Office et les modules de classes

par Emmanuel Tissot (Office Park)

Date de publication : 21/11/2008

Dernière mise à jour :

Cet article vous propose de découvrir les fonctionnalités des modules de classe du langage VBA des applications Office.

I - Préambule

Cet article se propose d'expliquer aussi clairement et complètement que possible ce que sont les modules de classe dans le cadre de VBA, langage de programmation utilisé notamment par les applications Microsoft Office. La première partie exposera quelques généralités, des exemples permettront ensuite de se familiariser avec le fonctionnement de ces modules, enfin la troisième partie approfondira l'aspect théorique afin de montrer l'étendue des possibilités.

Le thème de cet article veut qu'il ne s'adresse pas aux débutants, je suppose donc que vous maîtrisez les concepts de base de VBA (portée des variables, appels de procédure) ainsi que des notions plus avancées (programmation évènementielle, gestion des erreurs) bref que vous avez un minimum d'expérience.

La rédaction de cet article m'a permis d'approfondir mes connaissances sur ce thème, j'espère qu'il en sera de même pour vous et je vous souhaite une bonne lecture.

II - Généralités

II-A - Classe et instance

Un module de classe est un module qui permet de définir un nouvel objet. Mais alors qu'est ce qu'un objet ? Pour faire court disons simplement que c'est un ensemble indissociable de données et de procédures permettant de manipuler ces données.

Une classe (comprenez un module de classe) est donc un ensemble de propriétés, de méthodes et d'évènements qui définissent respectivement les caractéristiques, les capacités et le comportement des objets qui en sont issus. Pour donner une définition imagée une classe est un modèle, un moule, un plan, un concept, une matrice.

Une instance est la représentation en mémoire d'un objet définit à partir d'une classe, il s'agit donc de la concrétisation d'une abstraction. Bien évidemment plusieurs instances de la même classe peuvent exister simultanément, elles demeureront indépendantes l'une de l'autre.

II-B - Créer un module de classe

Dans l'éditeur VBE faites simplement Insertion > Module de classe. Affichez la fenêtre propriétés (F4) et modifiez le nom par défaut pour donner à votre objet un nom plus explicite en rapport avec sa nature.

Dans cette même fenêtre apparaît la propriété Instancing qui accepte deux valeurs, Private par défaut, et PublicNotCreatable. Cela devrait vous donner une idée de son usage mais nous y reviendrons plus tard.

II-C - Contenu d'un module de classe

Un module de classe peut contenir (avec quelques restrictions) tout ce que peut contenir un module standard, l'inverse n'est pas vrai.

Pour les déclarations:

- Déclarations de variables scalaires
- Déclarations de variables tableau (1)
- Déclarations de constantes (1)
- Instructions Enum définissant un groupe de constantes
- Variables objets dotées d'évènements déclarées avec WithEvents (2)
- Instructions Event pour définir des évènements personnalisés (2)
- Instructions Type pour définir un type de donnée personnalisé (1)

- Instructions Declare pour établir des références externes (ex API Windows) (1)
- Instructions Implements pour redéfinir l'interface d'une autre classe (2)

Pour les procédures:

- Procédures Sub, Function et Property
- Procédures évènementielles liées à des variables déclarées avec le mot-clé WithEvents (2)
- Procédures évènementielles Class_Initialize et Class_Terminate (2)

(1) Ces éléments sont obligatoirement privés.
(2) Ces éléments sont propres aux modules de classes et ne peuvent figurer dans un module standard.

II-D - Evènements d'un module de classe

Un module de classe est doté de deux évènements particuliers, Class_Initialize et Class_Terminate. Les noms sont assez explicite sur leur raison d'être et devraient vous rappeler UserForm_Initialize et UserForm_Terminate.

Class_Initialize se déclenche quand vous créez une nouvelle instance de la classe avec le mot-clé New. Il est employé pour fixer les valeurs initiales des variables du module.

Class_Terminate se déclenche quand l'instance de la classe est détruite, c'est-à-dire quand cette instance n'est plus désignée par aucune variable. Il sert à faire le ménage avant la destruction effective de l'instance. Par exemple si une classe génère des fichiers temporaires, il est bon de prévoir leur destruction dans cet évènement afin d'éviter une prolifération inutile.

En pratique l'évènement peut être déclenché en affectant la valeur Nothing à toutes les variables désignant la même instance. En admettant que nous ayons deux variables désignant le même objet:

```
Set MonObjet1 = Nothing
Set MonObjet2 = Nothing
```

L'évènement Terminate ne sera déclenché que par la deuxième instruction. En d'autres termes affecter Nothing à une variable ne détruit pas l'instance qu'elle désignait mais seulement la référence à cette instance, tant qu'une autre variable continue de désigner cette même instance l'évènement Terminate ne se déclenche pas.

Il peut aussi se déclencher implicitement lorsque toutes les variables désignant une instance cessent d'exister, par exemple lorsque une variable locale à une procédure est détruite à la fin de cette procédure, ou quand un Userform est déchargé avec une instruction Unload. Notez enfin que l'évènement Terminate ne se déclenche pas si le programme rencontre une instruction End.

II-E - Utiliser un module de classe

Il faut tout d'abord déclarer une variable qui représentera l'objet, et ensuite initialiser cette variable.

```
Dim MonObjet As NomClasse
Set MonObjet = New NomClasse
```

Il est possible de réunir ces deux instructions pour n'en former qu'une seule.

```
Dim MonObjet As New NomClasse
```

Dans ce cas de figure l'instance est créée à chaque utilisation de la variable sauf si elle est déjà initialisée. En plus de rendre le programme moins lisible ce comportement empêche de vérifier au cours du programme si la variable est initialisée.

```
If MonObjet Is Nothing Then
   MsgBox "Variable non initialisée."
Else
   MsgBox "Variable initialisée."
End If
```

Avec la deuxième option le test ci-dessus affichera systématiquement la deuxième MsgBox puisque si la variable n'est pas initialisée - donc égale à Nothing - elle le sera avant que la comparaison ne soit effectuée. La première option semble donc préférable. La variable se manipule ensuite comme n'importe quelle autre variable objet en utilisant la syntaxe classique.

```
Objet.Propriete = Valeur            'Affecte une valeur à une propriété

Valeur = Objet.Propriete            'Récupère la valeur d'une propriété

Objet.Methode Argument              'Exécute une méthode de l'objet

Resultat = Objet.Methode(Argument)  'Lit une valeur renvoyée par une méthode de l'objet
```

II-F - Intérêt des modules de classes

Ecrire un module de classe c'est écrire du code qui permet d'écrire du code, ce n'est donc pas un finalité en soi mais plutôt une étape intermédiaire du processus de développement. La plupart des projets se passent très bien de module de classe, ce qui n'empêche pas de réaliser des applications parfaitement fiables et adéquates. On peut donc estimer que les modules de classes n'ont rien d'indispensable et ne sont qu'une manière parmi d'autres d'arriver à ses fins. Ils ne sont en fait qu'une couche de programmation supplémentaire qui s'intercale entre les bibliothèques d'objets (VBA, Office, ADO...) utilisées par un projet et le code applicatif qui est écrit pour réaliser ce projet.

II-F-1 - Utiliser des variables évoluées

Si vous avez déjà ressenti les limites des variables classiques qui ne savent que stocker une information, vous avez peut être eu l'occasion d'utiliser des variables personnalisées définies avec une instruction Type. Cette instruction vous permet d'avoir facilement et en permanence accès à plusieurs informations connexes, c'est déjà beaucoup mieux qu'une simple variable. Ces variables personnalisées restent toutefois limitées en ce sens qu'elles n'offrent aucun contrôle (en dehors du type de donnée) sur le contenu de leurs différents champs. Les objets seront l'étape suivante de l'évolution de vos variables. Non seulement ils associeront une quantité quelconque d'informations variées et de procédures permettant de contrôler, manipuler, ou restituer ces informations mais en plus ils pourront envoyer des signaux à vos applications par le biais d'évènements ou d'erreurs personnalisées. Les objets sont donc des variables intelligentes conçues pour simplifier la vie du développeur.

II-F-2 - Cacher la complexité

En utilisant les objets fournis par une application telle que Word ou Excel vous ne vous demandez pas comment ils fonctionnent. Par exemple, la classe Workbook propose une méthode SaveAs qui permet d'enregistrer le classeur sous un autre nom. Cela implique d'accéder au disque dur, de vérifier si l'espace disponible est suffisant, mais aussi que le répertoire défini existe, et certainement d'autres choses. Toutes ces vérifications complexes sont cachées derrière un simple appel à une méthode qui résout l'ensemble des problèmes une fois pour toutes, dés l'instant que le comportement de cette méthode est reconnu fiable il devient inutile de réinventer un processus à chaque fois que l'on veut effectuer cette tache élémentaire, et ce même si on ignore comment le résultat est obtenu. Les modules de classes doivent offrir la même garantie et se comporter comme des "boites noires", leur contenu n'a plus d'importance,

la seule chose qui compte est qu'ils assurent les services pour lesquels ils ont été conçus afin que le développeur qui les utilise puisse se concentrer sur son objectif essentiel, réaliser son application.

II-F-3 - Produire un code fiable, lisible, réutilisable, auto documenté

Pour définir un nouvel objet, vous devrez définir ses propriétés, méthodes et évènements. Cela demandera au préalable un minimum d'analyse et rallongera d'autant le temps de développement mais il s'agit aussi d'un investissement à long terme, l'objectif ultime étant de produire un code plus compréhensible, facile à maintenir, auto documenté et surtout réutilisable, ce dernier aspect étant particulièrement important. Songez que si un problème est résolu au moyen d'un module de classe il n'y a rien de plus facile que de réutiliser cette classe dans un autre projet pour résoudre un problème similaire. Cela n'est toutefois possible que si le problème est exprimé sous une forme générique, abstraite, indépendante du contexte de l'application pour laquelle la classe a été développée initialement. Enfin, produire des classes réutilisables force une tolérance d'erreur égale à zéro et donc une phase de test rigoureuse, ce qui tend à augmenter la fiabilité globale du programme.

II-G - Cas particuliers

II-G-1 - Les modules de documents

Dans certains fichiers Office, le projet VBA de ces fichiers intègre par défaut des modules de classes. ThisDocument dans Word ou ThisWorkbook dans Excel, sont des modules de classe. A ce titre ils fournissent toutes les fonctionnalités prévues par ce type de module (ils sont en mesure d'héberger des procédures évènementielles), et en subissent toutes les contraintes (impossible d'y déclarer un tableau Public par exemple). Néanmoins ils ne peuvent pas être instanciés car ils ne définissent pas un objet, ils sont simplement une extension d'un objet.

La raison d'être de ces modules est de fournir un raccourci de programmation pour l'exploitation des objets auxquels ils sont rattachés. Grâce à eux nous pouvons nous passer d'écrire un module de classe, de déclarer une variable et de l'instancier. Il reste néanmoins possible de s'en passer et c'est même utile dans certains cas. Par exemple dans un projet Excel, le regroupement des procédures évènementielles des feuilles de calcul dans un module de classe, permet d'une part d'exporter les feuilles dans un autre classeur sans exporter le code qui y est attaché, et d'autre part préserve le code en cas de suppression accidentelle d'une feuille, cela sans polluer le module ThisWorkbook qui par nature peut jouer ce rôle.

II-G-2 - Les formulaires (Userform)

Tout comme les modules ci-dessus, un Userform fournit ses propres procédures évènementielles, ainsi que celles des objets qu'il contient (TexBox etc.) et celles de variables objets déclarées avec WithEvents. Il se rapproche un peu plus d'un module de classe normal quand on s'aperçoit qu'il est possible d'instancier un Userform, c'est-à-dire d'utiliser simultanément plusieurs objets basés sur un seul modèle.

```
Dim X As Userform1, Y As Userform1

Set X = New Userform1
Set Y = New Userform1
```

Ces instructions créent deux instances du même formulaire, sans les afficher. On peut ensuite manipuler ces deux instances indépendamment l'une de l'autre avant de les afficher tour à tour.

```
X.Caption = "Formulaire X"
Y.Caption = "Formulaire Y"

X.Show
Y.Show
```

En pratique, on a rarement besoin d'utiliser simultanément plusieurs exemplaires d'un même formulaire, et de fait un formulaire s'utilise le plus souvent comme un objet qu'il est inutile d'instancier, via son nom de classe.

```
Userform1.Caption = "Légende du formulaire 1"
Userform1.Show
```

Lorsque vous n'effectuez pas l'instanciation du formulaire de manière explicite c'est VBA qui s'en charge à votre place, conséquence pratique dans l'exemple ci-dessus la procédure Userform_Initialize est déclenchée par la première instruction faisant référence au formulaire. En résumé un Userform c'est un module de classe associé à une interface graphique personnalisable.

