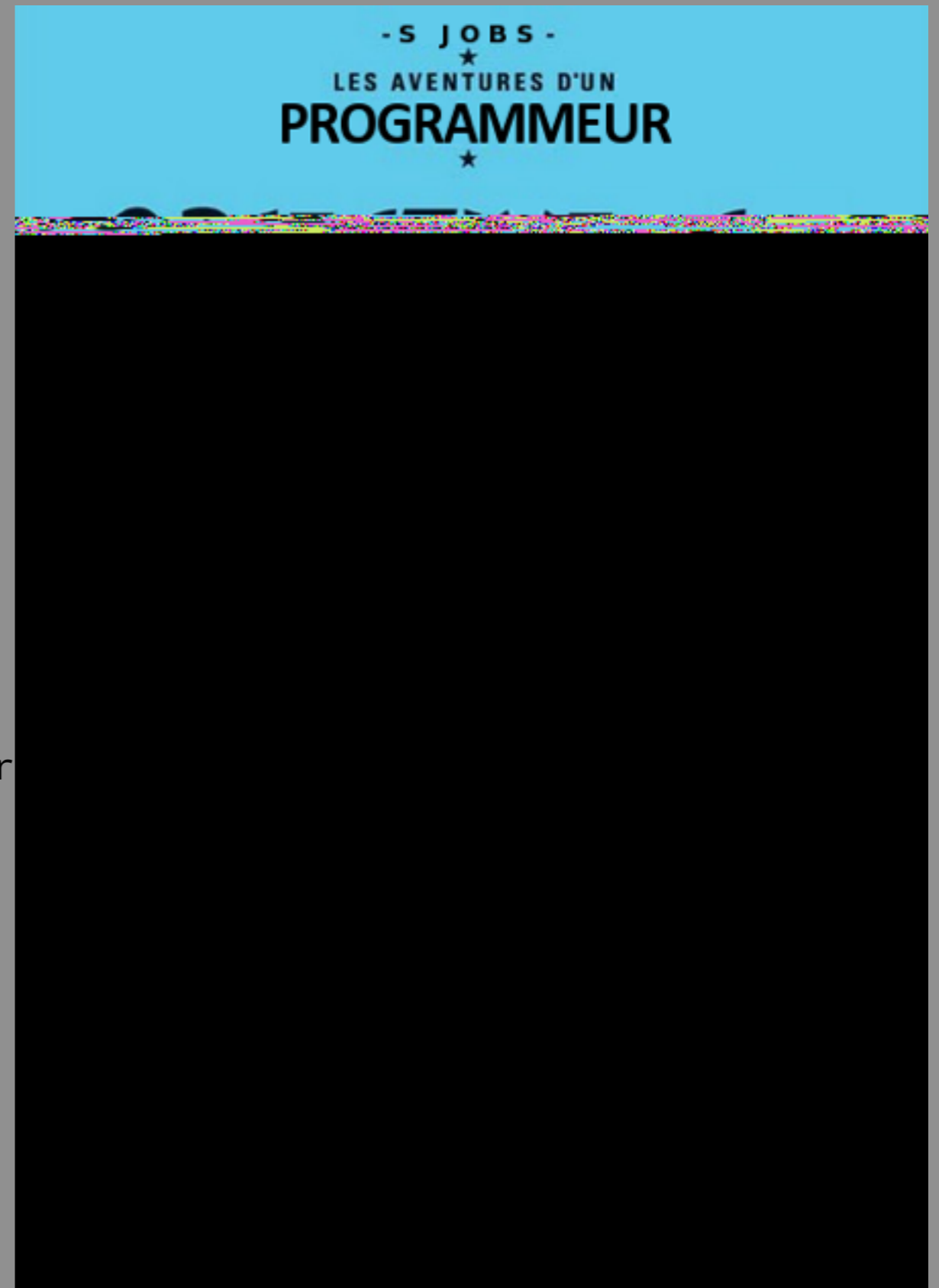
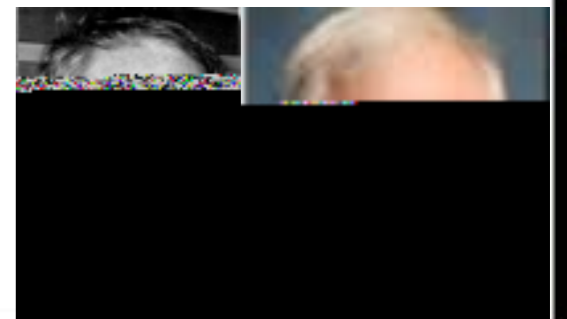


