- Syntaxe Objective C
  - Création d'un objet
    - Utilisation de méthodes statiques "convenience constructors" qui regroupent l'allocation et l'initialisation
    - Modifier le code pour utiliser
      - NSDate

```
NSDate *now = [NSDate date];
```

- Gestion de la mémoire - Historique
  - Allocation par la méthode “alloc”
  - Libération par la méthode “dealloc”
    - Cette méthode ne doit JAMAIS être appelée directement
  - Difficulté commune à tous les langages et environnement
    - Comment savoir à quel moment un objet doit être libéré ?
    - Un même objet peut être utilisé par plusieurs autres sans que ceux-ci n’en ait connaissance
  - Solution : utilisation d’un compteur de références (“retain count”)
    - Initialisé à 1 lors de l’allocation ([monObjet alloc])
    - Lorsqu’un objet n’est plus utilisé, il faut lui envoyer le message “release”

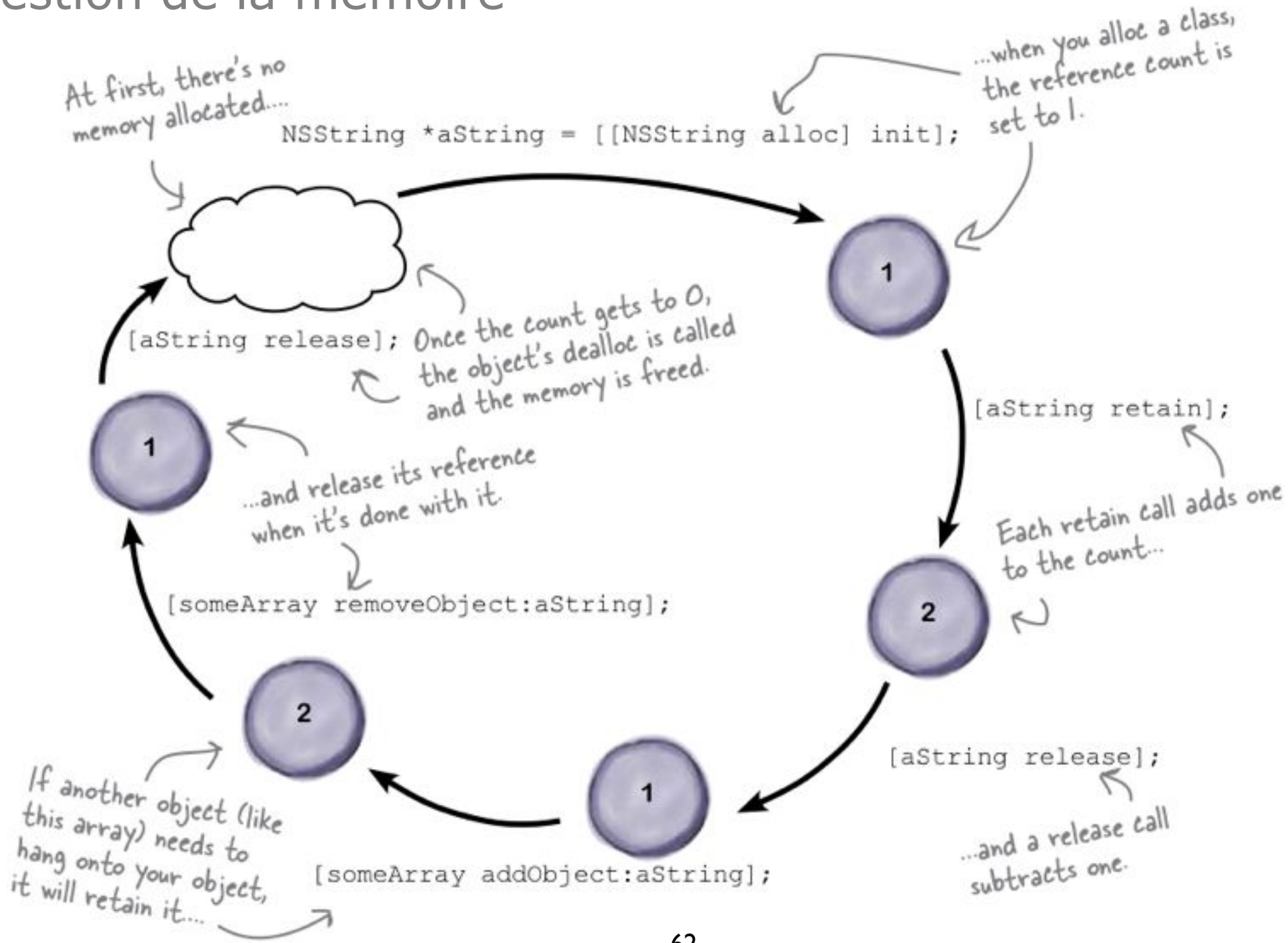
- Gestion de la mémoire
  - Retain count
    - Lors de la réception d'un message release, le compteur de référence est décrémenté et s'il tombe à zéro, le système appellera la méthode dealloc de l'objet
    - La méthode dealloc d'un objet doit relâcher tous les objets qui étaient référencés par l'objet désalloué

```
- (void)dealloc {
    [name release];
    [super dealloc];
}
```

