

```
/*! Booléen indiquant si le contexte Core Data doit être rechargé. Utilisé pour appeler la méthode reloadAfterCoreDataChanged si la vue n'était pas visible.*/
@property (nonatomic, retain) CLLocationManager *coreLocationManager;
@property (nonatomic, retain) CLPlacemark *currentPlacemark;
@property (nonatomic) BOOL shouldReloadCoreData;
@property (nonatomic, retain) NSNotification *coreDataChangeNotification;
-(void) releaseOutlets;
-(void) dismissModalViewController;
/*! Notification envoyée par le contexte CoreData lorsqu'il est sauvegardé.
 */
-(void) onCoreDataChanged:(NSNotification*)notification;
-(void) image:(UIImage *)image didFinishSavingWithError:(NSError *)error contextInfo:(void *)contextInfo;
@end

@implementation OBSBaseViewController
{
    CAGradientLayer *gradient;
    OBSEmailContactPickerController *_contactPicker;
}

@synthesize toolBoxButton = _toolBoxButton;
@synthesize popover = _popover;
@synthesize currentPreviewFile = _currentPreviewFile;
@synthesize file = _file;

- (void)dealloc
{
    self.currentPlacemark = nil;
    self.coreLocationManager = nil;
    self.currentLocation = nil;
    [super dealloc];
}
```

Le langage

Brendan GUEGAN
Segment « Multi Devices »
Développeur iOS – Référent technique iOS.
brendan.guegan@orange.com
+33 2 99 87 92 83

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

introduction

historique

- créé par Brad Cox au début des années 80
- ajout de la notion de classe au C
- créé à partir du C et du Smalltalk, et a influencé la création de Java
- utilisé par le système d'exploitation NeXTStep et récupéré ensuite dans Mac OS X
- gestion de mémoire simplifiée grâce aux compteurs de références
- Objective-C 2.0 créé en 2007 permet de simplifier encore plus la gestion de la mémoire

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

les bases de la syntaxe

déclaration et instantiation de variables

- tout objet est un pointeur
 - la création d'un d'objet se fait en deux étapes
 - allocation d'un message mémoire à l'aide la méthode `alloc`
 - Initialisation de l'objet alloué à l'aide d'une méthode `init`
 - la méthode `new` peut également être utilisée directement si l'on veut créer un objet sans paramètre
-  par convention, une méthode d'initialisation commence par `init` (exemple `initWithFormat:` de la classe `NSString`)
-  l'allocation et l'initialisation doivent se faire sur la même ligne

Exemple

```
MaClasse *objet = [[MaClasse alloc] init]; } appels  
MaClasse *objet = [MaClasse new]; } équivalents  
MaClasse *objet = [[MaClasse alloc] initWithFloat:3.0f];
```

les bases de la syntaxe

les types primitifs

- les types primitifs hérités du C existent : short, int, long, double, float, char, unsigned ...
- Un type ajouté : BOOL (valeur YES ou NO)
- Cocoa fournit quelques types primitifs basés sur le C à utiliser de préférence :
 - NSInteger \Leftrightarrow int (long en 64bits)
 - CGFloat \Leftrightarrow float (double en 64bits)
 - NSUInteger \Leftrightarrow unsigned int (unsigned long en 64bits)

Exemple

```
BOOL b = YES;  
int i = 0;  
CGFloat f = 4.0f;
```

les bases de la syntaxe

énumérés

Exemple

```
typedef enum
{
    Red,
    Green
} Color;
```

- le type de l'énuméré est par défaut un entier signé
- pour indiquer un type différent, il faut le spécifier de la manière suivante :

Exemple

```
typedef enum : unsigned char
{
    Red,
    Green
} Color;
```

les bases de la syntaxe

appel de méthode

- l'appel de méthode sur un objet correspond à lui envoyer un message
- syntaxe :
 - appel délimité par des crochets
 - possibilité d'enchaîner des appels de méthodes sur une même ligne
- possibilité d'appeler une méthode sur un objet nul

Exemple

```
[objet doSomething];
[objet doSomethingWithParam1:@"param1"];
[objet doSomethingWithParam1:@"param1" param2:34.0f];
```

les bases de la syntaxe

appel de méthode - les sélecteurs

- un sélecteur permet de sélectionner une méthode par rapport à sa signature, ou une chaîne de caractères

Exemple

```
SEL selector1 = @selector(doSomething);  
SEL selector2 = @selector(doSomethingWithParam1:);  
SEL selector1 = NSSelectorFromString(@"doSomething");
```

- on peut ensuite appeler une méthode grâce à un sélecteur

Exemple

```
[objet performSelector:selector1];  
[objet performSelector:selector2 withObject:@"doSomething"];
```

les bases de la syntaxe

appel de méthode



- pas de vérification de l'existence d'une méthode à la compilation,
seul un warning est indiqué
 - conséquences :
 - plantage si la méthode n'est pas trouvée à l'exécution
 - on peut appeler des méthodes non déclarées

les bases de la syntaxe

structures de contrôle

- structures et mots-clés hérités du C : for, if else, while, switch, do while, break, continue, default

Exemple

```
NSSet *set = [self doSomething];
for(id objet in set)
{
    [objet doSomething];
}
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

gestion de la mémoire

la théorie

- chaque objet possède un compteur de références
- dès que ce compteur vaut 0, l'objet est désalloué
- à sa création, le compteur d'un objet est à 1
- un objet peut être « retenu », son compteur augmente alors de 1
- un objet peut être « relâché », son compteur diminue alors de 1



en Objective-C, c'est chacun pour soi, si un objet désire en conserver un autre, il doit le retenir et le relâcher quand il n'en a plus besoin

gestion de la mémoire

la pratique

- pour retenir un objet, on utilise la méthode `retain`
- pour relâcher un objet, on utilise la méthode `release`
- on peut retarder le relâchement d'un objet avec la méthode `autorelease` (cela est utile pour une méthode qui renvoie un objet qu'elle a créé)
- on peut connaître le compteur d'un objet avec la méthode `retainCount`



on ne détruit jamais directement un objet

Exemple

```
MaClasse *objet = [[MaClasse alloc] init]; // compteur=1
[objet retain]; // compteur=2
[objet release]; // compteur=1
[objet release]; // compteur=0, l'objet est désalloué
```

gestion de la mémoire

la pratique - utilisation de l'autorelease

- l'autorelease permet de laisser un délai avant le relâchement d'un objet
- l'objet est relâché plus tard par le pool d'autorelease

Exemple

```
- (MaClasse*) maMethode
{
    MaClasse *objet = [[MaClasse alloc] init];
    return objet; // l'objet ne sera jamais libéré → fuite
    mémoire
}
```

gestion de la mémoire

la pratique - utilisation de l'autorelease

Exemple

```
- (MaClasse*) maMethode
{
    MaClasse *objet = [[MaClasse alloc] init];
    [objet release]; // le compteur passe à 0
    return objet; // l'objet est désalloué ➔ erreur
}
```

Exemple

```
- (MaClasse*) maMethode
{
    MaClasse *objet = [[MaClasse alloc] init];
    return [objet autorelease];
}
```

gestion de la mémoire

pour résumer

- alloc+init, retain, copy, new : + 1
- release, autorelease : - 1
- pour relâcher un objet, on utilise la méthode release
- on peut retarder le relâchement d'un objet avec la méthode autorelease (cela est utile pour une méthode qui renvoie un objet qu'elle a créé)
- on peut connaître le compteur d'un objet avec la méthode retainCount

Exemple

```
MaClasse *objet = [[MaClasse alloc] init]; // compteur=1
[objet retain]; // compteur=2
[objet release]; // compteur=1
[objet release]; // compteur=0, l'objet est désalloué
```

gestion de la mémoire

pour résumer



quand on fait un alloc+init, il faut en général faire un release ou un autorelease par la suite



quand on récupère un objet sans faire d'alloc+init, il ne faut pas faire de release ou autorelease

gestion de la mémoire

les pools d'autorelease - définition

- les pools d'autorelease sont chargés de relâcher les objets en autorelease
- un pool appartient à un thread, et un thread peut gérer plusieurs pools
- toutes les applications possèdent un pool d'autorelease par défaut dans le thread principal

gestion de la mémoire

les pools d'autorelease - quand s'en servir ?

- les pools d'autorelease sont utilisés dans des cas spécifiques
 - dans un thread
 - dans des parties de code dans lesquelles on crée beaucoup d'objets, on vide le pool afin de diminuer l'empreinte en mémoire
- pour relâcher un pool, on utilise la méthode drain

Exemple

```
NSAutoreleasePool *pool = [NSAutoreleasePool new];
NSArray *array = [NSArray array]; // tableau en autorelease
[pool drain]; // le tableau est relâché par le pool
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 **ARC**

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

ARC (Automatic Reference Counting)

- automatique sous Xcode 5
- gestion automatique de la mémoire
- plus d'appels à `release`, `retain` ou `autorelease` → ils sont rajoutés à la compilation par analyse statique du code qui évalue la durée de vie des objets

ARC (Automatic Reference Counting)

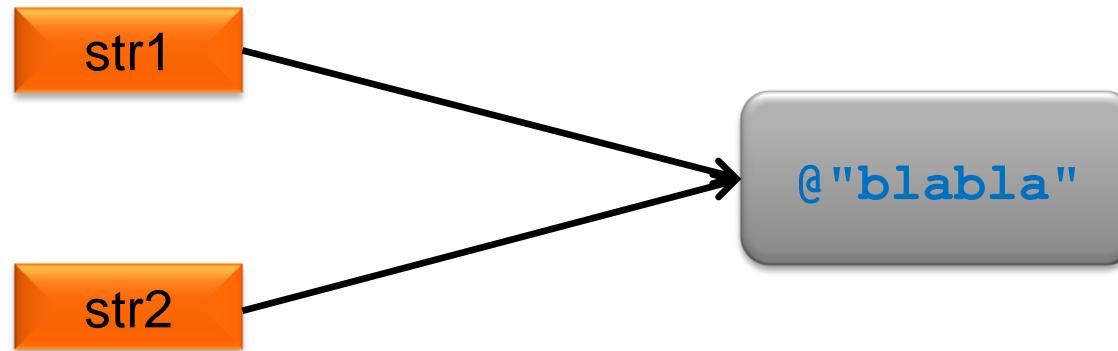
- nouveaux mots-clés à utiliser comme `const` :
 - `__weak` : pas de prolongation de la durée de vie de l'objet qui devient nul automatiquement. Utilisable pour des propriétés (`weak`).
 - `__strong` : durée de vie par défaut, l'objet existe tant qu'un objet pointe dessus. Utilisable pour des propriétés (`strong`).
 - `__unsafe_unretained` : identique à `weak`, mais le pointeur n'est pas remis à nul
 - `__autoreleasing` : pour indiquer des arguments passés par référence et relâchés automatiquement en retour de méthode
- en général plus besoin de surcharger la méthode `dealloc` sauf si la classe contient des variables locales car les propriétés sont relâchées automatiquement

