

# Access : Modules de classes

par [Michel Blavin](#)

Date de publication : Work in Progress...

Dernière mise à jour : 14 février 2006

Ce cours se présente comme un tutoriel sur les modules de classes et la Programmation Orientée Objets (POO) avec Visual Basic pour Application (VBA). Après un bref rappel sur les objets, nous mettrons en application les principes et fonctionnalités des Modules de classes.

Programmation modulaire

VBA, un Langage orienté objet

Pourquoi utiliser les modules de classes ?

A qui s'adresse ce document ?

Où trouver la dernière version de ce cours ?

Conventions

Remerciements

1 - Définitions et rappels.

1.1 - Les bienfaits l'abstraction.

1.2 - Les objets, les classes et leurs différences

1.3 - Qu'esce qu'une interface ?

1.4 - Rappels sur les objets, les pointeurs et leurs durée de vie.

2 - Première classe

2.1 - Interface

2.2 - Propriétés

2.2.1 - Implémentation des propriétés utilisant des variables publiques

2.2.2 - Implémentation des propriétés utilisant les Procédures Property

2.3 - Méthodes

2.4 - Les événements

2.5 - La propriété Parent

3 - Collections

3.1 - L'objet Collection de VBA

3.2 - Collections personnalisées

3.2.1 - Généralités

3.2.2 - Interface

3.2.3 - Implémentation

3.2.3 - Que manque-t-il aux collections personnalisées ?

4 - Les modules des formulaires et des Etats

Les liens qui ont la classe

## Programmation modulaire

De nos jours, l'un des plus grands défis pour un développeur est de ne pas devoir réinventer la roue à chaque nouvelle application. C'est pourquoi le développeur moderne est un développeur de modules. On développe des modules autonomes et plus ou moins génériques que l'on assemble pour former une application. La modularité et la réutilisation du code est un des principes de bases de la POO. Il est plus facile de débogger des modules de petite tailles qu'une grosse application.

En dehors de la programmation objet il est aussi très pratique d'utiliser des bibliothèques de code, qui, [une fois que l'on a résolu le problème des liens cassés](#), devient indispensable. Cela permet d'obtenir du code de meilleurs qualité, plus facile à maintenir, plus robuste.

## VBA, un Langage orienté objet

Les développeurs de langage orienté objet, égarés sur cette page, viennent de s'étrangler en lisant le titre de ce paragraphe. Faut bien s'amuser ;o).

Disons plutôt, par égard pour les puristes, que VBA n'est pas un langage orienté objet mais qu'il permet de faire de la Programmation Orientée Objet (POO) en VBA, malgré de nombreuses limitations (souvent frustrantes) :

Certaines des caractéristiques de la POO, comme **l'héritage** ou le **polymorphisme** , ne sont pas présents. Il est impossible de créer des classes dérivées.

La surcharge de méthode est impossible, puisqu'il est impossible en VB d'utiliser le même nom pour plusieurs méthodes dans une même classe. Nous verrons le cas particulier des procédures *Property*

D'autres part, les classes ne peuvent pas être directement instanciées si elle se trouve dans un complément (\*.mda). C'est à mon avis la plus grosse lacune et surtout la plus absurde, étant donné que la réutilisation du code est un des chevaux de bataille de la POO. Il est cependant possible d'utiliser cette classe si le code qui instancie l'objet se trouve dans un module de code simple dans le même mda que la classe.

## Pourquoi utiliser les modules de classes ?

La programmation procédurale a fait ses preuves et vous vous demandez peut-être pourquoi changer ? Les raisons sont multiples :

La création d'objets personnalisés permet de mieux organiser les développements en regroupant dans un même objet les données ainsi que les méthodes pour manipuler ses données.

Les classes sont des sous ensembles cohérents de vos applications. On commence par développer les classes dont on aura besoin (et qui pour certaines seront réutilisables ultérieurement) et ensuite on implémente la logique liée à l'application. Cela vous oblige à réfléchir à l'application que vous allez développer avant de commencer à coder, ce qui est rarement une mauvaise idée. ;)

Le découpage du code en classes permet d'accélérer le développement d'application complexe et facilite la maintenance, qui représente 80% de la vie d'un logiciel, en est grandement facilitée. Le temps que vous passez au départ à réfléchir c'est du temps gagné pour plus tard. Si vous avez eu à vous replonger dans un code écrit l'année dernière et que vous aviez du mal à comprendre, vous savez de quoi je parle.

Et malgré tout cela, il est difficile de prendre conscience du gain réel l'utilisation des modules de classes (à part pour frimer à côté de la machine à café), mais je peux vous dire que maintenant que j'y ai goûté, je ne développerai plus aucune application sans les utiliser.

## A qui s'adresse ce document ?

Cet article s'adresse à tous les développeurs Visual Basic pour Application, en particulier le développeurs Access. Cependant ce cours ne s'adresse pas à des débutants : vous devez connaître les bases de la programmation, des connaissances POO ne sont pas nécessaire mais seront un plus.

## Où trouver la dernière version de ce cours ?

Ce document a été écrit pour être publié sur [developpez.com](http://developpez.com) la dernière version est disponible à l'adresse suivante : [sinarf.developpez.com/access/vbaclass](http://sinarf.developpez.com/access/vbaclass). Une version pdf de l' [introduction de aux modules de classes](#) est disponible.