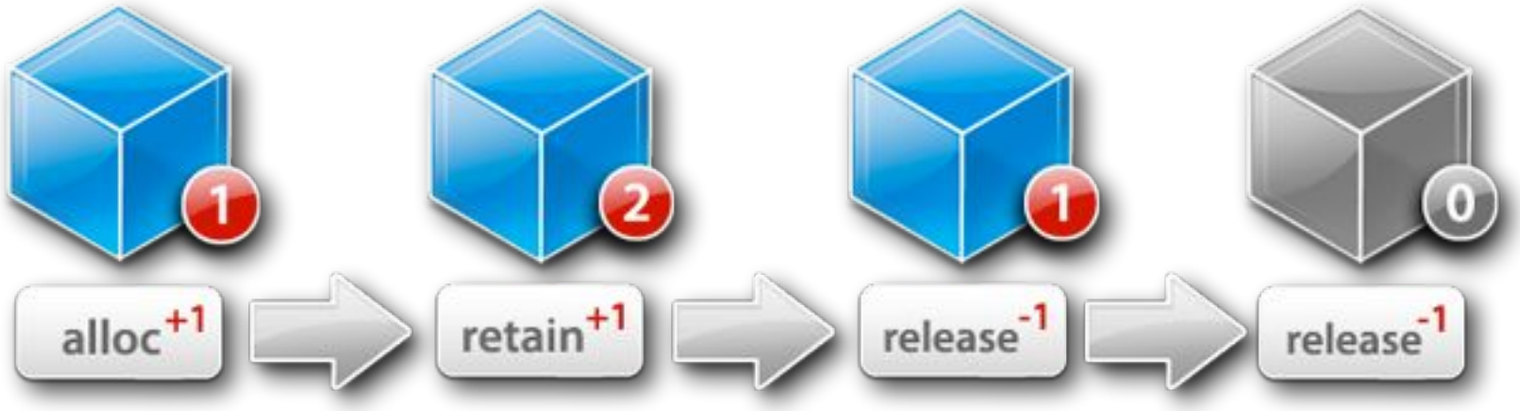
    - Pour s'assurer qu'un objet alloué par "quelqu'un d'autre" alors que l'on en a encore besoin il faut lui envoyer le message retain qui incrémente le compteur de références

