Objective-C :
l'autre orientation objet de C

Objective-C a été le langage de programmation adopté par NeXT pour concevoir l'un des plus impressionnants systèmes d'exploitation : NEXTSTEP.
Il persiste aujourd'hui comme langage de référence de Mac OS X mais aussi comme choix de prédilection du projet GNUstep.

Quelques dizaines de minutes suffisent à faire un tour complet de ce langage qui revendique ses emprunts à Smalltalk et qui à son tour s'est révélé une grande source d'inspiration pour les concepteurs du langage Java.

Objective-C

Diablotin 2004

Histoire

- Simula (milieu des années 60)
- Smalltalk
- Objective-C
  - Stepstone corp. (début des années 80 - Brad J. Cox)
  - NeXT (NEXTSTEP)
  - NeXT / Sun (OPENSTEP)
  - GNUstep
  - Apple (Mac OS X)
- Java

Principales caractéristiques

- N’est PAS un langage objet MAIS orienté objet
- Liaison dynamique par défaut (contrairement au C++)
  - Dynamic binding (ou encore late binding)
- Capacités d’introspection
- Favorise l’utilisation d’un type Objet générique : l’id
  - La précision du typage dans la déclaration des objets est à la discrétion du programmeur
- Présente les mêmes déficiences de performance (temps d’exécution) que la plupart des autres langages objets ou orientés objets

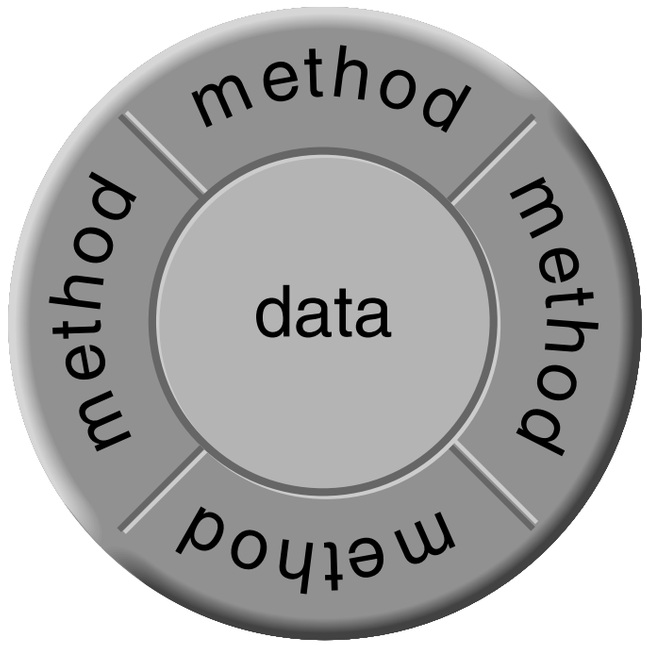
Dynamisme avant tout !

- Objective-C limite au maximum les prises de décisions à la compilation :
  - La détermination du type d'un objet peut être dynamique.
    Sa classe est trouvée durant l'exécution du programme.
  - Les liaisons sont dynamiques. Les méthodes à invoquer sont déterminées pendant l'exécution du programme. (Cela permet de faire évoluer un programme de manière incrementale.)
  - Des morceaux de programmes peuvent être chargés dynamiquement.

Concepts clefs

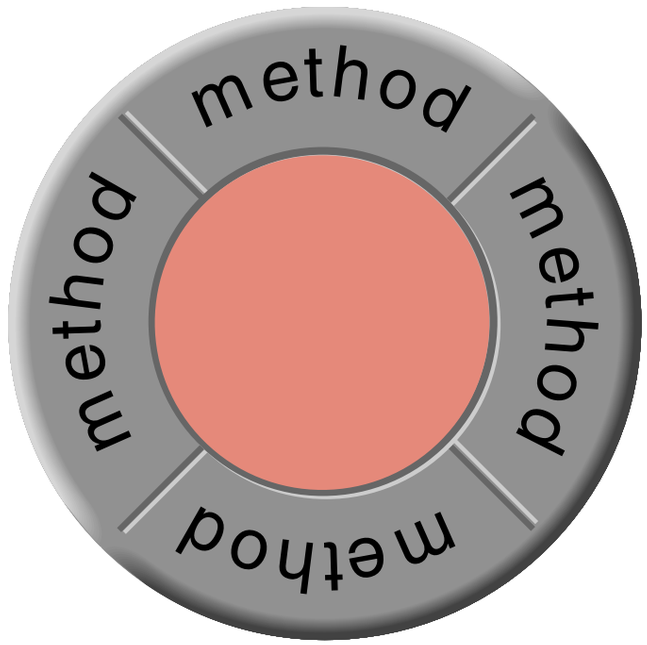
- Encapsulation
- Message
- Polymorphisme
- Héritage

Encapsulation



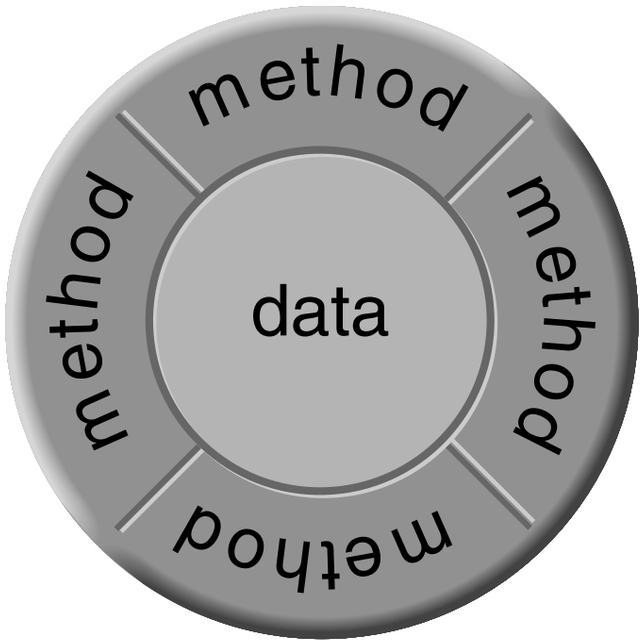
- Variables d’instance (data) : composante statique