III - Les gestionnaires d'évènements

Cette partie va présenter des exemples simples de modules de classes. Si d'un point de vue technique ces modules sont bels et bien de nouveaux objets, ils n'en sont pas vraiment d'un point de vue conceptuel si l'on considère que leur objectif n'est pas de définir une structure de donnée associée à des méthodes de manipulation mais plutôt d'exploiter les évènements de certains objets dans le cadre d'une application particulière.

III-A - Capter un évènement

Lorsque un objet déclenche un évènement, c'est pour permettre à l'application de réagir à cet évènement. Cette réaction passe par une procédure évènementielle associée à cet objet et rédigée dans un module de classe. L'exemple ci-dessous montre comment utiliser les évènements de l'application Excel.

III-A-1 - Associer une variable à un évènement

Dans un nouveau projet créez un nouveau module de classe que vous appellerez AppEvents. Ecrivez ensuite dans ce module la déclaration suivante.

```
Private WithEvents xlApp As Application
```

Le mot-clé WithEvents sert à indiquer que nous allons attacher des procédures évènementielles à la variable xlApp. On ne peut l'employer qu'avec des objets dotés d'un jeu d'évènements et vous pouvez remarquer que l'éditeur VBE ne propose qu'une liste restreinte d'objets répondants à ce critère, cette liste est directement dépendante de l'application avec laquelle vous travaillez et des bibliothèques que vous utilisez.

Une fois cette déclaration effectuée les procédures évènementielles de la classe Application deviennent disponibles dans la liste de droite du module lorsque vous sélectionnez xlApp dans la liste de gauche. Nous pouvons donc maintenant écrire une procédure qui se déclenchera à chaque fois qu'un classeur sera ouvert.

```
Private Sub xlApp_WorkbookOpen(ByVal Wb As Workbook)
   MsgBox Wb.FullName
End Sub
```

Cette procédure ne fait qu'afficher une MsgBox, elle pourrait servir à réorganiser les fenêtres, afficher ou masquer des barres d'outils, tenir un historique des fichiers ouverts etc.

III-A-2 - Activer les procédures évènementielles d'une variable

La seule chose à faire pour rendre opérationnelle une procédure évènementielle attachée à une variable est d'initialiser cette variable. Si vous avez remarqué que xlApp est déclarée Private vous conclurez rapidement que le seul moyen de l'initialiser consiste à utiliser la procédure Class_Initialize.

```
Private Sub Class_Initialize()
   Set xlApp = Application
End Sub
```

Il n'y a rien d'autre à ajouter à ce module de classe, il est tout à fait fonctionnel en l'état. Il ne reste maintenant qu'à créer dans le projet une instance de la classe. Pour ce faire déclarons une variable de type AppEvents dans un module standard.

```
Dim ThisApplication As AppEvents
```

Les règles générales relatives à la portée et à la durée de vie des variables s'appliquent bien entendu à notre variable ThisApplication. Pour qu'elle puisse perdurer nous devons donc la déclarer au niveau module, mais rien n'interdirait d'effectuer cette déclaration dans une procédure si nous n'avions qu'un besoin temporaire de surveiller ces évènements. Enfin, on initialise cette variable dans une procédure des plus basiques.

```
Public Sub Activer_Evenements_Application()
   Set ThisApplication = New AppEvents
End Sub
```

L'instanciation du module entraîne via sa procédure Class_Initialize l'initialisation de sa variable xlApp et par voie de conséquence les procédures évènementielles de cette variable deviennent opérationnelles. Elles le resteront tant que la variable ThisApplication contiendra une référence à une instance du module AppEvents.

La réception d'un évènement est donc finalement assez triviale et peut être mise en □uvre en appliquant le schéma suivant.

Dans un module de classe:

- Déclaration d'une variable avec le mot-clé WithEvents
- Ecriture des procédures évènementielles liées à cette variable

Ailleurs dans le projet:

- Déclaration d'une variable dont le type correspond au nom du module de classe
- Affectation à cette variable d'une instance de la classe avec le mot-clé New
- Connexion de la variable WithEvents de l'instance avec un objet existant

III-B - Procédures évènementielles commune à plusieurs objets

Nous venons de voir comment intercepter les évènements d'un objet au moyen d'un module de classe, nous allons maintenant affecter la même procédure évènementielle à plusieurs objets identiques, en l'occurrence une série de TextBox placés sur un Userform.

III-B-1 - Utiliser une collection d'instance

Dans un nouveau projet créez donc un nouveau formulaire et placez-y quelques TextBox. Notre but sera de n'autoriser dans chacun de ces TextBox que la saisie de nombre entier. Ce problème pourrait se résoudre en dupliquant pour chaque TextBox la procédure évènementielle suivante.

```
Private Sub TextBox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
   If KeyAscii < 48 Or KeyAscii > 57 Then KeyAscii = 0
End Sub
```

Bien évidemment cela est particulièrement lourd, fastidieux et peu évolutif. En cas d'ajout d'un TextBox supplémentaire il faut systématiquement ajouter une procédure, et si les TextBox sont renommés par la suite les procédures devront être réécrites. Voyons maintenant comment l'utilisation d'un module de classe permet de résoudre ces problèmes et d'obtenir d'autres avantages. Dans un module de classe que nous appellerons pour l'occasion NumBox, écrivons le code suivant.

```
Public WithEvents TargetBox As MSForms.TextBox

Private Sub TargetBox_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
   If KeyAscii < 48 Or KeyAscii > 57 Then KeyAscii = 0
End Sub
```

La déclaration d'une variable publique (TargetBox) équivaut à définir une propriété, nous y reviendrons.

Nous venons d'attacher notre procédure évènementielle à une variable de type TextBox, nous allons maintenant faire en sorte que cette variable pointe sur chaque TextBox du formulaire en créant dans celui-ci autant d'instances du module - et donc de la variable - que nécessaire au moyen d'une collection.

```
Dim NumBoxes As Collection              'Stocke les références à chaque instance

Private Sub UserForm_Initialize()
   Dim Ctl As MSForms.Control
   Dim MyNumBox As NumBox
   Set NumBoxes = New Collection
   For Each Ctl In Me.Controls
      If TypeOf Ctl Is MSForms.TextBox Then
         Set MyNumBox = New NumBox      'Crée une nouvelle instance
         Set MyNumBox.TargetBox = Ctl   'Connecte la variable à l'objet
         NumBoxes.Add MyNumBox          'Ajoute l'instance à la collection
      End If
   Next
End Sub
```

Comme vous pouvez le voir la démarche est exactement la même que dans l'exemple précédent, la seule différence étant que la classe est instanciée plusieurs fois afin de surveiller plusieurs objets. On s'aperçoit immédiatement que la quantité de code produite est bien moins importante qu'avec une approche sans module de classe ce qui implique une maintenance plus aisée. D'autre part ce qui fonctionne pour le formulaire A fonctionnera aussi pour le formulaire B, le code placé dans le module de classe est donc réutilisable à loisir.

III-B-2 - Objets et évènements dynamiques

Un autre bénéfice à utiliser un module de classe plutôt qu'une approche statique est qu'il devient possible d'affecter des évènements dynamiquement à des objets qui peuvent eux-mêmes être créés de manière dynamique. En voici l'illustration avec 3 TextBox ajoutés à l'exécution.

```
Dim NumBoxes As Collection                          'Stocke les références à chaque instance

Private Sub UserForm_Initialize()
   Dim Ctl As MSForms.Control
   Dim MyNumBox As NumBox
   Dim i As Long
   Set NumBoxes = New Collection
   For i = 1 To 3
      Set Ctl = Me.Controls.Add("Forms.TextBox.1") 'Crée un TextBox
      Ctl.Top = (i - 1) * 30 + 10                  'Positionne le TextBox
      Ctl.Left = 10
      Set MyNumBox = New NumBox                    'Crée une nouvelle instance
      Set MyNumBox.TargetBox = Ctl                 'Connecte la variable à l'objet
      NumBoxes.Add MyNumBox                        'Ajoute l'instance à la collection
   Next
End Sub
```

Aussi aisément qu'on peut affecter des procédures évènementielles à des objets on peut déconnecter ces procédures d'un objet particulier en détruisant l'instance du module qui pointe sur cet objet.

```
Private Sub CommandButton1_Click()
   MyNumBoxes.Remove 1
End Sub
```

Et pour une déconnexion globale il suffirait de détruire la collection.

```
Private Sub CommandButton1_Click()
   Set MyNumBoxes = Nothing
End Sub
```

III-B-3 - Réutilisabilité

Oublions un instant les objets et évènements dynamiques et supposons simplement que notre formulaire contienne trois TextBox auxquels on veuille associer une valeur plafond et un affichage en rouge lorsque l'utilisateur indique une valeur qui dépasse ce plafond. Voici à quoi pourrait ressembler notre module.

```
Public WithEvents TargetBox As MSForms.TextBox

Private Sub TargetBox_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
   If KeyAscii < 48 Or KeyAscii > 57 Then KeyAscii = 0
End Sub

Private Sub TargetBox_Change() 'Evènement Change de la variable
   Dim OverMaxValue As Boolean
   With TargetBox
      Select Case .Name
         Case "TextBox1"
            If .Value > 100 Then OverMaxValue = True
         Case "TextBox2"
            If .Value > 200 Then OverMaxValue = True
         Case "TextBox3"
            If .Value > 300 Then OverMaxValue = True
      End Select
      If OverMaxValue Then
         .ForeColor = vbRed
      Else
         .ForeColor = vbBlack
      End If
   End With
End Sub
```

Ceci est certes fonctionnel, simple à écrire et à comprendre, mais va à l'encontre de la philosophie des classes personnalisés. En effet une modification du formulaire (ajout d'un nouveau champ par exemple) nécessitera une modification du module de classe. De même un simple changement des valeurs plafonds associées à chacun des champs est impossible en cours d'exécution les valeurs étant ici codées en dur, tout comme le nom des champs eux-mêmes.

De tels changements seraient sans doute faciles à mettre en □uvre mais cela révèle néanmoins une liaison bien trop forte entre le module de classe et l'application, autrement dit un manque d'abstraction. Quant à la réutilisabilité du module elle est ici proche de zéro tout simplement parce qu'il contient des informations qui sont du domaine de l'application, il serait donc impossible de l'utiliser en l'état pour un formulaire ayant des besoins différents.

Pour préserver les avantages qu'une approche utilisant un module de classe est censée fournir (souplesse, réutilisabilité) il faut donc séparer strictement les taches entre la classe et l'application.

Pour l'application:

- Décider quels sont les champs à surveiller
- Définir une valeur maximale pour chacun d'eux

Pour la classe NumBox:

- Détecter les évènements relatifs à un champ de saisie
- Evaluer la saisie et modifier l'affichage si nécessaire

Il apparaît maintenant clairement que le module de classe n'a besoin que de deux informations:

- Un objet TextBox
- Une valeur numérique

Notre module de classe corrigé pourrait donc ressembler à ceci.

```
'Déclarations au niveau module
Public MaxValue As Long
Public WithEvents TargetBox As MSForms.TextBox

'Evènements de la variable TargetBox
Private Sub TargetBox_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
   If KeyAscii < 48 Or KeyAscii > 57 Then KeyAscii = 0
End Sub

Private Sub TargetBox_Change()
   On Error Resume Next 'NB: La gestion d'erreur est volontairement minimaliste
   If MaxValue > 0 Then
      With TargetBox
         If CLng(.Value) > MaxValue Then
            .ForeColor = vbRed
         Else
            .ForeColor = vbBlack
         End If
      End With
   End If
End Sub
```

La procédure TargetBox_Change ne fonctionnera que si l'application (comprenez le formulaire) initialise cette propriété avec une valeur strictement positive. Du coté de l'application, nous sommes libres de créer - statiquement ou dynamiquement - autant de champs de saisies que nous le voulons et de définir pour chacun d'eux une valeur plafond à n'importe quel moment.

```
Dim NumBoxes As Collection                          'Stocke les références à chaque instance

Private Sub UserForm_Initialize()
   Dim Ctl As MSForms.Control
   Dim MyNumBox As NumBox
   Dim i As Long
   Set NumBoxes = New Collection
   For i = 1 To 3
      Set Ctl = Me.Controls.Add("Forms.TextBox.1") 'Crée un TextBox
      Ctl.Top = (i - 1) * 30 + 10                  'Positionne le TextBox
      Ctl.Left = 10
      Set MyNumBox = New NumBox                    'Crée une nouvelle instance
      Set MyNumBox.TargetBox = Ctl                 'Connecte la variable à l'objet
      MyNumBox.MaxValue = i * 100                  'Définit une valeur plafond
      NumBoxes.Add MyNumBox                        'Ajoute l'instance à la collection
   Next
End Sub
```

En élevant le niveau d'abstraction de notre classe nous l'avons donc fait passer du statut de simple gestionnaire d'évènement dont l'usage était limité à un seul formulaire à celui d'objet personnalisé utilisable sans aucune modification pour n'importe quel formulaire de n'importe quel projet.