ARC (Automatic Reference Counting)

relation strong

Exemple

```
__strong NSString *str1 = @"blabla";
__strong NSString *str2 = str1;
```

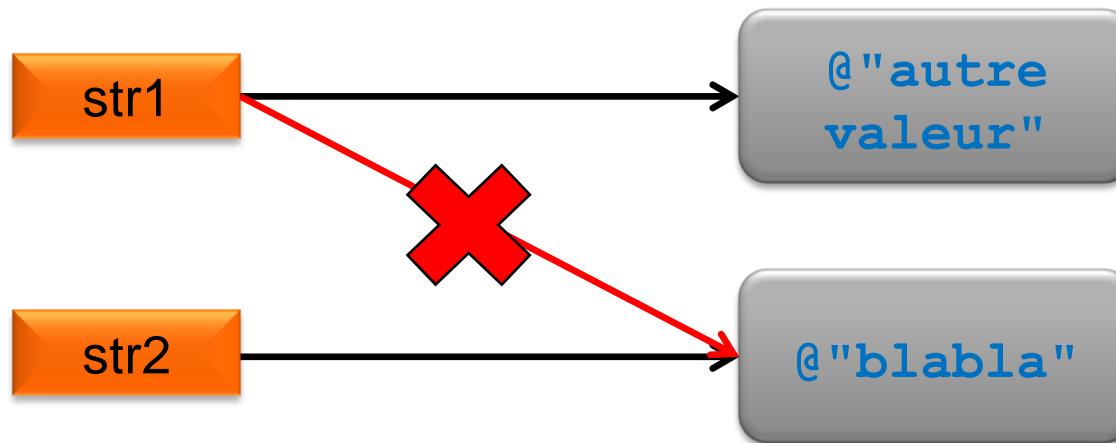


ARC (Automatic Reference Counting)

relation strong

Exemple

```
str1 = @"autre valeur";
```

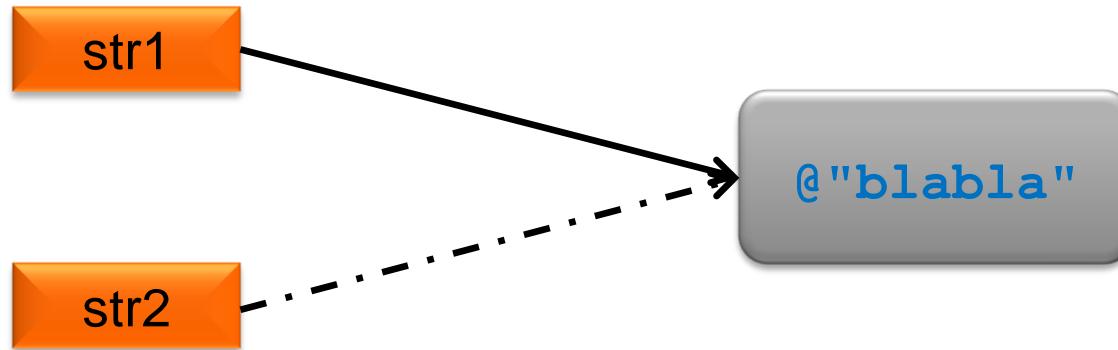


ARC (Automatic Reference Counting)

relation weak

Exemple

```
__strong NSString *str1 = @"blabla";
__weak NSString *str2 = str1;
```

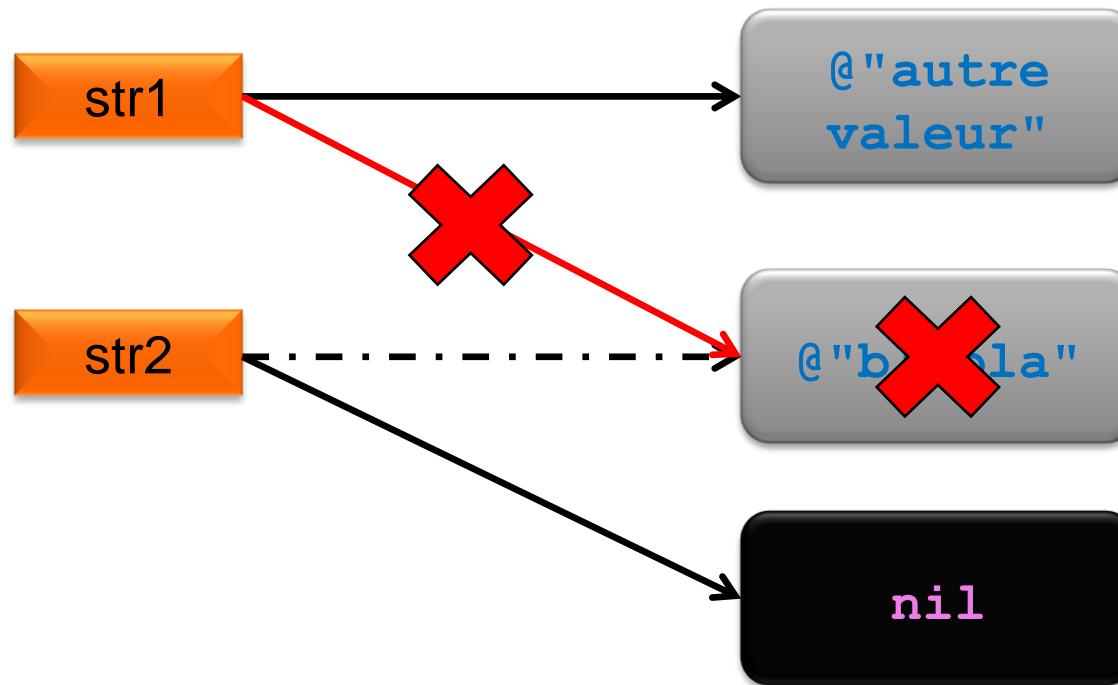


ARC (Automatic Reference Counting)

relation weak

Exemple

```
str1 = @"autre valeur";
```



sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 **création d'une classe**

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

création d'une classe

définition

- comme en C++, une classe est divisée en 2 fichiers :
 - fichier .h (header) : contenant sa déclaration
 - fichier .m : contenant son implémentation
- la déclaration d'une classe est une interface (ne pas confondre avec l'interface en Java)
- une interface hérite de `NSObject` et possède :
 - des variables membres
 - des propriétés
 - des méthodes d'instance
 - des méthodes de classe
- par convention, les méthodes d'initialisation sont préfixées par `init`

création d'une classe

définition

Exemple

```
// imports d'autres headers
@interface Evenement : NSObject
{
    // variables membres
    NSMutableArray *invites;
}

// propriétés
@property (nonatomic, strong) NSString *nom;
@property (nonatomic, strong) NSDate *date;

// méthodes d'instances
-(void) addInvite:(NSString*)invite;
-(void) removeInvite:(NSString*)invite;
-(NSArray*) invites;
-(instancetype) initWithNom:(NSString*)nom date:(NSDate*)date;

// méthodes de classe
+(instancetype) evenementWithNom:(NSString*)nom
date:(NSDate*)date;
@end
```

création d'une classe

définition - paramètres des propriétés

- une propriété peut être :
 - atomic ou nonatomic, i.e. thread-safe ou non
 - readwrite ou readonly
 - strong ou weak ou copy
- une propriété weak garde une référence forte à l'objet qui lui est assigné
- une propriété weak garde une référence faible à l'objet qui lui est assigné
- une propriété copy copie l'objet et pointe ensuite sur la copie
- pas de strong, weak ou copy sur une propriété de type primitif

création d'une classe

définition - appel d'une propriété

- 2 possibilités :
 - appel comme une méthode
 - utilisation de la notation pointée

Exemple

```
NSDate *nom = [monEvenement nom]; // getter  
[monEvenement setNom:@"evenement1"]; // setter  
// OU  
NSDate *nom = monEvenement.nom; // getter  
monEvenement.nom = @"evenement1"; // setter
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :

```
- (instancetype) initWithNom: (NSString*) nom date: (NSDate*) date;
```

- équivalent en Java :

```
publicinstancetype initWithNomAndDate (NSString nom, NSDate  
date) {}
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :

`- (instancetype) initWithNom: (NSString*) nom date: (NSDate*) date;`



Type de
retour

- équivalent en Java :

`publicinstancetype initWithNomAndDate (NSString nom, NSDate date) {}`

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :

```
- (instancetype) initWithNom: (NSString*) nom date: (NSDate*) date;
```

↑
Type de retour

↓
Étiquette 1

- équivalent en Java :

```
publicinstancetype initWithNomAndDate (NSString nom, NSDate  
date) {}
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :

The diagram shows the Objective-C method declaration `- (instancetype) initWithNom: (NSString*) nom date: (NSDate*) date;`. A green arrow labeled "Étiquette 1" points to the colon after "initWithNom". A pink arrow labeled "Type de retour" points to the `instancetype` keyword. A purple arrow labeled "Type du paramètre1" points to the `NSString*` type of the first parameter.

```
Étiquette 1
↓
- (instancetype) initWithNom: (NSString*) nom date: (NSDate*) date;
↑
Type de retour
↑
Type du paramètre1
```

- équivalent en Java :

```
publicinstancetype initWithNomAndDate(NSString nom, NSDate
date){}
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :

The diagram shows the following Objective-C method declaration with annotations:

```
- (instancetype) initWithNom: (NSString*) nom date: (NSDate*) date;
```

- ↑ Type de retour (pink arrow)
- ↓ Étiquette 1 (green arrow)
- ↑ Type du paramètre1 (purple arrow)
- ↓ Nom du paramètre1 (black arrow)

