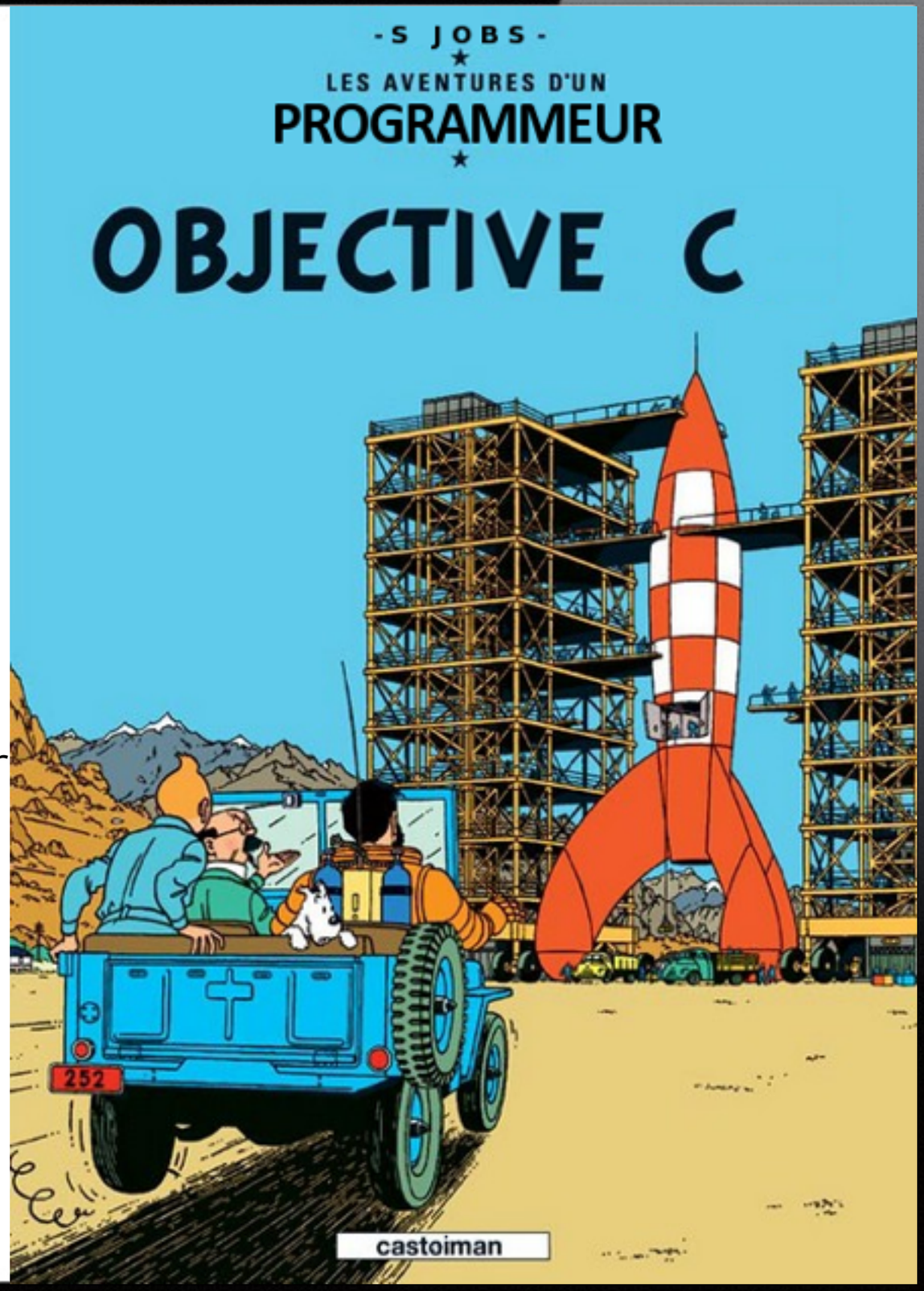
Objective-C

Jean-Baptiste.Yunes@univ-paris-diderot.fr

2014—2015

JBY is grateful to Frédéric Crassat (Orange) for helpful enhancements...