Encapsulation



- Méthodes : composante dynamique (comportement)

De l’encapsulation à l’objet



- La combinaison de données et des moyens de les faire évoluer ou d’y accéder donne un Objet

Fabrique à objets

- La création d'objets passe par la définition de leurs moules (ou modèles) : ce sont les classes

Fabrique à objets

- La création d'objets passe par la définition de leurs moules (ou modèles) : ce sont les classes
- En Objective-C, tout comme en C, on distinguera une partie déclaration d'une partie implémentation
  (ceci est en partie dû à la nature compilée du langage)

Fabrique à objets

- La création d'objets passe par la définition de leurs moules (ou modèles) : ce sont les classes

- En Objective-C, tout comme en C, on distinguera une partie déclaration d'une partie implémentation
(ceci est en partie dû à la nature compilée du langage)

  - Ces deux parties prendront place respectivement dans des fichiers .h (Header) et .m (iMplementation)

Définition d'une classe (interface)

```
@interface Cercle
{
        float x;
        float y;
        float rayon;
}

- assigneX: (float) uneAbcisse;
- assigneY: (float) uneOrdonnee;
- assigneRayon: (float) unRayon;

- (float) x;
- (float) y;
- (float) rayon;

- bougeX: (float) deltaX etY: (float) deltaY;

- affiche;

@end
```

Définition d'une classe (implémentation)

```
#import "Cercle.h"

@implementation Cercle

- assigneX: (float) uneAbcisse
{
    x = uneAbcisse;
}

- assigneY: (float) uneOrdonnee
    …


- (float) x
{
    return x;
}

- (float) y
    …


- bougeX: (float) deltaX etY: (float) deltaY
{
    x += deltaX;
    y += deltaY;
}

…
```

Constatations

- Mots-clefs ajoutés par la couche Objective-C :
  - @interface, @implementation
  - @end
    - Tous sont préfixés par le symbole @
      (à la manière du symbole # employé par la préprocesseur)

Constatations

- Une méthode a un accès automatique (implicite) aux variables d'instance de l'objet receveur

```
- assigneRayon: (float) unRayon
{
      rayon = unRayon;
}
```

Constatations

- Une méthode peut porter le même nom qu'une variable d'instance de l'objet

```
@interface
{
      float rayon;
}

- (float) rayon;
```

Constatations

- Terminologie :
  - On appelle les méthodes de consultation et d’affectation des variables d’instance les accessor methods

Constatations

- Mot-clef ajouté au préprocesseur C :
  - #import
    - syntaxe strictement identique à #include
      (les deux peuvent coexister)

Constatations

- Le type par défaut retourné par les méthodes n’est pas l’int comme en C

Constatations

- Conventions typographiques :
  - on s’attache à débuter les noms des classes avec une majuscule (exemple : **C**lasse)
  - on utilise une initiale minuscule pour les noms de méthodes et les variables (ex. : **v**ariable)
  - on préfère employer des majuscules plutôt que des soulignements à l’intérieur des noms (exemples : **u**ne**V**ariable, **U**ne**C**lasse et non Une_Classe)

Amélioration

```
/* Cercle.m - version 2 */


// RECOURIR PLUS SYSTEMATIQUEMENT AUX
//       "ACCESSOR METHODS"


…

- bougeX: (float) deltaX etY: (float) deltaY
{
     [self assigneX: [self x] + deltaX];
     [self assigneY: [self y] + deltaY];
}

…
```

Constatations

- Indépendance :
  - plus de référence directe aux variables d’instance, par conséquent un gain de liberté quant au mode de stockage ou à la représentation des données de l’objet

Découvertes

- Les messages : LA syntaxe qu'Objective-C ajoute au C

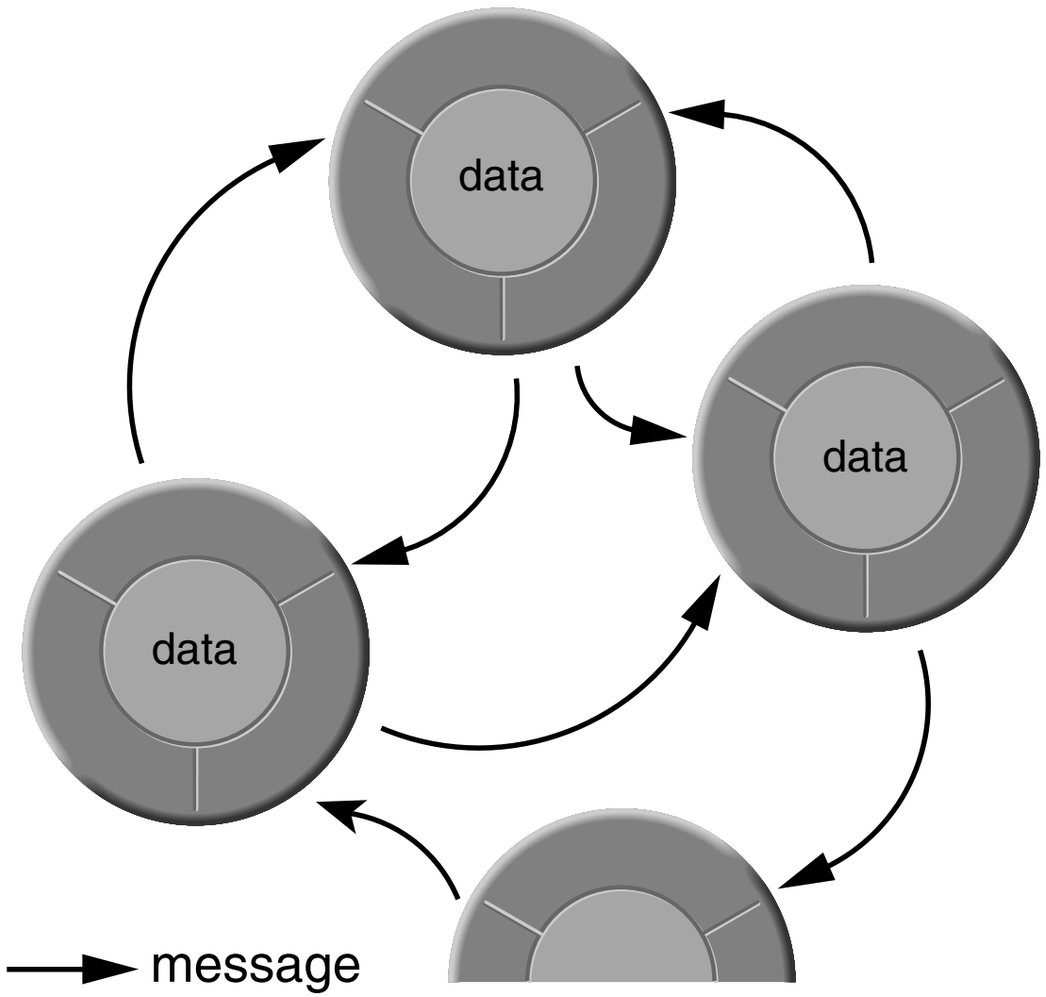
  [objet faitQuelqueChose]
  [objet faitQqChoseAvec: uneValeur]
  [objet faitAvec: uneValeur etAussi: uneSecondeValeur]