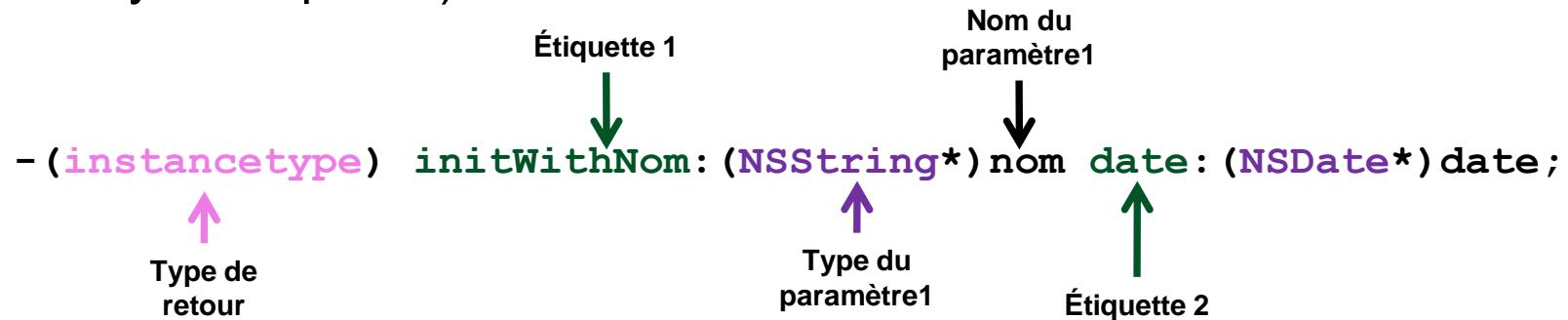
- équivalent en Java :

```
publicinstancetype initWithNomAndDate(NSString nom, NSDate  
date){}
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :



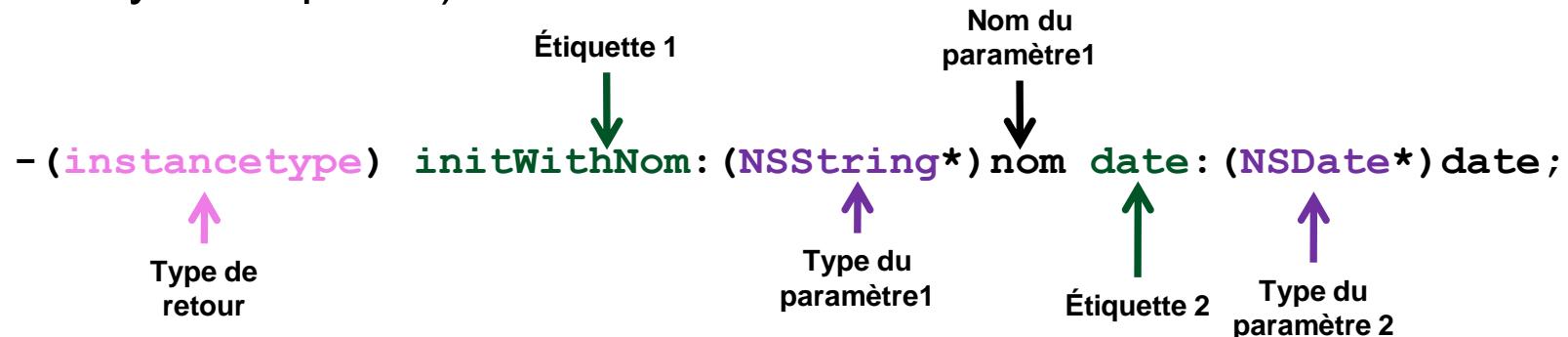
- équivalent en Java :

```
publicinstancetype initWithNomAndDate(NSString nom, NSDate  
date){}
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :



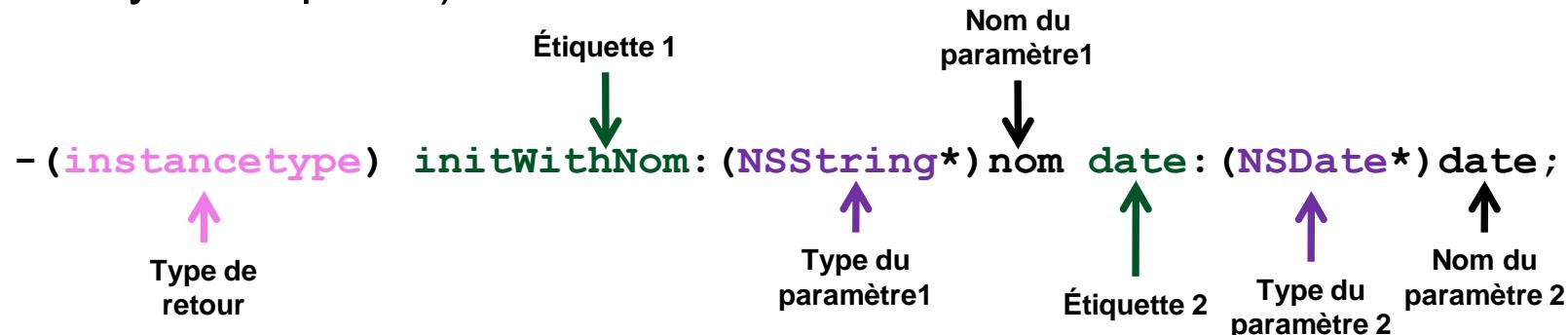
- équivalent en Java :

```
publicinstancetype initWithNomAndDate(NSString nom, NSDate  
date){}
```

création d'une classe

définition - déclaration de méthode

- 2 types :
 - d'instance : à appeler sur un objet
 - de classe : à appeler sur une classe
- la déclaration une méthode comprend un type, un type de retour et des étiquettes (chaque étiquette devant avoir un paramètre, sauf s'il n'y en a qu'une) :



- équivalent en Java :

```
publicinstancetype initWithNomAndDate(NSString nom, NSDate  
date){}
```

création d'une classe

définition - déclaration de méthode

- la signature de la méthode est `initWithNom:date:`
-  2 méthodes ne peuvent pas avoir la même signature (même si les paramètres sont de types différents)

création d'une classe

définition - déclaration de méthode

- une méthode de classe est l'équivalent d'une méthode statique en Java
- sert généralement à déclarer des constructeurs par « commodité » qui évite de faire un `alloc+init`
- par convention, un constructeur par commodité commence par le nom de classe, par exemple :
 - méthode `stringWithFormat:` de la classe `NSString`
 - méthode `arrayWithObjects:` de la classe `NSArray`

création d'une classe

implémentation

- l'implémentation d'une interface contient l'ensemble des méthodes définies dans sa déclaration
- en général, il faut surcharger la méthode de destruction de l'objet dealloc, sauf si l'interface ne retient aucune propriété ou variable membre

création d'une classe

implémentation

Exemple (1/2)

```
// imports de la définition de l'interface
#import "Evenement.h"
@implementation Evenement
@synthesize nom = _nom;
@synthesize date = _date;
-(instancetype) initWithNom:(NSString*)nom date:(NSDate*)date {
    self = [super init]; // appel obligatoire de la classe mère
    if(self) { // test que l'initialisation n'a pas échoué
        self.nom = nom;
        self.date = date;
        invites = [NSMutableArray new];
    }
    return self;
}
```

création d'une classe

implémentation

Exemple (2/2)

```
// méthodes de classe (utilisation d'autorelease)
+(instancetype) evenementWithNom:(NSString*)nom
date:(NSDate*)date{
    Evenement *objet = [[[self class] alloc] initWithNom:nom
date:date];
    return objet;
}
@end // OBLIGATOIRE, fin de l'implémentation
```

création d'une classe

implémentation - propriétés

- plusieurs possibilités :
 - manuellement
 - automatiquement
 - dynamiquement

création d'une classe

implémentation – propriétés manuelles

- il faut implémenter les méthodes de récupération/modification en fonction de ce qui la déclaration de la propriété

Exemple

```
- (NSString*) nom {
    return nom;
}
-(void) setNom:(NSString*) autreNom {
    // on teste que le pointeur sur le nouveau nom est différent
    if(nom != autreNom) {
        nom = autreNom;
    }
}
```

création d'une classe

implémentation – propriétés automatiques

- la directive `@synthesize` permet de ne pas implémenter soi-même des méthodes comme vu précédemment
- les méthodes sont générées à la compilation
- on peut coupler l'implémentation manuelle et automatique, par exemple, si la modification du nom d'un évènement entraîne le relâchement de sa date

création d'une classe

implémentation – encapsulation des données

- l'accès aux variables membres de la classe est configurable via des directives :
 - **@private** : la variable n'est accessible que dans l'implémentation
 - **@protected** : la variable n'est accessible que dans l'implémentation et dans celle des sous-classes
 - **@public** : la variable est accessible de partout
-  les variables membres publiques ne doivent être accédées qu'en lecture, sinon risque de problème mémoire
-  de même pour les variables protégées, il vaut mieux n'y accéder qu'en lecture car on ne connaît pas forcément leur utilisation dans la classe mère
-  ces directives ne s'appliquent pas sur les propriétés

création d'une classe

implémentation – encapsulation des données

Exemple

```
@interface MaClasse: NSObject
{
    @private
    NSInteger privateInt;
    @protected
    CGFloat protectedFloat;
    @public
    NSString *publicString;
}
@end

// l'accès aux données publiques se fait ensuite comme ceci
MaClasse *objet = [MaClasse new];
NSLog(objet->publicString);
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 **étendre une classe**

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

étendre une classe

l'héritage

- il n'y a pas d'héritage multiple



l'héritage de `NSObject` n'est pas implicite mais Xcode l'ajoute automatiquement en général

Exemple

```
// imports d'autres headers
@interface Evenement : NSObject
{// héritage de NSObject
}
@end
```

étendre une classe

les catégories - définition

- permettent d'ajouter des méthodes à une classe



ce n'est pas de l'héritage

- pas d'ajout de variables membres ou de propriétés
- méthodes ajoutées dynamiquement à la classe
- les nouvelles méthodes sont accessibles à tout le monde
- possibilité de surcharger une méthode dans certains cas



la portée de la surcharge est sur toute l'application

étendre une classe

les catégories - création

- par convention, les fichiers sources pour une catégorie sont nommés de la forme : <NomClasse>+<NomCategorie>.h (et .m)
- exemple : Evenement+Demo.h

Exemple

```
#import "Evenement.h">// import de la définition de l'interface
@interface Evenement (Demo)
-(NSString*) detail;
@end
@implementation Evenement (Demo)
-(NSString*) detail{
    return [NSString stringWithFormat:@"detail : %@ à %@", self.nom, self.date];
}
@end
```

étendre une classe

les catégories - utilisation

- par convention, les fichiers sources pour une catégorie sont nommés de la forme : <NomClasse>+<NomCategorie>.h (et .m)
- exemple : Evenement+Demo.h

Exemple

```
#import "Evenement+Demo.h" // import de la définition de la
catégorie
...
Evenement *objet = [Evenement new];
NSLog([objet detail]);
...
@end
```

étendre une classe

les extensions - définition

- ce sont des catégories anonymes
- la définition doit se faire dans le fichier d'implémentation de la classe originale, l'implémentation dans l'implémentation de la classe originale
- permet de définir des variables membres et propriétés



A quoi ça sert ?

- à définir des variables, méthodes et propriétés non visibles de l'extérieur
- redéfinir une propriété de lecture seule à lecture+écriture



cela n'empêche pas d'utiliser ces méthodes si on en a connaissance → en Objective-C, aucune méthode n'est privée

étendre une classe

les extensions - implémentation

Exemple

```
// implémentation dans le fichier Evenement.m
#import "Evenement.h"
@interface Evenement () // définition de l'extension
-(NSString*) test;
@end
@implementation Evenement// implémentation de la classe Evenement
...
-(NSString*) test{
    return @"blabla";
}
...
@end
```

étendre une classe

les protocoles - définition

- l'équivalent des interfaces en Java
- définit un contrat d'interface, i.e. un ensemble de méthodes qui peuvent être :
 - requises : la classe qui est conforme au protocole doit implémenter ces méthodes
 - optionnelles : la classe qui est conforme au protocole peut implémenter ces méthodes mais n'est pas obligée
- implique la nécessité de tester que la classe implémente une méthode optionnelle
- possibilité d'hériter d'un autre protocole, par convention un protocole hérite du protocole `NSObject` (attention différent de la classe)

étendre une classe

les protocoles - définition

Exemple

