubriques nous apprendra que nous n'allons pas trouver ici la méthode dont nous avons besoin pour afficher une valeur dans l'objet champ de texte. En raison du principe de l'héritage, nous devons aller voir la super classe immédiate de la classe NSTextField, qui est NSControl (et si nous échouons, nous devons examiner minutieusement sa super classe NSView, etc.). Puisque toute la documentation est en HTML, tout ce que nous avons à faire est de cliquer sur le mot NSControl (dans la liste Inherits From (Hérite de),

comme indiqué ci-dessus). Ceci nous amène aux informations de la classe NSControl:

NSControl

Hérite de NSView : NSResponder : NSObject

Vous pouvez voir que nous nous sommes déplacé d'une classe au dessus (avant, on avait NSControl : NSView : NSResponder : NSObject). Dans la liste de méthode, on remarque une sous-rubrique (la 5°):

Setting the control's value (Régler la valeur de contrôle)

C'est ce que nous voulons, nous voulons définir une valeur. En dessous de cette sous-rubrique, on trouve:

- setIntValue:

Cela semble prometteur, aussi nous consultons la description de cette méthode en cliquant sur le lien setIntValue:.

setIntValue:

- (void)setIntValue:(int)unEnt

Définit la valeur de la cellule destinataire (ou cellule sélectionnée) à l'entier unEnt. Si la cellule est en cours d'édition, il avorte toute édition avant de définir la valeur; si la cellule n'hérite pas de NSActionCell, il marque l'intérieur de la cellule comme devant être ré-affichée (NSActionCell effectue sa propre mise à jour des cellules).

Dans notre application, notre objet NSTextField est le destinataire, et nous devons lui fournir un entier. Nous pouvons aussi voir ça depuis la signature de la méthode:

```
- (void)setIntValue:(int)unEnt
```

En Objectif-C, le signe moins, marque le début d'une déclaration de méthode d'instance (par opposition à une déclaration de méthode de classe, dont nous parlerons plus tard). void indique que rien n'est retourné à l'invoqueur de la méthode. C'est à dire que lorsque nous envoyons un message `setIntValue:` à `textField`, notre objet MAFoo ne reçoit pas une valeur en retour de l'objet champ de texte. On est d'accord. Après le deux-points, (int) indique que la variable unEnt doit être un entier. Dans notre exemple, nous envoyons une valeur de 5 ou 0, qui sont des entiers, donc ça va.

Parfois, il est un peu plus difficile de découvrir la méthode appropriée à utiliser. Vous vous améliorerez en vous familiarisant avec la documentation, donc entraînez-vous.

Que faire si vous voulez lire la valeur de notre objet champ de texte textField? Vous vous souvenez du point important concernant les fonctions, à savoir que toutes les variables à l'intérieur étaient protégées? Il en va de même pour les méthodes. Cependant, les objets ont souvent une paire de méthodes connexes, dite "Accessors (Accesseurs)", une pour lire la valeur, et une pour définir la valeur. Nous connaissons déjà la dernière, c'est la méthode `setIntValue:`. La première ressemble à ça:

```
//[1]
- (int) intValue
```

Comme vous pouvez le voir, cette méthode retourne un entier. Donc, si nous voulons lire la valeur de l'entier associée à notre objet textfield, nous devons de lui envoyer un message de ce genre:

```
//[2]
int resultReceived = [textField intValue];
```

Encore une fois, dans les fonctions (et les méthodes) tous les noms des variables sont protégés. C'est formidable pour les noms de variables, parce que vous n'avez pas à craindre que la définition d'une variable dans une partie de votre programme affecte une variable du même nom dans votre fonction. Toutefois, les noms de fonctions doivent toujours être uniques dans votre programme. Objectif-C va plus loin dans la protection: les noms de méthode doivent être uniques au sein d'une seule classe, mais différentes classes peuvent avoir des noms de méthode en commun. C'est une grande fonctionnalité pour les gros programmes, car les programmeurs peuvent écrire des classes indépendantes les unes des autres, sans avoir à craindre des conflits dans les noms de méthodes.

Mais il y a mieux. Le fait que différentes méthodes dans des classes différentes peuvent avoir le même nom est appelé Polymorphisme en terme branché (et en grec, bien sûr), et c'est l'une des choses qui rend la programmation orientée-objet si particulière. Elle vous permet d'écrire des morceaux de code, sans devoir connaître par avance quelles sont les classes des objets que vous manipulez. La seule chose nécessaire c'est que, au moment de l'exécution, les objets comprennent les messages que vous leur envoyez.

En utilisant le polymorphisme, vous pouvez écrire des applications qui sont flexibles et extensibles de par leur conception. Par exemple, dans l'application graphique que nous avons créée, si on remplace le champ de texte par un objet d'une autre classe qui est capable de comprendre le message `setIntValue:`, notre application marchera toujours sans nous demander de modifier notre code, ou même de le recompiler. Nous pouvons même modifier l'objet au moment de l'exécution sans rien casser. C'est là que réside la puissance de la programmation orientée objet.

10: awakeFromNib

Introduction

Apple en a fait beaucoup pour vous, ce qui facilite la création de vos programmes. Pour votre petite application, vous n'avez pas eu à vous soucier du dessin de la fenêtre et des boutons à l'écran, entre autres choses.

Le plus gros de ce travail est mis à disposition au travers de deux frameworks (Un framework est un ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications). Le framework Fondation Kit, que nous avons importé dans l'exemple [12] du chapitre 4, fournit la plupart des services non associés à une interface utilisateur graphique. L'autre framework, appelé Application Kit, traite des objets que vous voyez à l'écran et des mécanismes d'interaction-utilisateur. Les deux frameworks sont bien documentés.

Revenons à notre application graphique. Supposons que l'on veuille que notre application affiche une valeur donnée dans l'objet champ de texte dès que la fenêtre s'affiche, au lancement de l'application.

Exercice

Toutes les informations concernant la fenêtre sont stockées dans un fichier **nib** (nib signifie NeXT Interface Builder). C'est une bonne indication de ce que la méthode nous avons besoin peut faire partie d'Application Kit. Voyons comment obtenir des informations sur ce framework.

Dans Xcode, accédez au menu Aide (Help) et sélectionnez Documentation. Dans la fenêtre de documentation assurez-vous que Full-Text Search (Recherche tout-texte) est activé (pour ce faire, sélectionnez Full-Text Search dans le menu déroulant du champ de recherche). Puis tapez Application Kit dans le champ de recherche et appuyez sur la touche Retour.

Xcode vous donne de nombreux résultats. Parmi eux se trouve un document appelé Application Kit Framework Reference. A l'intérieur de celui-ci vous trouverez une liste des services fournis par ce framework. Sous la rubrique Protocols, il y a un lien appelé `NSNibAwaking`. Si vous cliquez dessus, vous accédez à la documentation de la classe `NSNibAwaking`.