- L'objet à qui est adressé le message est appelé
  receveur

- Chaque paramètre est introduit par un mot-clef se terminant par ":" (le plus court admis est ":")

- Le nom composé formé du nom de la méthode et de ses différents mots-clefs est appelé
  sélecteur (exemple : **bougeX:etY:**)

Envoi de message



- Les données ne sont manipulées que par envoi de messages

Attention…

- … À l’ordre des mots-clefs, il N’est PAS libre !

  Si une interface décrit la méthode

  - peintEnRouge: (int) r vert: (int) v bleu: (int) b;

[unObjet peintEnRouge: 179 vert: 0 bleu: 0]; // OK !

[unObjet peintEnRouge: 179 bleu: 0 vert: 0];

Attention… (suite)

- … À l'ordre des mots-clefs, il N'est PAS libre !

  Si une interface décrit la méthode

  - peintEnRouge: (int) r vert: (int) v bleu: (int) b;

  [unObjet peintEnRouge: 179 vert: 0 bleu: 0]; // OK !

  [unObjet peintEnRouge: 179 bleu: 0 vert: 0];

- De plus, le nombre des mots-clefs est aussi contraint :

  toujours avec la méthode déclarée ci-dessus,

  [unObjet peintEnRouge: 179];

  le sélecteur est incomplet et par conséquent inconnu

Découvertes

- **self :**
  - une variable gérée automatiquement par le runtime Objective-C
  - une variable qui désigne systématiquement le receveur
    - permet à un objet de s'envoyer des messages à lui-même

      ```
      [self affiche];
      ```

    - permet à un objet de "se" retourner comme résultat d'une méthode et autorise par conséquent l'imbrication des messages (déconseillé à partir d'OpenStep)

      ```
      [[unCercle bougeX: 14.0 etY: 23.0] affiche];
      ```

Définition d'une classe (CORRECTIF dans l'implémentation)

```
#import "Cercle.h"

@implementation Cercle

- assigneX: (float) uneAbcisse
{
     x = uneAbcisse;
     return self;
}

     …


- (float) x
{
     return x;
}

     …


- bougeX: (float) deltaX etY: (float) deltaY
{
     [self assigneX: [self x] + deltaX];
     [self assigneY: [self y] + deltaY];
     return self;
}

…
```

Exploitation (… De la “fabrique”)

- La classe est constituée, qui détaille les attributs statiques des futurs objets (leurs variables d’instance) que leurs comportements dynamiques (les méthodes)

- Étape suivante, la création d’un objet, on parle aussi d’instanciation :

```
#import “Cercle.h”

Cercle unCercle = [Cercle new];          // juste bon
```

Exploitation (… De la “fabrique”)

- La classe est constituée, qui détaille les attributs statiques des futurs objets (leurs variables d’instance) que leurs comportements dynamiques (les méthodes)
- Étape suivante, la création d’un objet, on parle aussi d’instanciation :

```
#import “Cercle.h”

Cercle unCercle = [Cercle new];          // juste bon

id unCercle = [[Cercle alloc] init];     // MIEUX !
```

Constatations

- On privilégie l’emploi d’un type objet générique : id
  - Pour autant, l’emploi d’un type précis n’est pas prohibé

Constatations

- On préfère distinguer les actions d'allocation de celles d'initialisation (alloc, init) plutôt que de mélanger au sein d'une seule méthode (new)
- Cela facilite : la gestion des ressources
- Mais aussi la mise en œuvre des mécanismes de persistance
- Ou encore l'emploi d'objets distribués

Constatations

- Le receveur du message alloc dans

```
id unCercle = [[Cercle alloc] init];
```

est une classe et non un objet, ce qui signifie qu'il existe aussi, spécifiquement, des méthodes de classe

Méthodes de classe (interface)

```
@interface Cercle
{
        float x;
        float y;
        float rayon;
}

+ (float) numeroDeVersion;

- assigneX: (float) uneAbcisse;
- assigneY: (float) uneOrdonnee;
- assigneRayon: (float) unRayon;

- (float) x;
- (float) y;
- (float) rayon;

- bougeX: (float) deltaX etY: (float) deltaY;

- affiche;

@end
```

Méthodes de classe (implémentation)

```
#import “Cercle.h”

@implementation Cercle

+ (float) numeroDeVersion
{
     return 1.27;
}

- assigneX: (float) uneAbcisse
{
     x = uneAbcisse;
     return self;
}

…
```

Méthodes de classe

- l’ajout qui vient d’être fait permet d’écrire

```
printf (“Classe Cercle v%f\n”, [Cercle numeroDeVersion]);
```

Constatations

- On note qu'Objective différencie la définition des méthodes d'objet et de classe juste par l'emploi des symboles - et + respectivement

  - (type) methodeDObjet;

  + (type) methodeDeClasse;

Terminologie

- Les méthodes de classe sont aussi appelées Factory Methods

Ce nom dénote l'emploi le plus courant de ces méthodes, à savoir la "construction" de nouveaux objets.