## Conventions

Je suis un adepte de la normalisation des noms, je trouve que cela simplifie grandement la programmation surtout quand un projet commence à grossir. Le nom des classes est un peu une exception : **aucun préfixe**. Les noms des objets de leurs propriétés et de leur méthodes utilisent des noms explicites. Par soucis de lisibilité, j'utilise une majuscule pour séparer les différents mots par exemple une propriété contenant le nom d'un fichier pourra être appeler *NomFichier*.

Remarque : Pour ne pas allourdir les blocs de code présent dans cette article nous n'avons pas implémenté de gestion d'erreur, mais gardez à l'esprit que les développeurs sérieux implémentent **systematiquement** une gestion d'erreur. Alors tordons le coup aux rumeurs, comme quoi les développeurs VB, qui plus est VBA, ne sont pas des développeurs sérieux. ;)

## Remerciements

Merci à toute l'équipe de [developpez.com](#) qui m'a tant apporté. Je tiens à remercier [Maxence Hubiche](#), pour m'avoir tenu la main, encouragé, harcellement, flatté, flagorné, suggéré et pour son infinie patience, car je dois avouer que cet article est en gestation depuis des temps immémoriaux. Je remercie tous les correcteurs : [Christophe WARIN](#), Frank, [Etienne Bar](#), [Jean-Marc RABILLOUD](#), et sûrement plein d'autre que j'ai oublié, ingrat que je suis.

## 1 - Définitions et rappels.

Je donne ici quelques rappels sur les notions de base de la Programmation Orienté Objet, il ne s'agit pas de faire un cours complet sur le sujet, si vous cherchez des précisions à ce sujet je vous donne quelques liens en fins d'article mais je vous conseil en particulier [Pensez en java](#) de Bruce Eckel.

### 1.1 - Les bienfaits l'abstraction.

En encapsulant votre code dans des objets vous masquez l'implémentation. C'est à dire que les utilisateurs de vos classes n'ont pas besoin d'en connaître l'implémentation. Ces objets sont des boites noires, une fois développés, il nous suffit de savoir ce qui entre ou sort de la boite. Par exemple, vous utilisez souvent des objets Recordset, peut-être que par curiosité vous aimeriez connaître l'implémentation utilisée, mais reconnaissez que cela n'a aucune utilité pour vos développements.

#### **Tout ce dont vous avez besoin c'est de savoir comment utiliser cet objet**

Les classes peuvent aussi permettre de masquer des processus complexes derrière une interface simplifié. Une des utilisation peut être d'encapsulé les API pour les rendre plus convivial.

### 1.2 - Les objets, les classes et leurs différences

Les classes sont des **types évolués**. Contrairement aux types simples (entiers, flottant, chaîne de caractères ...) qui ne contiennent que des données, les objets contiennent à la fois les données et les méthodes qui permettent de manipuler ces données.

Il est facile de faire une confusion entre **objets** et **classes**, pour faire une analogie culinaire, nous dirons que si les objets sont des gâteaux, les classes sont les recettes qui permettent de faire ces gâteaux. La recette n'est pas un gâteau mais elle permet de faire autant de gâteaux que l'on désire. Dans cette image les ingrédients seraient les propriétés et les manipulations (mélanger, cuire) les méthodes.

Quand on crée un objet à partir d'une classe, on dit que l'on instancie un objet. Les différents objets instanciés à partir d'une classe, sont appelés instances.

Les objets ont des propriétés, qui contiennent les données, et des méthodes qui permettent de les manipuler. Dans certains langages, les propriétés sont appelées champs. Quant aux méthodes ce ne sont que des procédures ou des fonctions qui peuvent retourner une valeur et recevoir des paramètres.

### 1.3 - Qu'esce qu'une interface ?

Nous ne parlons pas ici d'interface graphique. L'interface est la partie visible d'une classe. Le créateur de la classe n'est pas forcément l'utilisateur de celle-ci, il est donc indispensable de définir une interface et de s'y tenir. Le créateur de la classe définit l'interface et met en place l'implémentation. Peu importe que l'implémentation change du moment que l'interface ne change pas. En gros, une mise à jour de la classe doit rester compatible avec le code existant des utilisateurs de cette classe. Il est donc très simple d'ajouter des propriétés et des méthodes mais pratiquement impossible d'en supprimer. Dans la réalité, cela se produit, dans ce cas on ne supprime pas tout de suite la méthode, propriété ou même la classe mais on dit que cet élément est "deprecated" pour indiquer qu'il ne faut plus l'utiliser qu'il sera supprimés dans les versions futures. En java, le compilateur émet un avertissement indiquant qu'il utilise une méthode deprecated, il n'y a pas, à ma connaissance, de processus similaire en VB.

Dans certains des langages orientés objet, comme Java, il existe un mot clé **interface**, il n'existe pas en VBA. Par

contre, il est possible créer une interface en créant une classe ne contenant que les prototypes des méthodes et propriétés.

## 1.4 - Rappels sur les objets, les pointeurs et leurs durée de vie.

Nous allons faire ici un petit rappel sur les objets. Lorsque vous déclarez un objet vous créez un pointeur vers cet objet. Plusieurs pointeurs peuvent pointer sur le même objet. Lorsque vous utilisez le code suivant :

```
set Objet1 = Objet2
```

Vous ne faites pas une copie de l'objet comme c'est le cas avec les types simples mais vous créez un deuxième pointeur vers votre objet. Lorsque vous passez un objet en paramètre d'une fonction ou procédure vous ne passez en fait qu'un pointeur vers l'objet. On dit que les objets sont passés par référence (c'est aussi le cas des types simple dans Visual Basic, bien que ces derniers puissent être passé [en tant que valeur](#)).

