Les types de données

Désignations	Tailles	Plages de valeurs
System.Boolean (type valeur)	2 octets	**True** ou **False**
System.Byte (type valeur)	1 octet	**0** à **255** (non signés).
System.SByte (type valeur)	1 octet	**-128** à **127**.
System.Char (type valeur)	2 octets	**0** à **65 535** (non signés).
System.DateTime (type valeur)	8 octets	**0:00:00** le **1er janvier 0001** à **23:59:59** le **31 décembre 9999**.
System.Decimal (type valeur)	16 octets	**0** à **+/-79 228 162 514 264 337 593 543 950 335** sans décimale ou **0** à **+/-7,9228162514264337593543950335** avec **28** décimales. Le plus petit nombre différent de zéro est : **+/-0,0000000000000000000000000001** (**+/-1E-28**).
System.Single (type valeur)	4 octets	**-3,402823E+38** à **-1,401298E-45** pour les valeurs négatives ; **1,401298E-45** à **3,402823E+38** pour les valeurs positives.
System.Double (type valeur)	8 octets	**-1,79769313486231E+308** à **-4,94065645841247E-324** pour les valeurs négatives et **4,94065645841247E-324** à **1,79769313486231E+308** pour les valeurs positives.
System.Int16 (type valeur)	2 octets	**-32 768** à **32 767**. Synonyme autorisé : **Short**
System.Int32 (type valeur)	4 octets	**-2 147 483 648** à **2 147 483 647**. Synonyme autorisé : **Integer**
System.Int64 (type valeur)	8 octets	**-9 223 372 036 854 775 808** à **9 223 372 036 854 775 807**. Synonyme autorisé : **Long**
System.UInt16 (type valeur)	2 octets	**0** à **65 535**. Synonyme autorisé : **UShort**
System.UInt32 (type valeur)	4 octets	**0** à **4 294 967 295**. Synonyme autorisé : **UInteger**
System.UInt64 (type valeur)	8 octets	**0** à **18 446 744 073 709 551 615**. Synonyme autorisé : **ULong**
System.Object (type référence)	4 octets ou 8 octets	N'importe quel type peut être référencé par une variable de type **Object**. (4 octets sur un système 32 bits, 8 octets sur un système 64 bits)
System.String (type référence)	Dépend de l'implémentation	**0** à environ **2** milliards de caractères Unicode (2 octets par caractère).
Type utilisateur System.ValueType (type valeur)	Dépend de l'implémentation	Chaque membre de la structure présente une plage déterminée par son type de données.

Chaque type de donnée repris dans le tableau précédent est assorti d'une indication *type valeur* ou *type référence*. La différence entre les deux réside dans la manière de leur réserver de la place en mémoire et de les initialiser.
La déclaration d'une variable de type *valeur* provoque la réservation de son emplacement en mémoire et son initialisation à la valeur nulle (sauf pour une variable de type **DateTime** pour laquelle la valeur d'initialisation est **111000**, c'est-à-dire **01/01/0001 01:01:01**).
La déclaration d'une variable de type *référence* provoque la réservation en mémoire de l'emplacement nécessaire pour stocker l'adresse où sont effectivement stockées les données désignées par cette variable. L'initialisation par défaut d'une telle variable, qui est donc un pointeur, est **Nothing**. Une variable de type *référence* doit être initialisée par des valeurs ou par l'usage de l'opérateur **New**. Cette initialisation doit permettre au compilateur d'évaluer et de réserver l'espace nécessaire en mémoire, et d'initialiser la variable avec l'adresse de la zone ainsi réservée.

Notation littérale de données

La notation littérale d'une donnée est l'expression de sa valeur dans le code. Par exemple : **`Pi = 3.141592`**

`Integer`	**`145, -7, &FF`**	**`&`** est, dans ce cas, le préfixe pour l'hexadécimal
`Long`	**`100000L`**	
`Double`	**`134.789, -45E-18`**	
`Single`	**`134.789F, -45E-18F`**	**`F`** comme float pour désigner la simple précision
`Decimal`	**`100000D`**	
`Char`	**`"A"c`**	**`c`**, pour transformer la chaîne de caractères **`"A"`** en caractère **`'A'`**
`Date`	**`New Date(2006, 3, 1)`**	instanciation d'un objet **`Date`** initialisé à **`01/03/2006`**
	`#1/3/2006#`	forme ordinaire du littéral de type **`Date`** et **`Time`** (**`#2:25:30 PM#`**)
	`"01/03/2006"`	littéral de type **`Date`** et **`Time`** sous forme de chaîne
`String`	**`"aujourd'hui"`**	si la chaîne doit contenir le caractère ", on double celui-ci comme dans **`"abcd""e"`** pour représenter la chaîne **`abcd"e`**.