Polymorphisme

- Ou le don d’appeler deux actions différentes par le même nom !

- Un exemple valant mieux qu’un long discours…

Programmation procédurale traditionnelle

```
affiche_rectangle (struct rectangle *
unRectangle…

affiche_cercle (struct cercle * unCercle…
```

Programmation procédurale traditionnelle (alternative)

```
typedef enum { RECTANGLE, CERCLE } FIGURE;

struct figure {
                    float x, y;
                    float largeur, hauteur;
                    float rayon;
                };

affiche (FIGURE typeFigure,
                    struct figure * uneFigure…
                …

    switch (typeFigure)
    {
        case RECTANGLE:
                    …
                    break;
        case CERCLE:
                    …
                    break;

struct figure leRectangle = {
            50.0, 100.0, 25.0, 25.0
        };

struct figure leCercle = {
            60.0, 95.0, 0.0, 0.0, 18.0
        };

affiche (RECTANGLE, &leRectangle…

affiche (CERCLE, &leCercle…
```

Programmation procédurale traditionnelle (alternative 2)

```
typedef enum { RECTANGLE, CERCLE } FIGURE;

struct figure {
                    FIGURE type;
                    float x, y;
                    float largeur, hauteur;
                    float rayon;
                };

affiche (struct figure * uneFigure…
                …

    switch (uneFigure->type)
    {
        case RECTANGLE:
                …
                break;
        case CERCLE:
                …
                break;


struct figure leRectangle = {
          RECTANGLE, 50.0, 100.0, 25.0, 25.0
       };

struct figure leCercle = {
          CERCLE, 60.0, 95.0, 0.0, 0.0, 18.0
       };

affiche (&leRectangle…

affiche (&leCercle…
```

Polymorphisme (en POO)

```
@interface Rectangle
{
        float x, y, largeur, hauteur;
}
- assigneX: (float) unX y: (float) unY;
- assigneLargeur: (float) uneLargeur          \
          hauteur: (float) uneHauteur;
- affiche;
@end

@interface Cercle
{
        float x, y, rayon;
}
- assigneX: (float) unX y: (float) unY;
- assigneRayon: (float) unRayon;
- affiche;
@end


     …


id unRectangle = [[Rectangle alloc] init];
id unCercle = [[Cercle alloc] init];

[unRectangle assigneX: 3.0 y: 8.0];
[unCercle assigneX: 60.0 y: 95.0];

[unRectangle affiche];
[unCercle affiche];
```

Héritage (… Ou comment ne pas réinventer la roue !)

- L’on remarque dans l’exemple qui précède que les deux classes Cercle et Rectangle possèdent des attributs (variables d’instances) et comportements (méthodes) non pas seulement similaires mais strictement IDENTIQUES
- Le concept d’héritage vient à la rescousse…

Héritage (mise en œuvre)

```
#import <objc/Object.h>

@interface Figure2D : Object
{
     float x;
     float y;
}

- definitPositionX: (float) uneAbcisse      \
         positionY: (float) uneOrdonnee;

- affiche;

@end
```

Héritage (mise en œuvre - suite)

```
#import “Figure2D.h”

@interface Rectangle : Figure2D
{
        float largeur, hauteur;
}
- definitLargeur: (float) uneLargeur          \
          hauteur: (float) uneHauteur;
- affiche;
@end

@interface Cercle : Figure2D
{
        float rayon;
}
- definitRayon: (float) unRayon;
- affiche;
@end


id unRectangle = [[Rectangle alloc] init];

[unRectangle definitPositionX: 60.0
                     positionY: 20.0];
[unRectangle definitLargeur: 15.0
                     hauteur: 18.0];
[unRectangle affiche];
```

Constatations

- Avec l'héritage, on crée des sous-classes autorisant :
  - l'ajout de méthodes
  - la redéfinition de méthodes
  - l'ajout de variables d'instance
- Par contre, on NE peut PAS :
  - redéfinir une variable d'instance
  - retirer quoi que ce soit, ni variable d'instance, ni méthode
  - hériter de plusieurs classes parentes

Question

- Comment procéder si l’on veut juste “enrichir” une méthode et non la redéfinir ?

Exemple avec la méthode init

```
#import <objc/Object.h>

@interface Figure2D : Object
{
     float x;
     float y;
}

- init;

- definitPositionX: (float) uneAbcisse       \
          positionY: (float) uneOrdonnee;

- affiche;

@end


/* ------ FICHIER Figure2D.m ------ */

#import “Figure2D.h”

@implementation Figure2D

- init
{
     x = y = 100.0;
     return self;
}

…
```

Exemple avec la méthode init

```
#import “Figure2D.h”

@interface Cercle : Figure2D
{
        float rayon;
}

- init;

- definitRayon: (float) unRayon;

- affiche;

@end
```

```
/* ------ FICHIER Cercle.m ------ */

#import “Cercle.h”

@implementation Cercle

- init
{
     x = y = 100.0;
     rayon = 25.0;

     return self;
}

…
```

