4

Le langage Visual Basic.NET

4. Le langage Visual Basic.NET

La conception d'une application Visual Basic se fait initialement en construisant l'interface utilisateur en dessinant les fenêtres et en déposant les composants nécessaires au fonctionnement de l'application à l'aide de l'environnement visuel de Visual Basic. Une fois l'interface utilisateur conçue, le temps vient de spécifier les instructions que l'application doit exécuter à l'aide de code en langage Basic.

Déclaration et utilisation des variables

Un programme est constitué d'événements exécutant des fonctions. Les données et informations traitées par ce programme peuvent être stockées dans la mémoire vive de l'ordinateur afin d'être ultérieurement récupérées pour être traitées ou comparées. Ces différents espaces mémoires utilisables par le programmeur sont appelés **variables** puisqu'elles stockent des valeurs qui peuvent varier selon le cours d'exécution du programme.

Le programmeur donne des noms aux différentes variables qu'il désire utiliser au sein de son programme afin de pouvoir les distinguer entre elles. De plus, le programmeur doit spécifier un type de données pour cette variable. Un type de données spécifie au compilateur le type d'informations qui sera tenu au sein de cette variable et les dimensions de l'espace mémoire nécessaires pour contenir ces informations. Le programmeur doit préalablement spécifier au compilateur l'existence d'une variable avant de pouvoir lui assigner une valeur ou en lire la valeur. Le programmeur doit donc procéder à la **déclaration de la variable** avant de pouvoir l'utiliser. La déclaration de variables s'effectue comme suit :

`Dim X As Integer`	La variable X est un nombre entier.
`Dim Nom As String`	La variable Nom est une chaîne de caractères.
`Dim a, b, c As System.Int64`	Les variables a, b et c sont des entiers longs.

Le contenu d'une variable peut changer au cours de l'exécution du code. Cependant, une variable ne peut contenir qu'une seule donnée à la fois. On peut parfois nommer ces variables *« scalaires »* en oppositions aux variables *« vectorielles »* pouvant contenir plusieurs valeurs que la programmation informatique nomme *« tableaux »*. Il sera question des tableaux plus loin.

Le nom d'une variable déclarée en Visual Basic doit respecter les critères suivants afin d'être reconnue par le compilateur :

- Le premier caractère est obligatoirement une lettre.
- Le nom peut être constitué de lettres, chiffres et du caractère de soulignement "_".
- Le nom ne peut excéder 1024 caractères… mais était-ce vraiment nécessaire de le préciser ?
- Aucune distinction n'est appliquée entre les lettres majuscules et minuscules, les deux étant considérées équivalentes.
- Les accents sont tolérés par Visual Basic mais non conseillés par les bonnes techniques de programmation.
- Les mots réservés du langage Visual Basic ne peuvent être utilisés comme noms de variables (`For, If, Select, Next, Function …`)

Quoique aucune autre règle formelle autre que celles énumérées précédemment ne soit prévue par Visual Basic au sujet de la nomination des variables, certaines conventions établies par la maturité de la programmation informatique proposent certaines pratiques permettant d'augmenter la lisibilité du code. Une de ces conventions stipule qu'il peut être bon de réserver certains des premiers caractères composants le nom d'une variable afin d'identifier le type de données qu'y sera stocké. Le tableau présente certains exemples mettant en œuvre cette convention :

	Type de variable	Exemple
`n`	Valeur numérique *(tous types confondus)*	`Dim nCompteur As Integer`
`b`	Valeur booléenne	`Dim bReponse As Boolean`
`d`	Date	`Dim dAujourdhui As Date`
`obj`	Variable-objet	`Dim objNet As Object`
`str`	Chaîne de caractères	`Dim strAdresse As String`

La déclaration d'une variable désigne le procédé par lequel le programmeur donne un nom à une espace mémoire dont l'endroit est choisie par le système d'exploitation mais dont la dimension est choisie par le programmeur à l'aide des types de données. Les types de données définissent le type d'information qui sera contenu au sein de la variable ainsi que la manière dont ces informations doivent être traitées et interprétées.

Les variables peuvent manipuler des données de différents types.

- Les variables **numériques** peuvent contenir des valeurs numériques entières ou réelles sur lesquelles diverses opérations mathématiques peuvent être effectuées.

```
nChiffreA  = 5
nChiffreB = 10
nChiffreC = (nChiffreA * 2) + nChiffreB
```

- Les variables **chaînes de caractères** peuvent contenir une suite de caractères qui, mis ensembles, forment un mot, une phrase ou une expression. Certaines opérations typiques aux chaînes de caractères peuvent être effectuées sur ces variables. Une nouvelle valeur est attribuée à ce type de variables en incluant les caractères entre deux guillemets *(" ")* :

```
strNom = "Pinchaud"
strVoiture = "Ford Mustang"
MsgBox (strNom & " conduit une voiture " & strVoiture)
```

- Les variables **dates** peuvent contenir des dates. Certaines opérations typiques aux dates peuvent être effectuées sur ces variables. Une nouvelle valeur est attribuée à ce type de variables en incluant la date entre deux dièses *(# #)*:

```
dNaissance = #15-03-2002#
```

Notez que deux désignations des types de données sont possibles en Visual Basic.NET. La première utilise les types de données spécifiques à Visual Basic tandis que la seconde, équivalente, utilise les types de données natifs à la plate-forme .NET et reconnus par l'ensemble des langages .NET.

Type de données VB	Type de données .NET	Taille	Plage de valeurs
Boolean	System.Boolean	1 octet	True ou False
Byte	System.Byte	1 octet	Entiers non-signés de 0 à 255
Char	System.Char	2 octets	Entiers non-signés de 0 à 65535
Date	System.DateTime	8 octets	Dates du 1er janvier 1 au 31 décembre 9999
Decimal	System.Decimal	12 octets	+/- 79 228 162 514 264 337 593 543 950 335 sans séparateur décimal ; +/-7,9228162514264337593543950335 avec 28 chiffres à droite du séparateur décimal ; le plus petit nombre différent de zéro est +/- 0.0000000000000000000000000001.
Single	System.Single	4 octets	Réels -3,402823E38 à -1,401298E-45 pour les valeurs négatives ; 1,401298E-45 à 3,402823E38 pour les valeurs positives
Double	System.Double	8 octets	Réels de -1.79769313486231E308 à -4.94065645841247E-324 pour les valeurs négatives ; Réels de 4.94065645841247E-324 à 1.79769313486232E308 pour les valeurs positives.
Short	System.Int16	2 octets	Entiers de -32 768 à 32 767
Integer	System.Int32	4 octets	Entiers de –2 147 483 648 à 2 147 483 647
Long	System.Int64	8 octets	Entiers de -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
String	System.String	10 octets + (2 x nbr. de caractères dans la chaîne)	0 à environ 2 milliards de caractères Unicode[1].
Object	System.Object	4 octets	N'importe quel type peut être stocké dans une variable de type Object.

[1] Un caractère Unicode est emmagasiné sur 16-bits et peut ainsi représenter 65536 caractères différents contrairement au caractère ASCII qui est limité à 256 caractères différents. L'Unicode a été prévu afin de pouvoir représenter les alphabets internationaux.

Même si deux désignations des types de données sont possibles en Visual Basic.NET. celle native à la plate-forme .NET prévoit d'avantage de types de données que celle native à Visual Basic. Ainsi, vous possédez la liberté d'utiliser les types de données de votre choix mais ceux de la plate-forme .NET exploitent mieux ce nouveau système. Par contre, les programmeurs des versions antérieures de Visual Basic seront certainement contents que leurs types de données aient été conservés pour fins de compatibilité antérieure.

Type de données VB	Type de données .NET	Taille	Plage de valeurs
N/a	System.IntPtr		Pointeur sur une variable ou sur une structure.
N/a	System.UIntPtr		Pointeur sur une variable ou sur une structure.
N/a	System.SByte	1 octet	Entiers signés de –128 à 127
N/a	System.GUID	16 octets	Globally Unique Identifier sur 128 bits.
N/a	System.UInt16	2 octets	Entier non-signé de 0 à 65535.
N/a	System.UInt32	4 octets	Entier non-signé de 0 à 4 294 967 295.
N/a	System.UInt64	8 octets	Entier non-signé de 0 à 184 467 440 737 095 551 615.
N/a	System.TimeSpan		Intervalle de temps.
N/a	System.Void	N/a	Rien.