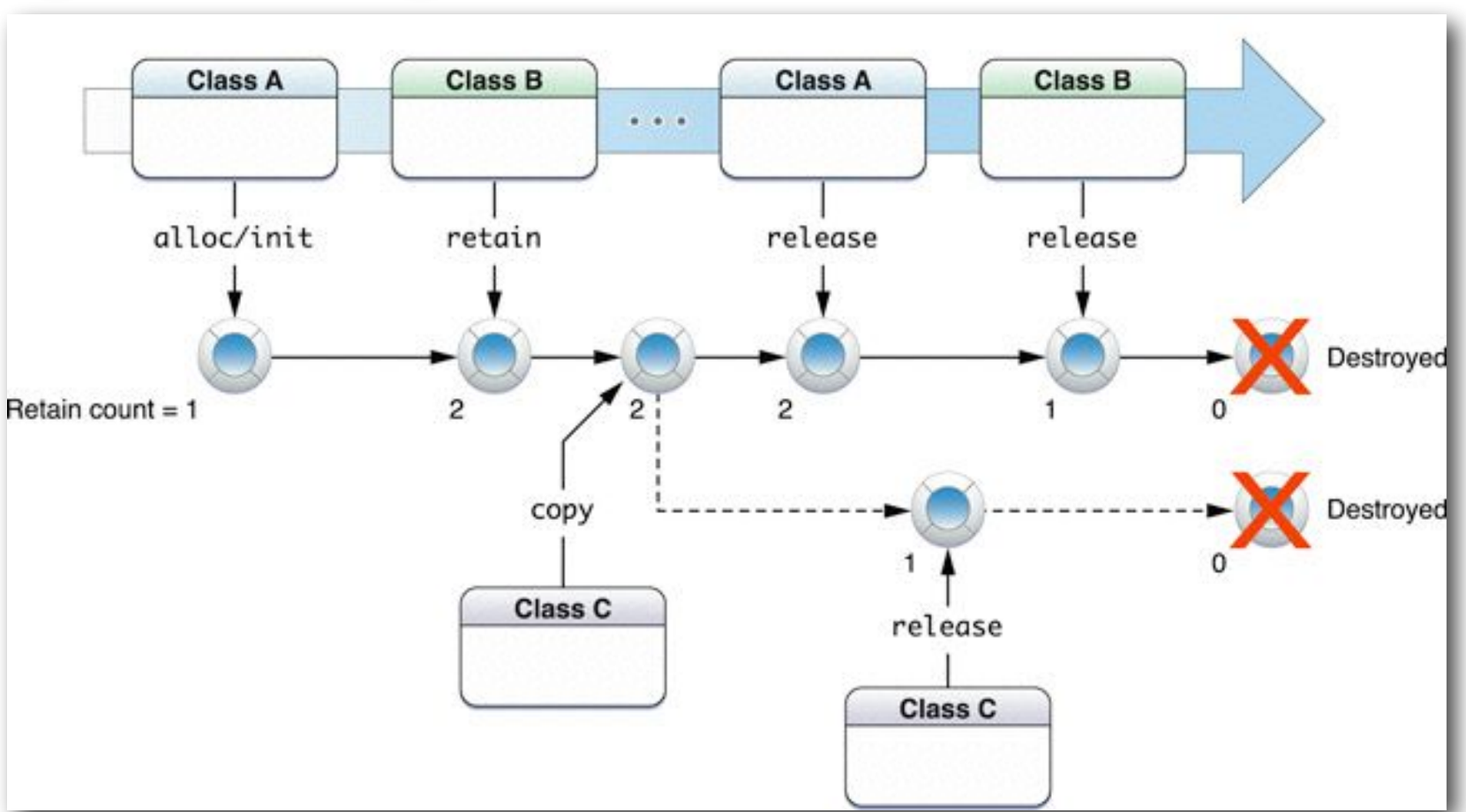
```
-(void)setName:(NSString *)newName{
   [newName retain];
   [name release];
   name = newName;
}
```

Gestion de la mémoire



Gestion de la mémoire - Résumé





- Gestion de la mémoire - Règle fondamentale
  - Envoyer release or autorelease aux seuls objets que vous possédez
    - Vous possédez un objet
      - si vous le créez en utilisant une méthode qui commence par “alloc”, “new” ou qui contient “copy” (par exemple, alloc, newObject, or mutableCopy),
      - ou si vous lui avez envoyé un message retain
  - Vous utilisez release ou autorelease pour abandonner la propriété d’un objet. autorelease signifie juste “envoie un message release dans le futur”

- Gestion de la mémoire
  - Si vous avez besoin de stocker un objet reçu (argument d'un message) dans une variable d'instance vous devez lui envoyer retain ou le copier.
  - Un objet reçu en argument est normalement garanti de rester valide à l'intérieur de la méthode dans laquelle il a été reçu (les exceptions concernent le multithreading et les objets distribués)
  - Les objets sont retenus lors de l'ajout dans une classe de collection (ex : NSArray) et relâchés lorsqu'ils en sont extraits