### La création des objets :

```
Dim obj1 as new Object           'création du pointeur vers l'objet1
Dim obj2 as object              'création du pointeur vers l'objet2

'Dans cette partie du code obj2 est réellement instancié
'mais obj1 ne l'est pas

' Maintenant si on utilise une propriété de obj1 cela provoque la création de l'objet
obj1.SomePrp = "What Ever" 'création de l'objet1

' On libère les pointeurs
set obj1 = Nothing
set obj2 = Nothing
'On ne peut plus accéder aux objets grâce aux pointeurs obj1 et obj2
```

Maintenant que nous savons quand nos objets sont effectivement créés, voyons comment se passe **la destruction des objets** :

**Un objet est détruit quand il n'existe plus de pointeur lui faisant référence.** Cette définition est simple mais l'on a souvent l'impression que lorsque le code suivant est exécuté :

#### Libération pointeur

```
Set objA = Nothing
```

l'objet est détruit, alors que ce n'est vrai que si objA est le dernier pointeur vers l'objet. Il est très important de prendre la précautions de bien "nettoyer" derrière soit et de ne pas laisser traîner de pointeurs.

Comme nous le verrons plus loin nous pouvons exécuter du code lors de la destruction d'un objet encore faut-il être certains que ce code s'exécutera.

## 2 - Première classe

Après ces brefs rappels théoriques, ce n'est pas sans une certaine émotion que nous allons voir en pratique comment créer notre première classe.

### 2.1 - Interface

Commençons par définir l'interface de la classe que nous allons créer. Celle-ci sera simple afin de mettre en relief le fonctionnement des modules de classes sans risquer de s'égarer avec une implémentation obscure.

Notre classe aura 4 propriétés :

- Nom : Une chaîne de caractères contenant le nom de la personne
- Prenom : Une chaîne de caractères contenant le prénom de la personne
- DateNaissance
- NomComplet : Une chaîne de caractère contenant le prénom suivi du nom séparés par un espace. Cette propriété prend les valeurs des propriétés *Nom* et *Prenom* sépar elle est en **lecture seule**.

et une méthode :

- age : renvoie l'âge de la personne.

### 2.2 - Propriétés

Puisque notre classe n'existe pas encore, nous devons commencer par créer un module de classe. Pour cela, dans l'éditeur VB, cliquer sur Insertion/Module de classe. Enregistrons le module. Ce nom ne doit pas être choisi au hasard car il s'agit du nom du type que vous allez créer. Notre classe s'appelle **Person**.

#### 2.2.1 - Implémentation des propriétés utilisant des variables publiques

La méthode la plus simple pour implémenter des propriétés est d'utiliser des variables publiques. Pour implémenter une propriété il nous suffit de déclarer des variables publiques dans notre module de classe.

Comme ceci :

##### Propriétés déclarées en tant que variables publiques

```
Public Nom As String           'Nom de la personne
Public Prenom As String       'Prénom de la personne
Public DateNaissance As Date  'Date de naissance de la personne
```

**Puisque l'on ne peut pas implémenter une propriété en lecture seule avec cette méthode nous laisserons la propriété *NomComplet* pour le moment**

Nous pouvons tester grâce au code suivant, dans un module de code classique.

##### Test de la classe Person

```
Sub testPerson()
Dim objperson As New Person           'création du pointeur vers l'objet
Dim strNom As String
Dim strPrenom As String
Dim dtmNaissance As String
```

## Test de la classe Person

```

strNom = "Blavin"
strPrenom = "Michel"
dtmNaissance = #9/14/1969#
With objperson
    .Nom = strNom                'provoque la création de l'objet
    .Prenom = strPrenom
    .DateNaissance = dtmNaissance
    Debug.Print "La méthode age renvoie : "; .Age; _
                " ans, pour "; .NomCompleet; " né le : "; _
                Format(.DateNaissance, "dddd d mmmm yyyy")
End With
Set objperson = Nothing
End Sub

```

Vous devriez obtenir le texte suivant sur la sortie de debug.

## Sortie de Debug

```

Michel Blavin est né le dimanche 14 septembre 1969

```

Cette façon de faire est vraiment très simple mais elle a plusieurs défauts :

- Impossible de créer une propriété en lecture (ou écriture) seule.
- Impossible de savoir quand une propriété est modifiée
- Impossible de vérifier la validité des valeurs (par exemple une date de naissance devrait toujours être une date révolue)

En fait cette méthode viole ce que l'on appelle l'encapsulation des données qui veut que les données ne devrait être accessible qu'au travers de méthodes. C'est une mauvaise implémentation, nous la réserverons à la phase de prototypage.

## 2.2.2 - Implémentation des propriétés utilisant les Procédures Property

Les procédures *Property* permettent de donner un accès complet ou limité aux propriétés d'un objet. Nous accèderons à ces données à travers des méthodes. Il existe 3 procédures Property :

- Property get : lecture de la propriété quel que soit son type.
- Property Let : écriture des propriétés de type simple.
- Property Set : écriture des propriétés de type Objet.

En séparant l'écriture de la lecture, il devient très simple d'implémenter une propriété en lecture seule. Pour cela, il suffira de créer une procédure *Get* mais pas la procédure *Set* ou *Let* correspondante. Et inversement, pour une propriété en écriture seule (cas plus rare mais utile, par exemple le cas de la propriété UID du type User du modèle de sécurité d'Access).

