# 1

# Le monde d'Objective-C

*Vous pensiez peut-être couper à l'inévitable historique, mais celui d'Objective-C est intéressant pour justifier le choix de ce langage par Apple. Il permet aussi d'éclairer sur ce que n'est pas Objective-C, afin d'éviter les confusions. Quelques points techniques relatifs à la compilation sont abordés.*

Ce chapitre a pour but de présenter Objective-C et ses satellites, pour définir clairement les rôles. Entre langage, normes, API, extensions et évolutions, quelques précisions sont nécessaires. Les détails techniques avancés sont toutefois reportés tout au long de cet ouvrage, pour ne pas complexifier inutilement cette présentation. Les évolutions du langage sont également récapitulées dans le dernier chapitre.

## Du C au modèle objet : les principaux protagonistes

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

Smalltalk-80 est un des tout premiers langages réellement objet, qui a donc fortement influencé les développements ultérieurs de ce concept. Le C++ et l'Objective-C sont deux branches différentes visant à créer un langage objet basé sur le C.

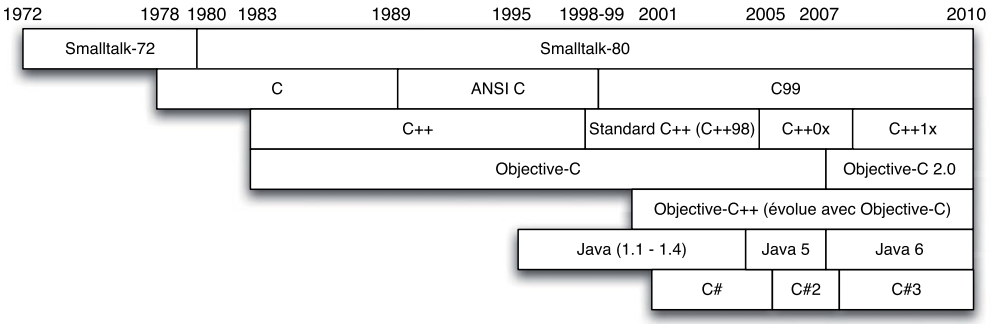


**Figure 1–1** Historique sommaire de Smalltalk, Java, C, C#, C++ et Objective-C

Tandis que C++ se tourne vers une forme de programmation très statique minimisant le « coût » du modèle objet pour les performances d'exécution, Objective-C choisit la voie du dynamisme de Smalltalk, dont il emprunte aussi une partie de la syntaxe.

DYNAMISME ET PERFORMANCES **L'un ne chasse pas l'autre**

Peut-être voyez-vous déjà dans cette divergence un avantage très net de C++ pour ce qui est des performances. C'est en partie vrai. Les performances d'Objective-C sont toutefois excellentes lorsqu'il est correctement utilisé. Une section y est consacrée dans le chapitre 13 intitulé « Les évolutions d'Objective-C : performances et extensions du langage », en fin d'ouvrage.

Java veut se placer en remplaçant du C++, toujours avec une syntaxe très proche du C mais un meilleur modèle objet, lui aussi inspiré de Smalltalk.

Le langage C#, développé par Microsoft, est un concurrent direct d'Objective-C, non pas du point de vue d'une compétition, mais des possibilités. Les similitudes sont nombreuses et dénotent en réalité un renouveau du développement objet pour la conception d'applications modernes.

L'Objective-C++ est une fusion des syntaxes de l'Objective-C et du C++. Attention : il ne s'agit pas vraiment d'un langage, mais plutôt d'une fonctionnalité du compilateur. Il ne s'agit pas ici de fusionner les concepts, mais d'autoriser les deux syntaxes dans le même code source. Ainsi, une classe C++ ne peut pas être interchangeable avec une classe Objective-C, mais les deux peuvent cohabiter. Voyez la section consacrée à Objective-C++ pour des explications plus poussées et connaître les limites de cette cohabitation, qui est tout de même assez poussée.