```
@protocol MonProtocole <NSObject>
@optional
- (NSString*) optionalMethod;
@required
- (int) requiredIntMethod;
@end
```

étendre une classe

les protocoles – déclaration d'une variable

Exemple

```
#import "MaClasse.h"
@interface MonAutreClasse : NSObject {
    id<MonProtocole> variableConforme;
}
@property (nonatomic, strong) id<MonProtocole> proprieteConforme;
// on peut également écrire
@property (nonatomic, weak) NSObject<MonProtocole>
*proprieteConforme2;
@end
```

étendre une classe

les protocoles - utilisation

- utilisation de la méthode `respondsToSelector:` définie dans le protocole `NSObject` (d'où l'importance d'hériter de ce protocole)

Exemple

```
...
- (NSString*) testMethod{
    // appel d'une méthode requise
    NSInteger i = [self.proprieteConforme requiredIntMethod];
    // appel d'une méthode optionnelle
    if([self.proprieteConforme
        respondsToSelector:@selector(optionalMethod)]) {
        NSString *s = [self.proprieteConforme optionalMethod];
    }
}
```

étendre une classe

les protocoles - déclaration d'une classe conforme

Exemple

```
#import "MaClasse.h"
@interface MonClasseConforme : NSObject <MonProtocole> {
    ...
}
...
@end
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 **délégation, dataSource et notifications**

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

délégation, dataSource et notifications

délégation

- un **délégué** est objet qui agit **au nom de**, ou **en coordination avec**, un autre objet
- il peut être utilisé dans plusieurs cas, par exemple :
 - une vue indique à son délégué qu'un de ses sous-vues a été sélectionnée
 - un objet responsable d'un téléchargement indique son avancement ou une erreur rencontrée
- mécanisme similaire aux **listeners** en Java
- un objet n'a en général **qu'un seul délégué**

délégation, dataSource et notifications

délégation - en pratique

- un délégué est objet implémenté un **protocole**
- la signature des méthodes suit une convention de nommage et commence toujours par le nom de **la classe de l'objet qui délègue** sans préfixe, et prend comme premier paramètre **l'objet qui délègue**
- après le nom de la classe, on trouve **en général** un verbe qui indique la temporalité de l'événement :
 - should ou will pour indiquer un événement qui va arriver
 - did ou has pour indiquer un événement qui s'est produit
- un délégué est déclaré sous forme d'une propriété **en weak**:

Exemple

```
@property (nonatomic, weak) id<UIApplicationDelegate>delegate;
```

délégation, dataSource et notifications

délégation - en pratique

Exemple

```
- (BOOL) application: (UIApplication*) application  
    shouldSaveApplicationState: (NSCoder*) coder;  
  
- (void) tableView: (UITableView*) tableView  
    didSelectRowAtIndexPath: (NSIndexPath*) indexPath;  
  
- (NSUInteger) application: (UIApplication*) application  
    supportedInterfaceOrientationsForWindow: (UIWindow*) window;
```

délégation, dataSource et notifications

délégation - en pratique

Exemple

```
- (BOOL) application: (UIApplication*) application  
shouldSaveApplicationState: (NSCoder*) coder;  
  
- (void) tableView: (UITableView*) tableView  
didSelectRowAtIndexPath: (NSIndexPath*) indexPath;  
  
- (NSUInteger) application: (UIApplication*) application  
supportedInterfaceOrientationsForWindow: (UIWindow*) window;
```

nom de la classe
qui délègue

délégation, dataSource et notifications

délégation - en pratique

Exemple

```
- (BOOL) application: (UIApplication*) application  
    shouldSaveApplicationState: (NSCoder*) coder;  
  
- (void) tableView: (UITableView*) tableView  
    didSelectRowAtIndexPath: (NSIndexPath*) indexPath;  
  
- (NSUInteger) application: (UIApplication*) application  
    supportedInterfaceOrientationsForWindow: (UIWindow*) window;
```

verbe clé

délégation, dataSource et notifications

dataSource

- un **dataSource** est objet similaire au délégué mais qui **contrôle les données** d'un autre objet
- il peut être utilisé dans plusieurs cas, par exemple :
 - combien de sections ma liste contient-elle ?
 - quel est l'élément à afficher à tel endroit dans ma liste ?

délégation, dataSource et notifications

dataSource - en pratique

- un dataSource est objet implémenté un **protocole**
- la signature des méthodes suit une convention de nommage et commence, en général, par le nom de **la classe à qui fournir les données** sans préfixe, et prend comme premier paramètre **l'objet à qui fournir les données**
- un dataSource est déclaré sous forme d'une propriété en `weak`:

Exemple

```
@property (nonatomic, weak) id<UITableViewDataSource>dataSource;
```

délégation, dataSource et notifications

dataSource - en pratique

Exemple

```
- (NSInteger) numberOfSectionsInTableView: (UITableView*) tableView;  
  
- (UITableViewCell*) tableView: (UITableView*) tableView  
    cellForRowAtIndexPath: (NSIndexPath*) indexPath;
```

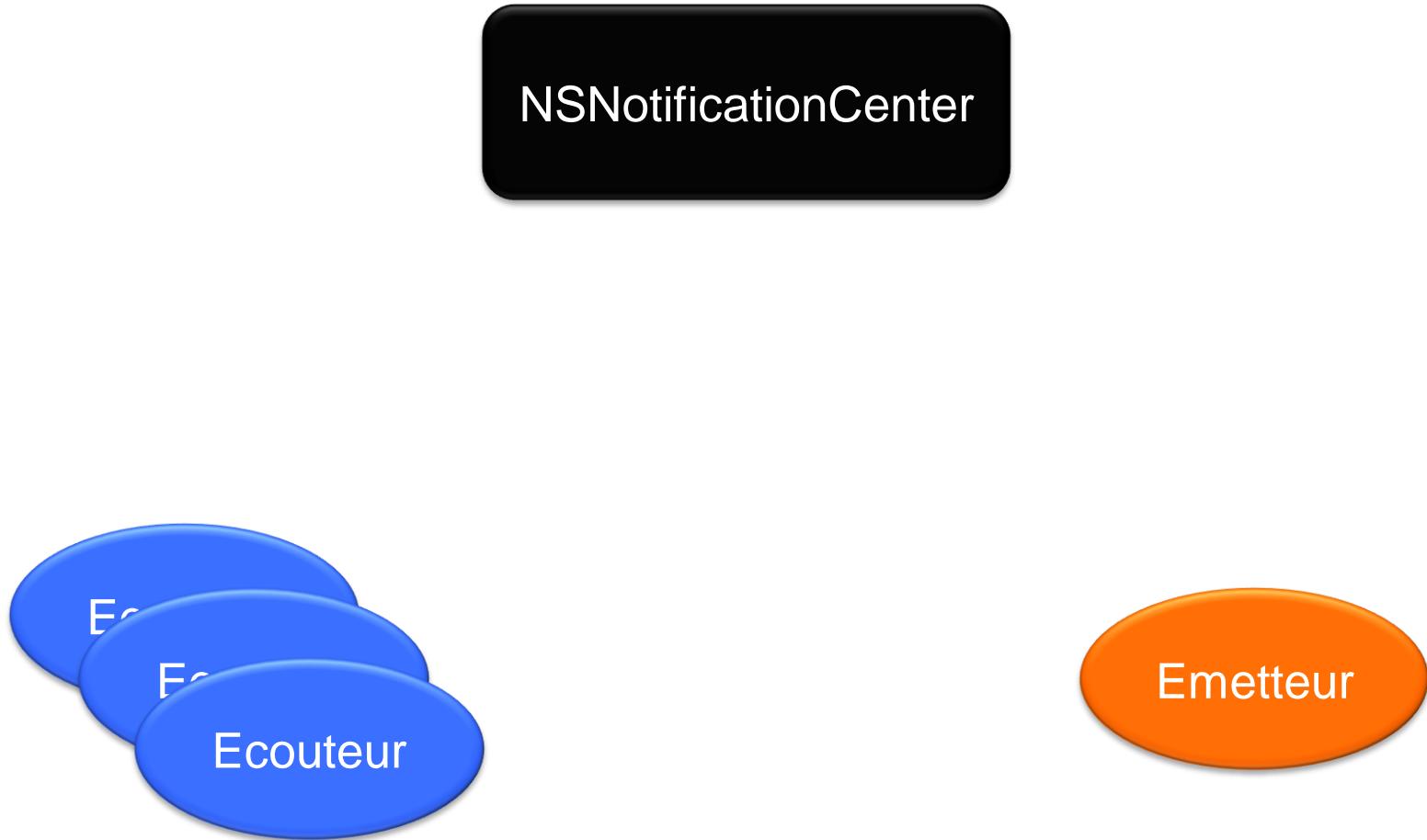
délégation, dataSource et notifications

notifications

- une notification permet d'envoyer un message à **plusieurs écouteurs**
- une notification a :
 - un nom
 - un émetteur (optionnel)
 - des informations complémentaires sous forme d'un dictionnaire (optionnel)
- pour écouter une notification, il faut s'enregistrer auprès du **centre de notifications**
- pour envoyer une notification, il faut passer par le **centre de notifications**