Une fois la variable déclarée, le programmeur peut lui assigner une valeur de départ avant de l'utiliser. Le programmeur procède alors à l'**initialisation de la variable** :

```
Dim X As Integer
X = 1

Dim Age As System.Int16 = 1

Dim Prix As Single = 10.0

Dim Chaine As String = "Un"
```

La variable X est un entier initialisé à 1.

La variable Age est un entier initialisé à 1.

La variable Prix est un nombre à virgules initialisé à 10.0.

Si le programmeur n'initialise pas la variable déclarée, celle-ci se voit attribuer la valeur 0 par défaut ou chaîne vide `""` dans le cas des variables de type `String`.

Pour ceux et celles qui auraient travaillé à l'aide des versions antérieures de Visual Basic, notez qu'il est désormais possible d'initialiser les variables au sein d'une seule et même instruction un peu comme le permet depuis longtemps le C. Cependant, notez que la déclaration de variables `Dim a, b, c As Long` procède à la création de trois variables de type `Long` et non à la création de deux variables de type `Variant` et d'une dernière variable de type `Long` comme l'auraient faits les versions antérieures de Visual Basic. De plus, notez que le programmeur est désormais obligé de spécifier un type de données lors de la déclaration des variables.

Déclaration de constantes

Une constante est, comme la variable, un espace-mémoire au sein duquel il est possible d'y stocker une valeur. La constante diffère cependant de la variable du fait que sa valeur est attribuée à la compilation du programme et qu'elle, à partir de ce moment, ne peut plus jamais changer de valeur. La constante ne doit en aucun cas être perçue comme un sous-produit de la variable mais plutôt comme un aide-mémoire ayant pour objectif de faciliter la vie du programmeur. Par exemple, le code suivant utilise certains calculs mathématiques pour résoudre des problèmes de géométrie :

Notez que la valeur 3.14159265 de `PI` revient régulièrement au sein du code et que, de plus, elle est fastidieuse à mémoriser et à taper. Nous préférerons avantageusement déclarer une constante afin d'y stocker la valeur de `PI` qui n'a, de toutes façons, pas besoin de changer de valeur pendant la durée du programme. Une constante est déclarée à l'aide du mot-clé **`Const`** et est suivi de la valeur que doit prendre la constante :

```
Const PI = 3.14159265
```

Il est conseillé de précéder la déclaration de la constante de l'identificateur de portée. Ce sujet sera cependant abordé un peu plus loin au cours du présent chapitre.

```
Public Const PI = 3.14159265
```

Ainsi, il sera aisément possible d'accéder à la valeur de `PI` au sein du code à l'aide de la constante :

Sachez que le compilateur ne compilera pas les constantes mais procédera à une séance de `rechercher/remplacer` au sein du code afin de compiler afin de modifier toutes les occurrences de la constante par sa valeur réelle. Ainsi, les constantes ne sont véritablement que des outils permettant de faciliter et d'agrémenter la programmation. Servons-nous-en alors !

La valeur PI est également accessible en tant que constante intrinsèque à la plate-forme .NET sous l'appellation `Math.PI` mais nécessite l'importation de `System.Math`.

Les opérateurs

Les opérateurs sont des symboles réservés par le langage Visual Basic permettant une pluralité d'opérations arithmétiques, logiques, comparatives et autres. Ainsi, vous ne pouvez vous servir de ces symboles pour d'autres fins que celles prescrites par le langage.

Opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des opérations arithmétiques telles l'addition, la soustraction, la multiplication, etc.

Opérateur	Description	Exemple
^	Soulève un nombre à la puissance spécifiée.	`Z = 4 ^ 2` ' donne 16.
-	Précise l'inverse arithmétique d'un nombre ou d'une expression.	`Z = -(3 * 4)` ' donne –12.
*	Multiplie deux nombres.	`Z = 3 * 4`
/	Divise deux nombres en conservant les décimales le cas échéant.	`Z = 9 / 4` ' donne 2,25.
\	Divise deux nombres en tronquant les décimales le cas échéant.	`Z = 9 \ 4` ' donne 2.
Mod	Retourne le reste d'une division entière *(modulo)*.	`Z = 9 mod 4` ' donne 1.
+	Additionne deux nombres.	`Z = 4 + 3`
-	Soustrait deux nombres.	`Z = 4 – 3`

Tâchez de bien distinguer les deux opérateurs de division dont l'un deux ne considère pas les décimales le cas échéant et qui peut produire des résultats inattendus s'il est incorrectement utilisé.

Opérateur de concaténation

La concaténation est une opération qui s'applique sur des chaînes de caractères seulement. Cette opération permet d'embouter deux chaînes de caractères afin de n'en former qu'une seule.

```
Dim strPrnom As String = "Joe"
Dim strNom As String = "Castagnette"

MsgBox (strPrnom & " " & strNom)            ‘ Affiche "Joe Castagnette"
```

L'opérateur de concaténation ne s'appliquant que sur des chaînes de caractères, il procèdera à une conversion explicite en chaînes de caractères si des valeurs numériques lui sont fournies.

```
Dim X As Integer = 2
Dim Y As Integer = 6
MsgBox (X & Y)                              ‘ Affiche "26"
```

Opérateur	Description	Exemple
&	Concatène deux chaînes de caractères.	`strA = strPrnom & strNom`

Opérateurs de comparaison

Ces opérateurs permettent de comparer deux valeurs. Le résultat de la comparaison s'évalue toujours par **True** ou **False**. L'utilité des opérateurs de comparaison prendra tout son sens lorsque nous aborderons les structures de contrôle plus loin dans ce chapitre.

Opérateur	Description	Exemple
=	Retourne `True` si les deux valeurs sont égales.	`bResultat = x = 5`
<>	Retourne `True` si les deux valeurs sont différentes.	`bResultat = x <> 5`
>	Retourne `True` si la valeur de gauche est plus grande que la valeur de droite.	`bResultat = x > 5`
<	Retourne `True` si la valeur de gauche est plus petite que la valeur de droite.	`bResultat = x < 5`
>=	Retourne `True` si la valeur de gauche est plus grande ou égale à la valeur de droite.	`bResultat = x >= 5`
<=	Retourne `True` si la valeur de gauche est plus petite ou égale à la valeur de droite.	`bResultat = x <= 5`
Is	Retourne `True` si les deux objets sont les mêmes.	`bResultat = objX Is objY`

Opérateurs logiques

Tout comme les opérateurs de comparaison, le résultat généré par l'utilisation des opérateurs logiques s'évalue toujours par **True** ou **False**. L'utilité des opérateurs logiques prendra tout son sens lorsque nous aborderons les structures de contrôle plus loin dans ce chapitre.

Opérateur	Description	Exemple
Not	Retourne la négation logique de l'expression.	`bR = Not (X < 1)`
And	Retourne la conjonction logique de deux expressions.	`bR = (X > 2) And (Y =1)`
AndAlso	Retourne la conjonction logique de deux expressions sans évaluer la seconde si cela ne s'avère pas nécessaire.	`bR = (X > 2) AndAlso (Y =1)`
Or	Retourne la disjonction logique de deux expressions.	`bR = (X > 2) Or (Y =1)`
OrElse	Retourne la disjonction logique de deux expressions sans évaluer la seconde si cela ne s'avère pas nécessaire.	`bR = (X > 2) OrElse (Y =1)`
XOr	Retourne la disjonction logique exclusive de deux expressions.	`bR = (X > 2) XOr (Y =1)`
Eqv	Retourne l'équivalence logique de deux expressions.	`bR = 2 Eqv (6 / 3)`
Imp	Retourne l'implication logique de deux expressions.	`bR = 2 Imp objNull`

Voici les tables de vérité des opérateurs logiques. Les expressions `Expr1` et `Expr2` combinées à l'aide de l'opérateur concerné donnent le résultat spécifié dans la colonne `=`.