- Gestion de la mémoire
  - Autorelase
    - “Convenience constructor” de la classe Person

```
+(Person *)createPersonWithAge:(UInt8)newAge {
    Person *person = [[Person alloc] initWithAge:newAge];
    return person;
}
```

- L’utilisateur de cette méthode ne serait pas responsable du relachement de la référence (release) car le nom de la méthode ne contient pas l’un des mots listés dans la règle (init, copy,...)
- La méthode “createPersonWithAge” appelle “alloc” et est donc reponsable de l’appel à release
- Impossible d’appeler “release” après return et si “release” est appelé avant return l’objet sera désalloué et la méthode retournera un pointeur sur un objet désalloué.
- Quand appeler “release” ? Plus tard...

- Gestion de la mémoire
  - Autorelease
    - Ici "Plus tard" signifie "après return" et suffisamment tard pour que l'appelant est le temps de stocker la valeur retournée dans une variable et d'appeler "retain" dessus si besoin.
    - Solution : autorelease

```
+(Person *)createPersonWithAge:(UInt8)newAge {
    Person *person = [[Person alloc] initWithAge:newAge];
    return [person autorelease];
}
```

    - Mise en oeuvre

```
Person *toto = [Person createPersonWithAge:24];
[toto retain];
```

    - Les objets qui reçoivent le message autorelease sont placés dans un pool d'autorelease (collection d'objets qui reçoivent le message release à la fin de la boucle de traitement des événements de l'application).
      - Voir la fonction main pour la création du pool d'autorelease

- Gestion de la mémoire
  - Depuis le passage à ARC (Automatic Reference Counting), toute cette mécanique est réalisée automatiquement par le compilateur
  - Il existe quelques situations qui nécessitent une intervention "manuelle"
    - Casser des cycles de références
    - Introduire un pool d'autorelease supplémentaire dans une boucle génératrice de nombreux objets temporaires
      - @autorelease

Syntaxe Objective C - Variables d’instances

- Par défaut les variables sont visibles par la classe qui les déclare et par ses sous-classes
- Modificateur par défaut : @protected

```
@interface Person : NSObject {
  NSString *name;
  UInt8 age;
}

équivalent à :

@interface Person : NSObject {
  @protected
  NSString *name;
  UInt8 age;
}
```

Syntaxe Objective C - Variables d’instances

Accès aux variables d’instance depuis l’implémentation

```
@implementation Person

- (BOOL)canLegallyVote {
  return (age >= 18);
}
@end
```

- Syntaxe Objective C - Variables d'instances
  - Accès aux variables d'instance depuis d'autres classes
    - Approche recommandée : accesseurs et mutateurs (getters and setters)
      - Convention de nommage :
        - L'accesseur est nommé en utilisant le nom de la valeur retournée
        - Le mutateur commence par "set" et est suivi du nom de la valeur retournée (avec la première lettre mise en majuscule)

```
- (UInt8)age;
- (void)setAge:(UInt8)age;
- (NSString *)name;
- (void)setName:(NSString *)value;
```

        - L'accesseur est préfixé par "get" lorsqu'il retourne un pointeur sur une variable qui n'est pas un objet (utile lorsque la méthode retourne plusieurs valeurs)

```
- (void)getRed:(CGFloat *)red green:(CGFloat *)green
blue:(CGFloat *)blue alpha:(CGFloat *)alpha;
```

- Syntaxe Objective C - Variables d'instances
  - Convention de nommage des accesseurs
    - Plus qu'une simple convention de nommage :
      - Base des mécanismes de Key-Value Coding et Key-Value Observing qui permettent de mettre simplement en oeuvre le design pattern "Observateur", l'intégration des objets dans le framework de persistance Core Data, ...
      - Utilisé par Interface Builder pour créer graphiquement des liens entre objets

- Syntaxe Objective C - Variables d'instances
  - Avantages des accesseurs
    - Flexibilité de l'implémentation
      - les propriétés peuvent être stockées dans des variables d'instances, ou dans une base de donnée, calculées en utilisant d'autres champs, récupérées sur un serveur, ...
      - L'utilisateur de la classe n'a pas besoin de connaître les détails (meilleure encapsulation)
      - L'implémentation peut-être modifiée sans impacter les utilisateurs
    - Maintenance allégée
      - Tous les accès à la variable se font à travers les accesseurs ➪ limite le nombre d'endroit où le code doit être modifié en cas de changement des propriétés