Objective-C

Jean-Baptiste.Yunes@univ-paris-diderot.fr
2012—2013



- deux extensions objet à C
 - C++ (Stroustrup)
 - Objective-C (Cox & Love)
- années 80
- des succès différents



- Objective-C est un sur-ensemble de C
- inspiré de smalltalk
 - typage dynamique
 - chargement dynamique
 - dispatch dynamique

- utilisé dans NextSTEP
- puis MacOSX
- puis iOS
- le langage « natif » mac

- a récemment évolué en Objective-C 2.0
- la documentation de référence

<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>

La syntaxe et la terminologie

Texte

- ObjC utilise une terminologie « pure »
 - objet
 - interface
 - implémentation
 - message
 - sélecteur
 - ...

- typique ObjC est l'expression d'envoi de message :

```
[maPile push:uneValeur];
```

- le receveur :

```
[maPile push:uneValeur];
```

- le message :

```
[maPile push:uneValeur];
```

- le sélecteur :

```
[maPile push:uneValeur];
```

- la méthode est le code qui sera finalement sélectionné...

Message expression

Message

Selector or keyword

[receiver **method:** parameter]

- une syntaxe spéciale pour les accesseurs

```
int x = [image sizeX];
```

peut s'écrire avantageusement :

```
int x = image.sizeX;
```

- même si `image` est un pointeur...

Rappel : un accesseur est une méthode permettant d'accéder à un attribut

Attention : en ObjC un getter pour un attribut de nom `couleur` se nomme simplement `couleur`, le setter se nomme `setCouleur`

- une syntaxe spéciale pour les accesseurs

```
[image setSizeX:1200];
```

peut s'écrire avantageusement :

```
image.sizeX = 1200;
```

Rappel : un accesseur est une méthode permettant d'accéder à un attribut

Attention : en ObjC un getter pour un attribut de nom couleur se nomme simplement couleur, le setter se nomme setCouleur

- deux typages

- dynamique :

`id uneReference;`

- statique

`Stack *maPile;`

- vérification des types à la compilation

- liaison dynamique

- la référence universelle nulle, mot-clé `nil`
`id uneReference;`
`uneReference = nil;`
`Stack *maPile = nil;`
- que ce passe t-il si l'on envoie un message à `nil` ?
`compte = nil;`
`[compte depose:1000000];`
- pas d'erreur! les valeurs éventuellement retournées sont toujours 0

- le type B00L est habituellement utilisé pour les booléens
- mots-clés YES, NO
- idem 0 ou 1

```
B00L test = (i==1);
```

```
if (test) ... else ...
```

```
if (!test) ... else ...
```

- il existe un type pour les sélecteurs : SEL
`SEL monSelecteur = @selector(changeName);`
- lequel peut être utilisé pour envoyer un message :
`[jack performSelector:monSelecteur withObject:@"dalton"];`
- équivalent de :
`[jack changeName:@"dalton"];`
- c'est l'équivalent ObjC des pointeurs sur fonctions du C

- il existe un type chaîne de caractères ObjC
 - type NSString
- dont les littéraux sont précédés du caractère @
 - attention : il s'agit bien d'objets à ne pas confondre avec les « chaînes » du C
 - @“je suis un objet chaîne ObjC”
 - “je suis un simple char *”