Not	
Expr	=
Faux	Vrai
Vrai	Faux

And		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

Or		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Vrai

XOr		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Faux

AndAlso		
Expr1	Expr2	=
Faux	*N\Eval*	Faux
Faux	*N\Eval*	Vrai
Vrai	Faux	Faux
Vrai	Vrai	Vrai

OrElse		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	*N\Eval*	Vrai
Vrai	*N\Eval*	Vrai

Eqv		
Expr1	Expr2	=
Faux	Faux	Vrai
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

Imp		
Expr1	Expr2	=
Faux	Faux	Vrai
Vrai	Faux	Faux
Vrai	*Null*	*Null*
Faux	Vrai	Vrai
Vrai	Vrai	Vrai
Faux	*Null*	Vrai
Null	Vrai	Vrai
Null	Faux	*Null*
Null	*Null*	*Null*

Les opérateurs `AndAlso` et `OrElse` sont particuliers du fait qu'ils peuvent outrepasser l'évaluation de la seconde expression si celle-ci s'avérerait inutile. Par exemple :

```
Dim X As Integer = 2
If (x > 4 AndAlso MsgBox("Certain ?", vbYesNo ) <> vbYes) Then

End If
```

La seconde expression ne sera pas évaluée puisque la première est fausse et que, conséquemment, il est assuré que l'expression entière sera évaluée négativement par un opérateur AND. Ainsi, la boîte de message ne serait jamais affichée ce qui n'est pas le cas si nous avions utilisé un opérateur AND.

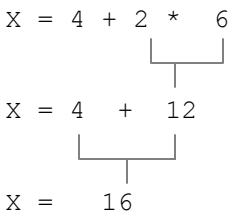
Ces opérateurs rejoignent, vous l'aurez remarqué, des techniques avancées de programmation.

Priorité des opérateurs

L'ensemble des opérateurs sont reliés selon un ordre de priorité très précise régissant l'ordre dans laquelle les différentes expressions doivent être évaluées afin d'en calculer le résultat. Ainsi, Visual Basic évaluera l'expression suivante :

```
X = 4 + 2 * 6
```

et obtiendra le résultat 16 et non le résultat 36. Le secret réside dans la priorité des opérateurs qui stipule entres autres que la multiplication est prioritaire à l'addition. Voyez comment Visual Basic évalue l'expression précédente :



Afin de s'assurer d'obtenir les résultats escomptés sans trop se soucier de la priorité des opérateurs, il suffit d'insérer explicitement des parenthèses qui demeurent les opérateurs possédant la plus forte priorité :

```
X = 4 + (2 * 6)          ' Donne 16 et le code est clair !
X = (4 + 2) * 6          ' Donne 36 et le code est clair !
```

Voici le tableau complet des opérateurs présentés en ordre décroissant de priorité.

Priorité	Opérateur(s)	Description(s)
14	( )	Parenthèses explicites.
13	^	Exposant.
12	-	Inverse arithmétique.
11	* / \	Multiplication, division réelle et division entière.
10	Mod	Modulo.
9	+ -	Addition et soustraction.
8	&	Concaténation de chaînes de caractères.
7	=, <>, >, <, >=, <=, Is	Comparaisons.
6	Not	Inverse logique.
5	And, AndAlso	Conjonction logique.
4	Or, OrElse	Disjonction logique.
3	XOr	Disjonction logique exclusive.
2	Eqv	Équivalence logique.
1	Imp	Implication logique.

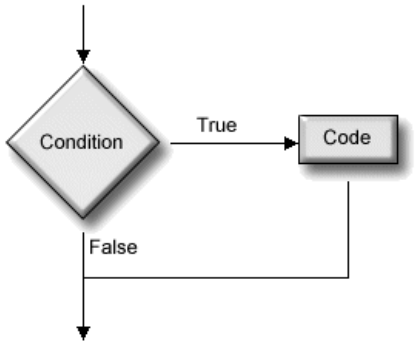
Structures de contrôle

L'exécution du code de votre code peut être contrôlé par diverses expressions appelées structures de contrôles. Les structures de contrôles sont réunies en deux groupes :

- Les **structures de branchement**, ou tests alternatifs, permettent d'exécuter un bloc de code lorsqu'une condition est rencontrée et d'exécuter un bloc de code différent lorsque la même condition n'est pas respectée.
- Les **structures répétitives**, ou itérations, permettent de répéter l'exécution d'un bloc de code tant qu'une condition est rencontrée ou qu'elle n'est pas respectée.

La structure de branchement If…Else…End If

Cette structure de branchement permet d'exécuter un bloc de code si une condition spécifiée est remplie.



Le code s'exécute seulement lorsque la condition peut positivement être évaluée par `True`. Dans tous les autres cas, aucun code supplémentaire n'est exécuté.

La syntaxe de cette structure de branchement est la suivante :

```
If <condition> Then
        <Code à exécuter>
End If
```

L'exemple suivant tire un nombre aléatoire entre 0 et 100 et en affiche la valeur. Lorsque X est un nombre pair, une boîte de dialogue le précise à l'utilisateur :

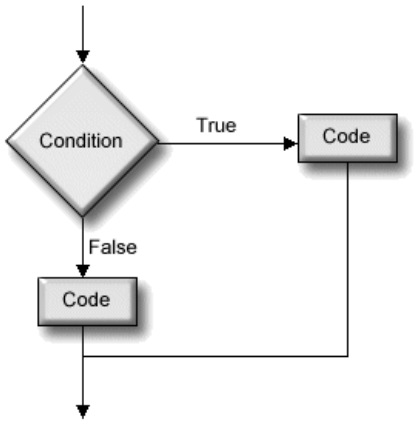
```
Randomize()
X = CInt(Rnd * 100)

MsgBox("X vaut " & X)
If (X Mod 2 = 0) Then
        MsgBox("X est un nombre pair." )
End If
```

Il est possible d'écrire un branchement `If` sur une seule ligne lorsque le code à exécuter est constitué d'une seule instruction au lieu d'un bloc d'instructions :

```
If (X Mod 2 = 0) Then MsgBox("X est un nombre pair.")
```

Cependant, cette syntaxe diminue la lisibilité du code et ne permet l'exécution d'une seule instruction au sein du bloc `If`. Notez que cette syntaxe ne nécessite pas l'insertion de l'instruction `End If` à la fin de la structure de branchement.

La structure peut être étendue afin d'intégrer un branchement alternatif **`Else`** dont le code ne s'exécutera que lorsque la condition ne sera pas rencontrée.



Le premier code s'exécute seulement lorsque la condition peut positivement être évaluée par `True`. Dans tous les autres cas, le second code est exécuté. Un des deux codes est nécessairement exécuté dans tous les cas et seulement un des deux codes est exécuté.

La syntaxe de cette structure de branchement est la suivante :

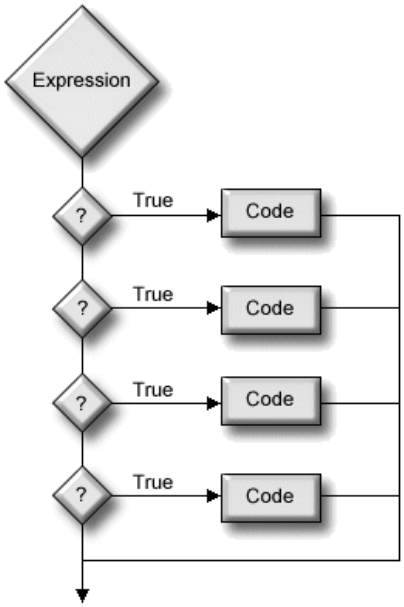
```
If <condition> Then

        <Code à exécuter>
Else

        <Code à exécuter>
End If
```

Voici le même exemple présenté précédemment à lequel un branchement `Else` a été ajouté :

```
Randomize
X = CInt(Rnd * 100)

MsgBox ("X vaut " & X)
If (X Mod 2 = 0) Then
        MsgBox ("X est un nombre pair.")
Else
        MsgBox ("X est un nombre impair.")
End If
```

La structure de branchement Select Case…End Select

Cette structure de branchement permet d'exécuter un bloc de code selon la valeur d'une expression spécifiée.



Un seul des blocs de code s'exécute selon la valeur de l'expression soumise à l'évaluation.

La syntaxe de cette structure de branchement est la suivante :

```
Select Case <expression>
  Case <valeur1>
       <Code à exécuter>

  Case <valeur2>
       <Code à exécuter>

  Case Else
       <Code à exécuter>

End Select
```

Si aucun des branchements ne correspond à l'expression spécifiée, le bloc facultatif `Case Else` est exécuté s'il a été spécifié.

L'exemple simple suivant montre l'utilisation de la structure `Select Case`. Notez que vous pouvez toujours insérer le nombre de blocs `Case` que désiré et que le bloc `Case Else` est facultatif.