Ce que nous allons faire maintenant est très important car nous allons modifier l'implémentation de notre classe et ceci sans modifier son interface.

Nous allons toujours utiliser des variables pour stocker les valeurs de nos propriétés mais nous allons utiliser des **variables privées**. Pour cela, nous allons transformer les déclarations des variables publiques en variables privées, ainsi seules les méthodes de la classe y auront accès.

## Propriétés en tant que variables privées

```

Private mstrNom As String        'Nom de la personne
Private mstrPrenom As String    'Prénom de la personne

```

## Propriétés en tant que variables privées

```
Private mdtmDateNaissance As Date 'Date de naissance de la personne
```

Maintenant nous devons mettre en place les procédures *Property*. Pour commencer nous n'allons mettre aucun contrôle, aucune validation, nous allons juste transformer les variables publiques en propriétés. Vous remarquerez que le type de retour de la procédure *Get* est le type du paramètre de la procédure *Let* (ou *Set*) correspondante, c'est une obligation. Vous remarquerez aussi que ces procédures portent exactement le même nom comme nous l'avons vu plus haut, c'est un cas unique en VB.

## Utilisation des procédures Property

```
' Propriété Nom
Property Let Nom(strNom As String)
    mstrNom = strNom
End Property
Property Get Nom() As String
    Nom = mstrNom
End Property

' Propriété Prenom
Property Let Prenom(strPrenom As String)
    mstrPrenom = strPrenom
End Property
Property Get Prenom() As String
    Prenom = mstrPrenom
End Property

' Propriété DateNaissance
Property Let DateNaissance(dtmDateNaissance As Date)
    mdtmDateNaissance = dtmDateNaissance
End Property
Property Get DateNaissance() As Date
    DateNaissance = mdtmDateNaissance
End Property
```

Si vous utilisez la procédure de test précédente vous constaterez que le fonctionnement des objets n'a pas changé alors que l'implémentation a évolué. On voit ici comment on peut faire évoluer une partie de notre code sans compromettre le fonctionnement de l'application.

Nous allons, maintenant, ajouter les contrôles suivants :

- Vérifier que les valeurs fournies ne sont pas nulles.
- Vérification que la date de naissance est bien révolue.

Ce qui nous donnera le code suivant :

## Procédures Property avec contrôles basiques

```
' Propriété Nom
Property Let Nom(strNom As String)
    If Len(strNom) = 0 Then
        MsgBox "Le nom ne peut pas être vide."
    Else
        mstrNom = strNom
    End If
End Property

Property Get Nom() As String
    Nom = mstrNom
End Property

' Propriété Prenom
Property Let Prenom(strPrenom As String)
    If Len(strPrenom) = 0 Then
        MsgBox "Le prénom ne peut pas être vide."
    Else
        mstrPrenom = strPrenom
    End If
```

## Procédures Property avec contrôles basiques

```

End Property

Property Get Prenom() As String
    Prenom = mstrPrenom
End Property

' Propriété DateNaissance
Property Let DateNaissance(dtmDateNaissance As Date)
    If dtmDateNaissance > Now Then
        MsgBox "Une date de naissance ne peut pas être dans le futur."
    Else
        mdtmDateNaissance = dtmDateNaissance
    End If
End Property

Property Get DateNaissance() As Date
    DateNaissance = mdtmDateNaissance
End Property

```

Il ne nous reste plus qu'à implémenter le propriété NomComplet en lecture seule.

## Propriété NomComplet (Lecture seule)

```

' Propriété NomComplet en lecture seule
Property Get NomComplet() As String
    NomComplet = Me.Prenom & " " & Me.Nom
End Property

```

Lorsque l'on fait référence à une propriété (ou une méthode) publique de la classe, nous utilisons le mot clé *Me*. Ce mot clé fait référence à l'instance en cours, il ne peut donc être utilisé que dans des modules de classe. En pratique, lors de l'instanciation d'un objet la propriété privée *Me* est automatiquement créée, il s'agit d'un pointeur vers l'objet courant.

S'il est utilisable dans les modules de formulaires et de rapports, c'est que ceux-ci, comme nous le verrons plus loin, sont des modules de classes.

## 2.3 - Méthodes

Comme dans un module simple les méthodes peuvent, soit ne pas renvoyer de valeur (procédure *Sub*), soit comme dans le cas de la méthode *Age* que nous allons implémenter, renvoyer une valeur (une fonction).

## Méthode Age

```

Public Function Age() As Integer
    'on calcule le nombre d'années entre l'année de naissance et aujourd'hui
    Age = DateDiff("yyyy", DateNaissance, Now, vbMonday, vbFirstFourDays)
    'si la jour de l'anniversaire n'est pas passé on retire 1 an
    If DateAdd("yyyy", Age, Me.DateNaissance) > Now Then
        Age = Age - 1
    End If
End Function

```

Nous avons maintenant une classe totalement fonctionnelle.