III-C - Conclusion

Avant de commencer la lecture du chapitre suivant, voici ce que vous devriez retenir de celui qui s'achève :

- Une analyse préalable est impérative pour ne pas mélanger le code de l'application et celui du module.
- Il est obligatoire d'instancier une classe avant de l'utiliser.
- La création d'une instance se fait avec le mot-clé New, cela déclenche la procédure Class_Initialize.
- Il n'existe pas de constructeur personnalisé, on est donc souvent amené à prévoir une procédure Initialize.
- Tout ce qui peut être déclaré privé devrait l'être, cela augment la fiabilité.
- Une instance n'est détruite que lorsqu'elle n'est plus désignée par aucune variable, à ce moment seulement la procédure Class_Terminate se déclenche.
- Dés lors qu'un objet dispose d'évènements, on peut les intercepter via une variable déclarée avec WithEvents.

IV - Les objets personnalisés

Maintenant que nous sommes familiarisés avec les modules de classes et leurs principes élémentaire de fonctionnement nous allons étudier leurs différentes facettes d'un point de vue plus théorique afin de comprendre comment il est possible de créer de nouveaux objets.

Dans cette section j'utilise les conventions suivantes :

- Toutes les procédures (Sub, Function et Property) étant publiques et par souci de lisibilité je ne précise pas leurs portées sauf si elles doivent être privées.
- Le nom des variables servant à conserver la valeur d'une propriété est composé du préfixe cp (Class Property) suivi du nom de la propriété correspondante.

IV-A - Les propriétés

IV-A-1 - Définition

C'est une caractéristique d'un objet c'est-à-dire un élément permettant de le décrire. Si cet élément est modifiable on parle alors de propriété en lecture écriture, dans le cas contraire il s'agit d'une propriété en lecture seule.

IV-A-2 - Créer une propriété

Le moyen le plus simple, que nous avons déjà utilisé, consiste à déclarer une variable publique au niveau du module.

```
Public Value As Double
```

La variable étant publique, elle sera librement accessible à l'application aussi bien en lecture qu'en écriture. Une autre façon de procéder consiste à utiliser une paire de procédures Property associée à une variable privée de niveau module.

```
Private cpValue As Double

Property Get Value() As Double
   Value = cpValue         'Renvoi la valeur actuelle
End Property

Property Let Value(ByVal NewValue As Double)
   cpValue = NewValue      'Mise à jour de la valeur
End Property
```

Le type de donnée retournée par une procédure Property Get doit correspondre au type de donnée de l'argument reçu par la procédure Property Let (ou Set) portant le même nom. Le rôle des procédures Property est de servir de relais entre la variable privée du module et l'application utilisant le module. Property Get sert à la lecture, Property Let à l'écriture, l'argument NewValue représentant la partie droite de l'instruction d'affectation.

Les arguments des procédures Property sont toujours passés par valeur et ce même si vous précisez le mot-clé ByRef, cette particularité permet de préserver avec certitude les variables de l'application des agissements du module. Les arguments des procédures Property peuvent - et doivent - donc être considérés comme des variables locales à la procédure.

Quelque soit l'option choisie pour définir la propriété l'application accède à la variable avec les mêmes instructions, ce qui implique la possibilité de transformer une variable publique en couple de procédures Property et vice-versa sans avoir à modifier l'application.

```
Dim MonObjet As NomClasse

Set MonObjet = New NomClasse

MonObjet.Value = 1            'Pour l'écriture
MsgBox MonObjet.Value         'Pour la lecture
```

IV-A-3 - Avantages des procédures Property

Les procédures Property sont bien plus souples et fiables qu'une simple variable publique. En effet une variable publique ne peut définir qu'une propriété en lecture écriture alors qu'avec les procédures Property il est possible de définir une propriété en lecture seule en n'écrivant pas de procédure Property Let (ou Set).

L'argument NewValue étant passé par valeur le module ne peut pas modifier la variable qui lui est transmise par l'application et de l'autre coté du miroir, la variable cpValue étant privée l'application ne pourra la modifier qu'en passant par la procédure Property Let. L'application reste donc maîtresse de ses propres variables et le module reste maître des siennes, cette étanchéité se traduit concrètement par une plus grande fiabilité.

Corollaire de ce qui précède, les procédures Property permettent de contrôler efficacement les valeurs qui sont assignées à la variable. En admettant que notre propriété Value représente un prix, nous ne pouvons admettre que

sa valeur soit inférieure à zéro. Il nous suffit de modifier la procédure Property Let pour empêcher une assignation de valeur incorrecte.

```
Property Let Value(NewValue As Double)
   If NewValue >= 0 Then
      cpValue = NewValue                                    'Mise à jour
   Else
      Err.Raise vbObjectError + 1, , "Valeur négative interdite"  'Erreur
   End If
End Property
```

Le même raisonnement s'applique à l'intérieur du module. Si d'autres procédures du module doivent modifier une variable de propriété il est préférable - même si ce n'est pas une règle absolue - d'appeler la procédure Property Let de cette propriété plutôt que d'accéder directement à la variable. Pour reprendre l'exemple ci-dessus un accès direct à la variable cpValue est susceptible de lui affecter une valeur qui ne serait pas conforme à la règle contenue dans la procédure Property Let, en passant systématiquement par la procédure Property Let une telle erreur ne peut pas se produire.

De même la modification d'une propriété peut nécessiter d'autres actions. Pour reprendre l'exemple de NumBox, lorsque le formulaire modifie la propriété MaxValue il est impératif de réévaluer la valeur courante du TextBox pour en synchroniser l'affichage, ce qui est possible si la propriété est implanté sous forme de procédures Property ne l'est pas avec une simple variable publique.

```
Private cpMaxValue As Long

Property Let MaxValue(NewMaxValue As Long)
   cpMaxValue = NewMaxValue
   TargetBox_Change              'Synchronisation de l'affichage
End Property
```

Enfin et ce n'est pas le moins important, les procédures Property sont évolutives. Si pour une raison quelconque vous décidez qu'elles sont inadaptées, vous pouvez les modifier pour ajouter ou supprimer des contrôles, cela sera sans incidence sur l'application qui utilise le module. Une variable publique n'offre évidemment pas cette possibilité.

IV-A-4 - Propriété renvoyant un objet

Il est tout à fait possible de définir une propriété de type objet, il faut pour cela utiliser une procédure Property Set au lieu de Property Let, et utiliser le mot-clé Set dans la procédure Property Get.

```
Private cpMyRange As Range                'Stocke une plage de cellule

Property Get MyRange() As Range
   Set MyRange = cpMyRange                'Renvoi la plage stockée
End Property

Property Set MyRange(NewRange As Range)
   Set cpMyRange = NewRange               'Définit une nouvelle plage à stocker
End Property
```

IV-A-5 - Propriété de type Variant

Les propriétés de type Variant pouvant contenir aussi bien des données intrinsèques (Double, String etc.) que des objets il faut prévoir les deux cas.

```
Private cpVariantProperty As Variant
```

Pour la lecture de la propriété il est nécessaire de tester le type de donnée de la variable avant de la renvoyer.

```
Property Get VariantProperty() As Variant
   If IsObject(cpVariantProperty) Then
      Set VariantProperty = cpVariantProperty
   Else
      VariantProperty = cpVariantProperty
   End If
End Property
```

Pour l'écriture il suffit de prévoir simultanément les procédures Property Let et Property Set, la procédure effectivement appelée sera déterminée par l'instruction appelante.

```
MonObjet.VariantProperty = 1                'Appellera Property Let
Set MonObjet.VariantProperty = AnyObject    'Appellera Property Set
```

```
Property Let VariantProperty(NewVariantProperty As Variant)
   clsVariantProperty = NewVariantProperty
End Property

Property Set VariantProperty(NewVariantProperty As Variant)
   Set clsVariantProperty = NewVariantProperty
End Property
```

Si l'usage de propriété de type Variant est possible il est toutefois déconseillé. D'une part parce que les variables de type Variant sont moins performantes que les variables typées, mais surtout parce qu'il est plus difficile d'en contrôler le contenu. Bien entendu si la variable n'est pas destiné à contenir des objets on se passera de Property Set, et inversement de Property Let si on ne veut stocker que des objets dans notre variable.

IV-A-6 - Propriété renvoyant un tableau

Pour exposer un tableau, deux stratégies sont envisageables, exposer un véritable tableau ou exposer une propriété de type Variant contenant un tableau. Par souci de simplification on admettra que la propriété définit un tableau unidimensionnel de type Long et d'indice inférieur égal à zéro et que le paramètre recu par Property Let est conforme à ces caractéristiques. Concrètement, il serait bien entendu nécessaire de vérifier ces caractéristiques avant de procéder à un quelconque traitement.

IV-A-6-a - Exposition d'un tableau typé

```
Private cpList() As Long

Property Get List() As Long()
   List = cpList
End Property

Property Let List(ByRef NewList() As Long)
   'Vérification des caractéristiques de NewList (base et dimensions)
   cpList = NewList
End Property
```

Notez que les dimensions du tableau ne sont pas fixées à la déclaration, ceci parce qu'il est impossible de procéder à des affectations entre tableaux de taille fixe, et que Property Get se termine par une paire de parenthèses indiquant qu'elle renvoie un tableau.

Bien qu'il soit déclaré ByRef (une déclaration ByVal engendre une erreur de compilation), l'argument NewList se comporte comme s'il était déclaré ByVal. La modification de ses éléments, de ses dimensions avec ReDim ou une éventuelle réinitialisation avec l'instruction Erase serait donc sans effet sur le tableau transmis par l'application.

```
Sub Test_Propriete_Tableau_1()
   Dim MonObjet As NomClasse, TabLong() As Long, i As Long
   Set MonObjet = New NomClasse
   ReDim TabLong(0 To 3)
   For i = 0 To 3
      TabLong(i) = i
   Next
   MonObjet.List = TabLong       'Affectation d'un tableau à la propriété
   Erase TabLong
   MsgBox UBound(MonObjet.List)  'Affiche l'indice supérieur
   TabLong = MonObjet.List       'Récupère le tableau stocké par la propriété
End Sub
```

En lecture l'application reste libre de récupérer la valeur de la propriété dans un tableau dynamique ou dans un Variant, les fonctions UBound et LBound sont applicables à la propriété Values.

En écriture, l'application doit fournir en paramètre un tableau dynamique de même type que la propriété. Toute affectation d'un tableau de taille fixe ou d'un autre type de donnée provoque une erreur de compilation même s'il s'agit d'un type compatible (Integer au lieu de Long par exemple). L'usage des fonctions Array ou Split, qui renvoient un Variant, est donc impossible.

Cette solution est peu satisfaisante lorsqu'il s'agit d'accéder aux éléments du tableau. En effet, si en lecture une syntaxe limite intuitive est fonctionnelle, la même syntaxe est inopérante en écriture.

```
MsgBox MonObjet.List()(1)          'Renvoie la valeur de l'élément 1
MonObjet.List()(1) = 10            'L'élément 1 n'est pas modifié
```

En conséquence si l'application veut modifier un élément du tableau il lui faut d'abord récupérer l'intégralité du tableau dans une variable temporaire, modifier ensuite l'élément dans cette variable et enfin réaffecter cette variable à la propriété.

```
TabLong = MonObjet.List            'Récupère l'intégralité du tableau
TabLong(1) = 10                    'Modifie un élément
MonObjet.List = TabLong            'Affecte l'intégralité du tableau à la propriété
```

Ceci n'étant ni convivial ni gage de performance, on utilisera donc une deuxième propriété dotée d'un argument identifiant l'élément auquel on accède.

```
Property Get ListItem(Index As Long) As Long
   ListItem = cpList(Index)
End Property

Property Let ListItem(Index As Long, NewListItem As Long)
   cpList(Index) = NewListItem
End Property
```

Ces procédures sont évidemment susceptibles de générer une erreur d'exécution si Index ne correspond pas à un indice valide. Enfin, si un contrôle sur la validité des éléments du tableau est nécessaire, il suffit de prévoir une simple fonction privée contenant les règles de validation.

```
Property Let List(ByRef NewList() As Long)
   Dim i As Long
```

```
    'Vérification des caractéristiques de NewList (base et dimensions)
    For i = 0 To UBound(NewList)                                          'Validation
        If Not CheckItem(NewList(i)) Then
            Err.Raise vbObjectError + 1, , "Valeur incorrecte à l'indice " & i   'Erreur
        End If
    Next
    cpList = NewList                                                      'Affectation
End Property

Property Let ListItem(Index As Long, NewListItem As Long)
    If CheckItem(NewListItem) Then                                        'Validation
        cpList(Index) = NewListItem                                       'Affectation
    Else
        Err.Raise vbObjectError + 1, , "Valeur incorrecte à l'indice " & Index  'Erreur
    End If
End Property

Private Function CheckItem(Item As Long) As Boolean
    CheckItem = Item >= 0                  'Refuse les valeurs négatives
End Function
```

Avantage principal de cette solution, on ne procède nulle part à un quelconque contrôle sur le type des données du tableau reçu en paramètre, cela facilite et accélère le traitement qui reste donc relativement simple. Du point de vue de l'application les contraintes d'usage peuvent néanmoins sembler trop rigides, ce qui nous amène à envisager la deuxième solution.