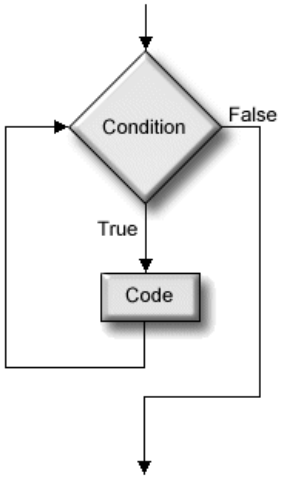
```
Select Case X
   Case 0
      MsgBox("X vaut 0")

   Case 1
      MsgBox("X vaut 1")

   Case 2
      MsgBox("X vaut 2")

   Case Else
      MsgBox("Désolé, la valeur est invalide!")
End Select
```

Voici un autre exemple de l'application de cette structure de branchement :

```
Select Case X
   Case 0 To 9
      MsgBox("X est entre 0 et 9.")

   Case 10 To 99
      MsgBox("X est entre 10 et 99.")

   Case Is > 99
      MsgBox("X est plus grand que 99.")
End Select
```

Structures répétitives Do While/Until … Loop

Ces structures répétitives sont utilisées afin de répéter une instruction ou un certain bloc de code.



Le code est exécuté tant que la condition est vraie *(boucle while)* ou jusqu'à ce que la condition soit vraie *(boucle Until)*. Lorsque la condition n'est plus remplie *(boucle while)* ou lorsqu'elle est remplie *(boucle Until)*, l'exécution quitte la boucle afin de poursuivre normalement le code suivant.

Les syntaxes de ces structures répétitives sont les suivantes :

```
Do While <condition>
      <Code à exécuter>
Loop


Do Until <condition>
      <Code à exécuter>
Loop
```

Voici un exemple simple de la mise en œuvre de ces structures répétitives. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
Dim nNbr As Integer = 0
Do While nNbr < 10
      MsgBox("nNbr vaut " & nNbr)
      nNbr += 1
Loop
```

Le même résultat peut être obtenu à l'aide de la boucle `Do Until` à la condition de modifier la condition :

```
Dim nNbr As Integer = 0
Do Until nNbr = 10
      MsgBox("nNbr vaut " & nNbr)
      nNbr += 1
Loop
```

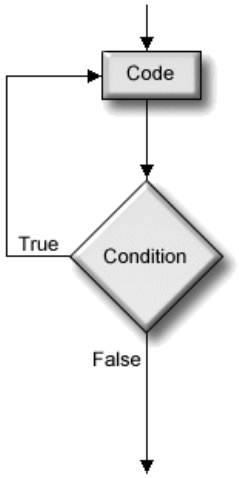
Il est possible de quitter prématurément une boucle à l'aide de l'instruction **Exit Do**. Lorsque cette instruction est rencontrée, la structure répétitive est quittée peu importe l'état de la condition gérant la boucle.

L'exemple suivant génère le même résultat que les deux boucles présentées précédemment :

```
Dim nNbr As Integer = 0
Do
      MsgBox("nNbr vaut " & nNbr)
      nNbr += 1

      If nNbr >= 10 Then Exit Do
Loop
```

Structures répétitives Do … Loop While/Until

Comme les structures présentées précédemment, ces structures répétitives sont utilisées afin de répéter une instruction ou un certain bloc de code.



Le code est exécuté tant que la condition est vraie *(boucle while)* ou jusqu'à ce que la condition soit vraie *(boucle Until)*. Lorsque la condition n'est plus remplie *(boucle while)* ou lorsqu'elle est remplie *(boucle Until)*, l'exécution quitte la boucle afin de poursuivre normalement le code suivant. Notez que, contrairement aux précédentes boucles `Do`, les présentes boucles exécutent le bloc de code au moins une fois puisque la condition est évaluée après que celui-ci ne soit exécuté.

Les syntaxes de ces structures répétitives sont les suivantes :

```
Do
      <Code à exécuter>
Loop While <condition>


Do
      <Code à exécuter>
Loop Until <condition>
```

Voici un exemple simple de la mise en œuvre de ces structures répétitives. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
Dim nNbr As Integer = 0
Do
      MsgBox("nNbr vaut " & nNbr)
      nNbr += 1
Loop While nNbr < 10
```

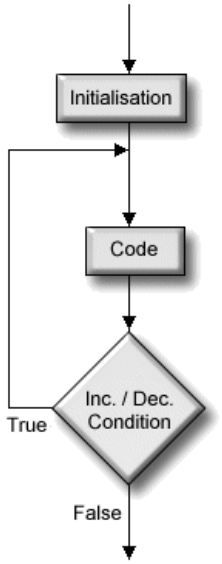
Le même résultat peut être obtenu à l'aide de la boucle `Do Until` à la condition de modifier la condition :

```
Dim nNbr As Integer = 0
Do
      MsgBox("nNbr vaut " & nNbr)
      nNbr += 1
Loop Until nNbr = 10
```

Ces structures répétitives acceptent également l'utilisation de l'instruction **Exit Do**.

Structure répétitive For… Next

Comme les structures présentées précédemment, cette structure répétitive est utilisée afin de répéter une instruction ou un certain bloc de code. La boucle `For… Next` présente cependant l'avantage d'inclure un compteur permettant d'initialiser le nombre d'itérations à exécuter.



Le compteur est d'abord initialisé ainsi que la valeur maximale ou minimale qu'il doit atteindre. Le code est ensuite répété tant et aussi longtemps que le compteur n'a pas atteint ou dépassé la valeur prescrite lors de l'initialisation de la boucle. Le compteur est automatiquement incrémenté ou décrémenté après l'exécution du code.

La syntaxe de cette structure répétitive est la suivante :

```
For <compteur> = <Valeur1> To <Valeur2>
      <Code à exécuter>
Next
```

La variable servant de compteur doit préalablement être déclarée.

Voici un exemple simple de la mise en œuvre de cette structure répétitive. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
For nCpt = 0 To 9
      MsgBox("nCpt vaut " & nCpt)
Next
```

Il est possible de créer des boucles inversées au sein desquelles la valeur du compteur est automatiquement décrémentée au lieu d'être incrémentée *(par défaut)* en spécifiant une valeur négative à l'instruction facultative **Step**. L'exemple suivant affiche les valeurs entre 0 et 9 inclusivement mais débute en affichant 9 avant de terminer par 0.

```
For nCpt = 9 To 0 Step -1
      MsgBox("nCpt vaut " & nCpt)
Next
```

De la même manière il est possible de créer des boucles dont l'incrémentation ou la décrémentation du compteur s'effectue par multiple de 2, de 3 ou de toute autre valeur. L'exemple suivant affiche les nombres pairs compris entre 0 et 20 inclusivement :

```
For nCpt = 0 To 20 Step 2
      MsgBox(nCpt & " est un nombre pair")
Next
```

Il est possible de quitter prématurément la boucle `For… Next` en introduisant l'instruction **Exit For** au sein du code à exécuter.

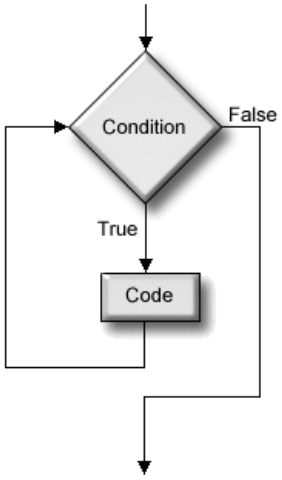
Structure répétitive For Each… Next

Comme les structures présentées précédemment, cette structure répétitive est utilisée afin de répéter une instruction ou un certain bloc de code mais utilise le nombre d'objets membres d'une collection afin d'initialiser le nombre le nombre d'itérations à effectuer. Ces concepts orientés objets vous seront exposés plus tard.

```
For Each File in objFiles
      MsgBox(File.Name)
Next
```

Structure répétitive While… Wend

Comme les structures présentées précédemment, cette structure répétitive est utilisée afin de répéter une instruction ou un certain bloc de code. Cependant, cette structure existe depuis les débuts du langage *Basic* et est supplantée par les autres formes de boucles. La boucle `While…Wend` a été conservée pour des fins de compatibilité antérieure mais il est conseillé d'utiliser les autres structures répétitives mieux structurées et plus flexibles.



Le code est exécuté tant que la condition est vraie. Lorsque la condition n'est plus remplie, l'exécution quitte la boucle afin de poursuivre normalement le code suivant.

La syntaxe de cette structure répétitive est la suivante :

```
While <condition>
      <Code à exécuter>
Wend
```

Voici un exemple simple de la mise en œuvre de cette structure répétitive. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
Dim nNbr As Integer = 0
While nNbr < 10
      MsgBox("nNbr vaut " & nNbr)
      nNbr += 1
Wend
```

Focus sur la condition

La condition qui régi l'exécution ou non d'un bloc de code au sein d'une structure de contrôle est toujours évaluée par une valeur booléenne `True` ou `False`. Visual Basic réduira toujours à ce niveau la valeur de la condition que votre code lui soumet. Dans l'exemple suivant, la condition est simplement composée d'une valeur numérique :

```
Dim nVar As Integer = 14
Do While nVar
      MsgBox ("nVar vaut " & nVar)
      nVar -= 1
Loop
```

Or, ce code affiche les différentes valeurs de la variable `nVar` de 14 à 1 puis le code cesse de s'exécuter. Pourquoi ? Ce comportement s'explique aisément par la façon qu'ont les langages de programmation d'évaluer les conditions selon l'algorithme booléen qui stipule que :

- Une **valeur nulle** (0) s'évalue négativement et prend la valeur **False**.
- **Toute autre valeur** s'évalue positivement et prend la valeur **True**.