NSNibAwaking Protocol Objective-C Reference (Référence en Objectif-C du protocole NSNibAwaking)

(protocole informel)

Framework	/System/Library/Frameworks/AppKit.framework
Declared in (Déclaré dans)	AppKit/NSNibLoading.h
Companion document (Document compagnon)	Loading Resources (Charger les ressources)

Protocol Description (Description du Protocole)

Ce protocole informel se compose d'une seule méthode, `awakeFromNib`. Les classes peuvent mettre en œuvre cette méthode pour initialiser les informations d'état après que des objets aient été chargés depuis une archive Interface Builder (fichier nib).

Si nous mettons en œuvre cette méthode, elle sera appelée lorsque notre objet sera chargé à partir de son fichier nib. Ainsi, nous pouvons l'utiliser pour atteindre notre objectif: afficher une valeur dans le champ de texte au lancement.

En aucun cas je ne veux insinuer qu'il est toujours aussi facile de trouver la bonne méthode. Souvent, il faudra pas mal naviguer et l'utilisation créative de mots-clés pour les recherches, avant de

trouver une méthode prometteuse. Pour cette raison, il est extrêmement important de vous familiarisez avec la documentation des deux frameworks, ainsi vous saurez quelles classes et méthodes vous sont appropriées. Vous n'en avez peut-être pas pour l'instant, mais cela vous aidera à découvrir comment obtenir que votre programme fasse ce que vous voulez.

Bien, maintenant que nous avons trouvé notre méthode, tout ce que nous nous devons faire c'est d'ajouter la méthode à notre dossier de mise en œuvre file MAFoo.m [1,15].

```
//[1]
#import "MAFoo.h"
@implementation MAFoo
- (IBAction)réinitialiser:(id)expéditeur
{
    [textField setIntValue:0];
}
- (IBAction)setTo5:(id)expéditeur
{
    [textField setIntValue:5];
}
- (void)awakeFromNib   // [1.15]
{
    [textField setIntValue:0];
}
@end
```

Lorsque la fenêtre est ouverte, la méthode awakeFromNib est appelée automatiquement. En conséquence, le champ de texte affiche zéro quand vous posez vos yeux sur la nouvelle fenêtre ouverte.

11: Pointeurs

Avertissement!

Ce chapitre contient des concepts avancés et traite de concepts implicites du langage C que les débutants pourront trouver intimidant. Si vous ne comprenez pas tout maintenant, ne vous inquiétez pas. Heureusement, en général (bien que la compréhension du fonctionnement des pointeurs soit utile) ce n'est pas indispensable pour commencer la programmation en Objectif-C.

Introduction

Lorsque vous déclarez une variable, votre Mac associe cette variable à de l'espace dans sa mémoire pour y stocker la valeur de la variable.

Par exemple, examinez les instructions suivantes:

```
//[1]
int x = 4;
```

Pour l'exécuter, votre Mac trouve de l'espace non encore utilisé dans sa mémoire puis note que cet espace est l'endroit où la valeur de la variable x doit être stockée (bien sûr, nous aurions pu et du utiliser ici un nom plus descriptif pour notre variable). Examinez à nouveau l'instruction [1]. Indiquer le type de la variable (ici int) permet à votre ordinateur de savoir combien d'espace mémoire est nécessaire pour stocker la valeur de x. Si la valeur était de type long long ou double, plus de mémoire devrait être réservée.

L'instruction d'affectation `x = 4` stocke le chiffre 4 dans cet espace réservé. Bien sûr, votre ordinateur se souvient d'où la valeur de la variable x est stockée dans sa mémoire, ou, en d'autres termes, quelle est l'adresse de x. De cette façon, chaque fois que vous utilisez x dans

votre programme, votre ordinateur peut aller au bon endroit (à la bonne adresse) et trouver la valeur réelle de x.

Un pointeur est simplement une variable qui contient l'adresse d'une autre variable.

Référencer les variables

Étant donné une variable, vous pouvez obtenir son adresse en écrivant & devant la variable. Par exemple, pour obtenir l'adresse de x, vous écrivez &x.

Lorsque l'ordinateur évalue l'expression x, il retourne la valeur de la variable x (dans notre exemple, il retournera 4). En revanche, lorsque l'ordinateur évalue l'expression &x, il retourne l'adresse de la variable x, et non la valeur qui y est stockée. L'adresse est un nombre qui désigne un endroit précis dans la mémoire de l'ordinateur (comme un numéro de chambre désigne une chambre précise d'un hôtel).

Utiliser les pointeurs

Vous déclarez un pointeur de cette façon:

```
//[2]
int *y;
```

Cette instruction définit une variable nommée y qui contiendra l'adresse d'une variable de type `int`. Encore une fois: Elle ne contiendra pas une variable int, mais l'adresse d'une telle variable. Pour stocker dans la variable y quelle est l'adresse de la variable x (en jargon informatique officiel: assigner l'adresse de x à y), vous faites:

```
//[3]
y = &x;
```

Maintenant y "pointe sur" l'adresse de x. Ainsi, à l'aide de y, vous pouvez retrouver x. Voici comment.

Etant donné un pointeur, vous pouvez accéder à la variable sur laquelle il pointe, en mettant un astérisque devant le pointeur. Par exemple, évaluer l'expression:

```
*y
```

retournera 4. Ceci est équivalent à évaluer l'expression x. Exécuter l'instruction:

```
*y = 5
```

est équivalent à exécuter l'instruction:

```
x = 5
```

Les pointeurs sont utiles parce que parfois vous ne voudrez pas vous référer à la valeur d'une variable, mais à l'adresse de cette variable. Par exemple, vous pourrez vouloir programmer une fonction qui ajoute 1 à une variable. Bon, ne pouvez-vous simplement le faire comme ça?

```
//[4]
void incrementer(int x)
{
    x = x + 1;
}
```

Eh bien, non. Si vous appelez cette fonction depuis un programme, vous n'obtiendrez pas les résultats que vous attendiez:

```
//[5]
int maValeur = 6;
incrementer(maValeur);
NSLog(@"%d:\n", maValeur);
```

Ce code afficherait 6 sur votre écran. Pourquoi? N'avez-vous pas incrémenté maValeur en appelant la fonction incrementer? Non, vous ne l'avez pas fait. Vous voyez, la fonction en [4] n'a fait que prendre la valeur de maValeur (c'est-à-dire le chiffre 6), l'incrémenter de un, puis ... en gros, l'a jetée. Les fonctions ne marchent qu'avec les valeurs que vous leur passez, pas avec les variables qui véhiculent ces valeurs. Même si vous modifiez le x (comme vous pouvez le voir dans [4]), vous ne faites que modifier la valeur que la fonction a reçue. Toute modification de ce type sera perdue lorsque la fonction retourne. En outre, ce x n'est même pas forcément une variable: si vous appelez `incrementer(5);`, que vous attendez-vous à incrémenter?