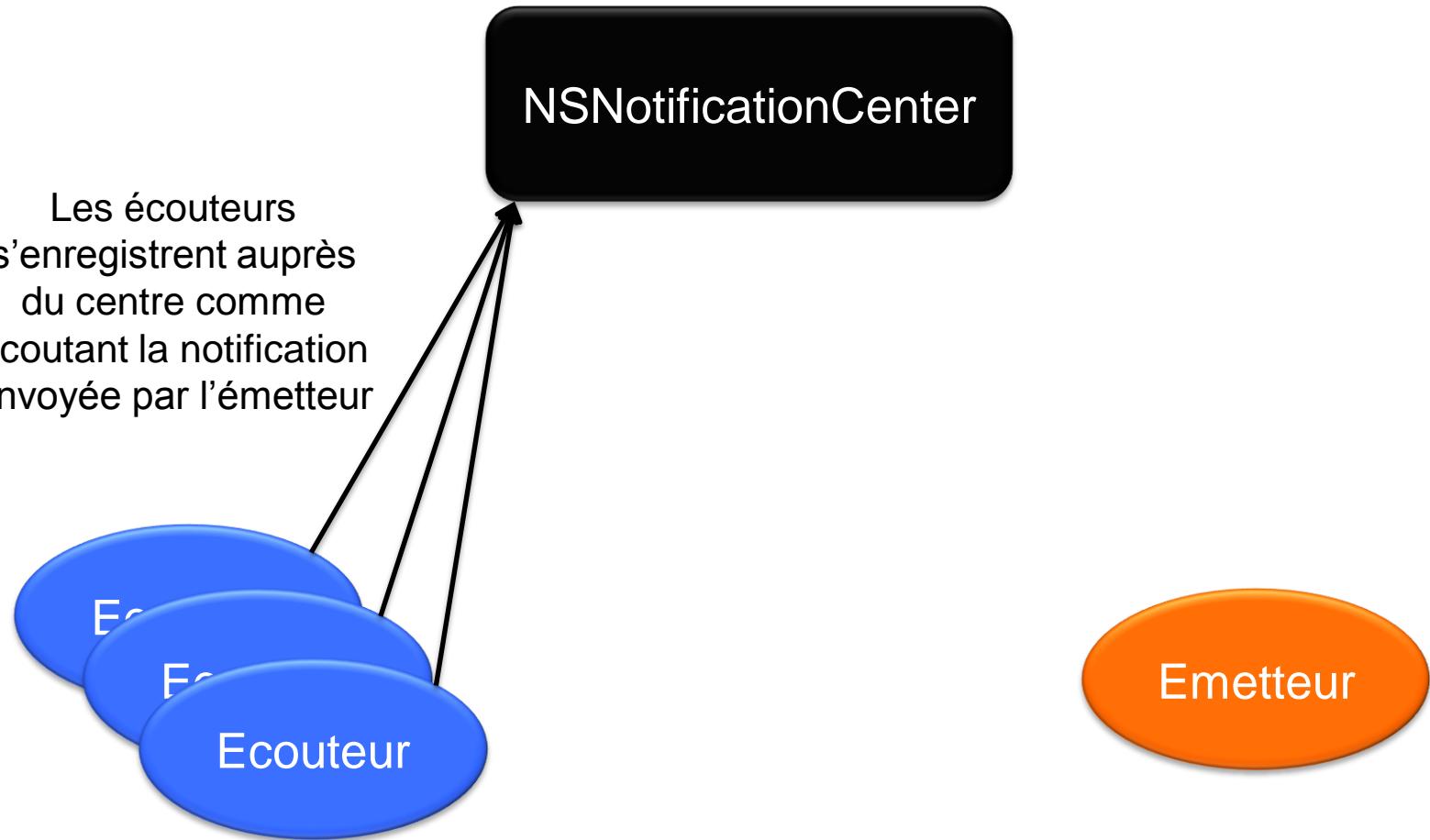
délégation, dataSource et notifications

notifications - fonctionnement



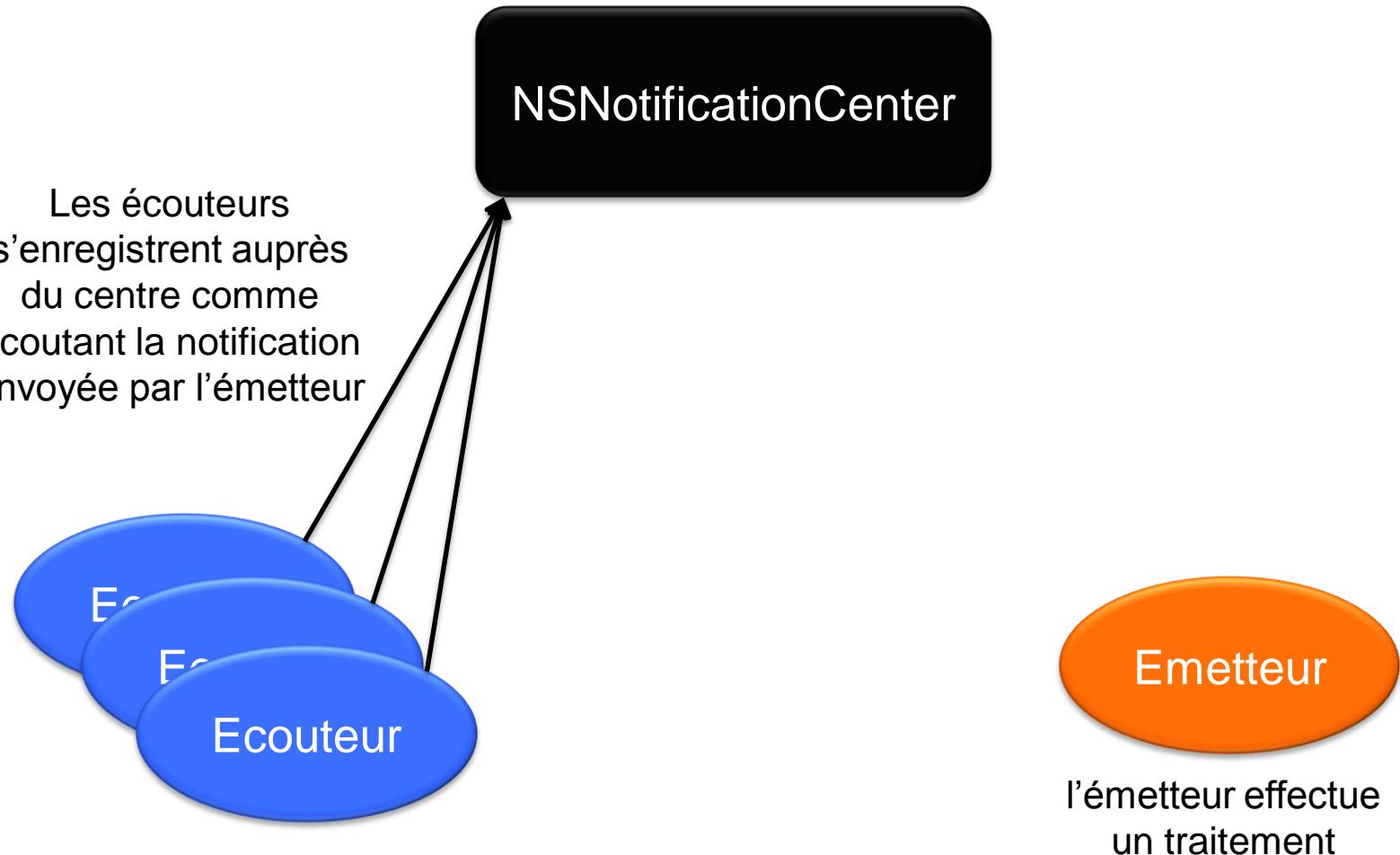
délégation, dataSource et notifications

notifications - fonctionnement



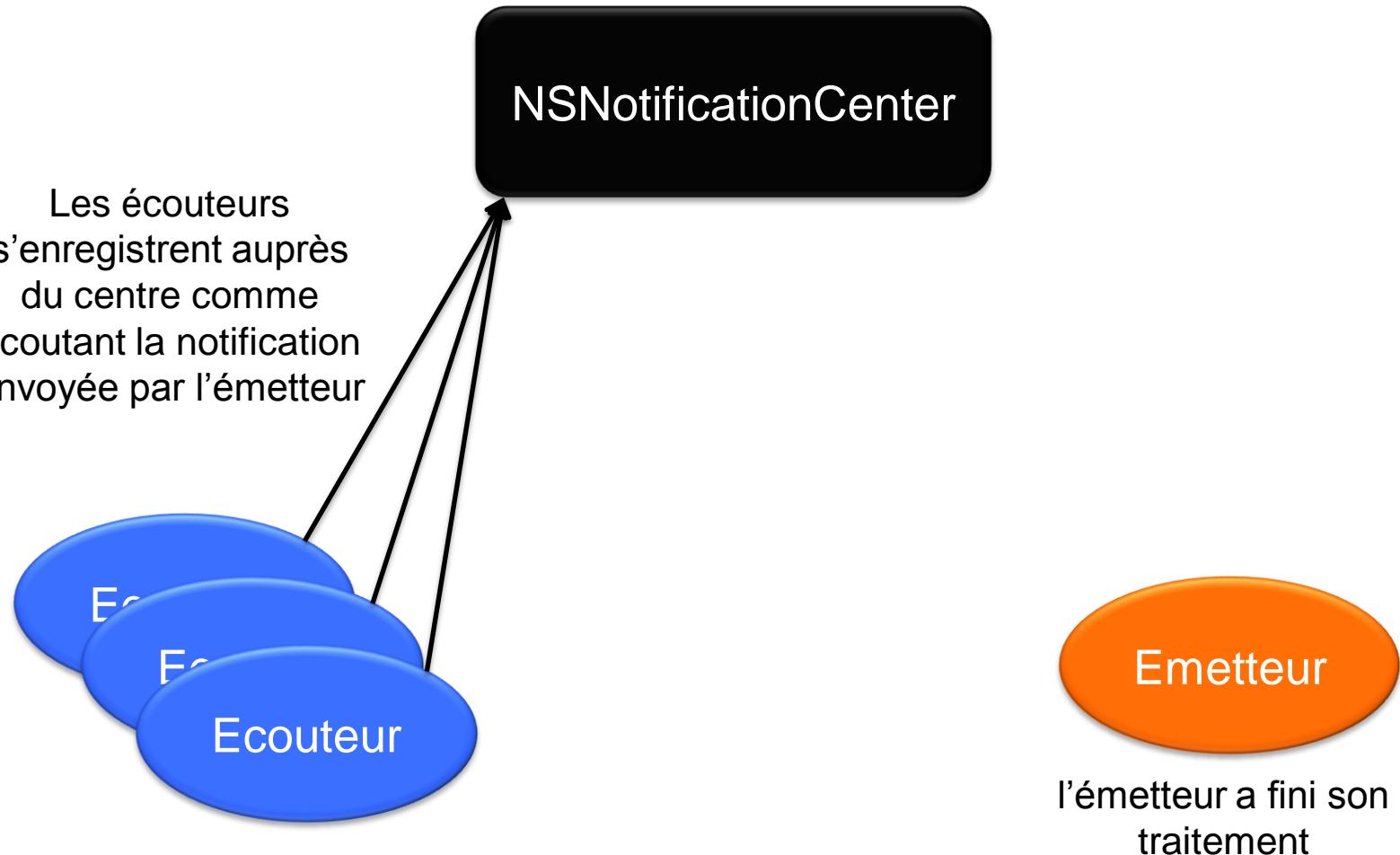
délégation, dataSource et notifications

notifications - fonctionnement



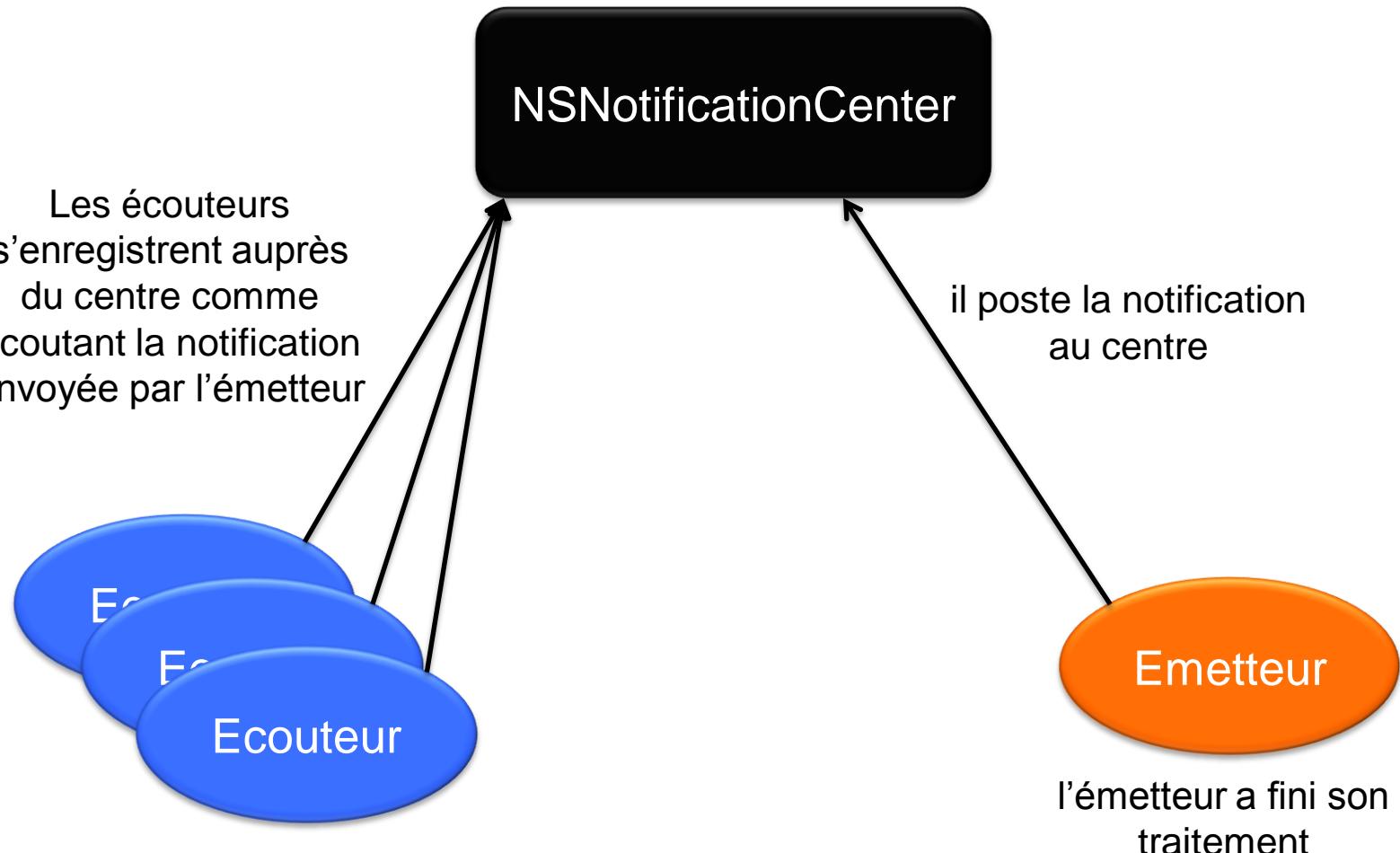
délégation, dataSource et notifications

notifications - fonctionnement



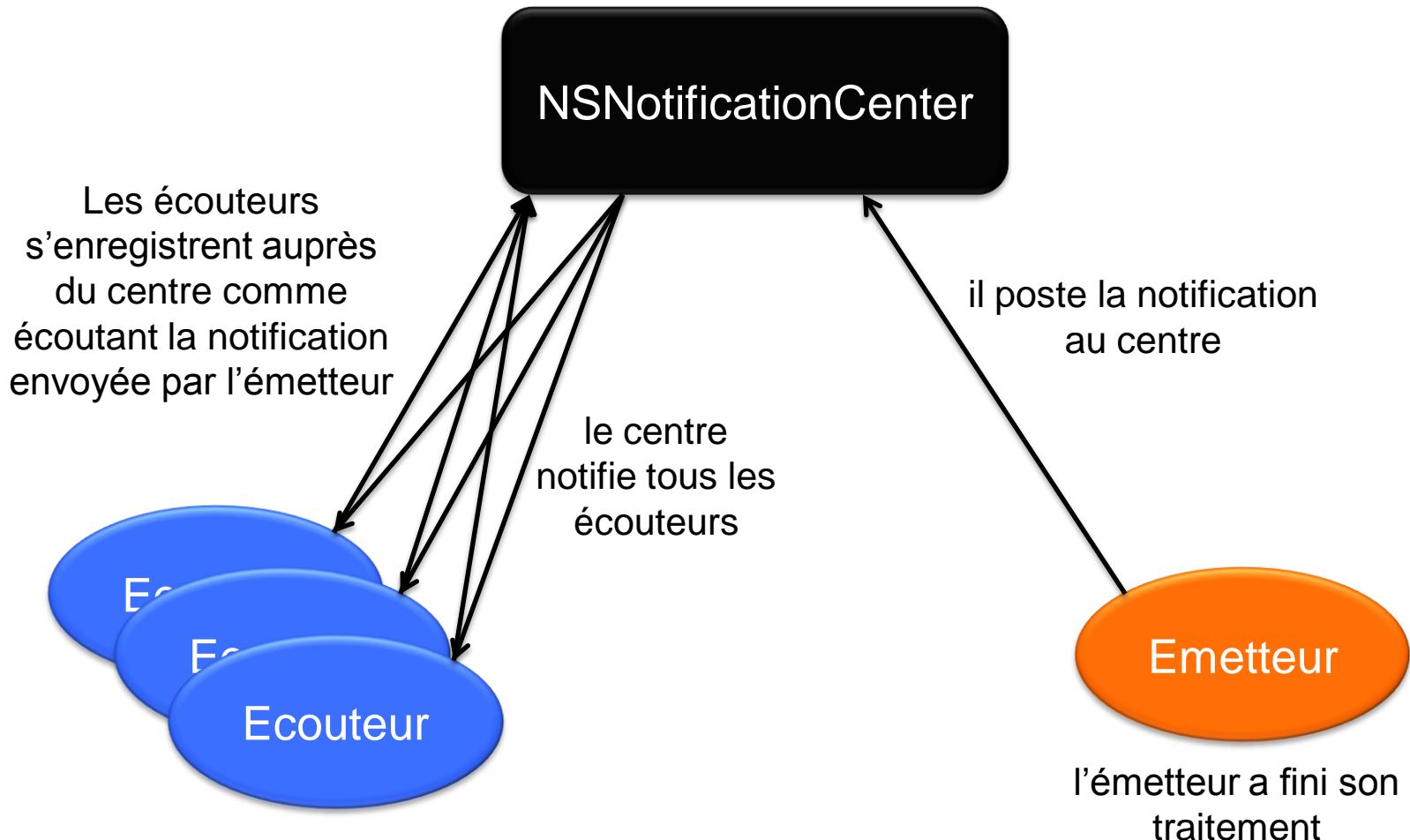
délégation, dataSource et notifications

notifications - fonctionnement



délégation, dataSource et notifications

notifications - fonctionnement



délégation, dataSource et notifications

notifications

- iOS émet déjà beaucoup de notifications :
 - affichage/masquage du clavier
 - l'application passe en arrière ou premier plan
 - changement des préférences de l'application
 - un affichage externe a été connecté/déconnecté
 - ...



un élément graphique ou contrôleur qui est enregistré auprès du centre de notifications doit se désinscrire lorsqu'il n'est plus visible



ne pas confondre avec les notifications de type push ou locales

délégation, dataSource et notifications

notifications - en pratique

Exemple

```
// enregistrement d'une saisie clavier dans un champ
// de texte auprès du centre
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(notificationMethodCalled:)
name:UITextFieldDidChangeNotification object:oneTextField];

// désinscription
[[NSNotificationCenter defaultCenter] removeObserver:self
name:UITextFieldDidChangeNotification object:oneTextField];

// méthode recevant la notification
-(void)notificationMethodCalled:(NSNotification*)notification
{
}
```

délégation, dataSource et notifications

notifications - en pratique

Exemple

```
// envoi d'une notification
[[NSNotificationCenter defaultCenter]
    postNotificationName:MyCustomNotification object:self
    userInfo:@{}];
```

délégation, dataSource et notifications

notifications - en pratique

Exemple

```
// envoi d'une notification
[[NSNotificationCenter defaultCenter]
    postNotificationName:MyCustomNotification object:self
    userInfo:@{}];
```



émetteur

délégation, dataSource et notifications

notifications - en pratique

Exemple

```
// envoi d'une notification
[[NSNotificationCenter defaultCenter]
    postNotificationName:MyCustomNotification object:self
    userInfo:@{}];
```

dictionary libre pour
ajouter du contenu à la
notification, peut être nul

émetteur

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 **les blocks**

partie 9 concurrence

partie 10 modern Objective-C

partie 11 KVC et KVO

les blocks

définition

- un morceau de code équivalent à une fonction
- exécuté « inline », on l'affecte à une variable ou en paramètre
- peut lire ou modifier des variables du bloc supérieur, des variables de classe