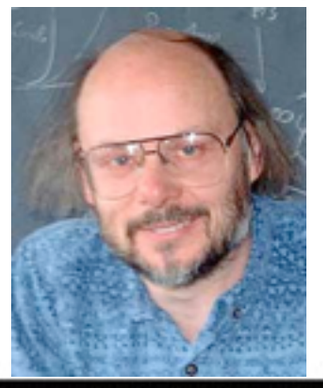


- deux extensions objet à C
  - C++ (Stroustrup)
    C++ = C + Simula
  - Objective-C (Cox & Love)
    Obj-C = C + Smalltalk
- années 80
- des succès différents

Brad Cox

Tim Love

Bjarne Stroustrup



- Objective-C est un sur-ensemble de C
- inspiré de smalltalk
  - une meilleure implémentation du paradigme objet
    - messages/protocoles/etc.
  - typage dynamique
  - chargement dynamique
  - dispatch dynamique

- utilisé dans NeXTSTEP
- puis MacOSX
- puis iOS
  - le langage « natif » mac
  - aujourd’hui l’alternative est Swift (que l’on utilisera pas pour des raisons logistiques)

- a évolué en Objective-C 2.0
  - garbage collector
  - extension syntaxique pour :
    - propriétés (attributs au sens de la conception)
    - énumération rapides (foreach)
    - extensions de classes (categories privées)
    - blocs (clôtures lambda)
- la documentation de référence

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

La syntaxe et la terminologie

- 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];
```

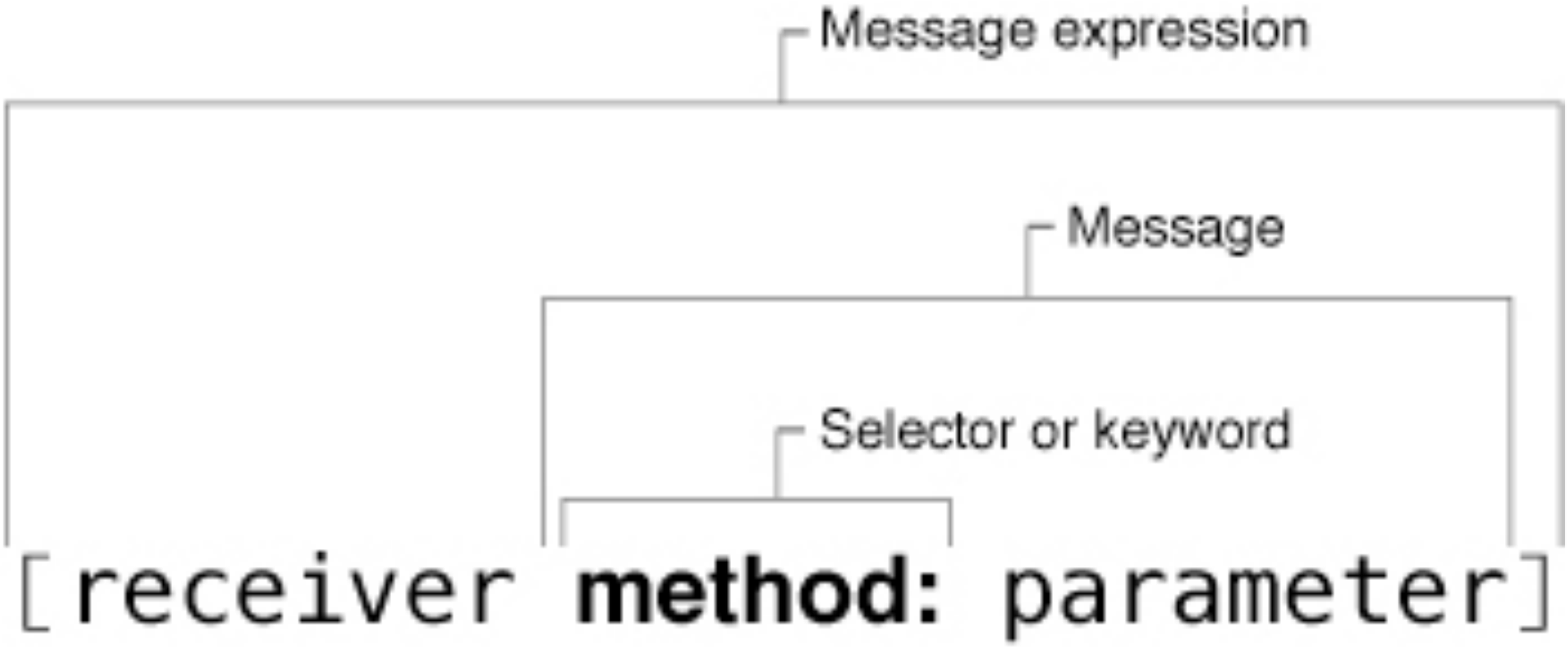
- le **message** :