LA BONNE SOLUTION !

```
/* ------ FICHIER Cercle.m ------ */

#import “Cercle.h”

@implementation Cercle

- init
{
     [super init];

     rayon = 25.0;

     return self;
}

…


// NE PAS OUBLIER DE FAIRE DE MÊME DANS LA
// METHODE init DE LA CLASSE Figure2D AFIN
// DE LAISSER UNE CHANCE À LA METHODE init
// DE LA CLASSE Object D’INITIALISER LES
// VARIABLES D’INSTANCE QUI SONT DEFINIES
// AU NIVEAU DE CETTE DERNIERE !
```

Réponse

- **super :**
  - une autre variable gérée automatiquement par le runtime Objective-C
  - une variable qui désigne systématiquement le receveur
    - permet à un objet de s'envoyer des messages à lui-même, comme pour self, MAIS avec recherche de la méthode à exécuter À PARTIR de la super-classe (la classe parente de celle dans laquelle est définie la méthode qui fait référence à super) et en remontant dans l'arbre d'héritage

```
[super affiche];
```

Héritage multiple

- Pour pallier l'absence de l'héritage multiple, des pointeurs sur des objets tiers sont employés
  - Exemple (emprunt aux "kits" OpenStep) :
    - NSButton est une sous-classe de NSControl
    - NSControl a besoin des attributs et comportements de la classe NSCell
    - NSControl inclut dans ses variables d'instance une référence (un pointeur) vers un objet de type NSCell
    - NSControl définit un ensemble de méthodes qui ne sont que des relais vers les méthodes éponymes de la classe NSCell et envoyé à l'objet de type NSCell sus mentionné

Héritage multiple (extrait du fichier NSControl.h)

```
#import <AppKit/NSView.h>


@interface NSControl : NSView
{
    /*All instance variables are private*/
    int _tag;
    id _cell;
        :
}


- (id)initWithFrame:(NSRect)frameRect;
- (void)sizeToFit;
- (id)cell;
- (void)setCell:(NSCell *)aCell;
- (id)target;
- (void)setTarget:(id)anObject;
- (SEL)action;
- (void)setAction:(SEL)aSelector;
- (int)tag;
- (void)setTag:(int)anInt;
```

Héritage multiple (fin)

- On remarque clairement la répartition des rôles entre les deux types d'objets qui se retrouvent combinés pour former un bouton :
  - NSControl (sous-classe de NSView) gère l'apparence
  - NSCell gère le comportement
- Note : il est possible de recourir à la méthode forward:: pour simuler l'héritage multiple

Astuces & techniques avancées

- Variables de classe :
  - elles n’existent pas !

    Qu’à cela ne tienne, il suffit de faire comme si…

“Pseudo” variable de classe

```
#import <objc/Object.h>

@interface Figure2D : Object
{
     float x;
     float y;
}

+ (int) nombreFigures;

- init;
- free;


…
```

“Pseudo” variable de classe

```
/* ------ FICHIER Figure2D.m ------ */

#import “Figure2D.h”

static int nombreDeFigures2D = 0;

@implementation Figure2D

+ (int) nombreFigures
{
     return nombreDeFigures2D;
}

- init
{
     [super init];
     x = y = 100.0;

     ++nombreDeFigures2D;

     return self;
}

- free
{
     --nombreDeFigures2D;

     return [super free];
}

…
```

“Pseudo” variable de classe (usage)

```
#import “Figure2D.h”

…

     int nbFigures = [Figures2D nombreFigures];
```

Astuces (suite)

- Méthodes privées :
  - elles n’existent pas !

    Qu’à cela ne tienne, il suffit de…

    … Ne pas en dévoiler les prototypes dans le fichier d’entête (.h) correspondant
  - Bien entendu, cela n’empêche en aucun cas les programmeurs “futés” d’invoquer ces méthodes
    - C’est tout juste la politique de l’autruche

Technique avancée

- L’introspection :
  - un petit ensemble de méthodes(définies essentiellement dans la classe racine Object) permettent à chaque objet, ou classe, de se dévoiler

```
printf (“Nom de la classe : %s\n”, [uneClasse name]);
printf (“Version de la classe : %s\n”, [uneClasse version]);
```

Et aussi instanceMethodFor:, methodFor:, isKindOf:, isKindOfClassNamed:, isMemberOf:, isMemberOfClassNamed:, isInstance, isClass, isMetaClass, instancesRespondTo:, RespondsTo:, descriptionForInstanceMethod:, descriptionForMethod:, class, superClass, metaClass

Technique avancée

- Gain de vitesse :
  - … Ou comment modérer le tempérament trop dynamique d'Objective-C quand il se révèle par trop pénalisant en temps d'exécution

```
long int indice = 100000000;
id (*evalue) (id, SEL, int);

while (indice--)
{
	evalue (unCercle, @selector (definitRayon:), 10.0);
	// est équivalent à [unCercle definitRayon: 10.0]
}
```

Technique avancée

- Résolution dynamique (“pointeur” sur méthode) :

```
id unObjet = monCercle;          // presque aussi facile en C++ :-)
SEL methode = @selector (rayon);     // plus aussi simple :-(

printf ("Rayon : %f\n", [unObjet perform: methode]);
```

ANNEXES

Mise en œuvre sur Linux

- Installation des paquetages gcc, gcc-objc et libobjc
- Compilation :

  cc -o executable sources.m -lobjc

  gcc [-Wno-import] -o executable sources.[m|c] … -lobjc [-lpthread]

- Le compilateur le plus répandu permettant de produire du code exécutable à partir de sources Objective-C est sans conteste gcc.

Mélanges

- Objective-C et C :
  - Le mariage n’est pas seulement possible, il est tout simplement implicite. Objective-C n’est véritablement qu’une couche ajoutée au langage C.
- Objective-C et C++ :
  - Pourquoi faire ?
  - On parlera plus de cohabitation ou coexistence que de mariage. Les représentations internes des objets manipulés par ces deux surcouches ne sont pas identiques.

Mise au point

- Ne pas confondre le langage en lui-même avec les environnements (frameworks) qui l'emploient.
  - Par exemple, il ne faut pas penser qu'OPENSTEP ou GnuStep sont des variantes d'Objective-C.

