Aide-mémoire LibreOffice

LibreOffice Basic
Les types structurés

v. 2.0 – 06/04/2021

Inter.

Rédigé avec LibreOffice v. 7.0.5 – Plateforme : Toutes

Tableaux (*Arrays*)

Regroupe des éléments ayant un rapport entre eux. Les éléments sont indexés (Long).

Dimensions

Un tableau peut comporter plusieurs dimensions (maxi : 60).

Déclarer

La base d'indexation est 0 (zéro) ! Modifiable pour 1, par `Option Base 1`.

Ci-dessous : `ix` = index.

Tableaux statiques	*Caractéristiques prédéfinies*
1 dimension (« vecteur »)	
`Dim T(ixMin To ixMax) As UnType`	`Dim T(1 To 12) As UnType` 1 dimension de 12 emplacements mémoire, indexés de 1 à 12.
ou `Dim T(ixMax) As UnType`	`Dim T(9)` : 1 dimension de 10 éléments, indexés de 0 à 9.
Plusieurs dimensions (« matrice »)	
`Dim T(ixMin1 To ixMax1, ixMin2 To ixMax2) As UnType`	Plusieurs dimensions (exemple avec 2). `Dim T(1 To 12, 1 To 31) As Integer` 2 dimensions (réserve 12 et 31 éléments)
ou `Dim T(ixMax1, ixMax2, ...) As UnType`	`Dim T(2,4) As String` : 2 dimensions. 3 éléments pour la 1è, 5 pour la 2è.

Tableaux dynamiques	*Caractéristiques connues à l'exécution*
`Dim Tablo() As UnType` `Dim Tablo As Variant`	Déclare un tableau de dimension inconnue. Un `ReDim` sera nécessaire par la suite.

Déclaré de type `Variant`, un tableau peut contenir des éléments de types différents.

Tableaux imbriqués (*Nested arrays/Jagged arrays*)

Ou tableaux de tableaux. Ex : utilisés pour accéder aux données des plages dans Calc.

Un tableau externe (de variants) a pour éléments des tableaux de données :

```
Dim T As Variant
T = Array(Array(1, 2, 3), Array(10, 20, 30), Array(7, 8, 9))
```

`T(0)(0)` vaut 1; `T(2)(2)` vaut 9, etc.

Accéder à un élément d'un tableau

Par son index :

Une dimension	**Deux dimensions**
`Dim Tablo(9) As Integer` `Tablo(5) = 123`	`Dim Tablo(11, 31) As Integer` `Tablo(5, 28) = 123`

Fonctions et instructions relatives aux tableaux

`Option Base 1`	(instruction en début de module – applicable au module courant seult) Force l'index des tableaux à débuter à 1 au lieu de 0.
`IsArray()`	Renvoie `True` si la variable est d'un type tableau. `OK = IsArray(Tablo)`

`Array()`	Renvoie un tableau créé à partir de valeurs discrètes.
	`Tablo = Aray(1, 50, 21, 97)  'tableau d'entiers`
	`Tablo = Array("Un", 2, Now())  'ici, tableau de variants`

`Redim`	(instruction) Redimensionne un tableau • Avec perte des données : `Redim Tablo(nlle dimension)` • Sans perte de données : `Redim Preserve Tablo(nlle dimension)`
`Erase`	(Instruction) Efface le contenu du tableau. `Erase Tablo` Si tableau dynamique, libère la mémoire.
`LBound()`	Renvoie la borne inférieure. Par défaut sur la 1ère dimension, sinon préciser : `LBound(Tablo, 2)`
`UBound()`	Renvoie la borne supérieure (idem). Le tableau n'a pas de dimension définie si `UBound(Tablo) = -1` **et** `LBound(Tablo) = 0`
`Split()`	Crée un tableau (vecteur) par découpe d'une chaîne sur un délimiteur. `T=Split("C:/file.txt", "/")` → `T(0)="C:", T(1)="file.txt"`
`Join()`	Réalise l'opération inverse de `Split()` : fusionne des éléments d'un tableau (vecteur) pour initialiser une chaîne. `Join(T, "|")` → `"C:|file.txt"`

Vérifier la validité d'un tableau

`Tablo` variable « tableau ». Elle est manipulable comme telle si elle passe les trois tests :