```
[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]

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

```
[unPoint setX:10 y:20];
```

- le **receveur** :

```
[unPoint setX:10 y:20];
```

- le **message** :

```
[unPoint setX:10 y:20];
```

- le **sélecteur** :

```
[unPoint setX:10 y:20];
```

- la **méthode** est le **code** qui sera finalement sélectionné…

- un méthode est un code du receveur qui sera appelé par le dispatch lors de l'envoi d'un message à l'aide d'un sélecteur

```
[unPoint setX:10 y:20];
```

- les méthodes ont une **signature** :

```
(void)setX:(int)x y:(int)y
```

- la signature est en quelque sorte un sélecteur typé.

- 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`

- attention il faut bien distinguer les écritures
  - [image setSizeX:1200] (accesseur explicite)
  - image.setSizeX = 1200 (accesseur implicite)
  - image->sizeX = 1200 (pas d'accesseur!)

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;
```

`id` est une sorte de `void *`

  - 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 `BOOL` est habituellement utilisé pour les booléens
  - mots-clés `YES`, `NO`
  - idem `0` ou `1`

```
BOOL 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 *“
```

- il existe un type tableau ObjC
  - type `NSArray` dont les littéraux sont précédés du caractère `@`

    `@{@"ceci",@"est",@"un",@"tableau",@"de",@"NSString"}`
  - type `NSDictionnary` dont les littéraux sont précédés du caractères `@`

```
@{@"lol"  : @"je rigole fort",
  @"ptdr" : @"je me marre trop",
  @"mdr"  : @"je me gausse à mort"}
```

Les classes

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2014–2015 Master II

- 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 malfonctions 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 isMemberOfClass:UneClasse];
[objet2 isKindOfClass:UneAutreClasse];
```

- l'instanciation ObjC est particulière car elle distingue :
  - **l'allocation**

    ```
    id ceb = [CompteEnBanque alloc];
    ```

  - et **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) :

```
+ (void)initialize {
  if (self == [MaClasse class]) {
    // initialisation de la 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“

// variables de classe

@implementation MaClasse

{

   //  variables privées 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;
+ (void)initialize;

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

@end
```

```objc
//  UneClasse.m
#import "UneClasse.h"

static int nbInstances;

@implementation UneClasse
{
  int value;
}
- (id)initWithValue:(int)v {
  self = [super init];
  if (self) {
    NSLog(@"init");
    value = v;
    nbInstances++;
  }
  return self;
}
- (int)value {
  return value;
}
+ (id)createWithValue:(int)v {
  return [[self alloc] initWithValue:v];
}
+ (void)initialize {
  if (self == [UneClasse class]) {
    nbInstances++;
  }
}
@end
```

```objc
//  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;
}
```

- `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 propriétés

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

- 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 `copy` de l'argument, l'ancienne valeur est `release` (nécessite le protocole `NSCopy`)
  - `assign` : simple affectation (par défaut)
  - `retain` : l'ancienne valeur est `release` et la nouvelle est `retain`

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

- L'implémentation d'une propriété peut s'effectuer 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 »

- L’utilisation de @synthesize est optionnelle
  - la construction par défaut suffit généralement
  - ne rien faire…