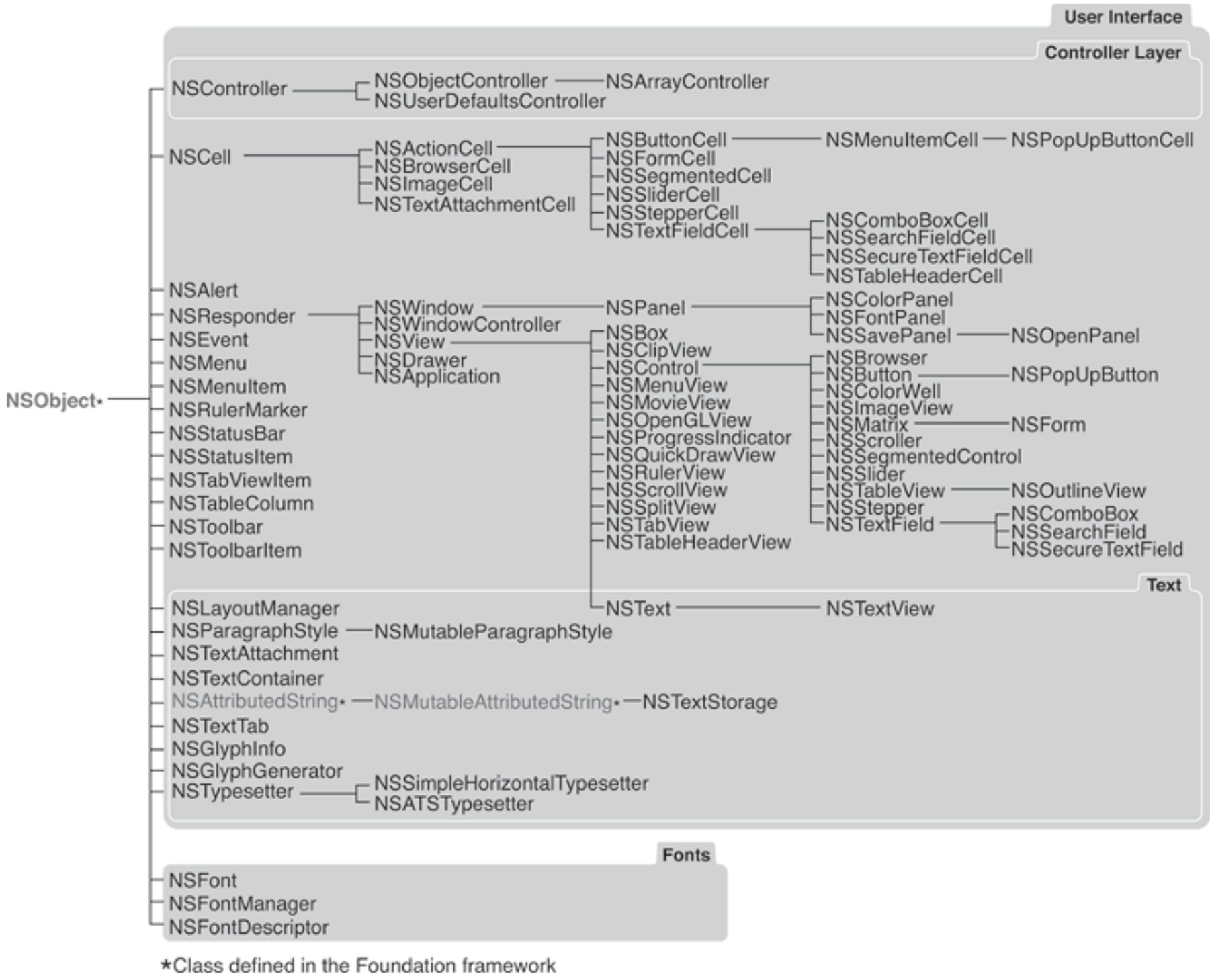
Humeurs et opinions

- Le programmeur C qui connaît Objective-C ne peut aimer le C++.
- Le programmeur C qui aime le C++ est celui qui n’a pas encore découvert Objective-C !