L'Objective-C 2.0 est une évolution d'Objective-C, publiée par Apple avec Mac OS 10.5. Quelques mots-clefs et fonctionnalités ont été rajoutés au langage.

Ces ajouts sont précisés au long des chapitres de ce livre, et résumés dans la section dédiée à Objective-C 2.0 dans le chapitre 13 « Les évolutions d'Objective-C : performances et extensions du langage ».

## Objective-C et ses API (Cocoa…) : les racines non Unix de Mac OS X

Une première précision me semble indispensable : il est courant de voir une confusion entre Objective-C et Cocoa. Soyons précis : Objective-C est un langage alors que Cocoa est une API (*Application Programming Interface*), c'est-à-dire un ensemble de classes permettant de programmer de façon native sous Mac OS X (et iPhone OS).

En réalité, Cocoa est l'implémentation par Apple, pour Mac OS X, du standard OpenStep publié par NeXT Computer en 1994. Cela n'a rien d'étonnant puisque Mac OS X est né du rachat des technologies NeXT. Cocoa est donc une bibliothèque de développement d'applications basée sur Objective-C.

Apple est actuellement le seul acteur majeur à promouvoir l'Objective-C. Il n'y a donc que peu d'homologues à Cocoa. On peut citer GNUStep et Cocotron, deux autres implémentations, libres… non pas d'OpenStep, cette fois, mais de Cocoa ! Cette dernière est en effet l'API au développement le plus actif puisqu'elle est maintenue par Apple, qui l'améliore sans cesse pour ses besoins propres au sein de Mac OS X. GNUStep et Cocotron poursuivent l'objectif de la réimplémenter sous forme de bibliothèque libre et portable, pour ouvrir ces technologies au développement d'applications sur d'autres systèmes que ceux d'Apple.

> **POUR EN SAVOIR PLUS** **Le projet GNUStep**
>
> GNUStep est un projet assez avancé, puisqu'en sus des API, il propose des outils de développement. Par exemple, l'IDE principale Xcode et le concepteur d'interfaces graphiques Interface Builder de Mac OS X trouvent des équivalents, ProjectCenter et Gorm, sous GNUStep. Vous pouvez visiter le site du projet.
> ▸ http://www.gnustep.org.

## La bibliothèque standard Objective-C

Le C, C++ et Java disposent d'une bibliothèque standard. Pour Objective-C, à l'instar de C#, ce n'est pas vraiment le cas. Objective-C est développé parallèlement à

Cocoa, les deux dépendant maintenant d'Apple, alors que C# évolue avec .NET, les deux émanant de Microsoft. Les autres implémentations (par exemple GNUStep pour Objective-C/Cocoa et Mono pour C#/.NET) ne sont que des portages, pas des concurrents. Il n'existe pas à ma connaissance de notion claire de bibliothèque standard pour ces langages, autrement que ces API auxquelles ils se sont trouvés intrinsèquement liés.

Pour le présent ouvrage, nous avons donc considéré que Cocoa définissait *de facto* un standard pour certains services (comme la gestion mémoire). Nous utiliserons donc par exemple la classe de base `NSObject` au lieu de `Object`, cette dernière étant pourtant présente dans le fichier d'en-tête `<objc/Object.h>`.

Nous avons cependant pris soin de ne pas nous tromper de cible : pour ce livre, nous ne voulons pas détailler Cocoa, nous ne faisons que nous servir de sa substantifique moelle. Pour connaître le fonctionnement complet d'une classe particulière, il faut toujours consulter la dernière version de sa documentation.

Enfin, pour étoffer et illustrer nos explications, nous faisons parfois référence à quelques classes utilitaires, en nous limitant à l'essentiel. Il pourra s'agir de `NSString` (chaîne de caractères), `NSArray` (tableau dynamique), `NSDictionary` (tableau associatif)… Lorsque nous utilisons ces classes et leurs méthodes, leurs noms sont toujours assez clairs pour être compris quasiment sans explications.