Déclaration des tableaux

Contrairement aux variables élémentaires de type ***valeur*** dont il a été question jusqu'à présent, les tableaux sont de type ***référence***.
Alors que **`Dim X As Integer`** réserve la place en mémoire pour un entier, **`Dim T() As Integer`** réserve la place en mémoire pour l'adresse d'un tableau d'entiers et non pour le tableau lui-même. Le tableau **`T`** doit être spécialement initialisé pour que la référence désigne effectivement son contenu.

Comme indiqué précédemment, l'accès aux données d'un tableau se fait par la désignation de son nom flanqué des indices nécessaires. L'indice minimal est toujours **`0`** et l'indice maximal est celui indiqué par l'initialisation.
Par exemple, **`Dim T(5) As Integer`** réserve l'espace pour **`6`** entiers, tous initialisés à **`0`**, accessibles par un indice variant de **`0`** à **`5`**.
Ou encore **`Dim T() As Integer = {0, 1, 2, 3, 4, 5}`** réserve l'espace pour **`6`** entiers, respectivement initialisés avec les valeurs **`0`**, **`1`**, **`2`**, **`3`**, **`4`** et **`5`**, également accessibles par un indice variant de **`0`** à **`5`**.

Déclarations des arguments des procédures, fonctions et méthodes

Un argument de procédure (ou fonction), et donc de méthode, définit une variable utilisable dans cette méthode pour y traiter les données qui lui sont passées en paramètres lors de l'appel. Cette variable est donc locale et bénéficie de l'effet de masque.

Toutefois, bien qu'une variable définie en déclaration d'argument ne soit jamais accessible de l'extérieur de sa procédure (ou fonction), sa modification peut changer la valeur de la variable correspondante dans le code appelant. Les spécifications **`ByRef`** et **`ByVal`** et le type (*Valeur* ou *Référence*) de la variable déterminent la sécurité des données transmises.

`ByVal`

Un argument déclaré **`ByVal`** est strictement local s'il est d'un type *Valeur*. C'est une copie de la variable d'origine qui est transmise et sa modification n'affecte en rien la variable d'origine. S'il s'agit d'un type *Référence*, toute modification de la variable dans la procédure (ou fonction) modifie également la variable utilisée dans le code appelant.
C'est la spécification par défaut. Le type **`String`** se comporte comme un type *Valeur*.

```
Private Sub ValByVal(ByVal a As Byte)
    a = a * 2                               ' Multiplie l'argument par 2
End Sub                                     ' La variable d'origine sera inchangée

Private Sub ChaineByVal(ByVal Ch As String)
    Ch = Ch & "+++"                         ' Concatène "+++" à l'argument
End Sub                                     ' La variable d'origine sera inchangée

Private Sub RefByVal(ByVal T() As Byte)
    T(2) = T(2) * 2                         ' Multiplie un élement de l'argument par 2
End Sub                                     ' La variable d'origine sera changée

Private Sub Test()
  Dim s As String = "AZE"
  Dim x As Byte = 5
  Dim T() As Byte = {10, 11, 12, 13, 14, 15}
  Console.WriteLine("Av.ValByVal X = " & x)        ' Affiche : Av.ValByVal X = 5
  ValByVal(x)
  Console.WriteLine("Ap.ValByVal X = " & x)        ' Affiche : Ap.ValByVal X = 5
  Console.WriteLine("Av.ChaineByVal s = " & s)     ' Affiche : Av.ChaineByVal s = AZE
  ChaineByVal(s)
  Console.WriteLine("Ap.ChaineByVal s = " & s)     ' Affiche : Ap.ChaineByVal s = AZE
  Console.WriteLine("Av.RefByVal T(2) = " & T(2))  ' Affiche : Av.RefByVal T(2) = 12
  RefByVal(T)
  Console.WriteLine("Ap.RefByVal T(2) = " & T(2))  ' Affiche : Ap.RefByVal T(2) = 24
End Sub
```

`ByRef`

Si l'argument est déclaré **`ByRef`**, alors la procédure (ou fonction) reçoit l'adresse mémoire de la variable et ce, quel que soit le type de cette variable. Dans ce cas, toute modification de la variable dans la procédure (ou fonction) entraîne la modification de la variable d'origine. Il faut être prudent quant aux manipulations effectuées sur cette variable à l'intérieur de la procédure ou de la fonction.

```
Private Sub ValByRef(ByRef a As Byte)
    a = a * 2                               ' Multiplie l'argument par 2
End Sub                                     ' La variable d'origine sera changée

Private Sub RefByRef(ByRef T() As Byte)
    T(2) = T(2) * 2                         ' Multiplie un élement de l'argument par 2
End Sub                                     ' La variable d'origine sera changée

Private Sub Test()
  Dim x As Byte = 5
  Dim T() As Byte = {10, 11, 12, 13, 14, 15}
  Console.WriteLine("Av.ValByRef X = " & x)        ' Affiche : Av.ValByRef X = 5
  ValByRef(x)
  Console.WriteLine("Ap.ValByRef X = " & x)        ' Affiche : Ap.ValByRef X = 10
```

```
  Console.WriteLine("Av.RefByRef T(2) = " & T(2))  ' Affiche : Av.RefByRef T(2) = 12
  RefByRef(T)
  Console.WriteLine("Ap.RefByRef T(2) = " & T(2))  ' Affiche : Ap.RefByRef T(2) = 24
End Sub
```

Optional

Avec **ByVal** et **ByRef**, **Optional** et **ParamArray** constituent les quatre spécifications des arguments des procédures (ou fonctions).