1. La variable existe-t'elle ?	`Not IsNull(Tablo)`
2. S'agit-il d'un tableau ?	`IsArray(Tablo)`
3. La dimension du tableau est-elle définie ?	`UBound(Tablo) >= LBound(Tablo)`

Parcourir un tableau à 1 dimension (vecteur)

Par ses index

```
Dim Tablo(9) As Integer, i As Long
For i = LBound(Tablo) To UBound(Tablo)
  Print Tablo(i)
Next i
```

Par ses éléments

```
Dim Element As 'type compatible avec les éléments
For Each Element In Tablo
  Print Element
Next Element
```

Parcourir un tableau à 2 dimensions

```
Dim Tablo(2, 4)    '3 lignes, 5 colonnes
Dim i As Long, j As Long
For i = LBound(Tablo) To Ubound(Tablo)
  For j = LBound(Tablo, 2) To UBound(Tablo, 2)
    Print Tablo(i, j)
  Next j
Next i
```

Trier des tableaux

Pas de fonctionnalité prédéfinie (chercher `QuickSort` sur le web).

Dupliquer des tableaux

Cas général Une boucle (`For..Next`) qui recopie les valeurs entre les 2 tableaux.
Peut être looong !

Astuce Par affectation puis `ReDim Preserve` :

```
Tablo2 = Tablo1  'les deux var. -> les mêmes données
ReDim Preserve Tablo2(Taille)  '-> 2 tableaux différents
```

S'applique uniquement aux tableaux de valeurs simples (non objet).

Utiliser les tableaux avec des sous-programmes

Comme paramètre d'une *Sub*

Sub avec paramètre tableau
```
Sub UseArray(ByRef Tablo() As String)
```
Appel de la Sub
```
Dim Tablo(9) As String
UseArray(Tablo)
```

Comme résultat d'une *Function*

Fonction avec résultat tableau
```
Function GetArray() As Integer()
  GetArray = UnTabloEntiers()
End Function
```
Retour de la fonction
```
Dim MonTablo As Integer
MonTablo = GetArray()
```

Tableaux et plages de classeur

(voir A-M n°3)

Types personnalisés (*Custom types*)

Permettent d'agréger plusieurs valeurs (membres) dans un type de donnée unique.
Simplifient la manipulation de données, le passage de paramètres ou le retour de fonction.

Type des membres : n'importe quel type simple, objet ou personnalisé (**sauf** tableau).

Déclaration du type

```
Type MonTypePerso
  UnMembre As TypeSimple
  AutreMembre As AutreTypeSimple
End Type
```

Exemple

```
Type Evenement
  Nom As String
  DateHeure As Date
End Type
```

Déclaration des variables de type personnalisé

```
Dim UneVar As MonTypePerso
```

Limitation : un type personnalisé n'est visible *que dans le module où il est déclaré*. La déclaration `As MonType` n'est donc possible qu'au sein du **même** module que celui où est déclaré le type `MonType`. Voir le contournement Fonction « usine » (factory) / Accesseur (accessor).

Utilisation

Déclaration
```
Dim VarPerso As MonTypePerso
```
Affectation
```
VarPerso.UnMembre = UneValeur
```
Lecture
```
UneVar = VarPerso.AutreMembre
```
Le mot-clé With

Permet d'abréger les références aux éléments

```
With VarPerso
  .UnMembre = UneValeur
End With
```

Notez la présence du point.

Collections

Structure qui permet l'accès rapide aux données en les indexant.
Une collection est manipulée comme un type `Object`.
Les éléments stockés peuvent être de n'importe quel type, y compris `Object`.

La clé d'indexation est de type `String`. Dans une collection, chaque clé est **unique**.

Déclarer une collection
```
Dim oColl As New Collection
```
On ne préjuge pas du contenu.

Ajouter un élément

`Elt` : tout type.

Avec clé (unique !)
```
oColl.Add(Elt, "LaClé")
```
Accès ultérieur par clé ou par index.
La clé n'est pas sensible à la casse.
Il est possible de préciser l'emplacement de l'ajout au moyen de `Before/After` :
```
oColl.Add(Elt, "LaClé", After:="Clé0")
```

Sans clé
```
oColl.Add(Elt)
```
Accès ultérieur par index seulement.

Vérifier l'existence d'un élément Tenter d'accéder et traiter l'erreur éventuelle (voir code ci-dessous).