Les classes

- Les classes ObjC sont des prototypes pour leurs instances
- ObjC autorise l'héritage
- NSObject est une classe prédéfinie qu'il est absolument conseillé d'employer comme classe racine (sous peine de malfunctions graves)

- Objc autorise l'introspection
- les objets-classes ont un type correspondant :

```
Class *c = [monObjet class];
```

- on peut tester l'appartenance d'un objet à une classe :

```
[objet1 isKindOfClass:UneClasse];
```

```
[objet2 isKindOfClass:UneAutreClasse];
```

- l'instanciation ObjC est particulière car elle distingue :
 - **l'allocation** et,
`id ceb = [CompteEnBanque alloc];`
 - **l'initialisation** :
`[ceb initWithEuros:1000];`
 - que l'on écrit généralement en une seule expression
`id ceb = [[CompteEnBanque alloc] initWithEuros:1000];`
- l'allocation est une factory statique de la classe
- l'initialisation est une méthode d'instance

- les variables de classe n'existent pas
- il faut se débrouiller avec le C (variables static)
- pour l'initialisation d'une classe, il existe la méthode statique `initialize` qui est appelée au moins une fois avant tout autre appel de méthode
- attention sa définition nécessite de garantir que le code ne soit exécuté qu'une seule fois à l'aide de l'idiome (on comprendra plus tard) :

```
+initialize {  
    if (self == [MaClasse class]) {  
        // initialisation de classe  
    }  
}
```

- la définition d'une classe ObjC s'effectue en deux étapes :
- la définition de son **interface**
 - relation avec les autres classes (héritage, ...)
 - propriétés et sélecteurs
- son **implémentation**
 - variables d'instances (ivars)
 - code ou synthèse des méthodes

- une **interface** ObjC (dans fichier .h)

```
#import "SaSuperClasse.h"
```

```
@interface MaClasse : SaSuperClasse
```

```
// propriétés
```

```
// sélecteurs
```

```
@end
```

- on notera l'usage du mot-clé `import` en lieu et place d'`include`

- une **implémentation** ObjC (dans fichier .m)

```
#import "MaClasse.h"
```

```
@implementation MaClasse
```

```
{
```

```
    // variables d'instance
```

```
}
```

```
// implémentation des méthodes
```

```
@end
```

- les méthodes peuvent être statiques ou d'instances
- statiques si le sélecteur est précédé du signe +
- d'instances si le sélecteur est précédé du signe –

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

@interface UneClasse : NSObject

+ (id)createWithValue:(int)v;

- (id)initWithValue:(int)v;
- (int)value;

@end
```

```
//  UneClasse.m
#import "UneClasse.h"
@implementation UneClasse
{
    int value;
}

- (id)initWithValue:(int)v {
    self = [super init];
    if (self) {
        NSLog(@"init");
        value = v;
    }
    return self;
}

- (int)value {
    return value;
}

+ (id)createWithValue:(int)v {
    return [[self alloc] initWithValue:v];
}

@end
```

```
//  main.m
#import <Foundation/Foundation.h>
#import "UneClasse.h"

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"Hello, World!");
        UneClasse *obj = [[UneClasse alloc] initWithValue:10];
        NSLog(@"%d",obj.value);
        UneClasse *obj2 = [UneClasse createWithValue:20];
        NSLog(@"%d",obj2.value);
    }

    return 0;
}
```

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- `self` est le pointeur sur l'instance typée avec sa classe de définition
- `super` est le pointeur sur l'instance typée avec la super-classe de sa définition
- attention : les classes sont aussi des objets, `self` et `super` ont aussi un sens dans une méthode de classe...
- attention : `self` est une variable (une vraie! Elle peut-être modifiée!)

Les protocoles

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- un **protocole** ObjC correspond à une interface Java
- cela ne définit qu'un protocole objet
 - un contrat à remplir auquel on peut se conformer
- on parle de protocoles formels (les catégories - voir plus loin - peuvent faire office de protocoles informels)