- un block est un objet
- apparu avec iOS 4

les blocks

usage

- fonctions de tri ou de filtrage
- callback sur un appel asynchrone (similaire à l'utilisation des listeners en Java)

les blocks

syntaxe

Exemple

```
type (^nom)(type_param1, ..., type_paramN) = ^(param p1, ..., param pn){  
    ...  
}  
// un exemple simple  
// déclaration  
int (^multiplication)(int, int) = ^(int p1, int p2){  
    return p1 * p2;  
}  
// utilisation  
int total = multiplication(4, 2); // 8
```

les blocks

utilisation d'une variable du contexte

- on peut accéder à une variable du contexte dans le block mais pas la modifier directement

Exemple 1

```
int x = 3;
void (^printXAndY) (int) = ^(int y) {
    NSLog(@"%@", x, y);
}
printXAndY(2); // écrit: 3 2
```

Exemple 2

```
int x = 3;
void (^printXAndY) (int) = ^(int y) {
    x += y; // erreur, ce n'est pas autorisé
    NSLog(@"%@", x, y);
}
printXAndY(2);
```

les blocks

utilisation d'une variable du contexte

- pour modifier une variable du contexte, il faut utiliser le mot-clé `_block` qui indique que la variable est en lecture-écriture

Exemple

```
_block int x = 3;
void (^printXAndY) (int) = ^(int y) {
    x += y;
    NSLog(@"%@", x, y);
}
printXAndY(2); // écrit: 5 2
// x vaut maintenant 5
```

les blocks

utilisation en paramètre

- un block peut être passé en paramètre de méthode ou de fonction

Exemple : tri d'un tableau de chaînes en une fois

```
NSMutableArray *tableau = ...;
// utilisation de la méthode sortWithComparator: qui prend
// en paramètre un block de type NSComparator
// signature de NSComparator :
// typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
_block int x = 3;
[tableau sortWithComparator:^(NSComparisonResult)(id obj1, id
obj2) {
    NSString *chaine1 = (NSString *)obj1;
    NSString *chaine2 = (NSString *)obj2;
    return [chaine1 compare:chaine2];
}];
```

les blocks

utilisation en paramètre

Exemple : tri d'un tableau de chaînes en deux fois

```
NSMutableArray *tableau = ...;
NSComparisonResult (^compareChaines)(id, id) =
    ^(id obj1, id obj2) {
        NSString *chaine1 = (NSString *)obj1;
        NSString *chaine2 = (NSString *)obj2;
        return [chaine1 compare:chaine2];
};
[tableau sortWithComparator:compareChaines];
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 **concurrence**

partie 10 modern Objective-C

partie 11 KVC et KVO

concurrence

introduction

- plusieurs possibilités :
 - utilisation de `NSThread`
 - utilisation des méthodes de `NSObject`
 - utilisation de Grand Central Dispatch (GCD)
 - utilisation des opérations

concurrence

utilisation de NSThread

- 2 possibilités :
 - utilisation de la méthode
`detachNewThreadSelector:toTarget:withObject`
 - création d'un thread et démarrage avec la méthode `start`

concurrence

utilisation de NSThread