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

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

Les protocoles

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2014–2015 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 blocs

Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2014–2015 Master II

- ObjC a introduit la notion de bloc
- il s'agit d'une fonction (anonyme) avec liaison dans l'environnement (clôture)
- 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# et des lambdas de C++
- utile pour les callbacks

- 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

```c
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 / 2014–2015 Master II

- Un pattern Key Value Coding
- il existe un protocole informel `NSKeyValueCoding` qui permet d'accéder au 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'écrira aussi :

```
[o setValue:red forKey:@“couleur“];
```

  - ou

```
cours.osx.description = @“cool“;
```

s'écrira 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 / 2014–2015 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]);
}
```

ou (plus moderne)

```
NSDictionary *dict = @{
  @"un":@"1",@"deux":@"2",@"trois":@"3"];

for (NSString *s in dict) {
    NSLog(@"%@ %@",s,dict[s]);
}
```

La gestion mémoire

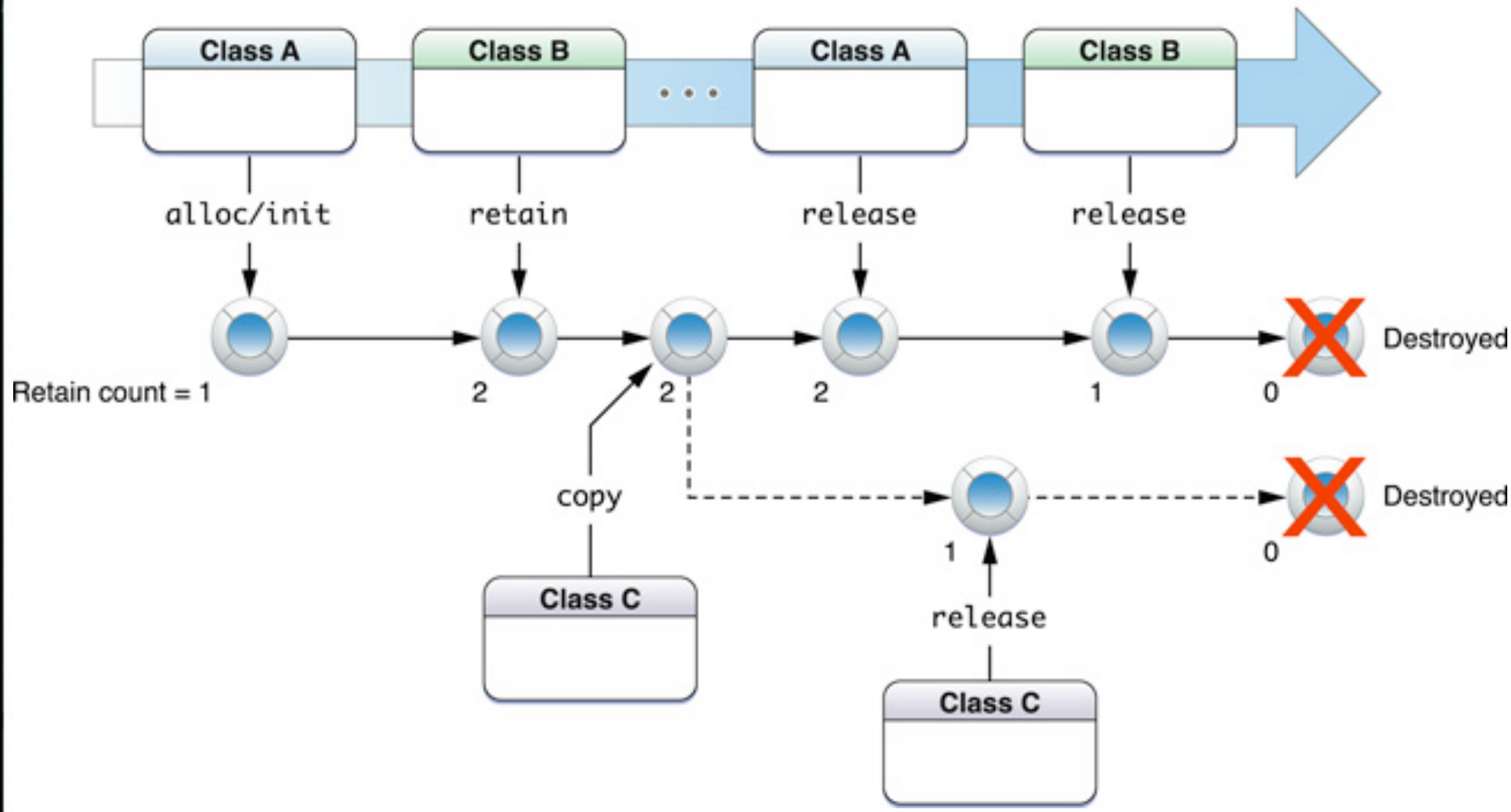
Jean-Baptiste.Yunes@univ-paris-diderot.fr / 2014–2015 Master II