Diffusion

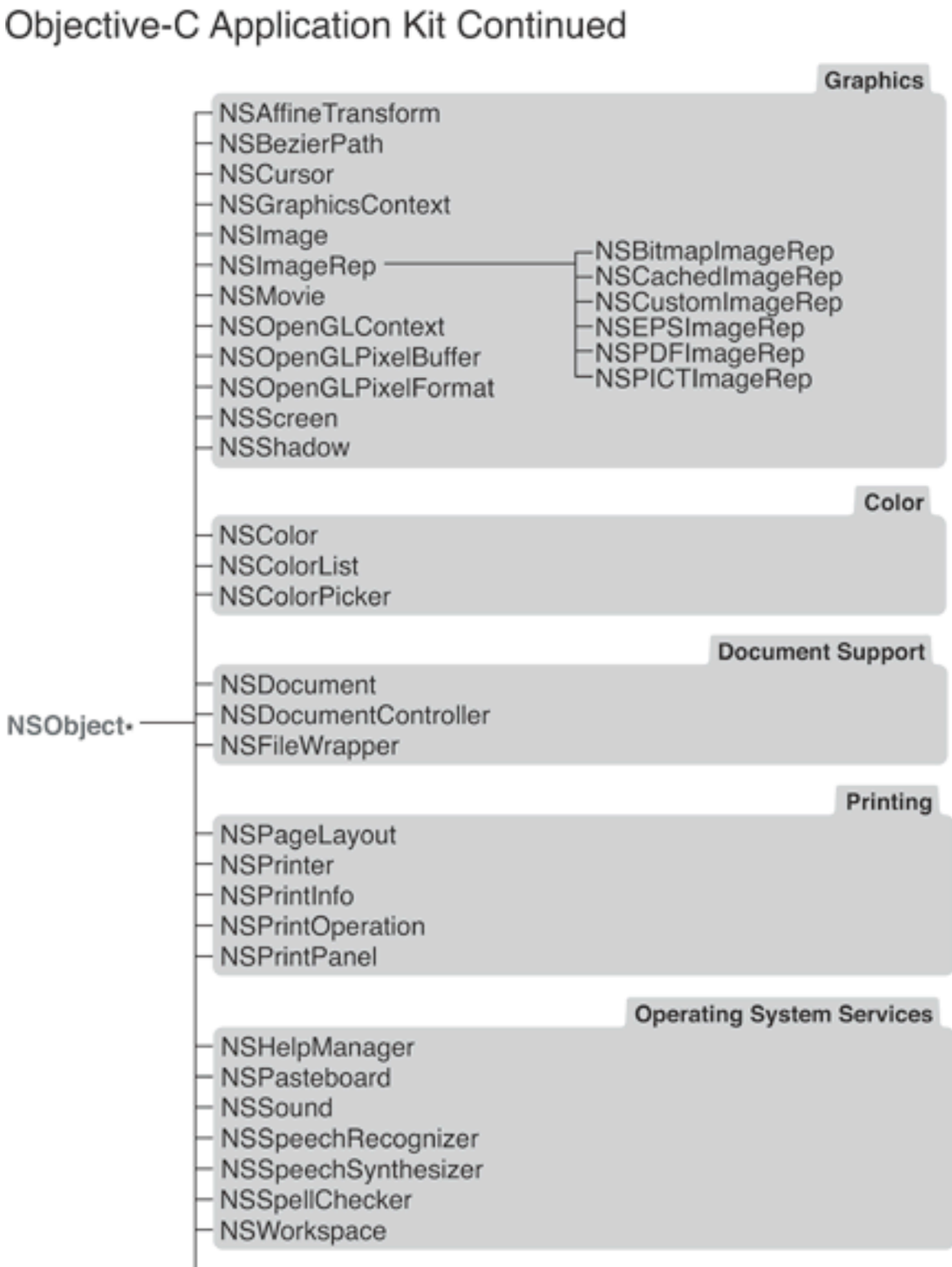
- Quelle est la couverture d'Objective-C à ce jour dans le monde informatique ?
  - Bien faible à vrai dire :
  - Mac OS X (successeur de NEXTSTEP)
  - GNUstep (implémentation libre d'OPENSTEP)
  - WebObjects (un autre produit Apple racheté avec NeXT)
  - GNUstep Web (implémentation libre de WebObjects)
  - Potentiellement tout système d'exploitation sur lequel on peut employer le compilateur gcc

Classes de l'Application Kit (Mac OS X)



*Class defined in the Foundation framework

Application Kit (suite de l'extrait)