- Syntaxe Objective C - Variables d'instances
  - Avantages des accesseurs
    - Gestion "centralisée" de la mémoire (mécanisme de comptage de références)
      - Restriction de la gestion de la mémoire aux accesseurs ➪ facilite l'application des conventions de Cocoa et limite la gestion de la mémoire à quelques méthodes isolées
    - Les mutateurs (setters) simplifient le débuggage
      - Si une propriété prend une valeur incorrecte ou suspecte, il est facile de mettre un point d'arrêt dans la seule méthode susceptible de modifier cette propriété (le mutateur "setMaVariable") pour identifier le code en cause
    - Les mutateurs (setters) permettent de contraindre la plage des valeurs autorisées ou de déclencher une mise à jour lors de la modification d'une propriété (ex : signaler au framework graphique qu'il faut rafraichir l'affichage lors de la modification d'une propriété de couleur, ...)

Syntaxe Objective C - Variables d'instances

- Avantages des accesseurs
  - Exemple
    - La propriété "age" de la classe Person est stockée dans une variable d'instance "UInt8 age"

```
@interface Person : NSObject {
  NSString *name;
  UInt8 age;
}
```

```
@implementation Person
//...
- (UInt8)age {
  return age;
}

- (void)setAge:(UInt8)value {
  age = value;
}
//...
@end
```

- Syntaxe Objective C - Variables d'instances
  - Avantages des accesseurs
    - Exemple
      - Stockage de la date de naissance plutôt que de l'age

```
@interface Person : NSObject {
   NSDate *birthDate;
}

@implementation Person
//...
-(UInt8)age
{
   NSDate *now = [NSDate date];
   double timeInterval = [now timeIntervalSinceDate:birthDate]/(365.25*24*3600);
   return (UInt8)(timeInterval);
}
//...
@end
```

      - Les utilisateurs utilisent toujours : `[test age]`

- Syntaxe Objective C - Propriétés
  - Déclaration de propriétés avec la directive **@property**
    - Doit figurer dans la liste des méthodes déclarées dans l'interface (équivaut à déclarer les accesseurs)
  - Possibilité de synthétiser automatiquement le code des accesseurs
    - Utilisation de **@synthesize**
    - On peut combiner @synthesize et implémentation de l'un des accesseurs ➪ seuls les accesseurs non trouvés dans l'implémentation seront générés automatiquement
    - La directive **@synthesize** peut être omise et XCode synthétise les accesseurs et peut également synthétiser les variables d'instance

Syntaxe Objective C - Propriétés

```
@interface HelloWorldViewController : UIViewController {
   IBOutlet UILabel *label;
}

@property(nonatomic, strong) IBOutlet UILabel *label;

-(IBAction)refresh:(id)sender;

@end
```

Déclaration d'une propriété et synthèse des accesseurs

```
@implementation HelloWorldViewController

@synthesize label;

-(IBAction)refresh:(id)sender{

   label.text = @"Hello World !";
}
```

Syntaxe Objective C - Propriétés

```
@interface HelloWorldViewController : UIViewController {
}

@property(nonatomic, strong) IBOutlet UILabel *label;

-(IBAction)refresh:(id)sender;

@end
```

Déclaration d'une propriété / synthèse des accesseurs et de la variable d'instance

```
@implementation HelloWorldViewController

@synthesize label;

-(IBAction)refresh:(id)sender{

   label.text = @"Hello World !";
}
```

Syntaxe Objective C - Propriétés

```
@interface HelloWorldViewController : UIViewController {
}

@property(nonatomic, strong) IBOutlet UILabel *label;

-(IBAction)refresh:(id)sender;

@end
```

Déclaration d'une propriété / synthèse des accesseurs et de la variable d'instance avec un nom différent

```
@implementation HelloWorldViewController

@synthesize label = _label;

-(IBAction)refresh:(id)sender{

   label.text = @"Hello World !";
}
```