Ainsi, l'exemple précédent évalue positivement la condition de la boucle tant que la variable `nVar` possède une valeur différente de zéro.

Il existe également quelques pièges qu'il faut faire attention d'éviter lors de la construction de boucles et de conditions. Un d'eux provient souvent de la méprise du programmeur à l'égard des opérateurs logiques AND et OR. Dans l'exemple suivant, le code attend une réponse de l'utilisateur avant de poursuivre l'exécution. À l'aide d'une boucle, le code s'assure que l'utilisateur ne puisse entrer que deux réponses possibles… ou, du moins, il croit le faire :

```
'********* Ce code est erroné… ne l'essayez-pas **********'
Do
      Reponse = InputBox("Désirez-vous continuer ? (Oui, Non)")
Loop While LCase(Reponse) <> "oui" OR LCase(Reponse) <> "non"
```

Cependant, la table de vérité de l'opérateur OR stipule que l'une des deux parties de l'expression évaluée doit être vraie pour que l'expression s'évalue positivement. Ainsi, si l'utilisateur entre la réponse *"non"*, le script évalue d'abord `Reponse <> "oui"` et conclu, en effet que cette affirmation est vraie et, concluant avec raison que la condition est vraie, procède à une nouvelle itération de la boucle. Le code aurait donc dû se lire comme suit :

```
Do
      Reponse = InputBox("Désirez-vous continuer ? (Oui, Non)")
Loop While LCase(Reponse) <> "oui" AND LCase(Reponse) <> "non"
```

Un autre piège provient des conditions mal construites provoquant des boucles infinies. Examinez l'exemple suivant et notez que la condition de cette boucle ne pourra jamais être rencontrée.

```
'********* Ce code est erroné… ne l'essayez-pas **********'
Dim nVar As Integer = 0
Do Until nVar > 10
      Y = nVar * X
Loop
```

Cet exemple illustre bien le phénomène des boucles infinies. La condition ne pouvant jamais être rencontrée, le code s'exécute éternellement et fait rapidement grimper à 100% le niveau d'utilisation du processeur.

Imbrication de structures de contrôles

Visual Basic permet au programmeur d'imbriquer les différentes structures de contrôles afin de donner plus de latitude à son code. L'imbrication de structures de contrôles emploie le principe dit LIFO *(Last-In, First-Out)* stipulant que la dernière structure de contrôle imbriquée doit être la première à se terminer. Voici l'exemple d'un code permettant de connaître tous les nombres premiers compris entre 1 et 100 :

```
For X = 1 To 100
      Premier = True
      For Y = 2 To X
            If (X Mod Y = 0) AND (X <> Y) Then
                  Premier = False
                  Exit For
            End If
      Next
      If Premier Then MsgBox(X & " est un nombre premier.")
Next
```

Notez que chacune des structures de contrôle se termine avant que ne se termine la structure de contrôle parent.

Utilisation des fonctions

Visual Basic.NET hérite de ses ancêtres *Basic* et *Visual Basic* d'une multitude de fonctions procurant une grande flexibilité à ce langage. Ces fonctions sont intrinsèques au langage et ne nécessite donc pas que le script référence un modèle d'objets externe tel que vous apprendrez à les utiliser dans les prochains chapitres. Lorsque votre code invoque une fonction, votre code demande au code qui y est contenu de s'exécuter.

Il est possible d'invoquer une fonction ou une procédure simplement en la nommant :

```
NomFonction()
```

Cet appel d'une fonction met en oeuvre la syntaxe la plus simple puisque la fonction ne nécessite pas que des paramètres d'entrée ne lui soient spécifiés. Les paramètres représentent des informations supplémentaires que la fonction pourrait attendre afin de préciser certains aspects de l'appel. Ainsi, lorsque vous demandez l'exécution de la commande *`Format`* afin de formater un disque, vous devez spécifiez la lettre du lecteur qui doit être formaté. Il s'agit là d'un paramètre, d'une information supplémentaire nécessaire pour l'exécution de la commande. Voici la syntaxe permettant d'invoquer une fonction ou une procédure et de préciser les paramètres attendus par celle-ci :

```
NomFonction  (Param1, Param2, ParamN)
```

Remarquez que chacun des paramètres sont séparés les uns des autres par une virgule (,). Notez également que certaines fonctions attendent des paramètres optionnels. Ainsi, si vous spécifiez une valeur pour ces paramètres, la fonction utilisera cette valeur mais utilisera une valeur définie par défaut à l'interne si aucune valeur n'est spécifiée pour ces paramètres.

Finalement, une fonction peut renvoyer une valeur de retour. Une valeur de retour peut être le fruit d'un calcul ou d'une opération quelconque. Une valeur de retour peut également être une information indiquant si la fonction s'est exécutée correctement ou si elle a échouée. Vous retrouverez au sein du présent chapitre la signification des valeurs de retour des fonctions intrinsèques à Visual Basic.NET. Voici la syntaxe permettant d'invoquer une fonction et d'en récupérer la valeur de retour :

```
MaVariable = NomFonction()
```

ou, dans le cas de méthodes paramétrables :

```
MaVariable = NomFonction(Param1, Param2, ParamN)
```

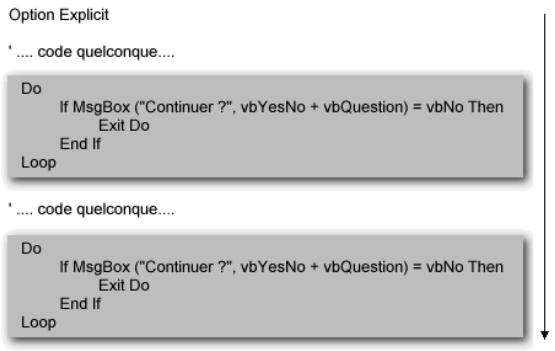
Dans laquelle syntaxe la valeur de retour de la méthode a été stockée au sein de la variable *`MaVariable`*. Votre code n'est pas obligé de récupérer les valeurs de retour des fonctions.

Création de fonctions et procédures

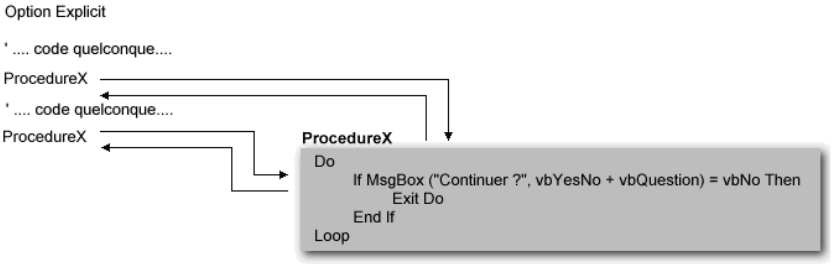
Lorsque vous écrirez vos codes, vous rencontrerez certainement des situations dans lesquelles vous êtes forcé de répéter intégralement ou à quelques différences près un code précédemment tapé. La première fois que cette situation se présente, c'est comique… la seconde fois, c'est sympathique mais, ensuite, la situation devient rapidement lassante de par sa redondance. De plus, le fait de copier un même code à plusieurs endroits peu provoquer des ennuis plus sérieux. Si pour une raison ou une autre un code déjà copié à maints autres endroits devait être modifié, ces mêmes modifications devraient obligatoirement être appliquées aux autres duplicata de ce code. Le mot *redondance* prend alors tout son sens.

Heureusement, Visual Basic vous permet d'encapsuler tout code redondant au sein de fonctions ou de procédures afin de pouvoir aisément le réutiliser ultérieurement.