Si vous voulez écrire une version de la fonction incrementer qui marche vraiment, c'est-à-dire accepte une variable comme son argument et incrémente en permanence la valeur de cette variable, vous devez lui passer l'adresse d'une variable. De cette façon, vous pouvez modifier ce qui est stocké dans cette variable, et non pas simplement utiliser sa valeur actuelle. Par conséquent, vous utilisez un pointeur comme argument:

```
//[6]
void incrementer(int *y)
{
    *y = *y + 1;
}
```

Vous pouvez alors l'appeler de cette façon:

```
//[7]
int maValeur = 6;
incrementer(&maValeur); // passe l'adresse
// Maintenant maValeur est égal à 7
```

12: Chaînes

Introduction

Jusqu'à présent, nous avons vu plusieurs types de données basiques: int, long, float, double, BOOL. Dans le précédent chapitre, nous avons introduit les pointeurs. Lorsque nous avons abordé le sujet des chaînes, nous n'en avons discuté que par rapport à la fonction NSLog (). Cette fonction nous a permis d'afficher une chaîne à l'écran, en remplaçant les codes commençant par un signe %, tel %d, par une valeur.

```
//[1]
float Valeurpi = 3.1416;
NSLog(@"Voici trois exemples de chaînes affichées à l'écran.
\n");
NSLog(@"Pi se rapproche de %10.4f.\n", Valeurpi);
NSLog(@"Le nombre de faces d'un dé est %d.\n", 6);
```

Nous n'avons pas parlé avant des chaînes en tant que types de données, pour une bonne raison. Contrairement aux entiers ou aux flottants, les chaînes sont de vrais objets, créés à l'aide de la classe NSString ou de la classe NSMutableString. Abordons ces classes, en commençant par NSString.

NSString

Encore les pointeurs

```
//[2]
NSString *favoriOrdinateur;
favoriOrdinateur = @"Mac!";
NSLog(favoriOrdinateur);
```

Vous comprendrez sans doute la seconde déclaration, mais la première [2.1] mérite quelques explications. Vous vous souvenez que, lorsque nous avons déclaré une variable pointeur, nous avons dû indiquer sur quel type de données le pointeur devait pointer? Voici une déclaration du chapitre 11 [3].

```
//[3]
int *y;
```

Ici on indique au compilateur que la variable pointeur y contient l'adresse d'un emplacement mémoire où un entier peut être trouvé.

Dans [2.1] nous disons au compilateur que la variable pointeur favoriOrdinateur contient l'adresse d'un emplacement mémoire où un objet de type NSString peut être trouvé. Nous utilisons un pointeur pour avoir prise sur notre chaîne, car en Objectif-C les objets ne sont jamais manipulés directement, mais toujours au travers de pointeurs sur eux.

Ne vous inquiétez pas trop si vous ne comprenez pas totalement ce point; il n'est pas essentiel. Ce qui est important, c'est de toujours se référer à une instance de NSString ou NSMutableString (ou bien sûr tout autre objet) en utilisant la notation *.

Le symbole @

Bien, pourquoi est-ce que ce drôle de signe @ apparaît tout le temps? Eh bien, Objectif-C est une extension du langage C, qui a sa propre façon de traiter avec les chaînes. Pour différencier le nouveau type de chaînes, qui sont des objets à part entière, Objectif-C utilise un signe @.

Un nouveau type de chaîne

En quoi Objectif-C améliore t-il les chaînes du langage C? Eh bien, les chaînes d'Objectif-C sont des chaînes Unicode au lieu de chaînes ASCII. Les chaînes Unicode permettent d'afficher les caractères de presque toutes les langues, comme le chinois, aussi bien que l'alphabet Romain.

Exercice

Bien sûr, il est possible de déclarer et initialiser la variable pointeur d'une chaîne d'un seul coup [4].

```
//[4]
NSString *favoriteActrice = @"Julia";
```

La variable pointeur favoriteActrice pointe sur un emplacement mémoire où l'objet représentant la chaîne "Julia" est stockée.

Une fois que vous avez initialisé la variable, c'est-à-dire favoriOrdinateur, vous pouvez donner une autre valeur à la variable, mais vous ne pouvez pas changer la chaîne elle-même [5.7], car c'est une instance de la classe NSString. Plus de détails sur ce point dans une minute..

```
//[5]
#import <foundation/foundation.h>
int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]
init];
    NSString *favoriOrdinateur;
    favoriOrdinateur = @"iBook";  // [5.7]
    favoriOrdinateur = @"MacBook Pro";
    NSLog(@"%@", favoriOrdinateur);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
MacBook Pro
```

Quand on dit qu'une chaîne ne peut être modifiée, on peut quand même (et en fait on vient de le faire) remplacer la totalité de la chaîne par une autre chaîne.

NSMutableString

Une chaîne de classe NSString est dite immuable, car elle ne peut être modifiée. Ce que nous entendons par là, c'est que les caractères individuels de la chaîne ne peuvent être modifiés.

A quoi bon une chaîne que vous ne pouvez modifier? Eh bien, pour le système d'exploitation, les chaînes qui ne peuvent être modifiées sont plus faciles à gérer, donc votre programme pourra être plus rapide. En fait, lorsque vous utiliserez Objectif-C pour écrire vos propres

programmes, vous verrez que la plupart du temps, vous n'avez pas besoin de modifier vos chaînes.

Bien sûr, parfois vous aurez besoin de chaînes que vous pouvez modifier. Donc, il y a une autre classe, et les objets chaîne que vous créez avec elle sont modifiables. La classe à utiliser est NSMutableString. Nous en parlerons plus loin dans ce chapitre.

Exercice

Tout d'abord, assurez-vous de bien comprendre que les chaînes sont des objets. Comme ce sont des objets, on peut leur envoyer des messages. Par exemple, nous pouvons envoyer le message longueur à un objet chaîne [6].

```
//[6]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]
init];
    int laLongueur;
    NSString * foo;
    foo = @"Julia!";
    laLongueur  = [foo length];  // [6.10]
    NSLog(@"La longueur est de %d.", laLongueur);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
La longueur est de 6.
```

Les programmeurs utilisent souvent foo et bar comme noms de variables pour expliquer les choses. En fait ce sont de mauvais noms, parce qu'ils ne sont pas descriptifs, tout comme x. Nous vous les mettons ici afin que vous ne soyez pas étonné quand vous les verrez dans des discussions sur Internet.