Syntaxe Objective C - Propriétés

```
@interface HelloWorldViewController : UIViewController {
}

@property(nonatomic, strong) IBOutlet UILabel *label;

-(IBAction)refresh:(id)sender;

@end
```

Déclaration d'une propriété / synthèse automatique des accesseurs et de la variable d'instance

```
@implementation HelloWorldViewController


-(IBAction)refresh:(id)sender{

   label.text = @"Hello World !";
}
```

Le compilateur ajoute automatiquement : `@synthesize label = _label;`

Syntaxe Objective C - Propriétés

- Attributs des propriétés
  `@property(nonatomic, strong)`
  - Le code généré par le mot clé @synthesize dépend des attributs spécifiés lors de la déclaration de la propriété
    - Atomicité
      - Par défaut les accesseurs générés sont prévus pour être accédés de façon concurrente depuis plusieurs thread
      - le code ainsi généré garantit l'atomicité des accès (mise en attente des threads si un thread est déjà "entré" dans la méthode)
      - Surcoût lors des appels aux méthodes (temps de prise et de libération des verrous associés)
      - Utilisation du mot clé "nonatomic" lorsque les accesseurs seront appelé depuis un thread unique

Syntaxe Objective C - Propriétés

Attributs des propriétés

Noms des accesseurs

- Par défaut les accesseurs pour une propriété

```
@property float value;
```

sont construits comme suit :

```
- (float)value;
- (void)setValue:(float)newValue;
```

- Modification du nom

```
@property(getter=retournerValue,setter=modifierValue) float value;
```

Syntaxe Objective C - Propriétés

- Attributs des propriétés
  - Caractère modifiable
    - Propriétés accessible en lecture et écriture

```
@property(readwrite) float value;
```

      - C’est la valeur par défaut
    - Propriétés accessible en lecture seule

```
@property(readonly) float value;
```

      - Le bloc d’implémentation ne nécessite que la présence d’un “getter”. Si @synthesize est utilisé aucun setter ne sera généré

Syntaxe Objective C - Propriétés

- Attributs des propriétés
  - Modification des setters
    - assign
      - La variable passée en paramètre est simplement assignée à la variable d'instance
      - Utile pour tous les types non objet (l'assignation est équivalent à une copie)

```
@property(assign) NSInteger value;
```

    - weak (équivalent d'assign pour les objets) - permet de casser des cycles de références

```
@property(weak) id target;
```

- Syntaxe Objective C - Propriétés
  - Attributs des propriétés
    - Modification des setters
      - strong
        - La variable (pointeur sur un objet) est retenue par le setter afin que l'objet ne soit pas désalloué

```
@property(strong) NSString *title;
```

      - copy
        - L'argument passé en paramètre est copié
        - La variable d'instance pointera vers un objet différent de celui qui a été passé en paramètre

```
@property(copy) NSString *title;
```

- Syntaxe Objective C - Propriétés
  - Attributs des propriétés
    - Modification des setters
      - copy
        - La copie est utile quand l'argument n'est pas immuable :

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"initial value"];
[someObject setStringValue:mutableString];
[mutableString setString:@"different value"];
```

- Si la méthode setStringValue se contentait de retenir la valeur passée en argument les modifications opérées sur l'objet mutableString se répercuteraient sur la propriété de l'objet someObject sans que celui-ci n'en soit informé

- Syntaxe Objective C - Variables d'instances
  - "Dot syntax"
    - Accès aux propriétés d'un objet à l'aide d'un "."
      - Provoque un appel aux accesseurs de la propriété

```
NSColor *color = graphic.color;
CGFloat xLoc = graphic.xLoc;
BOOL hidden = graphic.hidden;
int textCharacterLength = graphic.text.length;
if (graphic.textHidden != YES) {
    graphic.text = @"Hello";
}
graphic.bounds = NSMakeRect(10.0, 10.0, 20.0, 120.0);
```

```
NSColor *color = [graphic color];
CGFloat xLoc = [graphic xLoc];
BOOL hidden = [graphic hidden];
int textCharacterLength = [[graphic text] length];
if ([graphic isTextHidden] != YES) {
    [graphic setText:@"Hello"];
}
[graphic setBounds:NSMakeRect(10.0, 10.0, 20.0, 120.0)];
```