## Procédure de test de la classe Person complète

```

Sub testPerson()
    Dim objPerson As New Person          'création du pointeur vers l'objet
    Dim strNom As String
    Dim strPrenom As String
    Dim dtmNaissance As String

    strNom = "Blavin"
    strPrenom = "Michel"
    dtmNaissance = #9/14/1969#

```

## Procédure de test de la classe Person complète

```

objPerson.Nom = strNom           'provoque la création de l'objet
objPerson.Prenom = strPrenom
objPerson.DateNaissance = dtmNaissance

Debug.Print "La méthode age renvoie : "; objPerson.age; _
" ans, pour "; objPerson.NomComplet; " née le : "; _
Format(objPerson.DateNaissance, "dddd d mmmm yyyy")

Set objPerson = Nothing
End Sub

```

## Fenêtre de debug

La méthode age renvoie : 35 ans, pour Michel Blavin né le : dimanche 14 septembre 1969

## 2.4 - Les événements

En plus des propriétés et des méthodes, les modules de classes savent gérer des événements. Les modules de classes simples en implémentent 2, l'événement *Initialize* et l'événement *Terminate*.

*Il n'est pas ici question de constructeur ou de destructeur comme ils existent dans les langages comme java ou C++.*

L'événement *Initialize* se produit lors de la création de l'objet et l'événement *Terminate* se produit lors de sa destruction. Il est indispensable de savoir quand un objet est créé et quand il est détruit car si, à première vue, cela à l'air évident mais si les pointeurs ne sont pas bien libérés il se peut qu'un objet ne soit pas détruit et donc que le code de l'événement *Terminate* ne soit jamais exécuté.

Pour prendre conscience du moment exact où ces événements se produisent, nous allons ajouter le code suivant à notre classe :

## Événements Initialize et Terminate de la classe Person

```

Private Sub class_Initialize()
    Debug.Print "Événement Initialize - Création d'un objet Person"
End Sub

Private Sub class_Terminate()
    Debug.Print "Événement Terminate - destruction d'un objet Person : " & Me.NomComplet
End Sub

```

Ainsi, la fenêtre de debug nous informera à chaque fois que l'un de ces événements se produira. Pendant la phase d'apprentissage, je vous conseille fortement d'implémenter systématiquement ce genre de code, cela vous évitera probablement quelques heures de debug. Maintenant notre procédure de test donne :

## Fenêtre de Debug

Événement Initialize - Création d'un objet Person  
 La méthode age renvoie : 35 ans, pour Michel Blavin né le : dimanche 14 septembre 1969  
 Événement Terminate - destruction d'un objet Person : Michel Blavin

Vous pouvez aussi ajouter à vos classes des événements personnalisés (à partir de Office 2000 seulement). Nous allons ajouter un événement que l'on appellera *Completed* qui se produira lorsque les propriétés *Nom*, *Prenom* et *DateNaissance* auront été renseignés. Commençons par ajouter la déclaration de l'événement :

## Déclaration de l'événement Completed

```

Public Event Completed()
' Événement qui se produit lorsque les propriétés
' Nom, Prenom et DateNaissance auront été renseignés
' Cette déclaration provoque une erreur à la compilation
' sous Access 97.

```

Pour qu'un événement se produise, il faut utiliser la méthode *RaiseEvent*.

Dans notre cas, nous allons créer une méthode privée *IsCompleted* dans notre module de classe, qui va vérifier si toutes les propriétés concernées sont renseignées.

#### Ajout de 3 nouvelles propriétés privées

```
Private mblnNom As Boolean           'est à vrai lorsque le nom a été renseigné
Private mblnPrenom As Boolean       'est à vrai lorsque le prénom a été renseigné
Private mblnDateNaissance As Boolean 'est à vrai lorsque la date de naissance a été
renseignée
```

C'est 3 variables seront initialisées à *False* dans l'événement *Initialize* et nous allons transformer les procédures *Let* pour qu'elles appellent la procédure *IsCompleted* qui provoquera l'événement *Completed* si les trois propriétés ont été renseignées.

#### Procédure IsCompleted

```
Private Sub IsCompleted()
    If mblnNom And mblnPrenom And mblnDateNaissance Then
        'Si les 3 propriétés privées sont vrai alors l'événement se produit
        RaiseEvent Completed
    End If
End Sub
```

Comme me l'a fait remarquer un de mes relecteurs cette procédure aurait pu être publique ce qui aurait permis aux utilisateurs de la classe d'interroger cette méthode plutôt que de lever un événement, mais cela nous permet de implémenter une méthode privée.

#### Voici les procédures Let modifiées.

```
Property Let Nom(strNom As String)
    If Len(strNom) = 0 Then
        MsgBox "Le nom ne peut pas être vide."
        Exit Property
    Else
        mstrNom = strNom
        mblnNom = True
    End If
    IsCompleted
End Property

Property Let Prenom(ByVal strPrenom As String)
    If Len(strPrenom) = 0 Then
        MsgBox "Le prénom ne peut pas être vide."
        Exit Property
    Else
        mstrPrenom = strPrenom
        mblnPrenom = True
    End If
    IsCompleted
End Property

Property Let DateNaissance(ByVal dtmDateNaissance As Date)
    If dtmDateNaissance > Now Then
        MsgBox "Une date de naissance ne peut être dans le futur."
        Exit Property
    Else
        mdtmDateNaissance = dtmDateNaissance
        mblnDateNaissance = True
    End If
    IsCompleted
End Property
```

Maintenant il ne nous reste plus qu'à tester. Créons un nouveau formulaire et ajoutons les blocs de code suivant, pour la gestion de l'événement. Remarquez l'usage d'un nouveau mot clé *WithEvents*, qui, comme son nom l'indique signale lors de la déclaration de l'objet que nous allons utiliser tout ou partie des événements de l'objet instancié. La méthode exécutée lorsque l'événement se produit a un nom particulier composé du nom de l'objet

concerné et du nom de l'événement, séparés par un underscore ("\_").