**UN STANDARD QUI VARIE** **Cocoa pour Mac OS X et iPhone OS**

Cocoa n'est pas une API, mais plutôt un ensemble d'API. Certaines peuvent différer entre Mac OS X et iPhone OS. Par exemple, les classes d'interfaces graphiques sont remplacées sur iPhone OS par un jeu de contrôles plus adapté aux terminaux mobiles, regroupés dans le framework UIKit. Ainsi, certaines parties de Cocoa existent pour les deux plates-formes, d'autres non (soit remplacées, soit évincées pour une version plus compacte).
Les classes Cocoa utilisées dans cet ouvrage pour illustrer le code sont toujours assez basiques pour être valables sur ces deux plates-formes.

# La documentation officielle d'Objective-C

Actuellement, c'est la documentation d'Apple qui constitue la meilleure référence du langage. Le document http://developer.apple.com/mac/library/documentation/cocoa/Conceptual/ObjectiveC/ObjC.pdf a été mis à jour pour Objective-C 2.0 ; sa teneur est en fait une découverte de la programmation objet à travers les concepts d'Objective-C.

Pour le détail des classes et des fonctions utilitaires, c'est la documentation Cocoa (ou de quelques autres API de bas niveau) qu'il faut consulter.

- http://developer.apple.com/cocoa
- http://developer.apple.com/mac/library/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html

# Compilation d'un programme Objective-C

## La bibliothèque runtime d'Objective-C

Pour fonctionner, un programme Objective-C doit être « linké » avec un runtime Objective-C. Cette bibliothèque est la clé du dynamisme d'Objective-C. Il faut la voir comme une couche intermédiaire entre le code du développeur et l'exécution de ce code. Elle charge et référence les classes et les méthodes, et permet ainsi d'ajouter, manipuler, interroger des classes lors de l'exécution même du programme… Le chapitre 11 intitulé « Le dynamisme et les manipulations du runtime » expose les nombreuses possibilités de ce runtime.

Mac OS X l'intègre bien évidemment, sous la forme de la bibliothèque `/usr/lib/libobjc.dylib`. Une version de `libobjc` est aussi disponible pour Linux et Windows.

Précisons bien que cette bibliothèque n'est pas Cocoa. Il s'agit du cœur d'exécution de tout programme Objective-C.

## Compilation et linkage

L'utilisation d'Objective-C en dehors des systèmes d'Apple est très marginale. Ses deux plates-formes principales sont actuellement Mac OS X et iPhone OS, ce dernier étant une déclinaison de Mac OS X pour les périphériques mobiles que sont l'iPod Touch, l'iPhone et la tablette iPad.

De ce fait, compiler un programme Objective-C est le plus souvent synonyme de « compiler le programme sous Mac OS X ». Et dans ce cas, c'est souvent à travers l'IDE *(Integrated Development Environment)* XCode que cette compilation est réalisée. Il se charge d'ailleurs d'encapsuler l'exécutable produit dans la structure d'application propre à Mac OS X.

Cependant, rien n'empêche de compiler un programme en ligne de commande avec `gcc` ou `llvm`, ces deux outils ayant un *front-end* leur permettant de comprendre la syntaxe Objective-C. Par exemple, sous `gcc`, le *flag* de compilation `-x`, permettant de spécifier explicitement le langage, peut prendre la valeur `objective-c`.

**Compilation et link d'un programme Objective-C, sans Cocoa**

```
$>gcc -c monProgramme.m
$>gcc -o monProgramme -lobjc monProgramme.o
```

Un fichier Objective-C a traditionnellement l'extension « `.m` », connue de `gcc`. Lors de la simple compilation (`-c`), `libobjc` n'a pas d'intérêt. C'est au linkage qu'elle est intégrée *via* le flag `-l`.

Sous Mac OS X, il est peu pertinent de se passer de Cocoa. La compilation et le linkage y font donc quasiment toujours référence, comme dans l'exemple ci-dessous.