```
- (void) aThreadMethod: (NSObject*) object {
// ne pas utiliser le pool d'autorelease
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
    // code de la méthode
    ...
    [pool drain];
}

...
// utilisation
[NSThread detachNewThreadSelector:@selector(aThreadMethod:)
    toTarget:self withObject:anObject];
// ou
NSThread *thread = [[NSThread alloc]
    detachNewThreadSelector:@selector(aThreadMethod:)
    toTarget:self withObject:anObject];
[thread start];
...
```

concurrence

utilisation des méthodes de NSObject

- NSObject propose des méthodes pour exécuter un sélecteur dans un thread :
 - `performSelectorInBackground withObject:`
 - `performSelector:onThread withObject:waitUntilDone:modes:`
 - `performSelectorOnMainThread withObject:waitUntilDone:modes:` (appel le sélecteur sur le thread principal, donc utile pour la mise à jour de l'interface graphique)

concurrence

GCD - dispatch queues

- exécution simple de tâches concurrentes ou en série
- First In First Out
- 3 types :
 - en série (private dispatch queue) : les tâches sont exécutées les unes après les autres
 - concurrente (global dispatch queue) : les tâches sont exécutées en même temps
 - principale (main dispatch queue) : une queue en série qui exécute les tâches dans le thread principal
- exécution synchrone ou asynchrone
- ~~fonctionnent avec un compteur de références, il faut penser à les retenir (`dispatch_retain`) ou les relâcher (`dispatch_release`)~~

concurrence

GCD - dispatch queues - main queue

- s'exécute dans le thread principal
- créée automatiquement
- permet la mise à jour de l'interface graphique

Récupération

```
...
    dispatch_queue_t main_queue = dispatch_get_main_queue();
...

```

concurrence

GCD - dispatch queues - serial queue

- pour effectuer des tâches dans un ordre précis
- permet d'éviter l'utilisation de « lock » sur des ressources
- doivent être créées explicitement, et relâchées quand on en a plus besoin

Création

```
dispatch_queue_t serial_queue;
serial_queue = dispatch_queue_create("com.example.myQueue", NULL);
// le paramètre NULL peut être remplacé par DISPATCH_QUEUE_SERIAL
```

concurrence

GCD - dispatch queues - global queue

- pour effectuer des tâches en même temps
- possibilité d'en créer soi-même mais peu utiliser (utilisation de la fonction `dispatch_queue_create` en mettant en second paramètre `DISPATCH_QUEUE_CONCURRENT`)
- des queues globales existent par niveau de priorité :
 - `DISPATCH_QUEUE_PRIORITY_HIGH`
 - `DISPATCH_QUEUE_PRIORITY_DEFAULT`
 - `DISPATCH_QUEUE_PRIORITY_LOW`
 - `DISPATCH_QUEUE_PRIORITY_BACKGROUND`

Exemple

```
dispatch_queue_t globale_queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_LOW, NULL);
```

concurrence

GCD - dispatch queues - utilisation

```
dispatch_queue_t queue = ...;
serial_queue = dispatch_queue_create("com.exemple.myQueue", NULL);
// appel synchrone
dispatch_sync(queue, ^{
    // block de code
});
// appel asynchrone
dispatch_async(queue, ^{
    // block de code
});
```

concurrence

GCD - dispatch queues - utilisation

Exemple avec un block de compléction pour une addition

```
void addition(float *data, size_t len, dispatch_queue_t queue,
              void (^callback)(float)) {
    // on retient la queue pour qu'elle ne disparaisse pas
    dispatch_retain(queue);
    float total = 0.f;
    for(size_t i=0; i<len; i++) {
        total += data[i];
    }
    dispatch_async(queue, ^{
        callback(total);
    });
    // on relâche la queue une fois fini
    dispatch_release(queue);
}
```

concurrence

les opérations

- permettent d'effectuer des traitements asynchrones
- gèrent une priorité
- gèrent l'annulation
- gèrent des dépendances entre elles
- peuvent être monitorées, i.e. on peut connaître leur état d'exécution

concurrence

les opérations

- **classe de base** NSOperation
- **la classe** NSOperationQueue **gère une file d'opérations**

concurrence

les opérations - définition d'une opération

- chaque opération doit posséder au minimum :
 - une méthode d'initialisation
 - une méthode `main` qui contient le code d'exécution de l'opération
- l'annulation est gérée via la méthode `isCancelled`, l'opération doit appeler celle-ci régulièrement pour pouvoir s'interrompre le plus vite possible
- on démarre une opération avec la méthode `start`

concurrence

les opérations - définition d'une opération

Exemple d'implémentation de main

```
...
-(void) main{
    // on crée un pool car on est dans un thread secondaire
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
    @try {
        // code d'exécution de l'opération
    }
    @catch (NSEException *e) {
    }
    @finally {
        // nettoyage du pool dans finally au cas où une
        // exception aurait été levée
        [pool drain];
    }
}
```

concurrence

les opérations - les files

- permettent d'exécuter plusieurs opérations en même temps
- peut annuler toutes ses opérations en cours
- peut être suspendue

Exemple

```
...
NSOperationQueue *queue = [NSOperationQueue new];
// on indique que la file ne peut pas exécuter
// plus de 2 opérations en même temps
queue.maxConcurrentOperationCount = 2;
MyOperation *operation = [MyOperation new];
// l'opération démarre dès son ajout à la file
[queue addOperation:operation];
...
...
```

concurrence

synchronisation

- utilisation de la directive **@synchronized**

```
@synchronized(unObjet) { // verrou sur cet objet
    /* Section critique */
}
```

- utilisation de la classe NSLock

```
NSLock *lock = [NSLock new];
...
[lock lock]; // verrouillage
/* Section critique */
[lock unlock]; // déverrouillage
...
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 **modern Objective-C**

partie 11 KVC et KVO

modern Objective-C

- disponible à partir de Xcode 4.4 (LLVM Compiler 4.0)
- simplifie la création d'objets de type :
 - nombre
 - tableau
 - dictionnaire
- utilisation du symbole `@` comme pour les chaînes de caractères
les objets créés ne sont pas modifiables, il faut utiliser la méthode `mutableCopy` pour les transformer en objet modifiables



modern Objective-C

nombres

Avant

```
[NSNumber numberWithBool:YES];
[NSNumber numberWithBool:NO];

[NSNumber numberWithIntUnsigned:5];
[NSNumber numberWithInt:49];

[NSNumber numberWithLong:1205];
[NSNumber numberWithLongLong:42];

[NSNumber numberWithFloat:12.78f];
[NSNumber numberWithDouble:87.63f];
```

Après

```
@YES;
@NO;

@5U;
@49;

@1205L;
@42LL;

@12.78F;
@87.63;
```

modern Objective-C

nombres - expressions

- possibilité de faire des opérations
- exemple pour calculer PI sur 12

Avant

```
[NSNumber numberWithDouble:M_PI / 12.0];
```

Après

```
@(M_PI / 12.0);
```

modern Objective-C

nombres - enumérés

- création directe d'un nombre en fonction du type de l'énuméré

Exemple d'énuméré

```
typedef enum
{
    Red,
    Green
} Color;
```

Avant

```
NSNumber *red = [NSNumber
                  numberWithInt:Red];
```

Après

```
NSNumber *red =
@(Red);
```

modern Objective-C

nombres - enumérés

- si le type d'enuméré est unsigned char

Exemple d'énuméré

```
typedef enum : unsigned char
{
    Red,
    Green
} Color;
```

Avant

```
NSNumber *red = [NSNumber
    numberWithIntUnsignedChar:Red];
```

Après

```
NSNumber *red =
    @ (Red);
```

modern Objective-C

tableaux

Avant

```
NSArray *array = [NSArray  
arrayWithObjects:  
    [NSNumber numberWithBool:YES],  
    [NSNumber numberWithBool:NO],  
    [NSNumber numberWithFloat:5.f],  
    [NSNumber numberWithInt:49],  
    nil  
];
```

Après

```
NSArray *array = @ [  
    @YES,  
    @NO,  
    @5F,  
    @49  
];
```

modern Objective-C

dictionnaires

Avant

```
NSDictionary *dictionary = [NSDictionary  
dictionaryWithObjectsAndKeys:  
    @"key1",  
    [NSNumber numberWithBool:YES],  
    @"key2",  
    [NSNumber numberWithFloat:5.f],  
    nil  
];
```

Après

```
NSDictionary  
*dictionary = @{  
    @"key1": @YES,  
    @"key2": @5F,  
};
```

sommaire

partie 1 introduction

partie 2 les bases de la syntaxe

partie 3 gestion de la mémoire

partie 4 ARC

partie 5 création d'une classe

partie 6 étendre une classe

partie 7 délégation, dataSource et notifications

partie 8 les blocks

partie 9 concurrence

partie 10 modern Objective-C

partie 11 **KVC et KVO**

KVC et KVO

Key Value Coding

- mécanisme pour accéder ou modifier les propriétés d'un objet en utilisant des chaînes de caractères
- la classe `NSObject` est conforme au protocole `NSKeyValueCoding`
- 3 types de propriétés accessibles :
 - attribut : valeur simple (scalaire, chaîne, booléen)
 - relation unaire : un objet
 - relation n-aire : lien vers une collection d'objets, souvent représenté par un `NSArray` ou un `NSSet` ou un `NSDictionary`

KVC et KVO

Key Value Coding - keys et key paths

- une **key** est une chaîne qui identifie une propriété **d'un objet** :
 - elle commence par une minuscule
 - elle ne comporte pas d'espace
 - elle ne contient que des caractères ASCII
 - par convention on préfixe avec `is` pour une propriété booléenne
- un **key path** est une séquence qui permet de traverser **des propriétés d'objet**
 - c'est une chaîne contenant des keys séparées par des points
 - chaque key est interprétée sur l'objet traversé

exemple : pour un objet qui possède une propriété de type adresse qui possède une propriété de type ville, on peut accéder directement à la ville en utilisant le key path `address.city`

KVC et KVO

Key Value Coding - attribut et relation unaire

- accès par les méthodes `valueForKey:` ou `valueForKeyPath:`
- modification par les méthodes `setValue:forKey:` ou `setValue:forKeyPath:`
- la chaîne passée en paramètre correspond au nom de la propriété ou des propriétés traversées