Le schéma ci-contre représente un code au sein de lequel le même segment de code a été copié et répété. Si une modification était à apporter au premier segment, cette modification devrait également s'effectuer sur le second segment.



Le schéma ci-contre représente un code au sein de lequel un segment de code a été intégré au sein d'une procédure. Si une modification était à apporter au code de cette procédure, cette modification serait automatiquement perceptible lors de chacun des appels de procédure.



Vous procéderez à l'exécution de vos procédures et fonctions personnalisées de la même manière que celle vous permettant d'exécuter les fonctions intrinsèques à Visual Basic tel que présenté précédemment.

Création de procédures

Les procédures contiennent du code que vous pouvez réutiliser régulièrement au sein de votre code sans avoir à le réécrire ou le copier. À la place, vous inscrivez le code redondant au sein d'une procédure et en demandez l'exécution à chaque endroit désiré au sein de votre code.

La procédure doit préalablement être déclaré avant de pouvoir être référencé par le code. La déclaration d'une procédure s'effectue à l'aide de la syntaxe suivante :

```
Private|Public Sub NomProcédure ( [ListeArguments] )
        < Instructions>
End Sub
```

Voici un exemple au sein duquel les quatre opérations arithmétiques de base sont appliquées sur un nombre quelconque contenu au sein de la variable `N`. Remarquez l'appel à la procédure `Continuer` exécutée afin de confirmer l'intention de l'utilisateur de continuer l'exécution du code.

```
Dim N As Long =  3

MsgBox ("N + N = " & N + N )
Continuer

MsgBox ("N - N = " & N - N )
Continuer

MsgBox ("N * N = " & N * N )
Continuer

MsgBox ("N / N = " & N / N )


'******************** Procédure Continuer()  *****************'
Sub Continuer()
  Dim Reponse As String
  Do
      Reponse = LCase(InputBox("Continuer? (o)ui, (n)on"))
  Loop While (Reponse <> "o" And Reponse <> "n")
  If Reponse = "n" Then Application.Exit()
End Sub
```

Il est possible de quitter prématurément une procédure en inscrivant simplement l'instruction **`Exit Sub`**.

Création de fonctions

Les fonctions contiennent du code que vous pouvez réutiliser régulièrement au sein de votre code sans avoir à le réécrire ou le copier de manière plus flexible qu'une procédure puisque les fonctions permettent de retourner une valeur de retour. La procédure doit préalablement être déclaré avant de pouvoir être référencé par le code. La déclaration d'une procédure s'effectue à l'aide de la syntaxe suivante :

```
Private|Public Function NomFonction ( [ListeArguments] ) As TypeDonnées

    < Instructions>
    Return valeur

End Function
```

Voici un exemple au sein duquel les quatre opérations arithmétiques de base sont appliquées sur un nombre quelconque contenu au sein de la variable `N`. Remarquez l'appel à la fonction `Continuer` exécutée afin de confirmer l'intention de l'utilisateur de continuer l'exécution du code.

```
Dim N As Long = 3

MsgBox ("N + N = " & N + N )
If Continuer() = False Then Application.Exit

MsgBox ("N - N = " & N - N )
If Continuer() = False Then Application.Exit

MsgBox ("N * N = " & N * N )
If Continuer() = False Then Application.Exit

MsgBox ("N / N = " & N / N )


'******************** Procédure Continuer()  *****************'
Function Continuer() As Boolean
  Dim Reponse As String
  Do
      Reponse = LCase(InputBox("Continuer? (o)ui, (n)on"))
  Loop While (Reponse <> "o" And Reponse <> "n")
  Return Reponse = "o"
End Function
```

Il est possible de quitter prématurément une fonction en inscrivant simplement l'instruction **`Exit Function.`**

Bref, une fonction se distingue d'une procédure par sa capacité de retourner une valeur de retour. Notez que cette valeur de retour est précisée au sein de la fonction en assignant une valeur au nom de la fonction tel qu'on le ferait avec une variable :

```
fncAddition = x
```

ou en utilisant l'instruction `Return` :

```
Return x
```

Surcharge de procédures et de fonctions

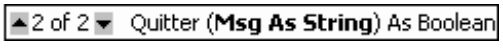
Lorsque le compilateur compile l'ensemble des procédures et des fonctions de votre programme, il les distingue à l'aide de leur prototype qui doit leur être propre et unique. Le prototype d'une procédure ou d'une fonction référence généralement le nom de la procédure, le nombre de paramètres attendus ainsi que le type de données de chacun de ceux-ci. La surcharge d'une procédure est un concept voulant qu'une procédure puisse posséder le même nom qu'une autre procédure dans la mesure où ces deux procédures présentent un prototype différent et diffèrent donc de par le nombre de paramètres attendus ou de par les types de données de ces derniers. La surcharge de procédures ou de fonctions est utile afin de permettre au programmeur de créer plusieurs procédures ou fonctions portant le même nom et servant les mêmes fins mais attendant des paramètres différents.

Avec la version .NET, Visual Basic permet désormais la surcharge de procédures et de fonctions. Il est possible de préciser qu'une fonction ou une procédure sera surchargée par une autre du même nom en incluant le **mot-clé** `Overloads` au sein de sa déclaration :

```
Public Overloads Function Quitter() As Boolean
      Application.Exit()
End Function


Public Overloads Function Quitter(ByVal Msg As String) As Boolean
      MsgBox(Msg, MsgBoxStyle.Information, "Quitter")
      Application.Exit()
End Function
```

Et cette pratique – bien connue des programmeurs C++ – est tout à fait légale. Lorsque la fonction `Quitter` sera invoquée au sein du code, le compilateur saura distinguer les deux fonctions du même nom puisqu'elles se différencient par leur prototype unique.

Lorsque vous tapez le nom d'une fonction au sein de votre code afin de l'invoquer, l'outil *Intellisense* de Visual Studio.NET vous affiche le prototype de la fonction correspondante au sein d'une info-bulle. Lorsque cette fonction est surchargée, vous verrez s'afficher deux petites flèches au sein de l'info-bulle vous permettant de naviguer parmi les différentes surcharges de la fonction afin de sélectionner celle que vous désirez.



Portée des variables

La portée d'une variable représente les limites au delà desquelles la variable n'est plus accessible au sein du code. Ainsi, certaines variables seront accessibles dans l'ensemble de votre code tandis que d'autres ne seront accessibles qu'à l'intérieure d'une procédure ou d'une fonction. Ce concept permet au programmeur d'éviter quantité d'erreurs imprévisibles et difficiles à diagnostiquer.

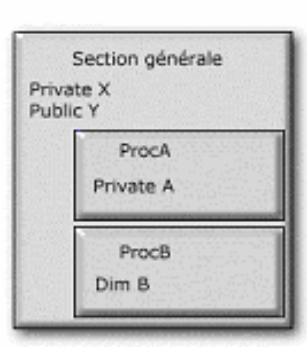
Le schéma ci-contre représente la disposition du code au sein d'un code *Visual Basic*. Le code dans son entier peut être nommé **Section générale** et peut contenir des procédures ou des fonctions.

Certaines variables ne seront accessibles qu'à l'intérieur d'une procédure donnée tandis que d'autres variables seront accessibles dans l'ensemble de la section générale et des procédures la composant.

La portée d'une variable est définie par le mot-clé utilisé de la déclaration de celle-ci :

- Les mot-clés **Dim** et **Private** déclarent une variable privée qui ne sera accessible qu'à l'intérieur du module au sein duquel elle est déclarée. Ainsi, si la variable est déclarée au sein d'une procédure, elle ne sera utilisable que par le code interne à cette procédure et demeurera invisible au reste du code. Cependant, si cette variable est déclarée dans la section générale du code, elle sera visible à l'ensemble des procédures et fonctions de ce code.

- Le mot-clé **Public** déclare une variable publique qui sera accessible au module au sein duquel elle est déclarée et à l'ensemble des procédures et fonctions lui appartenant. Ainsi, si cette variable est déclarée dans la section générale du script, elle sera visible à l'ensemble des procédures et fonctions de ce code.



Le schéma ci-contre représente la déclaration de quatre variables au sein d'un code.

Puisque déclarées au niveau de la section générale, les variables `X` et `Y` sont accessibles dans l'ensemble du code.

Puisque déclarées au niveau d'une procédure ou d'une fonction, les variables `A` et `B` ne sont accessibles que dans leur procédure ou fonction respective.