En ligne [6.10], nous envoyons à l'objet foo, le message `length` (longueur). La méthode `length` est définie comme suit dans la classe NSString:

```
- (unsigned int)length
Retourne le nombre de caractères Unicode dans le
destinataire.
```

Vous pouvez également mettre en majuscules les caractères de la chaîne [7]. A cette fin, envoyez à l'objet chaîne le message approprié, c'est-à-dire uppercaseString, que vous devriez être en mesure de trouver de par vous-même dans la documentation (examinez les méthodes disponibles dans la classe NSString). Dès réception de ce message, l'objet chaîne crée et retourne un nouvel objet chaîne ayant le même contenu, avec chaque caractère changé par sa valeur en majuscule correspondante.

```objc
//[7]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
    NSString *foo, *bar;
    foo = @"Julia!";
    bar  = [foo uppercaseString];
    NSLog(@"%@ est converti en %@.", foo, bar);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
Julia! est converti en JULIA!
```

Parfois, vous voudrez peut-être modifier le contenu d'une chaîne existante au lieu d'en créer une nouvelle. Dans ce cas, vous devrez utiliser un objet de classe NSMutableString pour représenter votre chaîne. NSMutableString fournit plusieurs méthodes qui vous permettent de modifier le contenu d'une chaîne. Par exemple, la méthode appendString: ajoute la chaîne passée comme argument à la fin du destinataire.

```
//[8]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
    NSMutableString *foo;                          // [8.7]
    foo = [@"Julia!" mutableCopy];                 // [8.8]
    [foo appendString:@" Je suis heureux"];
    NSLog(@"Voici le résultat: %@.",  foo);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

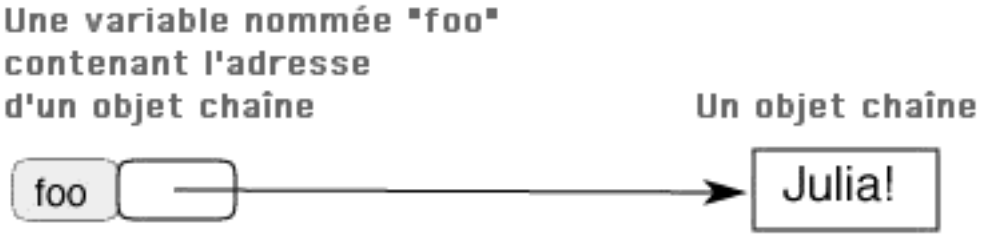
```
">Voici le résultat: Julia! Je suis heureux.
```

En ligne [8,8], la méthode mutableCopy (qui est fournie par la classe NSString) crée et retourne une chaîne mutable avec le même contenu que le destinataire. C'est à dire qu'après l'exécution de la ligne [8,8], foo pointe sur un objet chaîne mutable qui contient la chaîne "Julia!".

Encore plus de pointeurs!

Plus avant dans ce chapitre nous avons déclaré que, en Objectif-C, les objets ne sont jamais manipulés directement, mais toujours au travers de pointeurs sur eux. C'est pourquoi, par exemple, nous utilisons la notation du pointeur à la ligne [8.7] ci-dessus. En fait, lorsque nous utilisons le mot «objet» en Objectif-C, ce que l'on veut généralement dire c'est "pointeur sur un objet". Mais puisque nous utilisons toujours les objets au travers des pointeurs, nous utilisons le mot «objet»

comme raccourci. Le fait que les objets soient toujours utilisés au travers de pointeurs a une importante incidence que vous devez comprendre: plusieurs variables peuvent référencer le même objet en même temps. Par exemple, après exécution de la ligne [8.7], la variable foo référence un objet qui représente la chaîne "Julia!", quelque chose que nous pouvons représenter par l'illustration suivante:

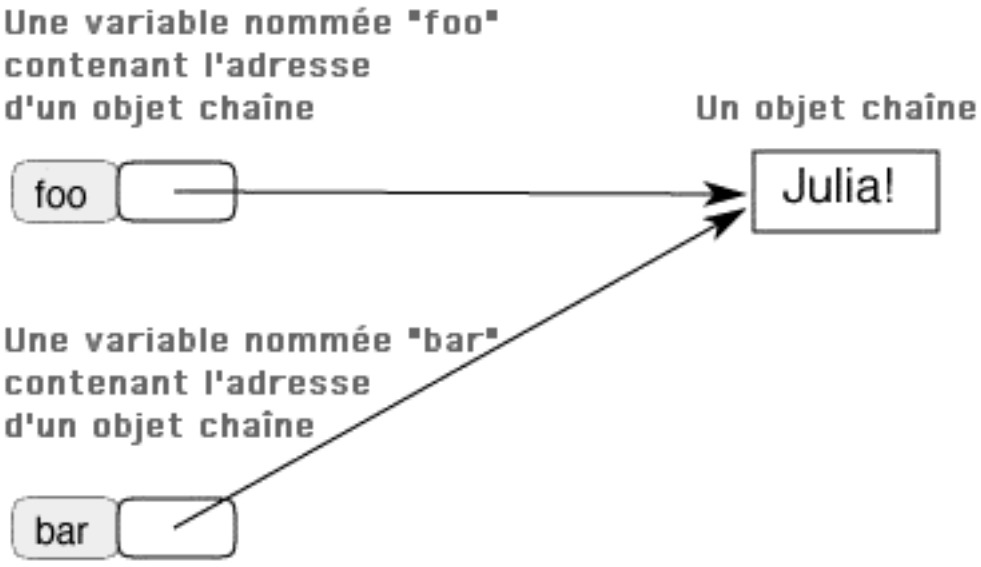


Les objets sont toujours manipulés au travers de pointeurs

Supposons maintenant que nous assignions la valeur de foo à la variable bar de cette façon:

```
bar = foo;
```

Le résultat de cette opération est que, et foo et bar pointent maintenant sur le même objet:



Plusieurs variables peuvent référencer le même objet

Dans une telle situation, l'envoi d'un message à l'objet en utilisant foo comme destinataire (par exemple [foo faireqqchose];) a le même effet que l'envoi du message en utilisant bar (par exemple [bar faireqqchose];), comme montré dans cet exemple:

```
//[9]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
    NSMutableString *foo = [@"Julia!" mutableCopy];
    NSMutableString *bar = foo;
    NSLog(@"foo pointe sur la chaîne: %@.",  foo);
    NSLog(@"bar pointe sur la chaîne: %@.",  bar);
    NSLog(@"\n");
    [foo appendString:@" Je suis heureux"];
    NSLog(@"foo pointe sur la chaîne: %@.",  foo);
    NSLog(@"bar pointe sur la chaîne: %@.",  bar);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
foo pointe sur la chaîne: Julia!
bar pointe sur la chaîne: Julia!
foo pointe sur la chaîne: Julia! Je suis heureux
bar pointe sur la chaîne: Julia! Je suis heureux
```

Etre capable d'avoir des références sur le même objet depuis différents endroits en même temps est une caractéristique essentielle des langages orientés-objets. En fait, nous l'avons déjà utilisée dans les précédents chapitres. Par exemple, au chapitre 8, nous avons référencé notre objet `MAFoo` depuis deux objets bouton différents.

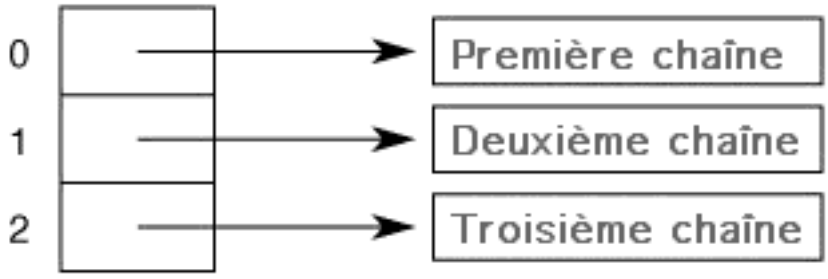
13: Tableaux

Introduction

Parfois vous aurez à organiser des collectes de données. Vous pourrez, par exemple, devoir entretenir une liste de chaînes. Il serait assez encombrant d'utiliser une variable pour chacune de ces chaînes. Bien sûr, il existe une solution plus pratique: le tableau.

Un tableau est une liste ordonnée d'objets (ou, plus exactement, une liste de pointeurs sur des objets). Vous pouvez ajouter des objets dans un tableau, les supprimer, ou demander au tableau de vous indiquer quel objet est stocké à un index donné (c'est-à-dire, à une position donnée). Vous pouvez également demander au tableau de vous indiquer le nombre d'éléments qu'il contient.

Lorsque vous comptez des articles, vous commencez habituellement en partant de 1. Toutefois, dans les tableaux, le premier article est à l'index zéro, le deuxième à l'index 1 et ainsi de suite.



Exemple: un tableau contenant trois chaînes

Plus loin dans ce chapitre, nous vous donnerons un exemple de code vous permettant de voir la conséquence du fait de compter à partir de zéro.

Les tableaux sont fournis par deux classes: NSArray et NSMutableArray. Comme pour les chaînes, il existe une version immuable et une mutable. Dans ce chapitre, nous étudierons la version mutable.

Ce sont des tableaux spécifiques à Objectif-C et Cocoa. Il existe un autre type de tableau, plus simple, en langage C (qui fait donc aussi partie d'Objectif-C), mais nous n'en parlerons pas ici. Ceci n'est juste qu'un rappel sur ce que vous pourrez lire ailleurs plus tard sur les tableaux C, et comprenez bien qu'ils n'auront pas grand chose à faire avec NSArrays ou NSMutableArrays.

Une méthode de classse

Une façon de créer un tableau est d'exécuter une expression comme celle-ci:

```
[NSMutableArray tableau];
```

Une fois évalué, ce code crée et retourne un tableau vide. Mais ... attendez une minute ... ce code semble étrange, non? En effet, dans ce cas nous avons utilisé le nom de la classe NSMutableArray pour spécifier le destinataire d'un message. Mais jusqu'à présent, nous n'avons envoyé des messages qu'à des instances, pas à des classes, d'accord?

Eh bien, nous venons d'apprendre quelque chose de nouveau: le fait que, en Objectif-C, nous pouvons aussi envoyer des messages aux classes (et la raison en est que les classes sont aussi des objets, instances de ce que l'on appelle des méta-classes, mais nous n'explorerons pas plus avant cette idée dans cet papier introductif).

Il conviendra de noter que cet objet est automatiquement autoreleased (auto-libéré) lors de sa création, c'est à dire qu'il est attaché à un NSAutoreleasePool et voué à la destruction par la méthode de classe qui l'a créé. Appeler la méthode de classe est équivalent à:

```
NSMutableArray *tableau = [[[NSMutableArray alloc] init] autorelease];
```

Dans l'éventualité où vous souhaitez que le tableau persiste plus longtemps que la durée de vie du pool autorelease, vous devez envoyer un message -retain à l'instance.

Dans la documentation Cocoa, les méthodes qui nous permettent d'appeler des classes sont signalées par un symbole «+», au lieu du symbole "-" que nous voyons habituellement devant le nom des méthodes (voir exemple au chapitre 8 [4.5]). Par exemple, dans la documentation, nous voyons cette description pour la méthode tableau:

array

+ (id)array
Crée et retourne un tableau vide. Cette méthode est utilisée par les sous-classes mutables de NSArray. Voir aussi: + arrayWithObject:, + arrayWithObjects:

Exercice

Revenons au codage. Le programme suivant crée un tableau vide, y stocke trois chaînes, puis affiche le nombre d'éléments du tableau.

```
//[1]
        #import <foundation/foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
    NSMutableArray *monTableau = [NSMutableArray tableau];
    [monTableau addObject:@"première chaîne"];
    [monTableau addObject:@"deuxième chaîne"];
    [monTableau addObject:@"troisième chaîne"];
    int count = [monTableau count];
    NSLog(@"Il y a %d éléments dans mon tableau", count);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
Il y a 3 éléments dans mon tableau
```

Le programme suivant est la même que le précédent sauf qu'il affiche la chaîne stockée à l'index 0 du tableau. Pour atteindre cette chaîne, il utilise la méthode objectAtIndex: [2.13].

```
//[2]
        #import <foundation/foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
    NSMutableArray *monTableau = [NSMutableArray tableau];
    [monTableau addObject:@"première chaîne"];
    [monTableau addObject:@"deuxième chaîne"];
    [monTableau addObject:@"troisième chaîne"];
    NSString *élément = [monTableau objectAtIndex:0];  //
[2.13]
    NSLog(@"L'élément à l'index 0 du tableau est: %@",
élément);
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
L'élément à l'index 0 du tableau est: première chaîne
```

Vous devrez souvent parcourir un tableau pour faire quelque chose avec chacun de ses éléments. Pour ce faire, vous pouvez utiliser une boucle comme dans le programme suivant qui affiche chaque élément du tableau avec son index:

```
//[3]
#import <foundation/foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
    NSMutableArray *monTableau = [NSMutableArray tableau];
    [monTableau addObject:@"première chaîne"];
    [monTableau addObject:@"deuxième chaîne"];
    [monTableau addObject:@"troisième chaîne"];
    int i;
    int count;
    for (i = 0, count = [monTableau count]; i < count; i = i
+ 1)
    {
       NSString *élément = [monTableau objectAtIndex:i];
       NSLog(@"L'élément à l'index %d du tableau est: %@", i,
élément);
    }
    [pool release];
    return 0;
}
```

Une fois exécuté, le programme affiche:

```
L'élément à l'index 0 du tableau est: première chaîne
L'élément à l'index 1 du tableau est: deuxième chaîne
L'élément à l'index 2 du tableau est: troisième chaîne
```

Notez que les tableaux ne sont pas limités aux chaînes. Ils peuvent contenir tout objet que vous voulez.

Les classes NSArray et NSMutableArray fournissent de nombreuses autres méthodes, et vous êtes encouragé à consulter la documentation de ces classes afin d'en apprendre davantage sur les tableaux. Nous allons finir chapitre en parlant de la méthode qui vous permet de remplacer un objet d'un index donné par un autre objet. Cette méthode se nomme replaceObjectAtIndex:withObject:.

Jusqu'à présent nous n'avons traité qu'avec des méthodes ne prenant au plus qu'un argument. Celle-ci est différente, et c'est pourquoi nous allons l'examiner ici: elle prend deux arguments. Vous pouvez le dire, car son nom contient deux deux-points. En Objectif-C, les méthodes peuvent avoir un nombre quelconque d'arguments. Voici comment vous pouvez utiliser cette méthode:

```
//[4]
[monTableau replaceObjectAtIndex:1 withObject:@"Bonjour"];
```

Après l'exécution de cette méthode, l'objet à l'index 1 est la chaîne @"Bonjour". Bien entendu, cette méthode ne doit être invoquée qu'avec un index valable. C'est à dire qu'il faut déjà avoir un objet stocké à l'index que nous donnons à la méthode, afin que la méthode soit en mesure de le remplacer dans le tableau par l'objet que nous passons.

Conclusion

Comme vous pouvez le voir, les noms de méthode en Objectif-C sont comme des phrases avec des trous dedans (préfixés par des virgules). Lorsque vous invoquez une méthode, vous remplissez les trous avec de vraies valeurs, créant une "phrase" sensée. Cette façon de désigner les noms de méthode et l'invocation de la méthode vient de Smalltalk et est l'une des plus grandes forces d'Objectif-C, car il rend le code

très expressif. Lorsque vous créerez vos propres méthodes, vous devrez vous efforcer de les nommer de façon à ce qu'elles forment des phrases expressives en cas d'appel. Cela aide à rendre lisible le code Objectif-C, ce qui est très important pour une maintenance plus aisée de vos programmes.

14: Accesseurs et Propriétés

Introduction

Nous avons vu qu'un objet pouvait être visible, comme une fenêtre ou un champ de texte, ou invisible, comme un tableau, ou un contrôleur qui répond à des actions de l'interface utilisateur. Alors, qu'est-ce qu'un objet exactement?

Par essence, un objet détient des valeurs (variables) et effectue des actions (méthodes). Un objet contient et transforme à la fois les données. Un objet peut être considéré comme un petit ordinateur en soi, qui envoie des messages et y répond. Votre programme est un réseau de ces petits ordinateurs travaillant tous ensembles pour produire le résultat désiré.

Composition d'objet

Le travail d'un programmeur Cocoa est de créer des classes qui contiennent un certain nombre d'autres objets (comme des chaînes, des tableaux et des dictionnaires) pour conserver les valeurs dont la classe aura besoin pour faire son travail. Certains de ces objets seront créés, utilisés et ensuite mis de côté au sein d'une seule méthode. D'autres pourront devoir rester tout le long de la durée de vie de l'objet. Ces derniers objets sont appelés variables d'instance ou propriétés de la classe. La classe peut également définir des méthodes qui marchent sur ces variables.

Cette technique est connue sous le nom de composition d'objet. De tels objets composés héritent généralement directement de NSObject.

Par exemple, une classe contrôleur de calculatrice pourra contenir comme variables d'instance: un tableau d'objets bouton, et une variable champ de texte résultat. Elle pourra aussi inclure des méthodes pour multiplier, additionner, soustraire et diviser des nombres et afficher le résultat dans l'interface graphique.

Les variables d'instance sont déclarées dans le fichiers d'en-tête d'interface de la classe. L'exemple de la classe de notre application contrôleur de calculatrice pourrait ressembler à ceci:

```
//[1]
@interface MaCalculatriceControleur : NSObject {
     //Instance variables
     NSArray * boutons;
     NSTextField * champRésultat;
}
//Methods
- (NSNumber *)mutiply:(NSNumber *)value;
- (NSNumber *)add:(NSNumber *)value;
- (NSNumber *)subtract:(NSNumber *)value;
- (NSNumber *)divide:(NSNumber *)value;
@end
```

Encapsulation

Un des buts de la programmation orientée-objet, c'est l'encapsulation: rendre chaque classe aussi autonome et réutilisable que possible. Et si vous vous souvenez du chapitre 3, les variables sont protégées de l'extérieur des boucles, des fonctions et des méthodes. La protection de variable est valable aussi pour les objets. Ce que cela signifie, c'est que d'autres objets ne peuvent pas accéder aux variables d'instance à l'intérieur d'un objet, elles ne sont disponibles que pour leurs propres méthodes.

De toute évidence, d'autres objets devront parfois modifier les variables contenues dans un objet. Comment?

Les méthodes sont disponibles à l'extérieur d'un objet. Rappelons que tout ce que nous avons à faire, c'est d'envoyer un message à notre objet pour déclencher cette méthode. Aussi, le moyen de rendre

disponibles des variables d'instance est de créer une paire de méthodes pour accéder à la variable d'instance et la modifier. Les méthodes sont collectivement appelées méthodes accesseur.

Au chapitre 8, nous avons découvert la méthode setIntValue: de NSTextField. Cette méthode est le pendant de la méthode intValue. Ce sont toutes deux des méthodes accesseur de NSTextField.

Accesseurs

A quoi cela ressemble t-il dans le code? Prenons l'exemple suivant.