*Le mot clé WithEvents ne peut pas être utilisé en même temps que le mot clé new.*

#### Gestion de l'événement

```
Private WithEvents mobjPerson As Person 'Provoque une erreur si placé ailleurs que
                                     'dans un module de
                                     classe

Private Sub mobjPerson_Completed()
    Debug.Print "l'Evénement Completed vient de se produire !"
End Sub
```

#### Sur Chargement (Load)

```
Private Sub Form_Load()
    Debug.Print "chargement du formulaire frmPerson"
    Set mobjPerson = New Person
    With mobjPerson
        .Nom = "Blavin"
        .Prenom = "Michel"
        .DateNaissance = #9/14/1969#
        'ici se produit l'événement completed
        'Et ensuite seulement on affiche la msgbox suivante
        Debug.Print .Prenom & " " & .Nom & " a " & .Age & " ans."
    End With
End Sub
```

#### Sur Libération (Unload)

```
Private Sub Form_Unload(Cancel As Integer)
    Debug.Print "Libération du formulaire frmPerson"
    Set mobjPerson = Nothing
End Sub
```

#### Fenêtre de Debug

```
chargement du formulaire frmPerson
Evénement Initialize - Création d'un objet Person
l'Evénement Completed vient de se produire !
Michel Blavin a 35 ans.
Libération du formulaire frmPerson
Evénement Terminate - destruction d'un objet Person : Michel Blavin
```

Comme vous le voyez, l'événement est déclenché par l'objet *Person*, par contre, l'action à effectuer lorsque cet événement se produit n'est pas contenu dans la classe *Person* mais dans le formulaire qui utilise la classe. C'est l'utilisateur de l'objet qui décide ce qu'il veut faire lorsque cet événement se produit.

## 2.5 - La propriété Parent

Lorsque vous créez une hiérarchie entre vos objets, il peut être intéressant de pouvoir remonter la hiérarchie, c'est le rôle de la propriété *Parent*. Vous pouvez d'ailleurs l'appeler autrement mais ce serait juste totalement stupide car cela rendrait votre code un peu plus obscur.

#### Propriété Parent

```
Property Set Parent(ByRef objPersons As Persons)
    If objPersons Is Nothing Then
        mobjParent = objPersons
    End If
End Property

Property Get Parent() As Persons
    Parent = mobjParent
End Property
```

Vous remarquerez que la propriété *Parent* ne peut être affectée qu'une seule fois (write once).

Pour renseigner cette propriété, il nous faudra, dans l'objet *Parent*, ajouter la ligne de code suivante :

#### Affectation de la propriété Parent

```
Set mobjPerson.Parent = Me
```

## 3 - Collections

Manipuler un objet c'est bien, mais en manipuler plusieurs, c'est encore mieux. Grâce aux Collections nous allons pouvoir agir sur un groupe d'objets.

### 3.1 - L'objet Collection de VBA

Dans certains cas, on peut utiliser tout simplement utiliser l'objet *Collection*, mais il accepte tous les objets. Vous risquez d'utiliser dans votre code des méthodes ou des propriétés absentes de certains objets, ce qui provoquera des erreurs d'exécution pas belles (Ugly runtime error;)).

Autre avantage de taille, nous pouvons choisir d'implémenter tout ou partie des méthode de l'objet *Collection*, voir même, comme nous allons le voir, d'en ajouter.

### 3.2 - Collections personnalisées

#### 3.2.1 - Généralités

Par convention, les collections portent le même nom que les objets qu'elles contiennent suivit d'un "s". Dans notre cas, Nous appellerons notre collection *Persons*.

Une collection personnalisée implemente les méthodes de l'objet *Collection*, qui sont les suivantes :

- *Add* : qui permet d'ajouter un élément à la collection.
- *Count* : qui renvoie le nombre d'éléments présents dans la collection
- *Item* : qui permet d'utiliser un des objets présents dans la collection
- *Remove* : qui permet d'enlever un objet de la collection

Ainsi que la propriété :

- *Count* : En lecture seule, renvoie le nombre d'éléments présents dans la collection

#### 3.2.2 - Interface

Pour créer une collection personnalisée, nous allons utiliser un objet *Collection* que nous allons encapsuler dans une classe. Commençons par créer un nouveau module de classe et enregistrons le sous le nom de *Persons*. Nous allons déclarer un objet *Collection* qui sera privé à notre classe.

##### Déclaration de la collection privée

```
Private mcolPersons As Collection
```

Dans l'événement *Initialize* nous instancions notre objet collection pour qu'il soit disponible dès que notre collection *Persons* sera instanciée et nous libérons notre variable dans l'événement *Terminate*.

##### Événement Initialize et Terminate

```
Private Sub class_Initialize()  
    Debug.Print "Événement Initialize - collection Persons"  
    Set mcolPersons = New Collection  
End Sub
```

### Événement Initialize et Terminate

```
Private Sub class_Terminate()
    Debug.Print "Événement Initialize - collection Persons"
    If Not (mcolPersons Is Nothing) Then Set mcolPersons = Nothing
End Sub
```

Ajoutons maintenant les 4 méthodes que nous avons citées au paragraphe précédent.