- 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
    - autorisé sous iOS depuis la version 5

- 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
    - compteur==0 ? suppression de l'objet
- Piège : le compteur de référence seul ne peut-être suffisant (pensez aux références circulaires)…

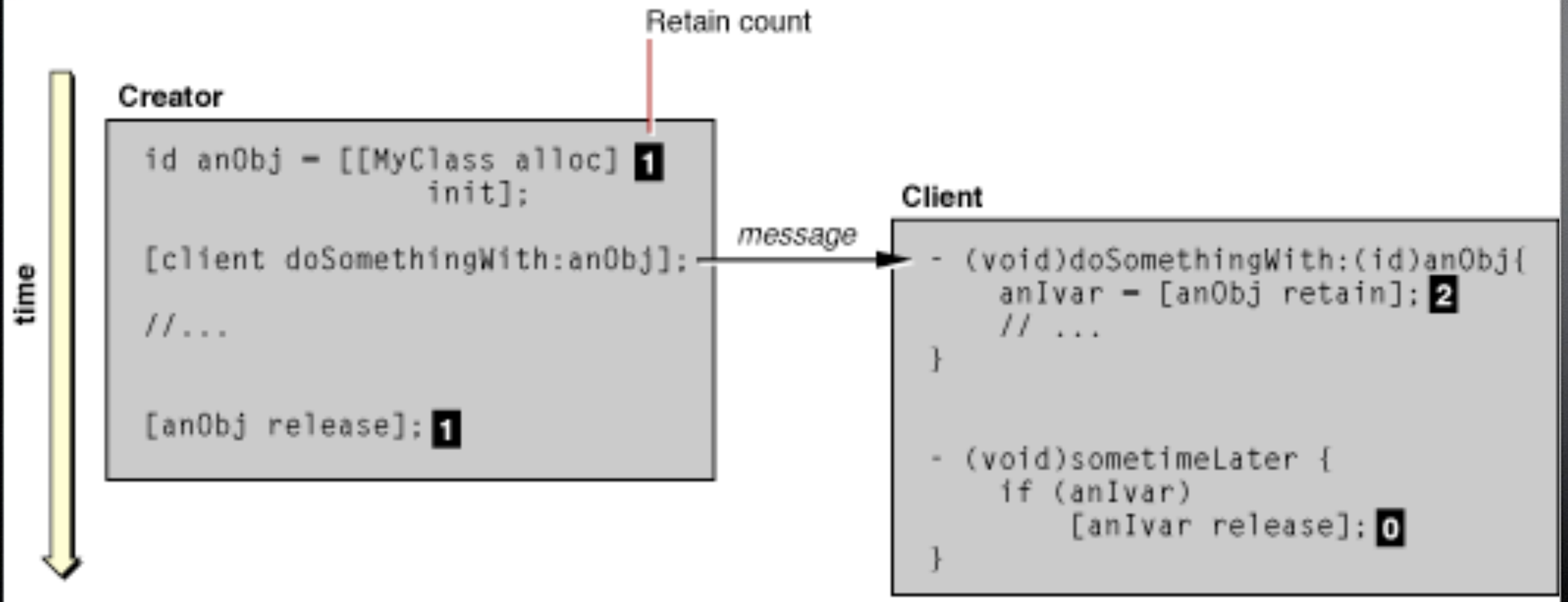
- `NSObject` propose trois méthodes de gestion du compteur :
  - `retain`
  - `release`
  - `autorelease`