```
//[2]
@interface MonChien : NSObject {
     NSString * _nom;   //[2.2]
}
- (NSString *)nom;
- (void)définirNom:(NSString *)value;
@end
```

Cette interface de classe définit un objet: MonChien. MonChien a une variable d'instance, une chaîne appelée _nom [2.2]. Afin de pouvoir lire _nom de MonChien ou de modifier ce _nom, nous avons défini deux méthodes accesseur, nom et définirNom:.

Jusqu'ici tout va bien. La mise en oeuvre ressemble à ça:

```
//[3]
@implementation MonChien
- (NSString *)nom {
    return _nom;   //[3.3]
}
- (void)définirNom:(NSString *)value {
        _nom = value;   //[3.6]
}
@end
```

Dans la première méthode [3.3], nous retournons simplement la variable d'instance. Dans la seconde méthode [3,6], nous avons mis la variable d'instance à la valeur passée. Notez que j'ai simplifié cette mise en œuvre pour plus de clarté; normalement vous devrez traiter tout code de gestion de mémoire nécessaire au sein de ces méthodes. L'exemple suivant [4] montre un ensemble d'accesseurs plus réaliste:

```
//[4]
@implementation MonChien
- (NSString *)nom {
    return [[_nom retain] autorelease];
}
- (void)définirNom:(NSString *)value {
    if (_nom != value) {
        [_nom release];
        _nom = [value copy];
    }
}
@end
```

Je ne vais pas entrer dans le détail du code supplémentaire ici (voir chapitre 15), mais en un coup d'œil vous pouvez voir le même schéma qu'en [3], avec juste quelques copying (copier), retaining (conserver) et releasing (libérer) enveloppés dedans. Différents types de valeurs requièrent différents codes de gestion de mémoire. (Notez aussi qu'en pratique, il est recommandé de ne pas utiliser de caractère de soulignement devant le nom de variable d'instance, j'en ai utilisé un ici pour plus de clarté. Dans votre code, vous pouvez simplement appeler la variable "nom". Puisque les méthodes et les variables ont des espaces de nom distincts, aucun conflit ne surviendra).

Propriétés

Léopard et Objectif-C 2.0 introduisent de nouvelles fonctionnalités de langage pour traiter plus économiquement ce modèle de programmation. La nouvelle fonctionnalité dont nous parlons est l'ajout de propriétés. Les accesseurs sont si communs que l'aide cette appréciable niveau de langage peut se traduire par beaucoup moins de code. Et moins de code veut dire moins de code à déboguer. :)

Alors, en quoi les propriétés sont-elles différentes des accesseurs? En gros, les propriétés synthétisent directement les accesseurs, en utilisant la gestion de mémoire la plus efficace et la plus appropriée. En d'autres mots, elles écrivent les méthodes accesseur pour vous, mais en arrière-plan; donc vous ne verrez même jamais le code.

En reprenant notre exemple [2] ci-dessus, en Objectif-C 2.0, nous pourrions écrire à la place:

```
//[5]
@interface MonChien : NSObject {
    NSString * nom;
}
@property (copy) NSString *nom;
@end
```

Et notre mise en oeuvre ressemblerait à ça:

```
//[6]
@implementation MonChien
@synthesize nom;
@end
```

Ce qui est logiquement équivalent à [4]. Comme vous pouvez le voir, cela simplifie quelque peu notre code. Si votre classe contient beaucoup de variables d'instance ayant besoin d'accesseurs, vous pouvez imaginer à quel point votre vie devient plus facile!

15: Gestion de la mémoire

15: Gestion de la mémoire

Introduction

Dans plus d'un chapitre je me suis excusé de ne pas expliquer deux ou trois déclarations dans les exemples. Ces déclarations traitent avec la mémoire. Votre programme n'est pas le seul programme sur votre Mac, et la RAM est une denrée précieuse. Donc, si votre programme n'a plus besoin d'une portion de mémoire, vous devez la restituer au système. Quand votre mère vous disait que vous deviez être poli et vivre en harmonie avec la communauté, elle vous enseignait comment programmer! Même si votre programme était le seul à tourner, la mémoire non libérée finira par dépeindre votre programme dans un coin et votre ordinateur traînera la patte.

Garbage Collection

Les techniques de gestion de mémoire utilisées par Cocoa et présentées plus loin dans ce chapitre sont communément connues sous le nom de Comptage de référence (Reference Counting). Vous trouverez des explications complètes sur ce système de gestion de la mémoire de Cocoa dans des livres ou des articles plus spécialisés (voir chapitre 15).

Mac OS X 10.5 (Léopard) introduit une nouvelle forme de la gestion de mémoire pour Objectif-C 2.0, connu sous le nom de Cocoa Garbage Collection (Nettoyage de mémoire). Garbage Collection gère automatiquement la mémoire, supprimant la nécessité d'expliciter les objets Cocoa retain, release ou autorelease.

La magie de Garbage Collection fonctionne sur tous les objets Cocoa qui héritent de NSObject ou NSProxy, et permet à un programmeur d'écrire simplement moins de code que dans les versions antérieures

d'Objectif-C. Il n'y a pas grand chose de plus à dire à son sujet, en pratique. Oubliez tout ce que vous avez appris dans ce chapitre!

Activer Garbage collection

Garbage Collection a besoin d'être activé, car il est désactivé par défaut dans un nouveau projet Xcode. Pour l'activer, sélectionnez votre application Cible (Target) dans la liste Source, et ouvrez l'Inspecteur. Mettez une coche à côté de l'item "Enable Objective-C Garbage Collection". Notez que tous les frameworks (ensemble de bibliothèques, d'outils et de conventions permettant le développement d'applications) que vous reliez à votre projet doivent également être "Garbage collectés".

Comptage de référence: Le cycle de vie de l'objet

Si vous êtes intéressé par les techniques de gestion de la mémoire pré-Leopard, lisez ce qui suit.

Lorsque votre programme crée un objet, l'objet occupe de l'espace en mémoire et vous devez libérer cet espace lorsque votre objet n'est plus utilisé. C'est à dire que si votre objet n'est plus utilisé, vous devez le détruire. Toutefois, déterminer quand un objet a fini d'être utilisé peut ne pas être facile à faire.

Par exemple, durant l'exécution du programme, votre objet peut être référencé par de nombreux autres objets, et ne doit donc pas être détruit tant que il y a une possibilité qu'il puisse être utilisé par d'autres objets (essayer d'utiliser un objet qui a été détruit peuvent amener votre programme à planter ou à se comporter de façon imprévisible).

Le compte de retenue (retain count)

Afin de vous aider à détruire les objets quand ils ne sont plus nécessaires, Cocoa associe un compteur à chaque objet, qui représente ce qui est appelé le "compte de retenue" de l'objet. Dans votre programme, lorsque vous stockez une référence à un objet,

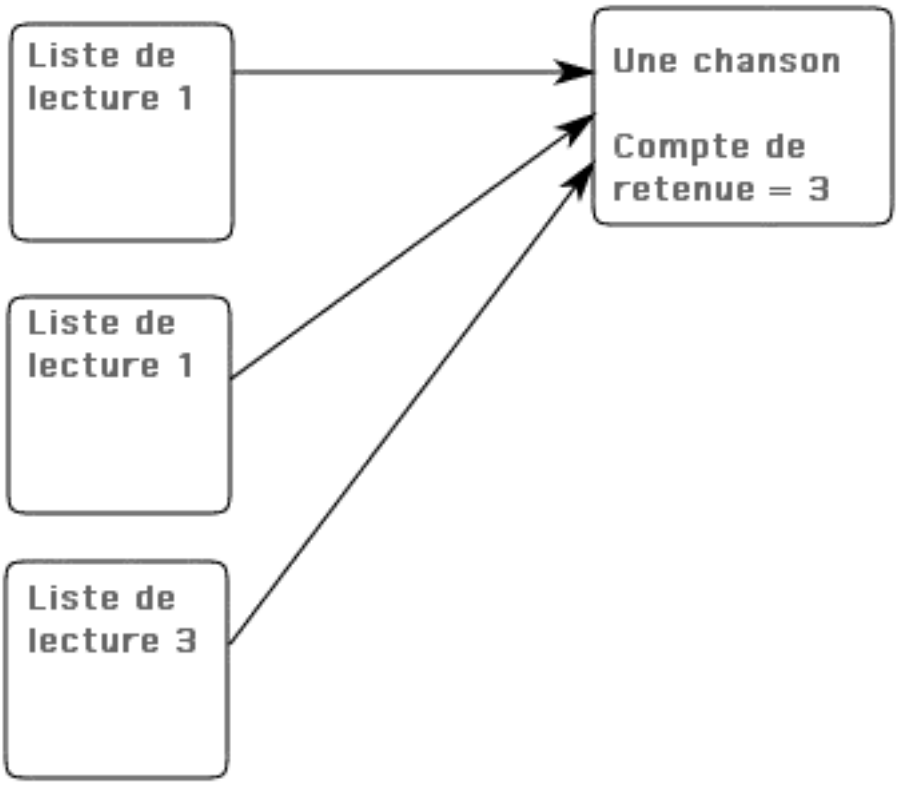
vous devez en informer l'objet en augmentant son compte de retenue de un. Lorsque vous supprimez une référence à un objet, vous en informez l'objet en diminuant son compte de retenue de un. Lorsque le compte de retenue d'un objet devient égal à zéro, l'objet sait qu'il n'est plus référencé nulle part et qu'il peut être détruit sans danger. L'objet se détruit alors lui-même, libérant la mémoire associée.

Par exemple, supposons que votre application est un juke-box numérique et que vous avez des objets représentant des chansons et des listes de lecture. Supposons qu'un objet chanson donné est référencé par trois objets listes de lecture. S'il n'est pas référencé ailleurs, votre objet chanson aura un compte de retenue de trois.

Liste de lecture 1, Liste de lecture 2, Liste de lecture 3

Une chanson

Compte de retenue = 3



Un objet sait combien de fois il est référencé, grâce à son compte de retenue

Maintenir et Libérer (Retain and Release)

Pour augmenter le compte de retenue d'un objet, il vous suffit d'envoyer à l'objet un message retain.