IV-A-6-b - Exposition d'un Variant

Exposer une propriété sous forme d'un Variant ne signifie pas pour autant que cette propriété soit destinée à contenir tout les types de données possibles. Par souci de cohérence avec l'exemple précédent on conservera donc en interne un tableau typé.

```
Private cpList() As Long

Property Get List(Optional Index As Variant) As Variant
    If IsMissing(Index) Then
        List = cpList                  'Renvoie le tableau
    Else
        List = cpList(CLng(Index))     'Renvoie un élément
    End If
End Property
```

Pour la lecture, la propriété se voit dotée d'un argument optionnel de type Variant permettant de déterminer ce que demande l'application à l'aide de la fonction IsMissing. Si Index est absent on renverra l'intégralité du tableau, et un seul élément dans le cas contraire. Bien évidemment si le type sous-jacent de l'argument Index n'est pas un entier ou une valeur convertible en entier cela provoquera une erreur d'incompatibilité de type, ce qui est le comportement normal d'un tableau.

Le même principe est utilisé pour l'écriture mais l'affectation du tableau est rendue plus complexe par la diversité des types de paramètres acceptables. Pour l'anecdote la déclaration de cette procédure est assez inhabituelle puisque elle inclus un argument obligatoire après un argument optionnel.

```
Property Let List(Optional Index As Variant, NewList As Variant)
    Dim Item As Long, i As Long, Temp() As Long
    If IsMissing(Index) Then
        'Vérification des caractéristiques de NewList (base et dimensions)

        If VarType(NewList) = vbArray + vbLong Then        'Selon le type sous-jacent de NewList
            For i = 0 To UBound(NewList)
                Item = CLng(NewList(i))                       '1 Conversion
                If Not CheckItem(Item) Then                   '2 Validation
```

```
                Err.Raise vbObjectError + 1, , "Valeur incorrecte à l'indice " & i      'Erreur
            End If
         Next
         cpList = NewList                                    '3 Affectation du tableau
      Else
         ReDim Temp(0 To UBound(NewList))
         For i = 0 To UBound(NewList)
            Item = CLng(NewList(i))                          '1 Conversion
            If CheckItem(Item) Then                          '2 Validation
               Temp(i) = Item                                '3 Stockage temporaire
            Else
               Err.Raise vbObjectError + 1, , "Valeur incorrecte à l'indice " & i      'Erreur
            End If
         Next
         cpList = Temp                                       '4 Affectation du tableau
      End If

   Else
      Item = CLng(NewList)                                   '1 Conversion
      If CheckItem(Item) Then                                '2 Validation
         cpList(CLng(Index)) = Item                          '3 Affectation de l'élément
      Else
         Err.Raise vbObjectError + 1, , "Valeur incorrecte à l'indice " & Index        'Erreur
      End If
   End If
End Property
```

Telle que ci-dessus, la propriété accepte des tableaux fixes ou dynamiques de tout les types numériques, des Variant contenant des tableaux, des tableaux de Variant. Cette solution est donc très souple du point de vue de l'application puisqu'elle permet par exemple l'utilisation de la fonction Array ou de parcourir la propriété avec For Each.

```
Sub Test_Propriete_Tableau_2()
   Dim MonObjet As NomClasse, TabLong() As Long, i As Long, Item As Variant
   Set MonObjet = New NomClasse
   MonObjet.List = Array(0, 1, 2, 3)            'Affectation d'un tableau à la propriété
   For i = 0 To 3
      MonObjet.List(i) = MonObjet.List(i) * 10  'Lecture/écriture d'un élément particulier
   Next
   For Each Item In MonObjet.List               'Parcours du tableau
      Debug.Print Item
   Next
   MsgBox UBound(MonObjet.List)                 'Affiche l'indice supérieur
   TabLong = MonObjet.List                      'Récupère le tableau stocké par la propriété
End Sub
```

Pour conclure, une fois votre propriété définie et quelque soit l'approche retenue, n'hésitez pas à prévoir des méthodes supplémentaires pour faciliter son usage. Des méthodes de tri ou de recherche sont en général les bienvenues. Enfin si aucune de ces approches ne répond à votre besoin une alternative consiste à implanter votre propriété sous la forme d'une collection.

IV-A-7 - Remarque

Le paragraphe précédent n'étant pas des plus digestes concluons cette section avec un petit hors sujet pour signaler que les procédures Property sont également utilisables dans les modules standard.

```
Private stdValue As Long

Property Get Value() As Long
   Value = stdValue
End Property

Property Let Value(NewValue As Long)
   If NewValue < 0 Then
```

```
        MsgBox "Valeur incorrecte.", vbCritical
    Else
        stdValue = NewValue
    End If
End Property
```

Cela permet par exemple de mettre en place un contrôle sur une variable globale sans avoir à le répéter à chaque utilisation de cette variable. On peut aussi déclencher une procédure particulière selon la valeur affectée à la variable qui devient alors réactive et ne se contente plus de stocker de l'information.

Il est également possible d'ajouter des propriétés aux objets existants tels que des UserForms afin par exemple de récupérer les choix de l'utilisateur dans la procédure appelante ou au contraire passer des paramètres à ce formulaire avant de l'afficher.

IV-B - Les méthodes

IV-B-1 - Définition

C'est une capacité dont est doté un objet, une action qu'il est capable de réaliser. Un objet Range est par exemple doté d'une méthode Clear qui supprime toutes les mises en formes de la cellule ainsi que son contenu.

IV-B-2 - Créer une méthode

Les méthodes d'un objet sont définies à l'aide de procédures Sub ou Function si la méthode doit renvoyer une valeur. Pour reprendre l'exemple de l'objet Range sa méthode ClearComments se contente d'effectuer des actions sans renvoyer de valeur, si nous devions écrire une telle méthode nous utiliserions une procédure Sub. Par contre la méthode SpecialCells qui permet d'extraire certaines cellules contenues dans un objet Range selon différents critères renvoie un autre objet Range, elle s'écrirait donc avec une procédure Function.

Contrairement aux procédures Property, les procédures Sub et Function acceptent le passage de paramètre par valeur ou par référence. Par défaut le passage par référence est utilisé, il est par nature plus rapide mais implique un risque pour les variables de l'application, et est donc à utiliser en connaissance de cause.

IV-B-3 - Contenu d'une méthode

Une classe est un monde à part, isolé du reste de votre application. En conséquence vous ne devriez jamais faire appel aux objets du monde extérieur, exception faite de ceux dont l'existence ne peut être remise en cause (Application, ThisWorkbook dans un projet Excel, ThisDocument dans un projet Word..) et de ceux que la méthode reçoit en paramètre. Les expressions de type ActiveCell, ActiveSheet ou Selection sont donc particulièrement malvenues dans la quasi-intégralité des cas.

La plupart des méthodes consisteront à manipuler les propriétés du module ou à effectuer des opérations basées sur ces propriétés. Voici l'exemple d'une méthode Reset qui se charge de réinitialiser deux propriétés nommées Value et Name.

```
Private cpValue As Double
Private cpName As String

Sub Reset()
    Value = 0
    Name = vbNullString
End Sub

Property Let Value(NewValue As Double)
    If NewValue >= 0 Then
        cpValue = NewValue                                    'Mise à jour
```

```
    Else
       Err.Raise vbObjectError + 1, , "Valeur négative interdite"  'Erreur
    End If
End Property

Property Let Name(NewName As String)
    Name = NewName
End Property
```

Et un exemple de procédure Function.

```
Public Enum DateStyle
    ds_USA
    ds_UK
    ds_Europe
End Enum

Function DateToString(AnyDate As Date, Style As DateStyle) As String
    Select Case Style
        Case ds_USA
            DateToString = Format(AnyDate, "YY/MM/DD")
        Case ds_UK
            DateToString = Format(AnyDate, "MM/DD/YY")
        Case ds_Europe
            DateToString = Format(AnyDate, "DD/MM/YY")
        Case Else
            Err.Raise 1004, , "Paramètre incorrect."
    End Select
End Function
```

Les procédure Sub et Function d'une classe sont en tout points identiques aux procédures que l'on trouve dans les modules standard, elles peuvent recevoir des arguments obligatoires ou optionnels, être privées ou publiques, ces caractéristiques sont déterminées par leur destination. Quant au choix d'écrire une méthode avec une procédure Sub ou Function, il n'est dicté que par la nécessité de renvoyer une valeur. N'ayant rien de plus à dire sur les méthodes, je profite de ce paragraphe pour faire une petite digression sur les énumérations.

La fonction ci-dessus prend pour paramètre une constante énumérée dont l'avantage principal est de permettre de documenter le code sans ajouter de commentaire, le nom de la constante étant suffisamment explicite. De plus lorsque l'on écrit l'appel de cette fonction l'éditeur propose automatiquement la liste des constantes disponibles, ce qui facilite l'écriture du code.

Les énumérations ne permettent cependant pas de contrôler la valeur effectivement passée à la fonction, elles sont toujours de type Long et si un appel à la fonction est effectuée avec une valeur de ce type n'apparaissant pas dans la liste cela ne produira ni erreur de compilation ni erreur d'exécution. En d'autres termes il s'agit seulement de suggestions pour l'utilisateur de la fonction et il est prudent de prévoir la réception d'un paramètre incorrect, soit en utilisant une des valeurs de la liste par défaut, soit en générant une erreur. Notez enfin que l'instruction Enum n'est disponible qu'à partir d'Office 2000.

IV-C - Les évènements

IV-C-1 - Définition

Un évènement est une réaction d'un objet à une action émanant soit d'un utilisateur soit de l'application qui contient l'objet. Chaque évènement est donc forcément lié à un objet qui subit une action et réagit en envoyant un message. La réception éventuelle de ce message est assurée par une procédure dite évènementielle. La définition d'évènements personnalisés est possible depuis Office 2000.

IV-C-2 - Déclaration

Un évènement se déclare au niveau module en utilisant le mot-clé Event. Voici la déclaration d'un évènement ValueChange doté d'un argument.

```
Event ValueChange(ByVal PreviousValue As Double)
```

Par nature les évènements sont toujours publics et ne renvoient jamais de valeurs, ils peuvent ou non recevoir des arguments mais ceux-ci sont alors obligatoires. Encore une fois, et toujours dans un souci d'étanchéité, il est préférable de passer les arguments par valeur plutôt que par référence afin que les variables du module ne puissent pas être modifiées par la procédure évènementielle de l'application.

IV-C-3 - Déclenchement

L'évènement ValueChange devant se déclencher à chaque fois que la propriété Value est modifiée il sera tout naturellement déclenché dans la procédure Property Let Value en utilisant l'instruction RaiseEvent suivie du nom de l'évènement et des arguments nécessaires.

```
Private cpValue As Double

Property Let Value(NewValue As Double)
   Dim Previous As Double
   If NewValue >= 0 Then
      Previous = cpValue                                    'Sauvegarde
      cpValue = NewValue                                    'Mise à jour
      RaiseEvent ValueChange(Previous)                      'Déclenchement
   Else
      Err.Raise vbObjectError + 1, , "Valeur négative interdite"  'Erreur
   End If
End Property
```

Le nombre d'arguments n'est absolument pas lié à la nature de l'évènement, ValueChange pourrait donc très bien recevoir des arguments qui n'ont rien à voir avec la propriété Value ou ne recevoir aucun argument.

IV-C-4 - Utilisation

Comme nous l'avons vu dans la première partie, une variable déclarée avec WithEvents permet d'accéder aux procédures évènementielles d'un objet.

```
Dim WithEvents MonObjet As NomClasse

Private Sub MonObjet_ValueChange(ByVal PreviousValue As Double)
   MsgBox PreviousValue
End Sub
```

Cette procédure sera déclenchée par toute instruction modifiant la propriété value de notre variable.

```
Sub TestEvent()
    Set MonObjet = New NomClasse
    MonObjet.Value = 1              'Appelle Let Value, déclenche MonObjet_ValueChange
End Sub
```

IV-C-5 - Annulation

Bien qu'un évènement ne retourne pas de valeur il est possible d'obtenir dans la procédure ayant déclenché l'évènement un retour d'information de la part de la procédure évènementielle en passant un argument par référence, ce procédé est utilisé par exemple sous Excel par la procédure Workbook_BeforeClose.

```
Event BeforeValueChange(ByVal NextValue As Double, ByRef Cancel As Boolean)
Event ValueChange(ByVal PreviousValue As Double)

Property Let Value(NewValue As Double)
   Dim CancelEvent As Boolean, Previous As Double
   If NewValue >= 0 Then
      'L'instance s'apprête à modifier sa propriété Value
      RaiseEvent BeforeValueChange(NewValue, CancelEvent)
      If CancelEvent Then
         'L'application a annulée la modification de la propriété
      Else
         Previous = cpValue                                    'Sauvegarde
         cpValue = NewValue                                    'Mise à jour
         RaiseEvent ValueChange(Previous)                      'Déclenchement
      End If
   Else
      Err.Raise vbObjectError + 1, , "Valeur négative interdite"     'Erreur
   End If
End Property
```

La variable CancelEvent est passée à la procédure évènementielle avec la valeur False, si au retour elle est égale à True cela signifie que le changement de valeur ne doit pas se produire.