Exemple

```
// accès à une propriété nommée myKey
[anObject valueForKey:@"myKey"];
// modification à une propriété nommée myKey
[anObject setValue:@"My value" forKey:@"myKey"];
// accès à la ville du client
[anOrder valueForKeyPath:@"customer.address.city"];
// modification de la ville du client
[anOrder setValue:@"Paris" forKeyPath:@"customer.address.city"];
```

KVC et KVO

Key Value Coding - attribut et relation unaire - compatibilité

- pour être compatible KVC, une classe doit avoir :
 - une méthode d'accès au nom de la propriété (ou préfixé par `is`)
 - une méthode de modification, si la propriété est modifiable, de la forme `set<Key>`
 - la méthode de modification ne doit pas effectuer de validation
 - une méthode de validation (optionnelle) de la forme `validate<Key>:error:`

KVC et KVO

Key Value Coding - relation n-aire

- utilisation possible des méthodes `valueForKey:`,
`valueForKeyPath:`, `setValue:forKey:` ou
`setValue:forKeyPath:`
- une relation n-aire peut être ordonnée ou non
- une relation n-aire peut être modifiable ou non

KVC et KVO

Key Value Coding - relation n-aire ordonnée - compatibilité

- pour être compatible KVC, une classe doit avoir :
 - une méthode d'accès au nom de la propriété qui retourne un tableau ou une méthode `countOf<Key>` et au moins une des deux méthodes `objectIn<Key>AtIndex:` et `<Key>AtIndexes:`
 - une méthode optionnelle `get<Key>:range:`
- pour une relation modifiable, il faut :
 - la méthode `insertObject:in<Key>AtIndex:` ou `insert<Key>:atIndexes:`
 - la méthode `removeObjectFrom<Key>AtIndex:` ou `remove<Key>AtIndexes:`
 - les méthodes optionnelles `replaceObjectIn<Key>AtIndex:withObject:` et `replace<Key>AtIndexes:with<Key>:`

KVC et KVO

Key Value Coding - relation n-aire non ordonnée - compatibilité

- pour être compatible KVC, une classe doit avoir :
 - une méthode d'accès au nom de la propriété qui retourne un set ou les méthodes `countOf<Key>`, `enumeratorOf<Key>` et `memberOf<Key>`
- pour une relation modifiable, il faut :
 - la méthode `add<Key>Object`: ou `add<Key>`:
 - la méthode `remove<Key>Object`: ou `remove<Key>`:
 - la méthode optionnelle `intersect<Key>`:

KVC et KVO

Key Value Coding - relation n-aire - opérateurs

- permettent d'effectuer des opérations sur la collection et d'en récupérer le résultat
- préfixés par le caractère @
- exemple avec un compte bancaire nommé dépenses

titre	montant	date
restaurant	47,35	12/12/2013
courses	98,21	10/09/2013
location skis	127,99	10/02/2014
forfaits skis	250,65	19/01/2014
location appartement	684,91	09/01/2014

KVC et KVO

Key Value Coding - relation n-aire - opérateurs - @avg, @count et @sum

- @avg permet d'obtenir une moyenne sur une propriété de tous les objets d'un tableau
- @count permet d'obtenir le nombre d'objets dans un tableau
- @sum permet d'obtenir la somme des valeurs d'une propriété sur tous les objets d'un tableau

Exemple

```
NSNumber *amountAvg = [depenses valueForKeyPath:@"@avg.montant"];  
NSNumber *count = [depenses valueForKeyPath:@"@count"];  
NSNumber *sum = [depenses valueForKeyPath:@"@sum.montant"];
```

KVC et KVO

Key Value Coding - relation n-aire - opérateurs - @max et @min

- @min permet d'obtenir la valeur minimale d'une propriété parmi tous les objets d'un tableau
- @max permet d'obtenir la valeur maximale d'une propriété parmi tous les objets d'un tableau

Exemple

```
NSNumber *amountMax = [depenses valueForKeyPath:@"@max.montant"];  
NSNumber *amountMin = [depenses valueForKeyPath:@"@min.montant"];  
NSDate *dateMax = [depenses valueForKeyPath:@"@max.date"];  
NSDate *dateMin = [depenses valueForKeyPath:@"@min.date"];
```

KVC et KVO

Key Value Coding - relation n-aire - opérateurs - autres

- il existe également des opérateurs pour extraire des tableaux à partir d'un tableau :
 - @distinctUnionOfObjects : un tableau ne contenant que des valeurs distinctes d'une propriété
 - @unionOfObjects : un tableau contenant toutes les valeurs d'une propriété
 - @distinctUnionOfArrays : fonctionne comme @distinctUnionOfObjects mais pour un tableau de tableaux d'objets
 - @unionOfArrays: fonctionne comme @unionOfObjects mais pour un tableau de tableaux d'objets
 - @distinctUnionOfArrays : fonctionne comme @distinctUnionOfArrays mais pour un set de sets d'objets

KVC et KVO

Key Value Observing

- mécanisme qui permet d'être informé des modifications sur une propriété d'un objet
- possibilité d'observer :
 - des attributs
 - des relations unaires
 - des relations n-aires
- l'observateur est informé du type de changement et des ancienne et nouvelle valeur de la propriété, ou des objets ajoutés ou supprimés dans le cas d'une relation n-aire

KVC et KVO

Key Value Observing - fonctionnement

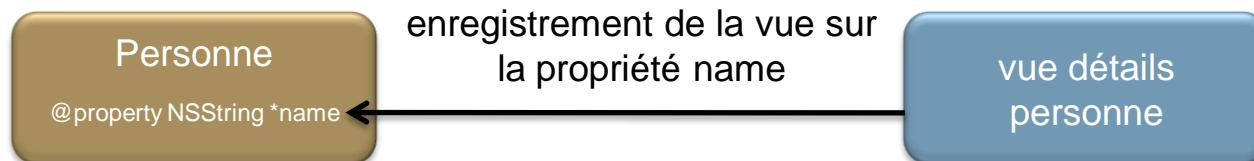
Personne

@property NSString *name

vue détails
personne

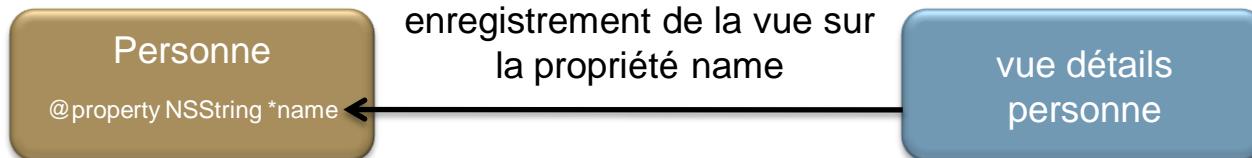
KVC et KVO

Key Value Observing - fonctionnement



KVC et KVO

Key Value Observing - fonctionnement



```
[personneObject addObserver:self  
    forKeyPath:@"name"  
    options:NSMutableArrayObservingOptionNew  
    context:NULL];
```

KVC et KVO

Key Value Observing - fonctionnement



KVC et KVO

Key Value Observing - fonctionnement



KVC et KVO

Key Value Observing - fonctionnement



appel automatique de la méthode suivante dans la vue :

```
- (void)observeValueForKeyPath: (NSString*)keyPath  
    obObject: (id) object  
    change: (NSDictionary*) change  
    context: (void*) context  
{  
}
```

KVC et KVO

Key Value Observing - fonctionnement



appel automatique de la méthode suivante dans la vue :

```
- (void)observeValueForKeyPath: (NSString*)keyPath
                      obObject: (id) object
                      change: (NSDictionary*)change
                     context: (void*)context
{
}
```

KVC et KVO

Key Value Observing - fonctionnement



KVC et KVO

Key Value Observing - fonctionnement



KVC et KVO

Key Value Observing - fonctionnement

la vue affiche une autre personne ou va être désallouée

Personne

@property NSString *name

vue détails
personne

suppression du lien

KVC et KVO

Key Value Observing - fonctionnement

la vue affiche une autre personne ou va être désallouée

Personne

@property NSString *name

vue détails
personne

suppression du lien

```
[personneObject removeObserver:self  
forKeyPath:@"name"];
```

KVC et KVO

Key Value Observing - compatibilité

- la classe possédant la propriété à observer doit être compatible KVC
- géré automatiquement lorsqu'on utilise le mot-clé `@property`
- il est possible d'empêcher l'envoi de notifications en surchargeant la méthode `automaticallyNotifiesObserversForKey`:

Exemple

```
- (BOOL) automaticallyNotifiesObserversForKey: (NSString*) key {
    if ([key isEqualToString:@"name"]) {
        return YES; // code d'exécution de l'opération
    }
    return [super automaticallyNotifiesObserversForKey: key];
}
```

KVC et KVO

Key Value Observing - notification manuelle

- la classe possédant la propriété doit informer les observateurs :
 - lors de l'accès
 - lors de la modification
 - lors de l'ajout ou la suppression d'un objet (dans le cas d'une relation n-aire)

KVC et KVO

Key Value Observing - notification manuelle - accès et modification

Exemple

```
- (NSString*) name {
    [self willAccessValueForKey:@"name"];
    [self didAccessValueForKey:@"name"];
    return _name;
}

- (void) setName: (NSString*) name {
    if (_name != name) {
        [self willChangeValueForKey:@"name"];
        _name = name;
        [self didChangeValueForKey:@"name"];
    }
}
```

KVC et KVO

Key Value Observing - propriétés dépendantes

- certaines propriétés peuvent dépendre de la valeur d'autres
- par exemple, une adresse peut dépendre :
 - d'une rue
 - d'un code postal
 - d'une ville
- si la valeur d'une de ces propriétés change, une notification peut être envoyée
- surcharge de la méthode `keyPathsForValuesAffectingValueForKey:` ou d'une méthode `keyPathsForValuesAffecting<Key>`



uniquement pour attribut ou relation unaire

KVC et KVO

Key Value Observing - propriétés dépendantes

Exemple

```
- (NSString*) adresse {
    return [@[_rue, _cp, _ville] componentsJoinedByString:@" "];
}

+(NSSet*)keyPathsForValuesAffectingValueForKey:(NSString *)key {
    NSSet *keyPaths = [super
        keyPathsForValuesAffectingValueForKey:key];
    if ([key isEqualToString:@"adresse"]){
        NSArray *affectingKeys = @[@"rue", @"cp", @"ville"];
        keyPaths = [keyPaths
setByAddingObjectsFromArray:affectingKeys];
    }
    return keyPaths;
}

+(NSSet *)keyPathsForValuesAffectingAdresse {
    return [NSSet setWithObjects:@"rue", @"cp", @"ville", nil];
}
```

références

- <http://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad>
- <http://www.siteduzero.com/tutoriel-3-200557-programmez-en-objective-c.html>
- <https://developer.apple.com/library/ios/navigation/index.html>

questions ?

Business
Services