**Compilation d'un programme Objective-C sous Mac OS X**

```
$>gcc -c monProgramme.m
$>gcc -o monProgramme -framework Cocoa monProgramme.o
```

> **POUR EN SAVOIR PLUS La notion de framework**
>
> Les bibliothèques dynamiques sous Mac OS X sont encapsulées dans un dossier appelé framework, dont la structure est très précise. Ce dossier contient les fichiers d'en-tête, le code compilé, les éventuelles ressources. Plusieurs versions de la bibliothèque peuvent se côtoyer dans le framework pour gérer la compatibilité.

Vous pouvez noter qu'aucune référence n'est ici faite à `libobjc`. En effet, Cocoa, à travers un autre framework (AppKit), est déjà linkée à `libobjc`. Cela peut d'ailleurs se vérifier avec l'outil `otool`, qui permet entre autres de lister les dépendances des bibliothèques partagées.

**Vérification du link vers libobjc à travers le framework Cocoa**

```
$>otool -L /System/Library/Frameworks/Cocoa.framework/Cocoa
/System/Library/Frameworks/Cocoa.framework/Cocoa:
(…)
/System/Library/Frameworks/AppKit.framework/Versions/C/AppKit
(compatibility version 45.0.0, current version 1025.2.0)
(…)

$>otool -L /System/Library/Frameworks/AppKit.framework/AppKit
/System/Library/Frameworks/AppKit.framework/AppKit:
(…)
   /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current
version 227.0.0)
(…)
```

**POINT TECHNIQUE** **Compilation pour iPhone OS**

Pour iPhone OS, le processus de signature numérique et l'architecture hardware différente de la machine hôte (ARM contre PPC/Intel) rendent la compilation et le lancement de programme plus complexes ; dans ce cas il vaut mieux passer par Xcode.

## Code minimal

Si vous voulez tester de petites portions de code sans passer par Xcode, vous pouvez créer un fichier contenant le code suivant.

**Code minimal pour un programme Objective-C**

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

int main(int argc, char* argv[]) //c'est le main() du langage C
{
  NSAutoreleasePool* bassin = [[NSAutoreleasePool alloc] init];

  //Votre code ici

  [bassin drain]; //De préférence à [bassin release]
                  //(cf. Gestion mémoire)
  return 0;
}
//À compiler avec gcc -framework Cocoa -o executable nomDuFichier.m
```

Le `NSAutoreleasePool` est décrit dans le chapitre 6 « Gestion de la mémoire et ramasse-miettes ».

Lorsque Xcode est utilisé pour créer la structure d'un projet Cocoa, le code de la fonction `main()` est sensiblement différent.

**Code minimal pour un programme Cocoa**