Pour démontrer la possibilité d'ajouter des méthodes nous allons créer une méthode *CreatePerson* qui, comme son nom l'indique, permet de créer un objet *Person* (et de l'ajouter à la collection). Elle reçoit en paramètre le nom, le prénom et la date de naissance. Elle renvoie bien évidemment un objet *Person* (en fait, comme vous l'avez compris, une référence vers le nouvel objet créé).

### 3.2.3 - Implémentation

Nous allons donc assez simplement implémenter nos 4 méthodes de base en encapsulant de la manière la plus simple les méthodes de l'objet *Collection*. Commençons par la méthode **Add** qui est la plus complexe.

#### implémentation de base de la collection

```
Property Get Count as Long
    Count = mcolPersons.Count
End Property

Public Sub Add(ByRef objperson As Person, _
    Optional ByVal Key As String = "")
    'Si aucune clé key n'est fournit on en génère une automatiquement
    If Len(Key) = 0 Then
        Key = objperson.NomComplet & Format(objperson.DateNaissance, "yyyymmdd")
    End If
    mcolPersons.Add objperson, Key
    'l'objet est inséré dans la collection on automatiquement renseigne la propriété Parent
    Set objperson.Parent = Me
    Set objperson = Nothing
End Function

Public Sub Remove(ByVal Index As Variant)
    mcolPersons.Remove (Index)
End Function

Public Function Item(ByVal Index As Variant) As Person
    Set Item = mcolPersons.Item(Index)
End Function
```

Pas de problème particulier avec ces méthodes, on remarquera juste que là où l'objet *Collection* utilise le type *Object*, nous utilisons le type *Person* ce qui empêchera d'insérer tout autre type d'objet dans la collection.

Vous remarquerez aussi que dans la méthode *Add* on renseigne automatiquement la propriété *Parent* de l'objet fils.

Le paramètre *Index* des méthodes *Item* et *Remove* peuvent être de deux types différents.

- Type *String* : il correspond à la valeur que vous avez donnée au paramètre *Key* de la méthode *Add*. **Ce paramètre doit être une chaîne de caractère**, vous ne pouvez donc utiliser un champ numérique autoincrémenté. Vous devez choisir pour cette valeur soit une des propriétés de vos objets ayant une valeur unique. Pour notre exemple, j'ai choisi de laisser le choix à l'utilisateur de la classe et s'il n'est fourni aucune valeur nous lui donnerons une valeur issue de la concaténation de la propriété *NomComplet* et de la date de naissance.
- Type *Long* : il s'agit de l'index de position dans la collection. Pour un objet donné, cette valeur peut varier lors de l'insertion ou de la suppression d'objet de la collection. Par exemple, si nous supprimons le élément avec l'index *n* de la collection, c'est le c'est l'élément *n+1* qui prendra l'index *n* et ainsi de suite. C'est pour cette

raison que l'on ne peut pas se baser sur cette valeur pour identifier un élément.

Nous allons maintenant ajouter une méthode *CreatePerson* à la collection qui permet d'instancier un objet *Person* sans avoir à créer de pointeur vers un objet personne. Cela nous permet comme vous allez le voir dans le code qui suit d'instancier et d'insérer un objet dans la collection.

#### méthode personnalisée CreatePerson

```
Public Function CreatePerson(strNom As String, _
                           strPrenom As String, _
                           dtmNaissance As Date) As Person
Dim objPerson As Person

Set objPerson = New Person
With objPerson
    .Nom = strNom
    .Prenom = strPrenom
    .DateNaissance = dtmNaissance
End With

Set CreatePerson = objPerson

If Not (objPerson Is Nothing) Then Set objPerson = Nothing
End Function
```

Maintenant, testons notre collection. Voici un petit code qui va nous permettre de tester les méthodes de notre classe. (Observez la fenêtre de debug pour voir à quel moment les objets sont créés et supprimés.)

#### Procédure de test de la classe Persons

```
Sub testCollection()
Dim colPersons As Persons
Dim i As Long
Set colPersons = New Persons
With colPersons
    .Add .CreatePerson("Blavin", "Michel", #9/14/1969#)
    Debug.Print "Nombre d'element(s) dans la collection : "; .Count
    .Add .CreatePerson("Hendrix", "Jimmy", #9/14/1969#)
    Debug.Print "Nombre d'element(s) dans la collection : "; .Count
    .Add .CreatePerson("Joplin", "Janis", #9/14/1969#)
    Debug.Print "Nombre d'element(s) dans la collection : "; .Count
    .Add .CreatePerson("Morisson", "Jim", #9/14/1969#)
    Debug.Print "Nombre d'element(s) dans la collection : "; .Count

    Debug.Print "on fait defiler la collection : "
    For i = 1 To .Count
        Debug.Print .Item(i).NomCompleet
    Next

    Debug.Print "suppression d'un élément"
    .Remove 2
    Debug.Print "Nombre d'element(s) dans la collection : "; .Count

    For i = 1 To .Count
        Debug.Print .Item(i).NomCompleet
    Next
End With
Set colPersons = Nothing
End Sub
```

Ce qui nous donne :

#### Fenêtre de Debug

```
Événement Initialize - Création d'une collection Persons
Événement Initialize - Création d'un objet Person
Nombre d'element(s) dans la collection : 1
Événement Initialize - Création d'un objet Person
Nombre d'element(s) dans la collection : 2
Événement Initialize - Création d'un objet Person
Nombre d'element(s) dans la collection : 3
Événement Initialize - Création d'un objet Person
```

## Fenêtre de Debug

```

Nombre d'element(s) dans la collection : 4
on fait defiler la collection :
Michel Blavin
Jimmy Hendrix
Janis Joplin
Jim Morisson
suppression d'un élément
Événement Terminate - destruction d'un objet Person : Jimmy Hendrix
Nombre d'element(s) dans la collection : 3
Michel Blavin
Janis Joplin
Jim Morisson
Événement Terminate - destruction d'une collection Persons
Événement Terminate - destruction d'un objet Person : Michel Blavin
Événement Terminate - destruction d'un objet Person : Janis Joplin
Événement Terminate - destruction d'un objet Person : Jim Morisson

```

## 3.2.3 - Que manque-t-il aux collections personnalisées ?

Du fait que notre objet *Collection* est encapsulé, il perd certaines fonctionnalités comme la propriété par défaut qui n'existe pas.