- `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)

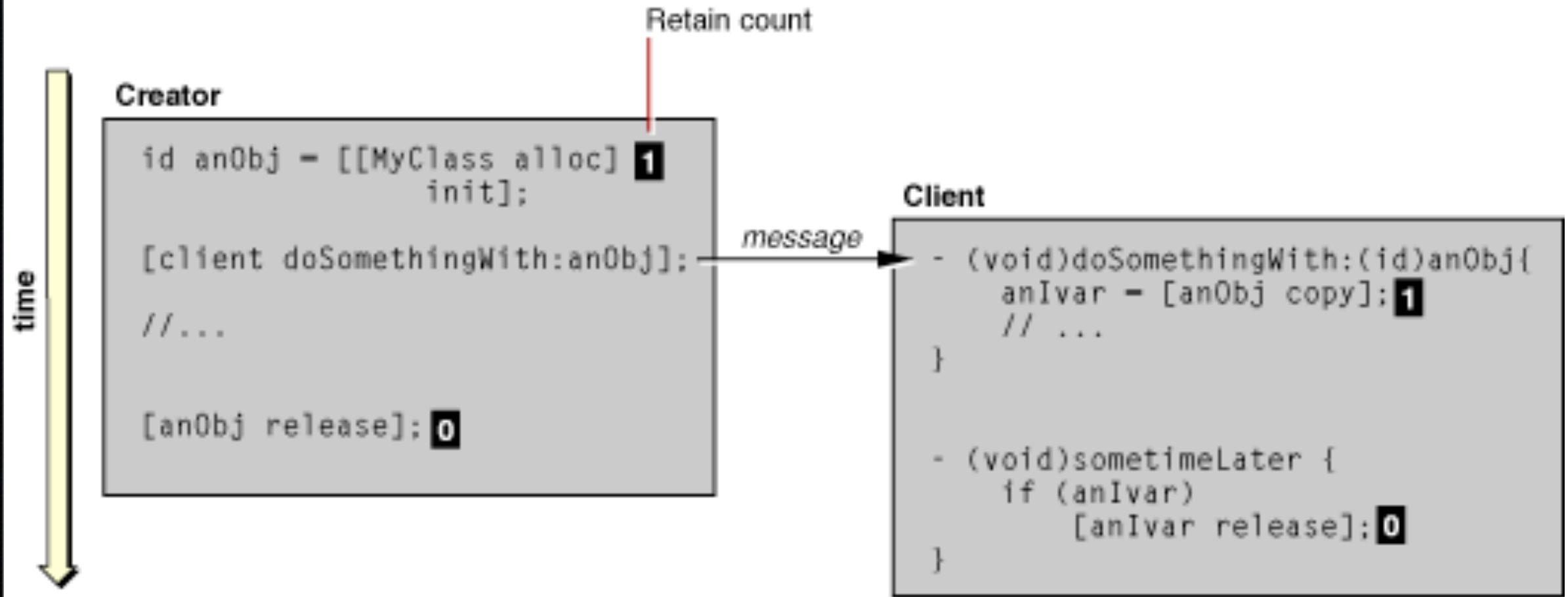

Class A
Class B
Class A
Class B
alloc/init
retain
release
release
Retain count = 1
2
2
2
1
0
Destroyed
copy
Class C
1
0
Destroyed
release
Class C


Message
Retain count
Comments
alloc (class method)
1
Object allocated; is a pointer set
init
1
Object initialized, can be used
doSomething
1
Message sent to object
retain
2
release
1
release
0
Object is no longer needed
dealloc (invoked)
Instance variables released, allocated memory freed
Object destroyed
time