```
[unObjet retain];
```

Pour diminuer le compte de retenue d'un objet, il vous suffit d'envoyer à l'objet un message release.

```
[unObjet release];
```

Autorelease

Cocoa offre également un mécanisme appelé "autorelease pool" qui vous permet d'envoyer à retardement un message release à un objet; pas immédiatement, mais plus tard. Pour l'utiliser, il vous suffit d'enregistrer l'objet avec ce qu'on appelle un autorelease pool, en envoyant un message autorelease à votre objet.

```
[unObjet autorelease];
```

L'autorelease pool s'occupera de l'envoi à retardement du message release à votre objet. Les déclarations traitant les autorelease pools que nous avons vu précédemment dans nos programmes, sont des instructions que nous donnons au système afin de configurer correctement la machinerie autorelease pool.

16: Sources d'information

Le modeste objectif de ce livre était de vous enseigner les bases d'Objectif-C dans l'environnement Xcode. Si vous avez parcouru deux fois ce livre, et essayé les exemples avec vos propres variations de ceux-ci, vous êtes prêt à apprendre comment écrire les applications mortelles que vous cherchez à créer. Ce livre vous a donné suffisamment de connaissances pour aborder rapidement les problèmes. Comme vous l'avez fait dans ce chapitre, vous êtes prêt à exploiter d'autres ressources, et celles mentionnées ci-après devraient retenir votre attention. Un conseil important avant de commencer à écrire votre code: ne commencez pas tout de suite! Vérifiez les frameworks, car Apple peut avoir déjà fait le travail pour vous, ou fourni des classes nécessitant un petit peu de travail pour arriver à ce dont vous avez besoin. De même, quelqu'un d'autre peut avoir déjà fait ce dont vous avez besoin, et rendu le code source disponible. Donc, économisez vous du temps en feuilletant la documentation et en cherchant sur Internet. Votre première visite devrait être pour le site développeur d'Apple à l'adresse suivante: http://developer.apple.com

Nous vous recommandons aussi fortement de mettre en signets:

- http://osx.hyperjeff.net/reference/CocoaArticles.php
- http://www.cocoadev.com
- http://www.cocoadevcentral.com
- http://www.cocoabuilder.com
- http://www.stepwise.com

Les sites ci-dessus comportent un grand nombre de liens vers d'autres sites et sources d'information. Vous devez aussi vous abonner à la liste de diffusion cocoa-dev à http://lists.apple.com/mailman/listinfo/cocoa-dev. C'est un endroit où vous pouvez poser vos questions. Des membres serviables feront de leur mieux pour vous aider. En retour, soyez poli et vérifiez d'abord si vous pouvez

trouver la réponse à votre question dans les archives (http://www.cocoabuilder.com). Pour des conseils sur l'envoi des questions dans les listes de diffusion, consultez la section "How To Ask Questions The Smart Way (Comment poser Les bonnes questions)" à http://www.catb.org/~esr/faqs/smart-questions.html

Il y a plusieurs très bons livres sur le développement Cocoa. Programming in Objective-C, de Stephen Kochan est destiné aux débutants. Certains livres présument que vous avez au moins certaines des connaissances que vous aurez tirées de ce livre. Nous avons aimé Cocoa Programming for Mac OS X de Aaron Hillegass du Big Nerd Ranch, où il enseigne Xcode pour vivre. Nous avons aussi apprécié Cocoa with Objective-C de James Duncan Davidson et Apple, publié par O'Reilly.

Enfin, un avertissement amical. Vous programmez pour le Mac. Avant de publier votre programme, assurez-vous qu'il est non seulement exempt de bogue, mais aussi impeccable, et respecte les directives de l'interface humaine d'Apple (qui sont décrites ici). Une fois cela fait, n'hésitez pas à rendre votre programme disponible! Les commentaires des autres vous aideront à affiner et développer votre programme et à mettre l'accent sur la qualité.

Nous espérons que vous avez apprécié ce livre et que vous allez poursuivre la programmation avec Xcode.

Bert, Alex, Philippe, Racky.

Licence

Version

Version 1.15

19 Février 2008

French translation by Racky