Cette technique apporte de la souplesse en permettant à l'application de mettre en place des contrôles supplémentaires très facilement. Imaginons que la propriété Value représente une quantité d'articles à commander. La procédure Property Let se charge de vérifier que NewValue est supérieur à zéro puisque par définition on ne commande pas de quantités négatives ou nulles. Par contre le conditionnement d'un article n'est pas une information stable. Le fournisseur de l'article A peut imposer des commandes par multiples de 50 et celui de l'article B par multiples de 100, ces quantités sont de plus susceptibles d'évoluer. Il est donc préférable de laisser le soin à l'application de procéder à ces contrôles via une procédure évènementielle qui se chargera de récupérer le conditionnement d'un article en fonction de l'article et du fournisseur dans une table externe afin de vérifier que NewValue est une valeur correcte ou non, et ainsi décider de mettre à jour Value ou d'informer l'utilisateur.

IV-C-6 - Erreurs et évènements

Si le déclenchement d'un évènement vous semble être une opération anodine pensez que cela revient à donner le contrôle de l'exécution à une procédure évènementielle dont le contenu est par définition inconnu, puisque cette procédure n'est pas encore écrite. Considérons donc le module de classe suivant, composé d'une propriété et d'un évènement.

```
Private cpValue As Double
Private ValueHistory As Collection
Event ValueChange(ByVal NewValue As Double)

Private Sub Class_Initialize()
   Set ValueHistory = New Collection   'Initialisation
End Sub

Property Let Value(NewValue As Double)
   On Error Resume Next
   ValueHistory.Add cpValue            'Sauvegarde les valeurs successives
   RaiseEvent ValueChange(NewValue)    'Déclenche l'évènement
   cpValue = NewValue                  'Mise à jour de la propriété
End Property
```

L'appel à la propriété et la procédure évènementielle suivante:

```
Dim WithEvents MonObjet As NomClasse

Sub TestEvenement()
   Set MonObjet = New NomClasse
   MonObjet.Value = 1            'Appelle Let Value, déclenche MonObjet_ValueChange
End Sub

Private Sub MonObjet_ValueChange(ByVal NewValue As Double)
   Dim x As Double
   x = NewValue / 0
End Sub
```

Dans cet exemple l'erreur qui survient dans la procédure évènementielle va s'avérer fatale et provoquer l'arrêt du programme en dépit de l'instruction On Error Resume Next précédant le déclenchement de l'évènement. Ce qui ressemble à une exception au mécanisme classique de gestion des erreurs entre procédures appelantes et procédures appelées peut aussi s'interpréter non pas comme un appel de procédure dans le cadre de la pile en cours mais plutôt comme le début d'une nouvelle pile d'appel qui suspend l'exécution de la première.

L'un de mes relecteurs m'a fait remarquer que la procédure évènementielle devrait elle-même contenir un gestionnaire d'erreur ce qui permettrait alors au programme de se poursuivre normalement. C'est la le point essentiel de ce paragraphe, rien ne vous garantit que cette gestion d'erreur est suffisante ni même qu'elle soit présente, en particulier si vos classes sont destinées à être utilisés dans plusieurs projets et/ou par plusieurs personnes.

Le meilleur moyen de se prémunir d'une gestion d'erreur absente ou défaillante au niveau des procédures évènementielles des applications clientes reste donc de respecter la chronologie suivante dans toute procédure source d'évènements:
- Déclenchement des évènements pour lesquels on attend un éventuel retour de l'application (ex : annulation)
- Traitement interne au module
- Déclenchement des évènements pour lesquels on attend aucun retour de l'application

En respectant ce schéma, vous n'avez d'une part pas à vous soucier des éventuelles erreurs qui peuvent se produire à l'extérieur du module et dont vous n'êtes pas responsable, et d'autre part cela vous permet de procéder sereinement à toute opération (ouverture de fichiers, modifications de paramètres) susceptible de modifier l'environnement de l'application. En résumé n'interrompez jamais un traitement par un évènement.

IV-C-7 - Désactiver les évènements d'une classe

Il n'existe aucun mécanisme pour désactiver les évènements d'une classe, le seul moyen de permettre cette fonctionnalité sera donc de prévoir une propriété à cet effet.

```
Private cpEnableEvents As Boolean
Event ValueChange(ByVal PreviousValue As Double)

Property Get EnableEvents() As Boolean
   EnableEvents = cpEnableEvents                    'Renvoi la valeur actuelle
End Property

Property Let EnableEvents(NewEnableEvents As Boolean)
   cpEnableEvents = NewEnableEvents                 'Mise à jour de la valeur
End Property
```

Chaque instruction RaiseEvent du module de classe doit être précédée d'une vérification de la propriété.

```
Property Let Value(NewValue As Double)
   Dim Previous As Double
```

```
    If NewValue >= 0 Then
        Previous = cpValue                                          'Sauvegarde
        cpValue = NewValue                                          'Mise à jour
        'Vérifie si l'évènement doit être déclenché
        If EnableEvents Then RaiseEvent ValueChange(Previous)
    Else
        Err.Raise vbObjectError + 1, , "Valeur négative interdite"  'Erreur
    End If
End Property
```

L'initialisation de la propriété se fait dans la procédure Class_Initialize.

```
Private Sub Class_Initialize()
    cpEnableEvents = True
End Sub
```

L'efficacité de cette approche reste toutefois limitée puisqu'elle ne permet pas de désactiver globalement les évènements pour toutes les instances d'une classe.

IV-D - Les collections

IV-D-1 - Introduction

En regardant une bibliothèque on retrouve souvent des classes fonctionnant par couple, les collections et les objets fonctionnels qu'elles regroupent. Pour ne citer qu'un exemple dans la bibliothèque Office la classe CommandBars (les barres d'outils) représente un ensemble d'objets CommandBar (une barre d'outils). Reproduire cette approche permet de proposer au développeur un package plus complet incluant des services supplémentaires comme par exemple des variables partagées entre chaque instance du module fonctionnel ou une détection des évènements au niveau de la collection.

Pour illustrer les avantages de cette approche nous allons donc définir deux nouvelles classes, la première représentera une collection et s'appellera Numbers, la seconde représentera un objet fonctionnel nommé Number servant à stocker un nombre positif et capable de générer des évènements, c'est assez basique mais ce sera suffisant pour notre propos et cela me permet surtout de poursuivre avec les procédures déjà évoquées. Dans une approche simple nous définirions ce module comme ceci.

```
'Le module Number
Private cpValue As Double
Private cpEnableEvents As Boolean

Event BeforeValueChange(ByVal NextValue As Double, ByRef Cancel As Boolean)
Event ValueChange(ByVal PreviousValue As Double)

Private Sub Class_Initialize()
    cpEnableEvents = True
End Sub

Property Get EnableEvents() As Boolean
    EnableEvents = cpEnableEvents
End Property

Property Let EnableEvents(NewEnableEvents As Boolean)
    cpEnableEvents = NewEnableEvents
End Property

Property Get Value() As Double
    Value = cpValue
End Property

Property Let Value(NewValue As Double)
```

```
    Dim CancelEvent As Boolean, Previous As Double
    If NewValue >= 0 Then
        If EnableEvents Then
            RaiseEvent BeforeValueChange(NewValue, CancelEvent)
        End If
        If Not CancelEvent Then
            Previous = cpValue                                      'Sauvegarde
            cpValue = NewValue                                      'Mise à jour
            If EnableEvents Then RaiseEvent ValueChange(Previous)
        End If
    Else
        Err.Raise vbObjectError + 1, , "Valeur négative interdite"  'Erreur
    End If
End Property
```

Pour une approche plus élaborée, nous ajouterons à cet objet deux nouvelles propriétés:

- Name: Une chaîne permettant d'identifier chaque instance de manière univoque.
- Parent: Cette propriété renverra un objet Numbers

```
'Dans le module Number
Private cpName As String
Private cpParent As Numbers

Property Get Name() As String
    Name = cpName
End Property

Property Let Name(NewName As String)
    cpName = NewName
End Property

Property Get Parent() As Numbers
    Set Parent = cpParent
End Property

Property Set Parent(NewParent As Numbers)
    If cpParent Is Nothing Then Set cpParent = NewParent
End Property
```

La propriété Name n'incluse à ce stade aucun contrôle visant à garantir qu'aucune instance ne porte le nom d'une autre, nous y reviendrons. Quant à la propriété Parent, il s'agit d'une propriété en "écriture unique", autrement dit on ne pourra la définir qu'une fois (au moment de la création de l'objet), cela semble logique si l'on admet qu'on n'a qu'une mère et qu'on ne peut en changer.

IV-D-2 - Adapter l'objet Collection

Les collections pouvant stocker simultanément plusieurs types d'objets, notre première tache sera de nous assurer que notre objet Numbers ne puisse contenir que des objets Number. Pour ce faire nous allons emballer une collection dans ce module, c'est-à-dire court-circuiter les méthodes et propriétés d'une collection pour y substituer nos propres méthodes et propriétés.

```
'Dans le module Numbers
Private cpNumbers As Collection                    'Déclare une collection

Private Sub Class_Initialize()
    Set cpNumbers = New Collection                 'Initialise la collection
End Sub

Function Count() As Long                           'Nombre d'éléments
    Count = cpNumbers.Count
End Function
```

```
Function Item(IndexOrName As Variant) As Number     'Renvoi un élément
   Set Item = cpNumbers.Item(IndexOrName)
End Function

Sub Remove(IndexOrName As Variant)                  'Retire un élément
   cpNumbers.Remove IndexOrName
End Sub
```

Count n'appelle guère de commentaires, Item et Remove acceptent un argument Variant afin que l'application puisse utiliser soit le nom de l'objet soit sa position dans la collection.

IV-D-3 - La méthode Add

Notre collection étant privée, la méthode Add sera l'unique point d'entrée permettant à l'application d'y ajouter de nouveaux objets. C'est donc cette méthode, et non pas l'application, qui se chargera de déterminer le nom initial de l'objet en prenant soin de s'assurer qu'il est unique afin que sa propriété Name puisse servir de clé d'identification. Cette méthode doit donc accomplir deux taches distinctes mais intimement liées, d'une part créer et initialiser un nouvel objet, et d'autre part ajouter cet objet à la collection.

```
'Dans le module Numbers
Function Add() As Number                            'Ajoute un nouvel élément à la collection
   Dim TheNumber As Number, TheName As String
   Static TheKey As Long
   On Error Resume Next
   Do                                               'Détermine un nom unique
      TheKey = TheKey + 1
      TheName = "Number" & CStr(TheKey)
   Loop Until cpNumbers.Item(TheName) Is Nothing
   On Error GoTo 0
   Set TheNumber = New Number                       'Crée un nouvel objet
   With TheNumber                                   'Initialise l'objet
      Set .Parent = Me
      .Name = TheName
   End With
   cpNumbers.Add TheNumber, TheName                 'Ajoute l'objet, utilise le nom comme clé
   Set Add = TheNumber                              'Renvoi l'objet à l'application
End Function
```

L'initialisation du nouvel objet se fait tout naturellement en appelant ses diverses propriétés. Cette construction permet si on le souhaite de transformer Add en constructeur personnalisé, il suffit pour cela de lui ajouter des paramètres (déclarés ici optionnels pour offrir plus de souplesse à l'application) que l'on retransmet ensuite aux propriétés correspondantes de l'objet.

```
'Dans le module Numbers
Function Add(Optional IValue As Double) As Number
   Dim TheNumber As Number, TheName As String
...
   On Error GoTo ErrH
   Set TheNumber = New Number                   'Crée un nouvel objet
   With TheNumber                               'Initialise l'objet
      Set .Parent = Me
      .Name = TheName
      .Value = IValue                           'Paramètre fourni par l'application
      .EnableEvents = True
   End With
   cpNumbers.Add TheNumber, TheName             'Ajoute l'objet, utilise le nom comme clé
   Set Add = TheNumber                          'Renvoi l'objet à l'application
   Exit Function
ErrH:
   Err.Raise Err.Number, , Err.Description      'Erreur
End Function
```

Notez l'apparition d'un gestionnaire d'erreur dans la procédure Add. Si aucune erreur ne se produit pendant l'initialisation de l'objet cela signifie que les paramètres transmis par l'application (ici IValue) sont conformes et Add renvoi le nouvel objet à l'application, dans le cas contraire la méthode Add générera une erreur d'exécution à destination de l'application en fournissant le message d'erreur défini par la propriété source de l'erreur.