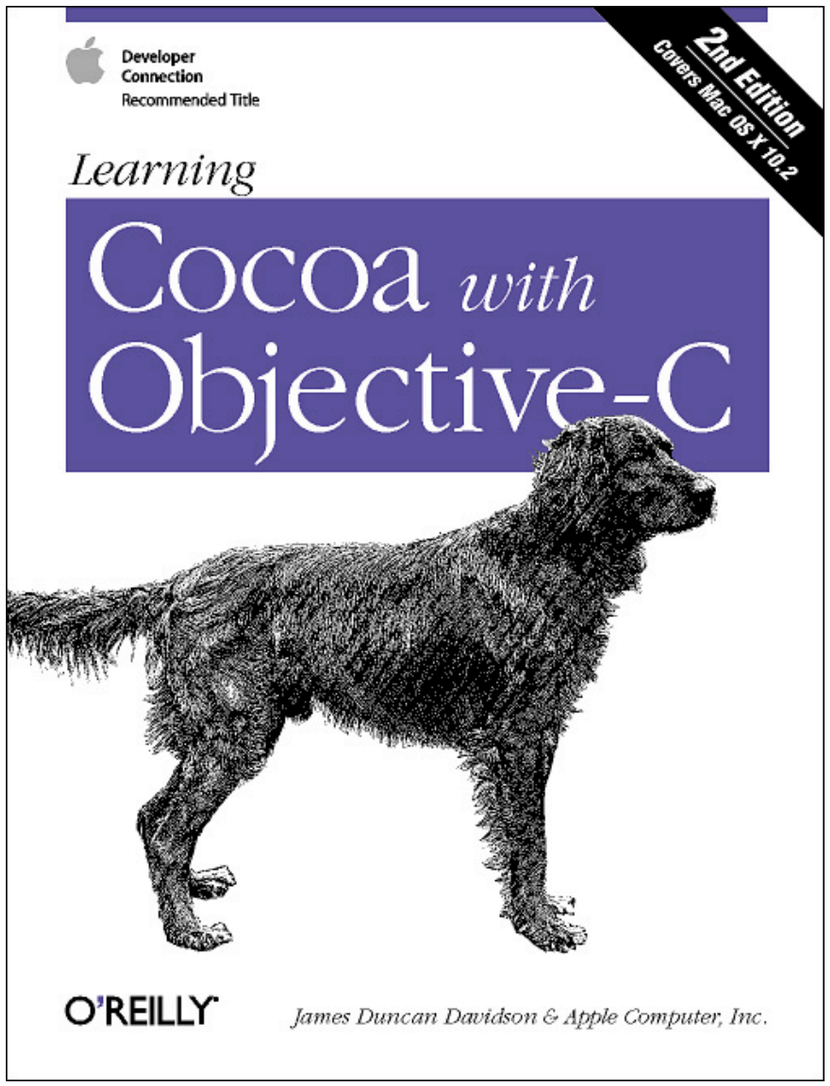


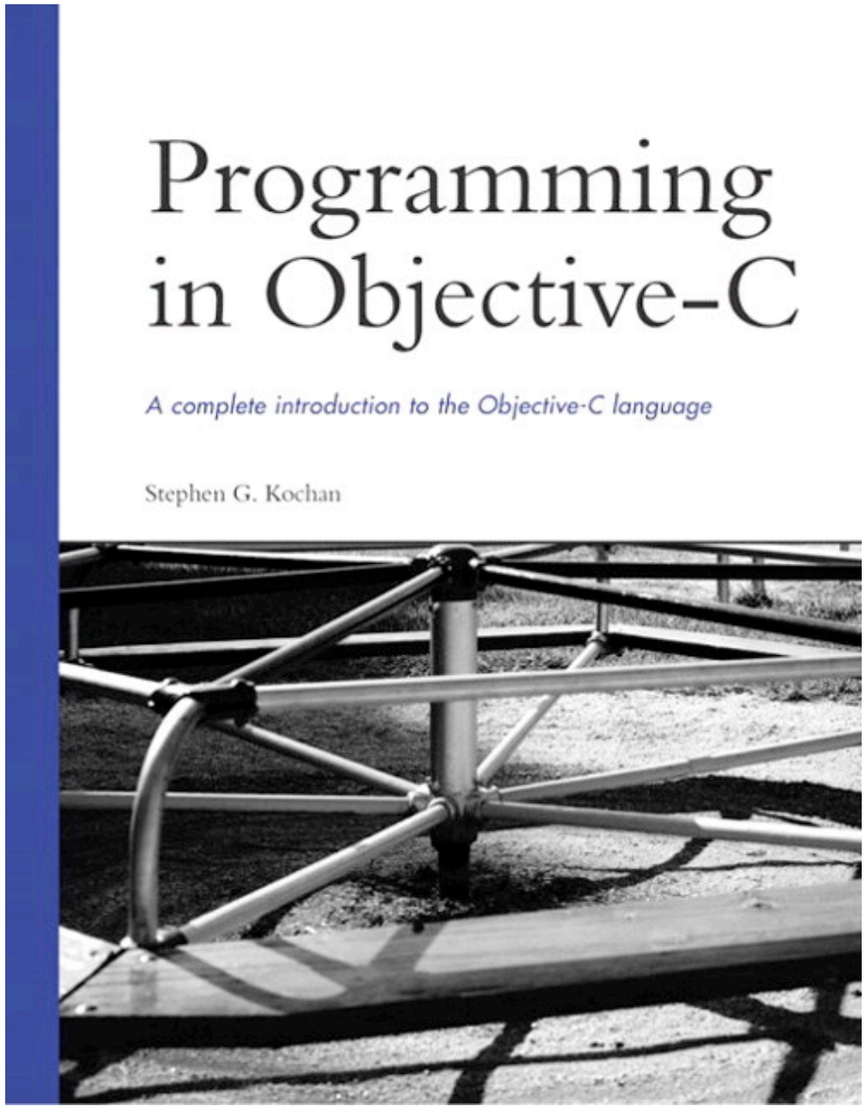
Comparatif

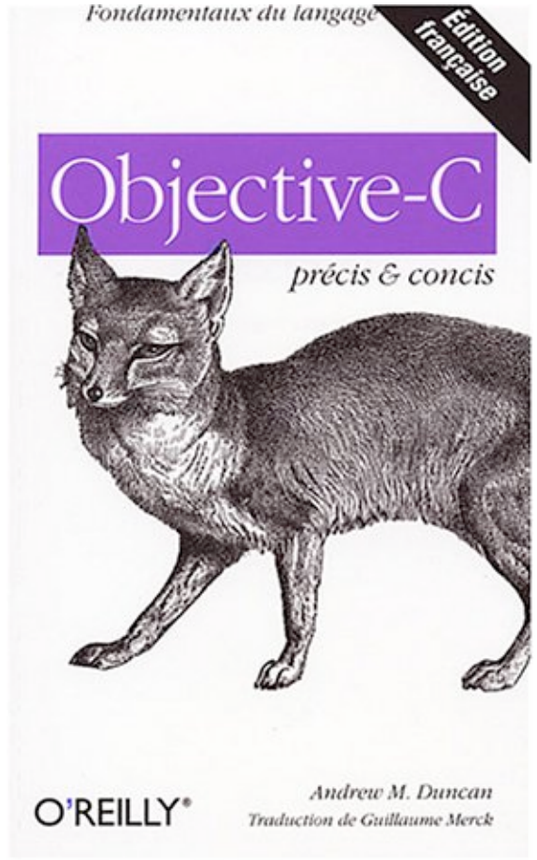
Caractéristique	Objective-C	Smalltalk-80	C++	Eiffel
typage	dynamique	dynamique	statique	statique
édition des liens dynamique	explicite	implicite	virtual (mot-clef)	OUI
accès aux noms des méthodes durant l'exécution	OUI	OUI	NON	NON
accès aux noms des classes durant l'exécution	OUI	OUI	NON	NON
accès aux noms des variables d'instance durant l'exécution	OUI	OUI	NON	NON
redirection (forwarding)	OUI	OUI	NON	NON
métaclasses	OUI	OUI	NON	NON
héritage	simple	simple	multiple	multiple
instantiation	explicite	explicite	new (mot-clef)	create (mot-clef)
accès aux méthodes de la super-classe	super	super	::	renaming
classe racine	Object (NSObject / Cocoa)	Object	multiple	multiple
nom du receveur	self	self	this	current
données privées	OUI	OUI	OUI	OUI
méthodes privées	NON	NON	OUI	OUI
variables de classe	NON	OUI	OUI	NON
"ramasse-miettes"	NON	OUI	NON	OUI

Ressources / Livres

- Chez O'Reilly
- Chez Sams







Ressources

- Retrouvez quelques documents sur

http://diablotin.info/objc/
http://www.gnustep.org
http://www.gnustepweb.org/
http://www.onlamp.com
http://www.apple.com
http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/
http://www.dekorte.com/Objective-C/
http://www.toodarkpark.net/computers/objc/
http://www.swarm.org/resources-objc.html

http://formation.diablotin.com