Accéder à un élément

Par sa clé
```
Valeur = oColl("LaClé")
```
Par son index
```
Valeur = oColl.Item(Index)
```

Remplacer un élément Même clé, nouvelle valeur d'élément.

Supprimer puis **Ajouter**.

Supprimer un élément

Par sa clé
```
oColl.Remove("LaClé")
```
Par son index
```
oColl.Remove(Index)
```

Supprimer tous les éléments
```
ReDim oColl As New Collection
```

Nombre d'éléments
```
Nombre = oColl.Count
```

Vérifier l'existence d'un élément

Tenter d'accéder à la clé et traiter l'erreur éventuelle.

```
Function ExistsItem(ByRef pColl As Object, pCle As String) As Boolean
  Dim Element As Variant, Existe As Boolean
  On Local Error Goto ErrHandler  'gérer l'erreur
  Existe = False
  Element = pColl(pCle)  'si pCle n'existe pas -> erreur
  Existe = True
  ErrHandler:
    'ne rien faire
  ExistsItem = Existe
End Function
```

Parcourir une collection

Vous pouvez récupérer les éléments d'une collection en la parcourant.

☞ Il n'existe pas de possibilité de lister les clés.

Par index

```
For i = 1 To oColl.Count
  Element = oColl.Item(i)    'accès à l'élément
Next i
```

Par accès direct aux éléments

```
Dim Element As 'type compatible avec les éléments de la collection.
For Each Element In oColl
  'faire qqch avec Element
Next Element
```

Classes

Créer des classes en Basic

Embryon de programmation orientée objet (POO).

☞ Limites : pas d'héritage (utilisez la délégation), pas de polymorphisme !

Une classe LibreOffice Basic pourrait donc être vue comme un type personnalisé amélioré auquel on ajoute un comportement (fonctions et sous-programmes).

Vocabulaire

Module de classe	**Module de code** destiné à contenir les déclarations de la classe. La classe est tout entière contenue dans ce module.
Classe	**Type** qui permet de créer (instancier) des variables objet.
Événements	Deux événements peuvent être interceptés : la **création** et la **destruction** de l'objet.
Membre	**Variable** interne à une classe (pas utilisée depuis l'extérieur).
Propriété	Reflète l'**état** de l'objet.
Méthode	Réalise une **action** sur l'objet.
Instance	**Objet** créé à partir d'un type classe.

Spécifier une classe

Les spécifications d'une classe (membres, événements, propriétés, méthodes) se font à l'intérieur d'un module de code dédié. Dans LibreOffice Basic, ce module ne se distingue d'un module standard que par ses options initiales.

✧ Conseil : adoptez une convention de nommage pour les modules de classes.

Options initiales

En début du module de classe, précisez les options :

```
Option Explicit  'comme d'habitude
Option Compatible
Option ClassModule
```

Variables membres

Elles sont internes, donc déclarées `Private`.

☞ Les membres d'une classe ne devraient **jamais** être appelés directement à travers l'instance mais uniquement au travers de propriétés créées à cet effet.

```
Private mName As String
Private mSheet As Object
```

Événements (*Events*)

Ce sont deux sous-programmes internes, donc déclarés `Private`.

Constructeur `Private Sub Class_Initialize()`
Pour initialiser l'objet en cours de création.

Destructeur `Private Sub Class_Terminate()`
Pour nettoyer les composants internes d'un objet en cours de destruction.

☠ **Faille de sécurité**. Il est **très fortement** déconseillé d'inclure ce destructeur dans vos classes : en raison d'un bug d'implémentation dans VisualBasic, `Class_Terminate()` constitue une faille de sécurité et, comme telle, est rejeté par les antivirus (voir CVE-2018-8174).

☞ **Limitation** : il n'est pas possible de passer des paramètres à ces sous-programmes.

Propriétés (*Properties*)

Propriété = **état** de l'objet.

Lorsque les propriétés sont visibles de l'extérieur, elles sont déclarées `Public`.

En lecture (Get) (tous types de données, y compris objets)

```
Public Property Get Name() As String
  Name = mName
End Property
```

En écriture (Let) (tous types de données sauf objets)

```
Public Property Let Name(ByRef pName As String)
  mName = pName
End Property
```

En écriture (Set) (objets seulement)

```
Public Property Set Sheet(ByRef pSheet As Object)
  Set mSheet = pSheet
End Property
```

☞ Il est possible de quitter prématurément une propriété par `Exit Property`

Une classe peut comporter des propriétés en lecture et en écriture : écrivez successivement les deux propriétés `Get` et `Let`/`Set` si nécessaire.