Dernière remarque, il est en général pertinent d'empêcher le déclenchement d'évènements tant que l'initialisation de l'objet n'est pas achevée. Dans notre cas ce résultat est obtenu en supprimant la procédure Class_Initialize de l'objet Number afin que la propriété EnableEvents soit la dernière (et non la première) à être initialisée.

A ce stade notre collection présente donc les caractéristiques suivantes:

- Elle se comporte de fait comme si elle était typée, en ne contenant que des objets Number.
- Chaque élément possède une propriété Parent renvoyant l'objet Numbers contenant la collection.
- Chaque élément possède une propriété Name dont la valeur correspond à sa clé.

IV-D-4 - La propriété Name

Cette propriété pose problème car elle doit absolument être synchronisée avec la clé d'identification de l'objet dans la collection faute de quoi celui-ci ne pourrait plus être référencé que par son index. A priori il est donc impossible de la modifier puisqu'il est impossible de modifier (et même de lire) la clé d'un élément d'une collection. La solution consiste à retirer provisoirement l'objet de la collection lorsque on modifie son nom, avant de l'y ajouter à nouveau, une coopération entre l'objet Number et son parent Numbers est donc nécessaire.

```
'Dans le module Number
Property Let Name(NewName As String)
   If cpName = vbNullString Then          'Vrai uniquement à l'initialisation
      cpName = NewName
   Else
      If NewName = vbNullString Then
         Err.Raise vbObjectError + 100, , "Impossible d'utiliser une chaine vide." 'Erreur
      End If
      If Parent.IsFreeName(Me, NewName) = True Then
         cpName = NewName                 'Le nom est accepté
      Else
         Err.Raise vbObjectError + 100, , "Impossible d'utiliser un nom existant." 'Erreur
      End If
   End If
End Property
```

Avant de modifier le nom on sollicite une fonction de l'objet parent afin de s'assurer de sa disponibilité.

```
'Dans le module Numbers
Function IsFreeName(TheNumber As Number, NewName As String) As Boolean
   'Vrai si le nom peut être modifié
   Dim OneNumber As Number, i As Long
   With cpNumbers
      On Error Resume Next
      Set OneNumber = .Item(NewName)
      On Error GoTo 0
      If OneNumber Is Nothing Then                    'Le nom est disponible
         i = TheNumber.Index                          'Mémorise la position
         .Remove i                                    'Retire l'objet de la collection
         If i > .Count Then                           'Insere l'objet avec sa nouvelle clé
            .Add TheNumber, NewName
         Else
            .Add TheNumber, NewName, i
         End If
         IsFreeName = True
      Else
         If OneNumber Is TheNumber Then IsFreeName = True   'Pas de changement
      End If
```

```
    End With
End Function
```

Ici la seule vérification porte sur l'unicité du nom mais on pourrait également implanter des règles relatives à sa longueur ou à sa composition (en fait ce serait même indispensable afin de s'assurer que le nom peut être utilisé comme clé). La méthode Index renvoyant la position de l'élément dans la collection est abordée un peu plus loin.

IV-D-5 - Propriétés communes à chaque instance

Via la propriété Parent, chaque objet Number est en mesure d'accéder à un membre public de l'objet Numbers, qui peut donc être vu comme un espace de stockage d'information partagé entre tous les objets Number faisant partie de sa collection. Illustrons cette capacité en revenant sur la propriété EnableEvents, qui permet à l'application de désactiver les évènements de chaque objet Number en effectuant une boucle sur la collection de l'objet Numbers.

```
Dim MyNumbers As Numbers, i As Long
...
For i = 1 To MyNumbers.Count
    MyNumbers.Item(i).EnableEvents = False
Next
```

Bien évidemment pour rétablir cette propriété il faudra à nouveau parcourir la collection, cela peut être évité en déplaçant cette propriété dans l'objet Numbers.

```
'Dans le module Numbers
Private cpEnableEvents As Boolean

Property Get EnableEvents() As Boolean
    EnableEvents = cpEnableEvents
End Property

Property Let EnableEvents(NewEnableEvents As Boolean)
    cpEnableEvents = NewEnableEvents
End Property
```

```
'Dans le module Number
Private Property Get EnableEvents() As Boolean
    EnableEvents = Parent.EnableEvents
End Property
```

Dans le module Number, la procédure Property Let EnableEvents et la variable cpEnableEvents doivent être supprimées. Cette propriété n'ayant plus vocation à être accessible à l'application, elle devient privée et plutôt que de renvoyer une variable interne appelle la propriété de l'objet Parent. Notez qu'il serait également possible d'en faire une fonction privée, le résultat serait le même. Enfin il faut modifier la méthode Add de l'objet Numbers pour y désactiver les évènements avant de créer l'objet.

```
'Dans le module Numbers
Function Add(Optional IValue As Double) As Number
    Dim TheNumber As Number, TheName As String, EE As Boolean
...
    EE = EnableEvents                       'Sauvegarde la valeur courante
    EnableEvents = False                    'Désactive les évènements
    On Error GoTo ErrH
    Set TheNumber = New Number              'Crée un nouvel objet
    With TheNumber                          'Initialise l'objet
        Set .Parent = Me
        .Name = TheName
        .Value = IValue                     'Paramètre fourni par l'application
    End With
    cpNumbers.Add TheNumber, TheName        'Ajoute l'objet, utilise le nom comme clé
```

```
    EnableEvents = EE                           'Rétabli la gestion des évènements
    Set Add = TheNumber                         'Renvoi l'objet à l'application
    Exit Function
ErrH:
    EnableEvents = EE                           'Rétabli la gestion des évènements
    Err.Raise Err.Number, , Err.Description     'Erreur
End Function
```

Désormais, à chaque modification de la propriété Value d'un objet Number, les évènements BeforeValueChange et ValueChange ne seront déclenchés qu'après avoir vérifié la propriété EnableEvents de l'objet Numbers. L'application peut donc désactiver les évènements de toutes les instances existantes dans la collection sans recourir à une boucle.

```
MyNumbers.EnableEvents = False
```

Nous avons donc gagné en espace mémoire, une seule variable étant utilisée par un nombre quelconque d'objet, mais surtout en simplicité d'utilisation. Enfin, cela nous assure que tous les objets Number se comporteront à l'identique à un instant donné.

IV-D-6 - Gérer les évènements au niveau de la collection

Il est également possible de faire appel à l'objet Parent lorsqu'on déclenche un évènement, ce qui permet à l'application de les gérer au niveau de la collection. Tout d'abord nous allons dupliquer dans le module Numbers les différents évènements de l'objet Number que nous souhaitons faire remonter.

```
'Dans le module Numbers
Event NumberValueChange(ByVal Nb As Number, ByVal PreviousValue As Double)
Event NumberBeforeValueChange(ByVal Nb As Number, ByVal NextValue As Double, Cancel As Boolean)
```

Ces évènements de l'objet Numbers sont les même que ceux de l'objet Number, avec un paramètre supplémentaire servant à identifier l'objet ayant déclenché l'évènement. Nous avons également besoin d'une méthode publique qui sera appelée par l'objet Number pour informer Numbers qu'un évènement doit être déclenché. Cette méthode aura besoin de deux paramètres, l'un permettant de déterminer quel évènement doit être déclenché et l'autre d'identifier l'objet Number à la source de l'évènement. Le premier paramètre sera choisi dans une liste de constantes.

```
Public Enum EventNames
    NbValueChange = 1
    NbBeforeValueChange = 2
End Enum
```

En plus de cela, certains évènements devant pouvoir être annulés, un paramètre supplémentaire est nécessaire, notre méthode devrait donc ressembler à ceci.

```
Sub RaiseAnEvent(EventName As EventNames, Source As Number, Cancel As Boolean)
```

Cette déclaration est incomplète, en effet les paramètres des évènements (NextValue et PreviousValue) n'y figurent pas, il sera donc impossible de les retransmettre à l'évènement que nous voulons déclencher. Ceci est volontaire car tous nos évènements n'ont pas forcément le même nombre de paramètres. Un moyen simple de gérer ces différents cas consiste à utiliser comme dernier argument un tableau déclaré avec le mot-clé ParamArray.

```
Sub RaiseAnEvent(EventName As EventNames, Source As Number, Cancel As Boolean, ParamArray Params())
```

Ce dernier paramètre est un tableau de type Variant dont la dimension n'est pas connue à l'avance, il est donc très souple. Seule restriction, lors de l'appel de cette méthode il sera impossible d'utiliser des arguments nommés, ils

seront donc obligatoirement passés par position. Nous pouvons maintenant modifier l'objet Number afin de faire remonter les différents évènements vers la collection.

```
'Dans le module Number
Property Let Value(NewValue As Double)
   Dim CancelEvent As Boolean, Previous As Double
   If NewValue >= 0 Then
      If EnableEvents Then
         'Déclenche l'évènement au niveau de l'instance, puis de la collection
         RaiseEvent BeforeValueChange(NewValue, CancelEvent)
         Parent.RaiseAnEvent NbBeforeValueChange, Me, CancelEvent, NewValue
      End If
      If Not CancelEvent Then
         Previous = cpValue                                    'Sauvegarde
         cpValue = NewValue                                    'Mise à jour
         If EnableEvents Then
            'Déclenche l'évènement au niveau de l'instance, puis de la collection
            RaiseEvent ValueChange(Previous)
            Parent.RaiseAnEvent NbValueChange, Me, False, Previous
         End If
      End If
   Else
      Err.Raise vbObjectError + 1, , "Valeur négative interdite"  'Erreur
   End If
End Property
```

Reste enfin à compléter la procédure RaiseAnEvent qui ne fera que retransmettre les paramètres qu'elle reçoit en respectant un ordre plus ou moins conventionnel, l'objet source d'abord, les différents paramètres ensuite, et enfin un éventuel paramètre d'annulation.

```
'Dans le module Numbers
Sub RaiseAnEvent(EventName As EventNames, Source As Number, Cancel As Boolean, ParamArray Params())
   Select Case EventName
      Case NbValueChange
         'Le paramètre Cancel est ignoré pour cet évènement
         RaiseEvent NumberValueChange(Source, CDbl(Params(0)))
      Case NbBeforeValueChange
         RaiseEvent NumberBeforeValueChange(Source, CDbl(Params(0)), Cancel)
      'Case un autre évènement
   End Select
End Sub
```

Concernant l'évènement BeforeValuechange la valeur finale de l'argument Cancel sera celle définie dans la procédure évènementielle de niveau collection puisqu'elle s'exécute après celle de niveau élément.

IV-D-7 - Itération avec For Each

La collection étant privée il est normalement impossible de la parcourir en dehors du module Numbers en utilisant l'instruction For Each. Il est toutefois possible de contourner cette limitation à l'aide d'une propriété particulière.

```
'Dans le module Numbers
Property Get NewEnum() As IUnknown
   Set NewEnum = clsNumbers.[_NewEnum]
End Property
```

_NewEnum est une propriété masquée de l'objet Collection et IUnknown est une classe de la bibliothèque stdole qui fait partie des bibliothèques par défaut de tout projet Office. Le but de cette propriété est de faire passer le module pour une collection au regard du compilateur, qui l'appellera à chaque fois que l'application utilisera For Each avec une variable de type Numbers. Les modifications suivantes ne peuvent être effectuées dans l'éditeur VBE, il faut

donc exporter le module et ouvrir le fichier .cls avec un éditeur de texte tel que Notepad pour ajouter des attributs à cette propriété.

```
'Dans le fichier Numbers.cls
Property Get NewEnum() As IUnknown
Attribute NewEnum.VB_UserMemId = -4
Attribute NewEnum.VB_MemberFlags = "40"
   Set NewEnum = cpNumbers.[_NewEnum]
End Property
```

Illustrons cette capacité en ajoutant une méthode Index à l'objet Number pour renvoyer sa position dans la collection.

```
'Dans le module Number
Function Index() As Long                'Position de l'élément dans la collection
   Dim OneNumber As Number, i As Long
   For Each OneNumber In Parent
      i = i + 1
      If OneNumber Is Me Then Exit For
   Next
   Index = i
End Function
```

Le parcours de la collection d'un objet Numbers se fait simplement en désignant cet objet, ici via la propriété Parent. En plus d'un confort pour le développeur cette possibilité apporte également un gain de performance en évitant de passer par la méthode Item de l'objet Numbers. Seul inconvénient la propriété NewEnum doit impérativement être déclarée Public et donc être visible depuis l'application.

Dans la même veine on peut aussi définir une propriété par défaut, même si cela n'apporte pas grand-chose, ici le choix se porte sur Item.

```
Function Item(IndexOrName As Variant) As Number 'Renvoi un élément
Attribute Item.VB_UserMemId = 0
   Set Item = cpNumbers.Item(IndexOrName)
End Function
```

Les lignes Attribute s'inscrivent immédiatement en dessous du nom de la procédure. Enregistrez le fichier .cls puis réimportez le dans votre projet, les lignes Attribute doivent être invisibles dans l'éditeur VBE. Enfin, vous ne devriez procéder à ces manipulations qu'après avoir finalisé vos modules de classes, les propriétés Attribute ayant parfois tendances à se perdre lorsqu'on modifie les procédures.