Si, par exemple, la procédure `ProcA` tentait d'accéder à la variable `B` déclarée au sein de la procédure `ProcB`, un message d'erreur afficherait que la variable n'a pas été déclarée. Cependant, la procédure ProcA peut sans encombre accéder à la variable `X` déclarée au sein de la section générale du code.

De manière générale, si votre variable doit être accessible de l'ensemble des procédures et fonctions, déclarez celle-ci au sein de la section générale du code. Autrement, déclarez vos variables à l'intérieur des procédures au sein desquelles l'accès à cette variable est nécessaire.

Passage de paramètres

Les exemples précédents de procédures et fonctions ne vous permettaient pas de préciser des paramètres. Or, le passage de paramètres est une pratique fort utile dans l'élaboration de procédures et de fonctions flexibles et réutilisables.

Le passage de paramètres permet à une procédure ou à une fonction de réagir différemment selon la valeur des paramètres qu'elle a reçue. Par exemple, je peux demander à une calculatrice d'effectuer une addition sur deux valeurs numériques 7 + 4. Cependant, je peux demander à cette même calculatrice d'effectuer une même addition mais, cette fois, avec deux valeurs différentes : 6 + 5. Les instructions que doit exécuter la calculatrice demeurent les mêmes lors de l'exécution des deux opérations pourtant différentes puisque seules les valeurs changent. Dans de telles situations, le programmeur a intérêt à créer une fonction qui, comme la calculatrice, exécutera toujours les mêmes instructions en y modifiant seulement les valeurs.

Vous pouvez déclarer une fonction ou une procédure et spécifier qu'elle nécessite le passage de certains paramètres afin de s'exécuter adéquatement en précisant ces paramètres entre les parenthèses de la déclaration de la fonction ou de la procédure et en séparant les différents paramètres par des virgules :

```
Function|Sub NomProcedure (Param1 [,Param2] [,ParamN])
```

L'exemple suivant montre une fonction qui permettrait de prendre deux nombres et d'en retourner la somme :

```
Public Function Addition (ByVal Nbr1 As Short, ▼
                          ByVal Nbr2 As Short) As Integer
      Addition = Nbr1 + Nbr2
End Function
```

On aurait également pu écrire:

```
Public Function Addition (ByVal Nbr1 As Short, ▼
                          ByVal Nbr2 As Short) As Integer
      Return Nbr1 + Nbr2
End Function
```

Cette fonction pourrait maintenant être exécutée au sein du script à l'aide de l'instruction suivante :

```
Dim Somme
Somme = Addition(2, 4)          ' Somme vaut désormais 6.
```

Les paramètres peuvent être attendus par une procédure ou une fonction selon deux modes :

- **Par valeur** *(ByVal)* indique qu'une copie de la valeur est effectuée au sein de la nouvelle variable qu'est le paramètre. Puisque le paramètre est une nouvelle variable possédant une copie de la valeur, si ce premier est altéré, la valeur originelle n'en subira aucun effet. Ce mode de passage des paramètres devrait être celui que vous utiliserez le plus régulièrement afin d'éviter toute erreur difficile à diagnostiquer.

- **Par référence** *(ByRef)* indique que l'adresse de la valeur est assignée à la nouvelle variable qu'est le paramètre. Puisque le paramètre est pointe directement sur la valeur, si ce premier est altéré, la valeur originelle en subira les mêmes modifications, phénomène appelé effet de bord.

Le mode de passage des paramètres est précisé dans le prototype de la procédure ou de la fonction à l'aide des mots-clés `ByVal` et `ByRef`. Si aucun mot-clé n'est précisé par le programmeur, *Visual Basic.NET* assumera le mode `ByVal` par défaut :

`Private|Public Sub` *NomProcédure* `(ByBal|ByRef` *Param* `As` *Type*`)`

Par exemple, la fonction suivante ne pose aucun problème puisque les paramètres sont passés par valeur. Ainsi, lorsque le code de la fonction altère la valeur de `N1`, la valeur originelle n'est pas modifiée mais seul le paramètre local à la fonction l'est :

```
Public Function Additionner (ByVal N1 As Short, ByVal N2 As Short) ▼
                                                          As Integer
        N1 += N2
        Return N1

End Function

'****************************************************************'
Dim xA, xB As Short
Dim xReponse As Integer

xA = 5
xB = 6
xReponse = Additionner(xA, xB)
```

À la fin de l'exécution de ce code, `xReponse` possède correctement la valeur 11 et les deux valeurs passées en paramètres *(`xA` et `xB`)* ont conservé leur valeur originelle. Cela est explicable du fait qu'elles ont été copiées au sein de nouvelles variables avant d'être utilisées au sein de la fonction ou de la procédure.

Ceux qui auraient utilisé les versions antérieures de Visual Basic remarqueront que Visual Basic.NET considère le passage de paramètres par valeur comme mode de passage de paramètres par défaut contrairement à ses prédécesseurs qui, par défaut, considéraient plutôt le passage de paramètres par référence.

Cependant, il en est tout à fait autrement pour l'exemple suivant où les paramètres sont passés par référence et où les valeurs originelles sont modifiées par le code de la fonction :

```
Public Function Additionner (ByRef N1 As Short, ByRef N2 As Short) ▼
                                                                  As Integer
        N1 += N2
        Return N1

End Function

'*****************************************************************'
Dim xA, xB As Short
Dim xReponse As Integer

xA = 5
xB = 6
xReponse = Additionner(xA, xB)
```

Si on faisait ici afficher la valeur de la variable `xA`, celle-ci vaudrait également 11 puisqu'elle a été altérée du fait que le paramètre par référence `N1` a altéré sa valeur.

Quoique les paramètres passés par référence représente un danger certain pour les programmeurs novices, ce mode de transmission des paramètres permet entres autres à une procédure ou à une fonction de retourner plusieurs valeurs de retour simultanément :

```
Public Sub MultiExposant(ByRef N1 As Short, ByRef N2 As Short)

      Dim x1 As Short = N1
      Dim x2 As Short = N2

      N1 = x1 ^ x2
      N2 = x2 ^ x1

End Sub


'*****************************************************************'
Dim xA, xB As Short

xA = 2
xB = 3

MultiExposant(xA, xB)

MsgBox("2 à la 3 vaut " & xA)    '  Affiche 8
MsgBox("3 à la 2 vaut " & xB)    '  Affiche 9
```

Paramètres optionnels

Il est possible qu'une fonction ou une procédure attende un paramètre optionnel lorsqu'une valeur par défaut peut aisément lui être appliquée dans la majorité des appels de cette fonction ou de cette procédure.

Par exemple, on pourrait venir à la conclusion qu'une fonction qui calculerait le salaire hebdomadaire selon le nombre d'heures travaillées et le salaire horaire de l'employé puisse appliquer le calcul sur une semaine de 40 heures dans la grande majorité des cas sauf exceptions. Dans ce cas, si la valeur de ce paramètre était omise, la fonction appliquerait la valeur par défaut pour ce paramètre et ne causerait aucune erreur. Cependant, si la valeur de ce paramètre était fournie, cette dernière serait utilisée et aurait préséance sur la valeur par défaut.

On peut déclarer des paramètres optionnels au sein de notre déclaration de procédure ou de fonction en incluant le mot-clé `Optional`. Dans ce cas, il est nécessaire de fournir une valeur par défaut :

```
Public Function CalcSalaire(  ByVal SalHoraire As Single,
                              Optional ByVal NbrHr As Short = 40
                            ) As Single
```

Ainsi, cette fonction pourrait être invoquée de la manière suivante :

```
Dim SalaireHr As Single
Dim SalaireHebdo As Single

SalaireHr  = 18.55
SalaireHebdo  = CalcSalaire( SalaireHr )

'********** SalaireHebdo vaut 18.55 * 40 = 742.00  **********'
```

autant que de la manière suivante :

```
Dim SalaireHr As Single
Dim SalaireHebdo As Single

SalaireHr  = 18.55
SalaireHebdo  = CalcSalaire( SalaireHr, 35 )

'********** SalaireHebdo vaut 18.55 * 35 = 649.25  **********'
```

Notez que si le mot-clé `Optional` est utilisé, tous les paramètres suivants, s'il y a, doivent également être facultatifs et être déclarés à l'aide du mot clé `Optional`. Finalement, les paramètres optionnels doivent être déclarés à la fin de la liste des paramètres attendus par la procédure ou la fonction.