## Il n'y a pas de méthode par défaut !

```

colPersons.Item(i).NomCompleet           ' Correct
colPersons(i).NomCompleet                 ' Incorrect renvoie une erreur.

```

Il existe un "workaround" pour simuler la propriété par défaut mais la solution ne me plaisant pas, je ne vous la montrerai pas, na ! (Pour les tordus, vous la trouverez dans les liens, en cherchant bien...)

L'autre manque est celui du *for each* qui ne fonctionne pas, la solution consiste à utiliser une boucle *for* classique. Attention, si vous supprimer des éléments dans votre boucle vous risquez d'avoir des surprises.

## Remplace la boucle For Each inopérante

```

For i = 1 To colPersons.Count
    Debug.Print colPersons.Item(i).NomCompleet
Next

```

## 4 - Les modules des formulaires et des Etats

Les modules de formulaires et d'états sont, eux aussi, des modules de classe. Ils n'ont pas d'événement *Initialize* et *Terminate* mais ils en ont bien d'autres que vous utilisez d'ailleurs tous les jours. Les modules attachés aux formulaires étant des modules de classe, vous pouvez leur ajouter les propriétés, méthodes et événements.

Rappelez-vous juste que, pour les utiliser, vous devez avoir un objet instancié, Le formulaire doit donc être ouvert même s'il est caché.

Un petit cas pratique, rien de tel pour illustré notre propos :

Il s'agit d'un formulaire pop-up contenant un controle calendrier il acceptera une propriété personnalisée contenant le nom du contrôle dans lequel on retournera la date qui sera sélectionner (par un double clic dans notre cas) dans le formulaire calendrier.

Commençons par créer un formulaire contenant le contrôle calendrier d'un bouton *Annuler* et enregistrer le sous le nom "frmDateChooser" ce qui nous donnera ceci :



Maintenant ajoutons le code gérant la propriété personnalisée.

### Ajout d'une propriété à un formulaire

```
Dim mprpCtlCible As Control

Property Set ctlCible(strCtl As Control)
    Set mprpCtlCible = strCtl
End Property
```

Comme vous voyez cette propriété est en écriture seule car la lecture de la propriété est inutile dans le contexte.

Maintenant, il faut que nous renvoyons la valeur que nous avons sélectionnée au contrôle de contenu dans la

propriété.

transmission de la valeur au contrôle cible.

```
Private Sub CtlCalendar_DblClick()
    On Error GoTo ErrManager
    'On affecte la valeur du controle calendrier à la valeur du controle contenu
    ' propriété
    mprpCtlCible.Value = Me.CtlCalendar.Value

    ToiTuSors:
        DoCmd.Close acForm, Me.Name
    Exit Sub

    ' Gestionnaire d'erreurs
    ErrManager:
        Select Case Err.Number
            Case 91
                MsgBox "Pas de contrôle cible !"
            Case Else
                MsgBox "L'erreur suivante s'est produite : " & vbCrLf & Err.Description, vbCritical, _
                    "Erreur N° " & Err.Number
        End Select
        Resume ToiTuSors
    End Sub
```

Puis le code pour le bouton *Annuler* :

Code du bouton annuler

```
Private Sub cmdCancel_Click()
    'Pour annuler il suffit de fermer le formulaire
    DoCmd.Close
End Sub
```

Pour que le formulaire s'ouvre à la date d'aujourd'hui, ajoutons le code nécessaire dans l'événement *Open* du formulaire.

Mise à la date d'aujourd'hui

```
Me.CtlCalendar.Value = Now
```

Pour tester créons un nouveau formulaire et mettons une Textbox dessus ajoutons dans l'événement *DblClick* le code suivant.

Utilisation du datechooser

```
Private Sub Texte0_DblClick(Cancel As Integer)
    DoCmd.OpenForm "frmdatechooser"
    Set Form_frmDateChooser.ctlCible = Me.Texte0
End Sub
```

Enregistrez votre formulaire et faites un test.

[Le DateChooser est disponible en téléchargement](#)

## Les liens qui ont la classe

### D'autres cours sur les modules de classes en VBA (en Anglais)

[Advanced Class Modules](#)

[Creating Reusable Class Modules in VBA](#)

[Class Modules: Simplify and Accelerate Your Development Processes](#)

### la programmation objet dans les autres langages.

Les [autres cours Access](#) sur [developpez.com](#)

[FAQ Java](#) et [FAQ C++](#).

Je vous conseille grandement la lecture (au moins de la première partie) de [Pensez en java](#), excellent livre sur Java, expliquant les concepts de la POO de manière claire et précise.

### Divers

La [FAQ Access](#)

Le [forum Access](#)