- 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`



Rétention d'un objet reçu



Copie d'un objet reçu

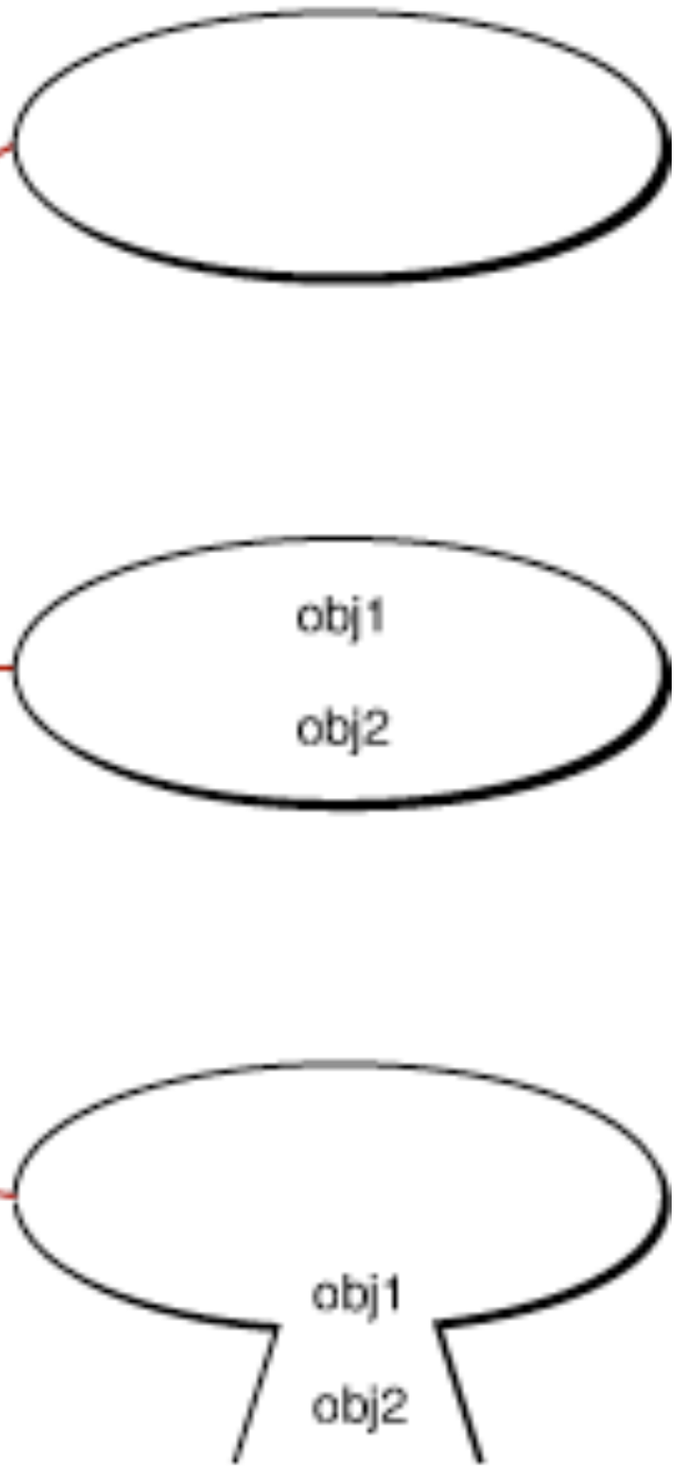
- `autorelease` ?
  - permet de relâcher la propriété mais de délayer la suppression de l'objet en un point futur (autorelease pool)
    - utile si une méthode créé un objet qu'elle retourne mais qu'elle ne désire pas en être propriétaire…
- un autorelease pool est une collection d'objet pour lesquels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

```
NSAutoreleasePool *arp =
   [[NSAutoreleasePool alloc] init];

int count = 2;
while (count--) {
    NSFoo *obj1 = [[[NSFoo alloc]
       init] autorelease];
    NSFoo *obj2 = [[[NSFoo alloc]
       init] autorelease];

}

[arp release];
```



Autorelease Pool