```
// définition d'un protocole
// Printable.h

#import <Foundation/Foundation.h>

@protocol Printable <NSObject>

- (id)print;

@end
```

```
// adoption d'un protocole  
//  UneClasse.h
```

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

```
@interface UneClasse : NSObject <Printable>
```

```
+ (id)createWithValue:(int)v;
```

```
- (id)initWithValue:(int)v;
```

```
- (int)value;
```

```
@end
```

```
// l'implémentation nécessite la définition de la méthode du protocole
//  UneClasse.m
#import "UneClasse.h"

@implementation UneClasse
{
    int value;
}
- (id)initWithValue:(int)v {
    self = [super init];
    if (self) {
        NSLog(@"init");
        value = v;
    }
    return self;
}
- (int)value {
    return value;
}
+ (id)createWithValue:(int)v {
    return [[self alloc] initWithValue:v];
}
- (id)print {
    NSLog(@"my value is %d",self.value);
    return self;
}
@end
```

```
// un usage possible
//  main.m
#import <Foundation/Foundation.h>
#import "UneClasse.h"

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        id<Printable> obj2 = [UneClasse createWithValue:20];
        [obj2 print];
    }
    return 0;
}
```

- les sélecteurs définis dans un protocole peuvent être qualifiés dans des sections :
 - `@required`, par défaut, les classes s'y conformant doivent définir les méthodes
 - `@optional`, aucune obligation
- pour l'introspection on dispose de :
`[unObjet conformsToProtocol:@protocol(UnProtocole)];`
- un protocole peut incorporer d'autres protocoles (hiérarchie de types)

Les propriétés

- Les propriétés permettent de déclarer des attributs (en fait des accesseurs)
- Elles peuvent être déclarées dans
 - une interface
 - un protocole
 - une catégorie

- La déclaration d'une propriété :
`@property (attributs) type nom;`
- correspond à la déclaration des accesseurs :
 - `(type)nom;`
 - `(void)setNom: (type)newNom;`
- bien entendu, les attributs raffinent cette déclaration

- Les attributs de nommage, permettent de choisir le nom des accesseurs :
 - `getter=nom_de_la_méthode`
 - `setter=nom_de_la_méthode`
- Les attributs d'accès :
 - `readwrite` (attribut par défaut)
 - `readonly` (bien entendu interdit l'existence d'un setter)

- Les attributs sémantiques :
 - `strong` : notion de propriété
 - `weak` : pas de lien fort
 - `copy` : signifie que l'affectation s'effectue avec une copie de l'argument, l'ancienne valeur est relâchée (nécessite le protocole `NSCopy`)
 - `assign` : simple affectation (par défaut)
 - `retain` : l'ancienne valeur est relâchée et la nouvelle est retenue

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- L'attribut d'atomicité
 - `nonatomic`
- par défaut c'est atomique, donc thread-safe

- L'implémentation d'une propriété s'effectue via `@synthesize` ou `@dynamic`
 - `@synthesize nom, ..., nom1=nom2, ...;`
 - permet d'obtenir la synthèse automatique des propriétés nommées correspondantes, le second cas permet de synthétiser `nom1` en utilisant le support de la variable `nom2`
 - `@dynamic` permet d'indiquer que les accesseurs seront fournis par un autre moyen (en général dynamique). À n'utiliser qu'en connaissance de cause
 - ou « manuellement »

```
// déclaration d'une propriété
//  UneClasse.h
#import <Foundation/Foundation.h>

@interface UneClasse : NSObject

@property () int value;

+ (id)createWithValue:(int)v;

- (id)initWithValue:(int)v;

@end
```

```
//  
//  UneClasse.m  
#import "UneClasse.h"  
@implementation UneClasse  
@synthesize value;  
- (id)initWithValue:(int)v {  
    self = [super init];  
    if (self) {  
        NSLog(@"init");  
        value = v;  
    }  
    return self;  
}  
+ (id)createWithValue:(int)v {  
    return [[self alloc] initWithValue:v];  
}  
@end
```