J'ai découvert cette astuce bien utile en lisant cette **discussion**, que je vous recommande d'ailleurs.

IV-D-8 - Objets orphelins

De par leur conception Number et Numbers sont faits pour fonctionner ensemble et non pas l'un sans l'autre, or le fonctionnement de ce couple présente un défaut, la possibilité pour l'application d'obtenir des objets Number orphelins, c'est-à-dire qui ne sont pas référencés dans la collection de leur objet Parent ou qui n'ont pas d'objet Parent. Ceci peut se produire par exemple si l'application crée un objet Number sans passer par la méthode Add de l'objet Numbers.

```
'Dans un module standard
Dim MyNumber As Number

Set MyNumber = New Number
MyNumber.Value = 3.141592653  'Cause une erreur d'exécution inattendue
```

Parent n'ayant pas été défini la manipulation de l'objet sera inévitablement source d'erreur nombre de ses méthodes ou propriétés essayant vainement d'y faire appel. L'objet ainsi créé est de fait quasi inutilisable. Autre hypothèse désagréable, l'application retire un objet Number de la collection alors qu'une variable désigne cet objet.

```
'Dans un module standard
Dim MyNumber As Number, MyNumbers As Numbers

Set MyNumber = MyNumbers.Item(1)
MyNumbers.Remove 1
MsgBox MyNumber.Index   'Renvoi Count mais MyNumber n'appartient plus à la collection
```

Dans ce cas de figure le comportement va devenir erratique, certaines propriétés ou méthodes échoueront mais d'autres renverront des informations erronées, par exemple un appel à la méthode Index renverra le nombre d'objets dans la collection de MyNumbers ce qui peut laisser croire que MyNumber figure toujours dans la collection, en dernière position. Pour éviter une telle erreur de logique qui serait difficile à cerner, mieux vaut faire en sorte de définir Parent à Nothing lorsque on appelle Remove afin qu'une erreur d'exécution se produise lors de l'appel à Index.

```
'Dans le module Numbers
Sub Remove(IndexOrName As Variant)
   Set Item(IndexOrName).Parent = Nothing    'Brise le lien ascendant
   cpNumbers.Remove IndexOrName              'Brise le lien descendant
End Sub
```

Ceci nous oblige également à modifier la procédure Set Parent de l'objet Number afin que Remove produise l'effet attendu.

```
'Dans le module Number
Property Set Parent(NewParent As Numbers)
   If (cpParent Is Nothing) Or (NewParent Is Nothing) Then Set cpParent = NewParent
End Property
```

On voit ici que cette propriété n'a vocation à être appelée qu'à deux moments bien précis, soit lors de l'ajout de l'objet à la collection soit lorsque on le retire de cette collection. Autrement dit, et dans l'idéal, elle devrait être en lecture seule pour l'application tout en restant accessible par l'objet Numbers. Bien que ces deux exigences soient contradictoires nous verrons plus bas comment les concilier.

Pour faire un parallèle avec l'objet Worksheet (une feuille de calcul Excel), il ne peut être créé qu'en passant par la méthode Add de la collection Worksheets, et retirer cet objet de la collection implique de le détruire affectant du même coup toutes les variables le désignant. Ceci est impossible à reproduire, et si en théorie la méthode Remove devrait pouvoir servir à détruire l'objet qu'elle retire de la collection, dans la pratique elle ne fait que détruire une référence à l'objet sans affecter les autres références éventuelles.

Etant donné qu'il est impossible d'empêcher l'application de produire des objets orphelins le mieux que l'on puisse faire est donc de préserver la cohérence de la relation entre deux objets Number et Numbers, soit ils se connaissent mutuellement, soit ils s'ignorent complètement, mais en aucun cas l'un ne doit connaître l'autre à son insu.

IV-D-9 - L'évènement Terminate

Je vous disais au début de cet article que l'évènement Terminate d'un objet était déclenché lorsque aucune variable ne désignait plus cet objet, cette condition est en fait nécessaire mais insuffisante. Si un objet A pointe vers un objet B et que cet objet B pointe vers l'objet A, alors et même si aucune variable ne désigne A ou B, l'évènement Terminate de ces deux objets ne se déclenche jamais et ils demeurent en mémoire, tout en restant inaccessibles.

Ce problème peut se produire avec le modèle que je viens de vous décrire. En effet chaque objet Number pointe vers un objet Numbers via sa propriété Parent, pendant que cet objet Numbers pointe vers chaque objet Number via sa collection, il y a donc une référence circulaire entre ces objets. Pour vous en convaincre voici une petite démonstration.

```
'Dans le module Numbers
Private Sub Class_Terminate()
   MsgBox "Numbers.Terminate"
End Sub
```

```
'Dans le module Number
Private Sub Class_Terminate()
   MsgBox "Number.Terminate " & Me.Name
End Sub
```

```
'Dans un module standard
Sub Test_Terminate_1()
   Dim i As Long, MyNumbers As Numbers
   Set MyNumbers = New Numbers          'Crée un objet Numbers
   With MyNumbers
      For i = 1 To 10                   'Crée 10 objets Number
         .Add
      Next
   End With
   Set MyNumbers = Nothing              'Les objets ne sont pas détruits
End Sub
```

Lorsque cette procédure s'achève, les objets créés (une instance de Numbers et dix instances de Number) ne sont pas accessibles (aucune variable ne pointe vers l'un de ces objets), et pourtant aucun des évènements Terminate ne s'est produit, conclusion logique la mémoire n'a pas été libérée. Pour éviter ce genre de situation il est donc prudent de briser ces références circulaires en retirant chaque objet Number de la collection de son parent lorsque on veut détruire celui-ci.

```
'Dans le module Numbers
Sub Clear()                             'Retire tous les éléments de la collection
   Dim OneNumber As Number
   For Each OneNumber In cpNumbers
      Set OneNumber.Parent = Nothing    'Brise les liens ascendants
   Next
   Set clsNumbers = New Collection      'Brise les liens descendants
End Sub
```

```
'Dans un module standard
Sub Test_Terminate_2()
   Dim i As Long, MyNumbers As Numbers
   Set MyNumbers = New Numbers          'Crée un objet Numbers
   With MyNumbers
      For i = 1 To 10                   'Crée 10 objets Number
         .Add
      Next
      .Clear                            'Vide la collection
   End With
   Set MyNumbers = Nothing              'Les objets sont détruits
End Sub
```

Bien évidemment l'appel à Clear n'a rien d'automatique et ne garantit pas qu'il ne subsistera aucune trace de ces objets. Par exemple si une variable de niveau module pointait vers un élément de la collection au moment de cet appel, cet élément continuera d'exister en tant qu'objet orphelin. Il appartient donc à l'application de gérer rigoureusement ses variables et au concepteur des modules de classes d'utiliser l'évènement Terminate avec précaution.

IV-D-10 - Résumé

Au final, si le montage peut sembler complexe, il reste néanmoins très avantageux du point de vue de l'application puisque en déclarant une seule variable de type Numbers le développeur:

- Dispose d'un nombre quelconque d'instances de l'objet Number, via la collection
- Est assuré que chaque objet porte un nom unique
- Peut librement modifier ce nom
- Peut accéder à un objet Number par son nom ou sa position
- Est en mesure de parcourir la collection en utilisant l'instruction For Each
- Dispose de deux niveaux de procédures évènementielles
- Peut simplement désactiver ces évènements

Ce modèle basique illustrant la relation entre un module fonctionnel (Number) et un module faisant office de collection peut ensuite facilement être enrichi d'autres propriétés, méthodes ou évènements usuels, par exemple:

Pour l'objet Number:
- Tag, une propriété pour stocker des informations variées

Pour l'objet Numbers:
- Sort, une méthode pour trier la collection
- Find, une méthode pour rechercher un (ou plusieurs) éléments
- NewNumber, un évènement signalant l'ajout d'un élément à la collection

IV-E - Les bibliothèques

Une bibliothèque est un ensemble de code dont la seule finalité est de se mettre au service d'une application, autrement dit il s'agit d'un fichier contenant des modules et/ou des fonctions à caractères génériques. Concrètement il s'agira d'un fichier externe à l'application mais de même type que celle-ci (une macro complémentaire .xla pour une application Excel, une base de donnée .mde pour une application Access..).

IV-E-1 - Création

Modifions tout d'abord le nom du projet hébergeant nos modules afin de faciliter son identification: VBE > Outils > Propriétés > Onglet Général, remplacez le nom par défaut (VBAProject) par un nom plus significatif.

Affichons maintenant la fenêtre propriétés (F4) pour revenir, comme promis au tout début de cet article sur la propriété Instancing des modules de classes. Jusqu'à présent nos modules étaient Private, et donc visibles uniquement dans leur projet de résidence, en modifiant cette propriété à PublicNotCreatable nos deux modules deviennent accessibles depuis l'extérieur (Public), cependant la création d'une nouvelle instance n'est possible qu'à l'intérieur du projet (NotCreatable). Pour rendre ces modules disponibles nous n'avons pas d'autre choix que d'écrire dans un module standard une fonction publique renvoyant une nouvelle instance.

```
Public Function NewNumbers() As Numbers
   Set NewNumbers = New Numbers
End Function
```

Notre bibliothèque ne proposera pas de fonction équivalente pour l'objet Number, ainsi l'application devra d'abord passer par la fonction NewNumbers pour créer un objet Numbers, puis par la méthode Add de cet objet pour créer les objets Number dont elle à besoin, ce qui permet au passage de diminuer le risque de création d'objets orphelins par l'application. Ci-dessus cette fonction ne sert qu'à créer une nouvelle instance, mais on pourrait en profiter pour lui assigner le rôle de constructeur de l'objet Numbers.

```
Public Function NewNumbers(Name As String) As Numbers
   Dim N As Numbers
   Set N = New Numbers
   N.Name = Name         'En supposant qu'une propriété Name existe pour Numbers
   Set NewNumbers = N    'Renvoie le nouvel objet à l'application
End Function
```

Il ne reste plus qu'à enregistrer le fichier dans le format convenable.

IV-E-2 - Avantages

L'avantage le plus évident à utiliser des bibliothèques est que le code est disponible pour plusieurs applications. D'autre part, le fait de n'avoir qu'un seul fichier simplifie la maintenance, la mise à jour se résumant à remplacer ce fichier.

Autre avantage induit, plus spécifique aux classes, il nous est désormais possible de masquer aux applications clientes les procédures destinées à faire coopérer nos modules entre eux afin de nous assurer qu'elles ne seront pas appelées en dehors de leur contexte normal d'utilisation. Il suffit pour cela de modifier leurs déclarations en utilisant le mot-clé Friend qui restreint la visibilité d'une procédure à son projet de résidence, définissant ainsi une portée intermédiaire entre Public et Private.

```
'Dans le module Numbers
Friend Function IsFreeName(...) As Boolean

Friend Sub RaiseAnEvent(...)

'Dans le module Number
Friend Property Set Parent(NewParent As Numbers)
```

En procédant ainsi la propriété Parent devient de fait une propriété en lecture seule au regard de l'application tout en restant disponible pour le module Numbers, ce qui accroît la robustesse de l'ensemble.

IV-E-3 - Evolution

En termes d'évolution il convient de distinguer les éléments privés, librement modifiables, et les éléments publics qui forment l'interface de la bibliothèque. Si rien n'empêche d'ajouter de nouveaux membres publics (méthodes ou propriétés), la modification de la définition d'un membre public (ou sa suppression) représente un risque majeur d'incompatibilité avec le code des applications existantes.

Quelques modifications sont toutefois possibles, par exemple ajouter un argument à une procédure à condition de l'ajouter après les arguments existants et de le définir optionnel. Modifier le type d'un paramètre d'une procédure Sub ou Function est envisageable si vous optez pour un type moins restrictif (de Long à Double par exemple) et à condition qu'il soit passé par valeur, l'inverse est beaucoup plus risqué. Dans l'autre sens la valeur de retour d'une fonction peut passer de Double à Long sans perturber le fonctionnement des applications clientes, l'inverse est susceptible de provoquer un dépassement de capacité.

En ce qui concerne les propriétés, si l'on opte pour un type moins restrictif (Long vers Double) on a un risque d'erreur en lecture, et à l'inverse un type plus restrictif (Double vers Long) engendre un risque d'erreur en écriture. Il est donc fondamental de bien choisir vos types de données au départ.

Le contenu des procédures, même publique, peut par contre évoluer à votre gré, à condition de ne pas les dénaturer. Une procédure de tri définie initialement comme effectuant un tri croissant ne doit pas être modifiée pour effectuer un tri décroissant, mais la manière dont vous effectuez le tri n'a aucune importance du point de vue de l'application.