Les propriétés peuvent accéder aux : membres, autres propriétés, méthodes de la classe.

Méthodes (*Methods*)

Méthode = **action** sur/de l'objet.

Sub et `Function` propres à la classe, internes (`Private`) ou visibles (`Public`). S'écrivent comme des Sub ou des `Function` standard précédées du mot-clé `Public` ou `Private`. Elles ont accès aux membres et aux propriétés de la classe.

Utiliser des classes en Basic

Déclarer/créer un objet

Instanciation immédiate `Set MonObjet = MaClasse`

Instanciation différée (au 1er appel)

```
Dim MonObjet As New MaClasse
MonObjet = New MaClasse
```

Le constructeur de la classe est appelé au moment de l'instanciation de l'objet.

La déclaration d'un objet n'est pas possible en dehors de la bibliothèque dans laquelle le module de classe existe. Vous devrez généralement déclarer l'objet `As Object`.

☠ **Limitation** : Une classe n'est **pas** visible en dehors de la bibliothèque dans laquelle elle est déclarée. Voir Fonction « usine » (factory) / Accesseur (accessor).

Accéder aux propriétés et méthodes d'un objet

Syntaxe d'accès aux éléments d'un objet : `objet.propriété` ou `objet.méthode`

Comme pour les types personnalisés, vous pouvez utiliser la syntaxe `With..End With`

Libérer un objet

Lorsqu'un objet n'est plus nécessaire, vous pouvez le détruire : `Set oObjet = Nothing`

Le destructeur de la classe est appelé à ce moment là.

☞ Cette instruction n'est pas strictement nécessaire dans des Sub ou des `Function`, puisque leurs variables internes sont détruites en sortie. Cependant la destruction effective de la variable n'est alors pas sous votre contrôle.
L'instruction `Set oObjet = Nothing` :
– montre clairement l'intention,
– assure le contrôle du moment de la destruction de l'objet.

Visibilité des types personnalisés et des classes

- Un **type personnalisé** n'est visible que dans le **module** où il est déclaré.
- Une **classe** n'est visible que dans la **bibliothèque** où elle est déclarée.

	☠ Utilisation	☠ Visibilité	☞ Usine créée dans	☞ Déclaré
Type perso	`As MonType`	Même module	Même module	`As Variant`
Classe	`As MaClasse`	Même bibli	Même bibli, autre module	`As Object`

Fonction « usine » (*factory*) / Accesseur (*accessor*)

Pour contourner les problèmes de visibilité, préparez une fonction « usine » (ou accesseur) pour créer les variables. Cette fonction pourra être appelée de n'importe quel module/bibliothèque.

Création de la fonction « usine » / accesseur

✧ La fonction usine peut aussi être utilisée pour initialiser la variable (passez-lui des paramètres).

Types personnalisés

Créez l'usine dans le même module que celui où est déclaré le type personnalisé.

```
Function CreateMonTypePerso() As MonTypePerso
  Dim oVar As MonTypePerso
  CreateMonTypePerso = oVar
End Function
```

Classes

Créez l'usine dans un autre module de la même bibli que celle où est déclarée la classe.

```
Function CreateMaClasse() As MaClasse
  Dim oVar As New MaClasse
  CreateMaClasse = oVar
End Function
```

Utilisation de la fonction « usine » / accesseur

Types personnalisés

```
Dim MaVar As Variant
MaVar = CreateMonTypePerso()
```

Classes

```
Dim MaVar As Object
MaVar = CreateMaClasse()
```


Crédits

Auteur : Jean-François Nifenecker – jean-francois.nifenecker@laposte.net

Nous sommes comme des nains assis sur des épaules de géants. Si nous voyons plus de choses et plus lointaines qu'eux, ce n'est pas à cause de la perspicacité de notre vue, ni de notre grandeur, c'est parce que nous sommes élevés par eux. (Bernard de Chartres [attr.])

Historique

Version	Date	Commentaires
2.0	06/04/2021	Renommage de l'A-M n°9 ; mise à jour de la mise en forme ; actualisations.