```
//  
//  main.m  
#import <Foundation/Foundation.h>  
#import "UneClasse.h"  
  
int main(int argc, const char * argv[])  
{  
    @autoreleasepool {  
        UneClasse *obj = [UneClasse createWithValue:20];  
        obj.value = 34;  
        NSLog(@"%d",obj.value);  
        [obj setValue:45];  
        NSLog(@"%d",[obj value]);  
    }  
    return 0;  
}
```

- La notation pointée fait appel aux accesseurs (qu'ils soient synthétisés ou non)

Les blocs

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- ObjC a introduit la notion de bloc
- il s'agit d'une fonction (anonyme) avec liaison dans l'environnement
- c'est assez similaire aux pointeurs sur fonctions mais avec capture de l'environnement
- un peu similaire aux inner-classes de Java ou plus proche des fonctions anonymes de C#
- utile pour les callbacks

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- La syntaxe utilise ^

```
int (^unBloc)(int) = ^int (int a) {  
    return a+v;  
};
```

- Cette définition suppose qu'il existe dans l'environnement une variable v...
- si v est globale alors elle est capturée par référence (donc modifiable)
- si v est locale (auto), elle est capturée comme const, donc par valeur (non modifiable). Ce comportement peut-être modifié en employant la directive `__block` pour la déclaration de v

```
int v1;
int main(int argc, const char * argv[]) {
    v1 = 1;
    int v2 = 2;
    __block int v3 = 3;
    int (^add)(int) = ^int (int num) {
        return num + v1 + v2 + v3;
    };
    printf("%d\n", add(4));
    v1 = 10;
    v2 = 20;
    v3 = 30;
    printf("%d\n", add(40));
    return 0;
}
```

KVC

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- Un pattern Key Value Coding
- il existe un protocole informel `NSKeyValueCoding` qui permet d'accéder aux propriétés d'un objet de façon indirecte
- non pas à travers la variable d'instance (notation pointée) ou un accesseur mais à l'aide des sélecteurs (entre autres)
`setValue:(id) forKey:(NSString *)`
`valueForKey:(NSString *)`
- cela permet de « scripter » facilement le code

- pour qu'une classe soit KVC-compliant :
 - le cas le plus simple (suffira bien) est que la propriété possède bien les accesseurs adéquats ainsi :
`o.couleur = red;`
s'écrit aussi :
`[o setValue:red forKey:@"couleur"];`
 - ou
`cours.osx.description = @"cool";`
s'écrit aussi :
`[cours setValue:@"cool" forKeyPath:@"osx.description"];`
- permet donc de décrire le modèle de données via des chaînes

Les énumérations

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- Les énumérations rapides permettent de faciliter l'énumération des éléments d'une collection
- le protocole correspondant est `NSFastEnumeration`
- en ce cas on peut écrire
`for (type var in collection) { ... }`
- ces énumérations sont safe

```
NSDictionary *dict = [NSDictionary  
    dictionaryWithObjectsAndKeys:  
        @"un",@"1",@"deux",@"2",@"trois",@"3",nil];  
  