IV-E-4 - Utilisation

Pour qu'une application puisse utiliser les classes et fonctions isolées dans une bibliothèque il faut lui ajouter une référence vers cette bibliothèque. VBE > Outils > Références cochez la case correspondante au nom de projet choisi plus haut (le fichier en question doit bien sur être ouvert, sinon passez par le bouton Parcourir).

Toute médaille ayant son revers, vos applications ne sont plus auto suffisantes, et si la bibliothèque est déplacée ou renommée vous aurez une référence manquante et des erreurs de compilation 'Type défini par l'utilisateur non défini' les rendront inutilisables.

Enfin, si l'application doit être distribuée à plusieurs utilisateurs vous devrez également distribuer la bibliothèque et prévoir son référencement. Je n'aborde pas ici cette question, les solutions à mettre en □uvre étant largement dépendantes du logiciel (Excel, Access, Word) cible.

IV-F - Programmation orientée objets ?

VBA n'est en aucun cas un langage orienté objet comme peut l'être Java, il permet néanmoins de mettre en □uvre certains principes de la POO soit en intégralité comme l'encapsulation des données avec les procédures Property, soit seulement en partie comme le polymorphisme ou l'héritage.

IV-F-1 - Héritage et polymorphisme

L'héritage est un concept qui veut qu'une classe puisse utiliser des propriétés et méthodes définies par une autre classe. Par exemple les classes Label et CheckBox héritent des méthodes et propriétés définies dans la classe Control. Cela permet de connaître la position d'un Label ou d'un CheckBox sur un formulaire grâce aux propriétés Top et Left bien que la définition de ces propriétés ne fasse pas partie intégrante de ces classes. On pourra qualifier les classes CheckBox et Label de classes dérivées de la classe Control.

Le polymorphisme peut être considéré comme l'autre face de l'héritage, la capacité d'une classe à se comporter comme n'importe laquelle de ses classes dérivées. L'objet Control peut ainsi servir à définir la bordure du Label qu'il représente grâce à la propriété BorderStyle de celui-ci, mais aussi à définir l'état d'un CheckBox grâce à la propriété Value de celui-ci. L'objet Control est donc un objet polymorphe, mais aussi incomplet par nature, puisque il ne peut exister sans un objet (Label, CheckBox etc.) sous-jacent.

Pour illustrer ces concepts nous allons créer trois modules de classes nommés Polygone, Rectangle et Cercle. Vous aurez deviné que Polygone sera notre classe polymorphe, Rectangle et Cercle en seront les classes dérivées. Certains objecteront qu'un cercle n'est pas un polygone, merci d'en faire abstraction pour les besoins de la cause.

IV-F-2 - Implémenter une classe abstraite

Les classes Label et CheckBox s'appuient sur un socle commun défini dans la classe Control, et se différencient l'une de l'autre par un ensemble de propriétés et méthodes qui leurs sont propres. Nous allons donc commencer par définir ce socle commun dans notre classe Polygone.

```
'Dans la classe Polygone
Public Function Perimeter() As Double
   'Renvoie le périmètre d'un Polygone
End Function

Public Function Surface() As Double
   'Renvoie la surface d'un Polygone
End Function

Private Sub Class_Initialize()
   Err.Raise vbObjectError + 50, , "Impossible de créer une instance de la classe Polygone"
```

```
End Sub
```

Les deux fonctions sont des prototypes, c'est-à-dire des membres d'une classe réduits à une simple déclaration. Etant donné qu'elles ne contiennent aucune instruction elles renverront toujours zéro, et la classe Polygone ne contenant que des prototypes est donc inutilisable en l'état, il s'agit d'une classe abstraite. Comme il n'est pas question d'instancier cette classe il serait utile de la définir comme telle, mais rien ne le permet. On peut donc éventuellement prévoir de générer une erreur dans la procédure Class_Initialize afin d'empêcher l'application de créer un objet qui serait vide de contenu et donc de sens.

Deuxième étape, la définition des éléments particuliers à chacune des classes qui dériveront de la classe abstraite.

```
'Dans la classe Rectangle
Public Height As Double     'Hauteur du rectangle
Public Width As Double      'Largeur du rectangle
```

```
'Dans la classe Cercle
Public Ray As Double        'Rayon du cercle
```

Etablissons maintenant le lien entre la classe abstraite et ses classes dérivées au moyen de l'instruction Implements. Vous constaterez que la classe Polygone apparaît dans la liste de gauche de chacun de ces modules, et que les prototypes qu'elle contient apparaissent dans la liste de droite. Implements impose aux classes Cercle et Rectangle de redéfinir l'intégralité de l'interface de la classe Polygone, c'est-à-dire de redéclarer les prototypes de la classe Polygone afin de définir pour chacun d'eux un contenu adapté à la nature de l'objet qu'elles définissent. En effet on ne calcule pas la surface d'un cercle de la même manière que celle d'un rectangle.

```
'Dans la classe Rectangle
Implements Polygone

Private Function Polygone_Perimeter() As Double
    Polygone_Perimeter = 2 * (Height + Width)
End Function

Private Function Polygone_Surface() As Double
    Polygone_Surface = Height * Width
End Function
```

```
'Dans la classe Cercle
Implements Polygone

Private Function Polygone_Perimeter() As Double
    Polygone_Perimeter = Ray * 3.141592 * 2
End Function

Private Function Polygone_Surface() As Double
    Polygone_Surface = Ray * Ray * 3.141592
End Function
```

Enfin, les procédures implémentées étant privées il nous reste à écrire dans les classes Cercle et Rectangle une procédure publique permettant de renvoyer le code qu'elles contiennent lorsque le contrôleur de la classe n'est pas un Polygone mais simplement un Rectangle ou un Cercle.

```
'Dans les classes Cercle et Rectangle
Public Function Perimeter() As Double
   Perimeter = Polygone_Perimeter
End Function

Public Function Surface() As Double
   Surface = Polygone_Surface
```

```
End Function
```

Ici les fonctions implémentées (Polygone_Surface et Polygone_Perimeter) sont appelées par les fonctions publiques (Surface et Perimeter) mais l'inverse serait bien sur possible. Utilisons maintenant nos différentes classes.

```
Sub Test()
   Dim i As Long
   Dim P(1 To 2) As Polygone     'Déclare un tableau contenant 2 polygones
   Dim R As Rectangle
   Dim C As Cercle

   Set P(1) = New Rectangle      'Instancie un rectangle
   Set R = P(1)
   With R
      .Height = 10
      .Width = 20
   End With

   Set P(2) = New Cercle         'Instancie un cercle
   Set C = P(2)
   C.Ray = 10

   For i = 1 To 2
      Debug.Print "Surface: " & P(i).Surface & " Perimetre: " & P(i).Perimeter
   Next
End Sub
```

En décortiquant cette procédure somme toute assez simple, il en ressort plusieurs choses. En premier lieu une variable de type Polygone peut s'initialiser avec une instance de n'importe quelle classe qui l'implémente, démontrant ainsi le rôle d'une classe abstraite qui est de servir de réceptacle à des objets partageant des caractéristiques communes et non pas de définir un objet.

Ensuite le processus mis en □uvre par VBA pour afficher les caractéristiques de nos objets consiste à substituer à l'appel d'une fonction de la classe Polygone un appel à la fonction correspondante de la classe ayant servi à initialiser la variable. Une variable de type Polygone est donc susceptible de répondre à une instruction identique de manière différentes selon sa nature (Cercle ou Rectangle) du moment, il est donc possible de la qualifier de polymorphe.

Enfin, et c'est le plus important, la classe abstraite permet d'écrire du code sans connaître à l'avance le type de l'objet sur lequel ce code va s'appliquer à condition de se limiter aux membres implémentés, c'est ici le cas dans la boucle.

A contrario, si vous connaissez la nature de l'objet présent dans la variable il serait logique d'utiliser cette variable pour définir les dimensions (Height, Width ou Ray) de l'objet mais c'est hélas impossible, VBA ne connaissant de Polygone que son interface et pas sa nature il vous renverra immanquablement le message 'Membre de méthode ou de données introuvable'. Une classe implémentée ne permet donc pas d'accéder aux propriétés particulières des classes qui en sont dérivées, et le polymorphisme est donc incomplet.

Il va sans dire que cette limite est particulièrement frustrante et réduit fortement l'intérêt de la classe abstraite lorsque on la compare au type de donnée générique fourni par VBA, Object. Ce type de donnée peut en effet être considéré comme une classe abstraite universelle qui présente l'avantage d'offrir l'accès à tous les membres publics de l'instance de la classe qu'il contient en contrepartie d'une baisse - relative - des performances, d'une moindre lisibilité du code, d'un renoncement à l'autocomplétion du code et d'un moindre contrôle sur la syntaxe par VBA.

IV-F-3 - Implémenter une classe non abstraite

Si l'intérêt de la classe Polygone peut donc sembler limité, il reste possible d'utiliser cette classe pour définir des membres dont ses classes dérivées vont pouvoir hériter.

```
'Dans la classe Polygone
```

```
Private clsName As String

Public Property Get Name() As String
   Name = clsName
End Property

Public Property Let Name(NewName As String)
   If Len(NewName) > 2 Then
      clsName = NewName
   Else
      Err.Raise vbObjectError + 60, , "Le nom est trop court."
   End If
End Property
```

Nous définissons ici une propriété Name incluant un contrôle sur la longueur du nom, le but de la manœuvre étant de faire profiter Cercle et Rectangle de cette vérification sans la reproduire dans chaque module. Pour cela nous déclarons dans chacune de ces classes une variable privée de type Polygone.

```
'Dans les classes Cercle et Rectangle
Private clsPolygone As Polygone

Private Sub Class_Initialize()
   Set clsPolygone = New Polygone
End Sub
```

Evidemment, cette approche est incompatible avec le déclenchement d'une erreur dans la procédure Class_Initialize de la classe Polygone, qui n'est plus tout à fait abstraite. Les procédures implémentant la propriété Name se contenteront d'appeler la propriété Name de cette Variable.

```
'Dans les classes Cercle et Rectangle
Private Property Let Polygone_Name(RHS As String)
   clsPolygone.Name = RHS
End Property

Private Property Get Polygone_Name() As String
   Polygone_Name = clsPolygone.Name
End Property
```

Enfin, il ne reste qu'à définir dans les classes dérivées une propriété publique, faute de quoi la propriété Name d'un Cercle ou d'un Rectangle ne serait accessible qu'en passant par une variable de type Polygone.

```
'Dans les classes Cercle et Rectangle
Public Property Get Name() As String
   Name = clsPolygone.Name
End Property

Public Property Let Name(NewName As String)
   clsPolygone.Name = NewName
End Property
```

Si l'instruction Implements permet de profiter d'un polymorphisme limité elle n'est en revanche d'aucun secours en matière d'héritage et VBA ne propose en fait aucun mécanisme permettant d'atteindre ce but. La solution se résume donc finalement à emballer la classe dont on veut hériter, indépendamment du fait qu'elle soit ou non implémentée par la classe héritière.

Cette simulation d'héritage fonctionne parfaitement et ne présente pas d'autre défaut que sa lourdeur si ce n'est qu'il vous appartient de veiller à ce que le processus soit mis en œuvre de la même façon dans toutes les classes dérivées. Il va de soi que l'intérêt de ce type d'opération - la factorisation du code - augmente avec le nombre de classes dérivées, de membres hérités et avec la complexité de ces membres.

Pour conclure, le polymorphisme n'étant que partiellement pris en charge et l'héritage ne pouvant qu'être simulé, la mise en □uvre de ces concepts reste assez anecdotique.

V - Conclusion

Bien entendu les exemples proposés dans cet article ne le sont qu'à titre d'illustration, l'essentiel étant d'en comprendre l'esprit. J'espère que cela vous aura permis de mieux appréhender le fonctionnement et les possibilités offertes par les modules de classes, et éventuellement donné envie de les utiliser dans vos futurs développements.

Je tiens à remercier **Pierre Fauconnier** , **Maxence Hubiche** , **Starec** , **dolphy35** , **jeannot45** et particulièrement **PapyTurbo** , qui ont pris la peine de relire cet article, de valider les informations présentées, de tester les exemples et de me faire des suggestions ainsi que **Silkyroad** pour m'avoir encouragé à franchir le pas et aidé à publier ce premier article.

Enfin quelques liens sur ce thème:

- L'article de Microsoft (en anglais): **http://msdn.microsoft.com/en-us/library/aa140954(office.10).aspx**
- L'article de **sinarf** plus orienté vers Access: **http://sinarf.developpez.com/access/vbaclass/**
- Celui de **Pierre Fauconnier** traitant également de VB6: **http://fauconnier.developpez.com/articles/vba/general/classes/**

Et quelques exemples trouvés sur DVP :

- **ClasseFileSearch** , pour remplacer l'objet FileSearch dans les versions 2007 d'Office, par **Silkyroad** .
- **DateBox**, pour saisir une date dans un formulaire, par moi-même.
- **clGdiPlus** , classe de gestion d'image, par **Arkham46** .