```
int main(int argc, char *argv[])
{
  return NSApplicationMain(argc, (const char **) argv);
}
```

La raison est la suivante : dans un programme Cocoa, l'interaction avec l'utilisateur passe par une boucle événementielle (la *run-loop*) qui enregistre et distribue les événements, jusqu'à la demande de terminaison du programme. La fonction `NSApplicationMain()` se charge donc de créer et lancer cette boucle. Elle est d'ailleurs conçue pour que chaque itération (chaque gestion d'événement, donc) soit

encadrée d'un `NSAutoreleasePool`, ce qui est important dans le cycle de vie des objets (voir le chapitre 6 sur la gestion de la mémoire).

> Plus de détails sur la run-loop sont donnés dans le chapitre 9 « Le multithreading et les outils associés ».

## Différences 32/64 bits

La documentation de Cocoa fait parfois état d'une différence de comportement selon que le programme est compilé en 32 ou 64 bits. Même si cela peut sembler étrange, il y a au moins deux raisons à cela :

- le jeu d'instructions 64 bits du processeur peut effectivement permettre des améliorations pour certains mécanismes de base (les exceptions par exemple sont gérées différemment d'un point de vue technique entre le runtime 32 bits et le runtime 64 bits. Voyez le chapitre 8 traitant des exceptions pour en connaître les conséquences.) ;
- Apple a amorcé une transition vers le 64 bits. Les nouveaux développements du runtime Objective-C intégrés au système sont donc concentrés sur le 64 bits. Le 32 bits n'est préservé que pour maintenir une compatibilité, mais ne bénéficiera plus d'avancées majeures. Un exemple : les propriétés (voir chapitre 10 sur les propriétés) peuvent en 64 bits être « @synthétisées » en lien avec une variable d'instance implicite (non déclarée).

**POINT TECHNIQUE Cocoa et le 64 bits**

La migration vers le 64 bits a bien évidemment eu des incidences sur les API aussi. Par exemple, Cocoa propose les *`typedefs`* `NSInteger` et `NSInteger` pour les types de base que devraient être `int` et `unsigned int`. En effet, Apple a choisi le modèle LP64 où les `int` restent à 32 bits tandis que les `long` (et pointeurs) passent à 64 bits. Pour les types flottants c'est `CGFloat` qui adapte le `float` (le `double` est toujours à 64 bits).
Cela ne concerne pas directement le langage Objective-C, mais il est bon de le savoir pour pouvoir lire du code ou des documentations le mentionnant — ne serait-ce que le prototype de certaines méthodes de Cocoa.

## 32 et 64 bits côte à côte

Actuellement, un exécutable pour Mac OS X (lequel est basé sur un noyau *Mach*) peut contenir différentes sections de code compilées, une par architecture supportée, et cela au sein du même fichier. On appelle « binaires universels » (*universal binaries*) ceux qui peuvent s'exécuter sur toutes les architectures qu'a connu Mac OS X depuis son avènement : PowerPC 32 bits, PowerPC 64 bits, X86 (Intel 32 bits), X86_64 (Intel 64 bits).

Un code Objective-C peut être compilé séparément pour ces quatre architectures ; Xcode automatise d'ailleurs la fusion des compilations. Mais pour le moment, ces différents mondes ne se mélangent pas ; un programme 64 bits ne peut pas charger de greffons (plug-ins) 32 bits. Safari, ou Préférences Système, par exemple, sont souvent redémarrés en 32 bits pour supporter des plug-ins qui n'ont pas encore été mis à jour.

## Interopérabilité d'Objective-C avec d'autres langages

Si un programme écrit dans un langage donné est trop lent ou inadapté pour certaines sous-tâches spécifiques, il est toujours possible de scinder le code en plusieurs parties implémentées par différents langages. C'est la facilité d'interopérabilité de ces parties entre elles qui détermine la faisabilité de cette séparation.

En l'occurrence, Objective-C dispose d'un grand nombre de ponts vers d'autres langages.

- Bien évidemment, un code Objective-C peut être du simple C.
- Le compilateur peut permettre de mélanger du C++ et de l'Objective-C dans le même code ; on parle alors d'Objective-C++. Une section spécifique est consacrée à Objective-C dans le dernier chapitre de cet ouvrage, « Les évolutions d'Objective-C : performances et extensions du langage ».
- Le projet libre PyObjc implémente un pont Python/Objective-C.
- Les projets libres RubyObjc/RubyCocoa/MacRuby implémentent des ponts Ruby/Objective-C.
- Le projet libre MonoObjc implémente un pont Mono/Objective-C, Mono étant une implémentation libre de C#/.NET.
- Pendant longtemps, Apple a maintenu un pont entre Objective-C et Java, mais ce projet semble définitivement abandonné.

Moins pertinents pour ce qui est de l'interopérabilité, d'autres projets en rapport avec Objective-C peuvent être cités.

- Novell publie le kit MonoTouch, autorisant la programmation pour iPhone en C#/.NET. Le P/Invoke de .NET y permet même des échanges avec Objective-C ;
- À travers l'API WebKit fournie par Apple, Objective-C peut interpréter du JavaScript.
- Le projet libre Cappuccino a créé Objective-J, un langage basé sur JavaScript mais qui se veut le plus ressemblant possible à Objective-C/Cocoa.