for (NSString *s in dict) {  
    NSLog(@"%@ %@",s,[dict objectForKey:s]);  
}
```

La gestion mémoire

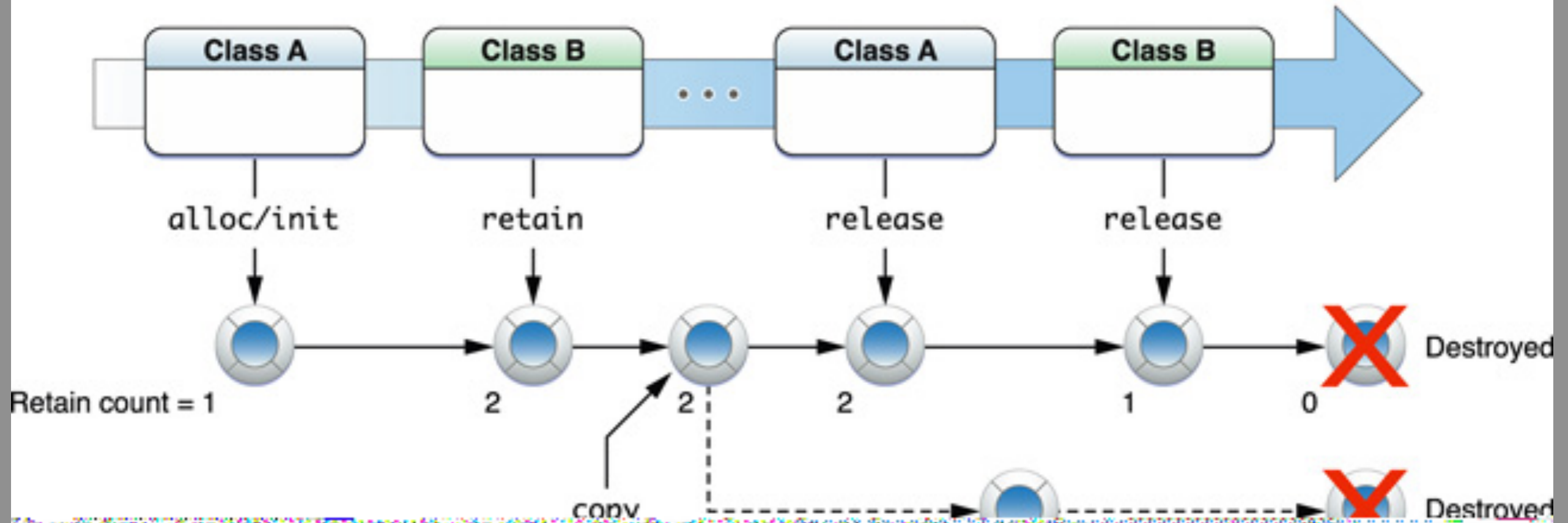
- il y a trois modes de gestion du cycle de vie des objets :
- (historique) par comptage explicite de référence (Manual Retain Release)
- (nouveau) par comptage implicite de référence (Automatic Reference Counting)
- (assez récent) par gestion implicite via le garbage-collector
 - interdit sous iOS

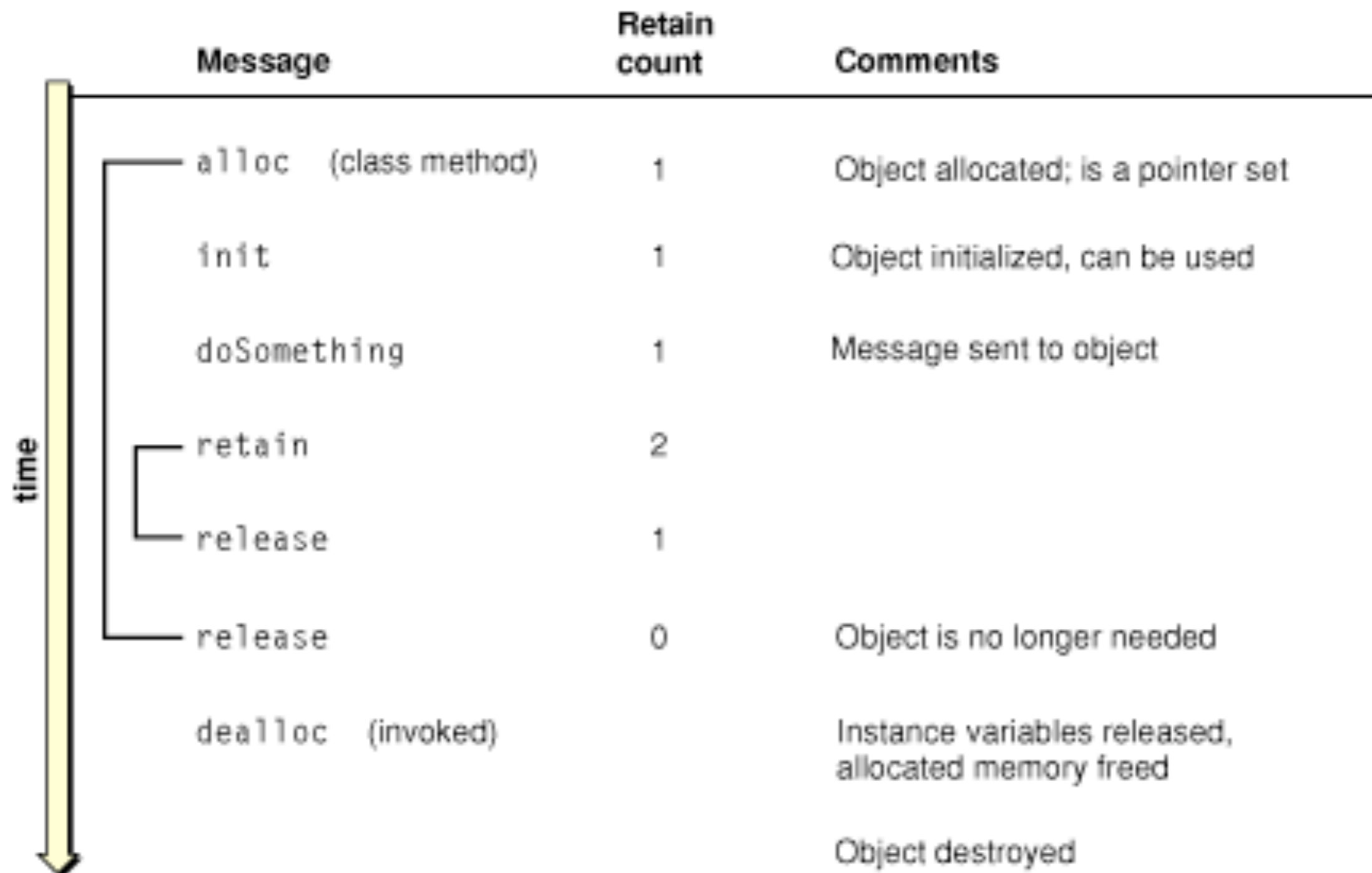
- Le comptage de référence
 - problème : y a t-il des pointeurs désignant un objet ?
 - idée simple : associer à tout objet un compteur indiquant combien de pointeurs le désigne
 - $\text{compteur} == 0$? suppression de l'objet
 - Piège : le compteur de référence seul ne peut-être suffisant (pensez aux références circulaires)...

- NSObject propose trois méthodes de gestion du compteur :
 - retain
 - release
 - autorelease

- retain :
 - incrémente le compteur de référence de l'objet
- release
 - décrémente le compteur de référence de l'objet, de plus si le compteur tombe à 0 la méthode dealloc de l'objet est appelée
- autorelease (plus loin)

Texte





- à savoir :
 - on est propriétaire de tout objet créé via `alloc`, `new`, `copy`, `mutableCopy`
 - on peut devenir propriétaire d'un objet à tout instant via appel à `retain`
 - en temps normal, tout objet reçu en paramètre est valide durant l'intégralité de l'appel
 - `retain` est le plus souvent utilisé par les setters afin garantir la conservation de l'objet au-delà de l'appel
 - quand on ne désire plus utiliser un objet que l'on possède, on le relâche via `release` ou `autorelease`

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2012–2013 Master II

- autorelease ?
 - permet de relâcher la propriété mais de délayer la suppression de l'objet en un point futur (autorelease pool)
 - utile si une méthode crée un objet qu'elle retourne mais qu'elle ne désire pas en être propriétaire...
- un autorelease pool est une collection d'objet pour lesquels la gestion de leur relâchement est délayée...