Le mot **Optional** indique que l'argument qu'il précède est facultatif. Cet argument peut donc être absent lors de l'appel de la procédure (ou fonction). Sa valeur par défaut doit être indiquée lors de la déclaration des arguments.
Quand plusieurs arguments sont facultatifs, l'emploi de virgules lors de l'appel permet d'en omettre certains et pas d'autres. Les arguments sont évalués selon leur ordre de définition.
Il est toutefois possible de désigner explicitement les arguments par leur nom lors de l'appel. Les arguments ainsi utilisés sont dits *arguments nommés*. Cette faculté, qui n'est pas réservée aux arguments facultatifs, permet de passer les variables dans un ordre différent de celui de la définition.

```
Private Sub Optionel(ByVal X As Byte, Optional ByVal Y As String = "Y absent")
    Console.WriteLine("Optionel " & X & " / " & Y)
End Sub

Private Sub Nommes(ByVal X As Byte, Optional ByVal Y As String = "Y",
                                                Optional ByVal Z As String = "Z")
    Console.WriteLine("Nommes " & X & " / " & Y & " / " & Z)
End Sub

Private Sub Test()
  Optionel(1)                          ' Affichage de Optionel : Optionel 1 / Y absent
  Optionel(2, "Pierre")                ' Affichage de Optionel : Optionel 2 / Pierre
  Nommes(3, Z:="Pierre")               ' Affichage de Nommes : Nommes 3 / Y / Pierre
  Nommes(4, , "Pierre")                ' Affichage de Nommes : Nommes 4 / Y / Pierre
  Nommes(Z:="Pierre", Y:="Bill", X:=5)' Affichage de Nommes : Nommes 5 / Bill / Pierre
End Sub
```

ParamArray

Cette spécification, qui doit être la dernière de la liste d'arguments, signifie que l'argument qui suit est facultatif et que s'il est présent, il est un tableau d'arguments du type précisé et de taille indéterminée. Un argument **ParamArray** est toujours passé à l'aide de **ByVal**.

```
Private Sub ParamArra(ByVal ParamArray PX() As Integer)
    Dim XX As Integer
    For Each XX In PX
        Console.Write(XX & " / ")
    Next XX
    Console.WriteLine()
End Sub

Private Sub Test()
  Dim T () As Integer = {101, 102, 103, 104}
  ParamArra()                          ' N'affiche rien dans ParamArra
  ParamArra(110, 120, 130)             ' Affichage dans ParamArra :   110 / 120 / 130 /
  ParamArra(T(0), T(1), T(2))          ' Affichage dans ParamArra :   101 / 102 / 103 /
  ParamArra(T)                         ' Affichage de tous les éléments de T
End Sub
```

Type de donnée des arguments

Indépendamment des spécifications **ByVal**, **ByRef**, **Optional** et **ParamArray**, les arguments doivent être typés, c'est-à-dire que leur type de donnée doit être explicité. Les arguments peuvent **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single** ou **String** ou bien le nom d'une énumération, d'une structure, d'une classe ou d'une interface.

Valeur de retour des fonctions

Le type de la valeur retournée par une fonction doit être précisé dans la ligne de définition de la fonction.

Les fonctions utilisent l'instruction **`Return`** pour simultanément envoyer le résultat de leur travail au code appelant et quitter la fonction.

```
Public Function MaFonction() As Byte
  '  … code de la fonction
  Return 10
  '  … code non exécuté après la sortie inconditionnelle de la fonction …
```

Une ancienne méthode consiste à affecter le résultat au nom de la fonction. Cette opération ne provoque pas la sortie de la fonction. Le résultat n'est effectivement transféré qu'à la fin de la fonction ou lors d'une sortie forcée par **`Exit Function`**.

```
Public Function MaFonction() As Byte
  '  … code de la fonction
  MaFonction = 10
  '  … code encore exécuté avant la sortie de la fonction …
```

Lors d'une sortie forcée par **`Exit Function`**, si le nom de la fonction n'a subi aucune affectation, la fonction retourne une valeur nulle selon le type de la valeur de retour : **`0`** pour les numériques, **`Nothing`** pour les objets, … .