Ceux qui auraient utilisé les versions antérieures de Visual Basic remarqueront que Visual Basic.NET ne tolère plus l'omission de paramètre sans valeur par défaut. Ainsi, la fonction IsMissing qui permettait de tester l'existence de paramètres a disparu du langage Visual Basic depuis .NET.

Création et manipulation de tableaux

Différentes variables de même type de donnée peuvent être stockées conjointement sous l'appellation d'une même variable afin de faciliter l'accès à des valeurs similaires. Quoiqu'on puisse parler de variables vectorielles (en opposition aux variables scalaires traditionnelles), on utilisera plus généralement le terme tableau *(Array)* afin d'identifier ce type de regroupement.

La déclaration d'un tableau s'effectue simplement en spécifiant l'index supérieur désiré du tableau entre parenthèses lors de la déclaration de la variable vectorielles :

```
Dim MonTableau(9) As Integer
```

Cette déclaration provoque la création d'un tableau nommé `MonTableau` possédant 10 entiers accessibles à l'aide des index de 0 à 9. Notez que la déclaration suivante est équivalente à la précédente même si les parenthèses sont insérées après la déclaration du type de données :

```
Dim MonTableau As Integer(9)  'Équivalant au précédent
```

Dans les deux cas, vous obtenez un tableaux de dix entiers pouvant être représenté comme suit. Notez que l'ensemble des valeurs du tableau est initialisée à zéro comme le serait une variable scalaire sous *Visual Basic.NET* :

	Index	Contenu	
MonTableau	0	0	MonTableau(0)
	1	0	MonTableau(1)
	2	0	MonTableau(2)
	3	0	MonTableau(3)
	4	0	MonTableau(4)
	5	0	MonTableau(5)
	6	0	MonTableau(6)
	7	0	MonTableau(7)
	8	0	MonTableau(8)
	9	0	MonTableau(9)

Sous l'ensemble des langages de *Visual Studio.NET*, les tableaux sont indexés à partir de zéro, c'est-à-dire que le premier élément du tableau sera référencé à l'aide de l'index zéro :

```
Dim MonTableau(9) As Integer  'Tableau de 10 éléments
Dim Premier As Integer = 68
MonTableau(0) = Premier
```

De même manière, le dernier élément du tableau sera référencé à l'aide du nombre d'éléments contenus au sein du tableau moins un :

```
Dim MonTableau(9) As Integer  'Tableau de 10 éléments
Dim Dernier As Integer = 92
MonTableau(9) = Dernier
```

Ainsi, le code suivant permet d'assigner une valeur au cinquième élément du tableau:

```
Dim MonTableau(9) As Integer  'Tableau de 10 éléments
Dim Cinquieme As Integer = 43
MonTableau(4) = Cinquieme
```

Et nous obtiendrions finalement le tableau suivant après exécution des trois codes précédents :

	Index	Contenu	
MonTableau	0	68	MonTableau(0)
	1	0	MonTableau(1)
	2	0	MonTableau(2)
	3	0	MonTableau(3)
	4	43	MonTableau(4)
	5	0	MonTableau(5)
	6	0	MonTableau(6)
	7	0	MonTableau(7)
	8	0	MonTableau(8)
	9	92	MonTableau(9)

Il est donc aisé de parcourir l'ensemble des éléments d'un tableau à l'aide de boucles. Notez, à ce sujet, que la boucle `For … Next` est spécialement bien adaptée à la gestion de tableaux. L'exemple suivant assigne une lettre de A à Z aux différents éléments du tableau :

```
Dim Lettres As Char(25)
Dim N As Short

For N = 0 To 25
      Lettres(N) = Chr(N + 65)
Next
```

Tandis que l'exemple suivant affiche consécutivement le contenu de chacun des éléments du tableau précédent :

```
For N = 0 To 25
      MsgBox ( Lettres(N) )
Next
```

Vous pouvez passer un tableau en tant que paramètre à une fonction ou une procédure comme s'il s'agissait d'une variable scalaire. La déclaration du paramètre nécessitera cependant l'inclusion des parenthèses précisant les dimensions du tableau attendu comme l'illustre l'exemple suivant.

```
Dim Lettres As Char(25)
Dim N As Short

For N = 0 To 25
      Lettres(N) = Chr(N + 65)
Next

Afficher(Lettres)
```

```
Private Sub Afficher (ByVal Ltr As Char(25))
      Dim K As Short
      For K = 0 To 25
            MsgBox(Ltr(K))
      Next
End Sub
```

Tableaux dynamiques

Il est souvent impossible de connaître à l'avance le nombre d'éléments qu'un tableau devra posséder tout au long de l'exécution du code. Ainsi, si on désire créer un code qui emmagasinera les informations concernant l'ensemble des dossiers d'un disque au sein d'un tableau, le code devra prévoir un tableau aux dimensions variables qui saura s'ajuster au nombre de dossiers présents sur la machine sur laquelle il s'exécute. Le code doit alors procéder à la création d'un tableau dynamique.

La déclaration d'un tableau dynamique s'effectue simplement en omettant de spécifier toute dimension au tableau lors de la déclaration de celui-ci :

```
Dim MonTableau() As Integer
```

Notez que la déclaration suivante est équivalente à la précédente même si les parenthèses sont insérées après la déclaration du type de données :

```
Dim MonTableau As Integer()   'Équivalant au précédent
```

Dans les deux cas, la déclaration provoque la création d'un tableau dynamique nommé `MonTableau` ne possédant aucun entier et ne pouvant conséquemment pas être utilisé sans qu'une erreur ne soit soulevée. Il est cependant possible d'initialiser certaines valeurs du tableau dynamique lors de sa déclaration en spécifiant celles-ci entre accolades, chacune d'elle séparée de la suivante par une virgule :

```
Dim MonTableau() As Integer = { 1, 3, 9, 2, 7 }
```

Il est certain que le tableau possède désormais cinq éléments mais il demeure un tableau dynamique extensible tout au long de l'exécution du code qui suivra.

Une fois le tableau dynamique déclaré, il est possible d'en modifier les dimensions à l'aide de l'instruction `Redim`. Cette instruction redimensionne le tableau spécifié en l'amenant à atteindre l'index supérieur spécifié en tronquant ou en étendant les dimensions de celui-ci.

```
Dim MonTableau() As Integer   'Tableau dynamique vide

Redim MonTableau(9)     'Le tableau possède maintenant 10 items
```

Cependant, l'instruction `Redim` provoque le fâcheux résultat de supprimer l'ensemble des valeurs existantes au sein du tableau visé à moins qu'elle ne soit accompagnée du mot-clé `Preserve` qui s'assurera que le tableau conserve ses valeurs.

```
Redim Preserve MonTableau(9)  'Les éléments conservent leur valeur
```

L'exemple suivant saisi des valeurs entières auprès de l'utilisateur tant que ce dernier ne saisi pas une valeur nulle (0). Alors, une moyenne des valeurs saisies est calculée et affichée. Remarquez surtout comment le tableau est redimensionné à chaque fois que l'utilisateur saisi une nouvelle valeur.

```
Dim MesValeurs() As Integer   'Tableau dynamique vide
Dim NbrValeurs As Integer     'Conserve le nombre d'éléments
Dim UneValeur As String

'********************* Saisie les valeurs *********************'
Do
   Do
      UneValeur = InputBox("Saisissez une valeur")
   Loop Until IsNumeric(UneValeur)

   If UneValeur = 0 Then Exit Do

   Redim Preserve MesValeurs(NbrValeurs)
   MesValeurs(NbrValeurs) = CInt(UneValeur)
   NbrValeurs += 1
Loop

'******************** Calcule la moyenne **********************'
Dim N As Integer
Dim Total As Long

For N = 0 To NbrValeurs - 1
   Total += MesValeurs(N)
Next N

'******************** Affiche la moyenne **********************'
MsgBox "La moyenne est : " & Total / NbrValeurs
```

Consultez le chapitre 5 *(Les fonctions de Visual Basic.NET)* afin d'obtenir d'avantage d'informations concernant les diverses fonctions permettant à Visual Basic de traiter d'avantage les tableaux dynamiques.

Gestion des erreurs

Différentes erreurs peuvent survenir lors de l'exécution de votre code et ce même si la nature de cette erreur n'est pas tragique et ne devrait logiquement pas interrompre l'exécution de celui-ci. Ne voyez pas les erreurs comme des bêtes monstrueuses mais plutôt comme un message vous informant de l'état de l'exécution du code. Par exemple, une erreur est générée lorsque le code tente d'écrire ou de lire sur un support non disponible (i.e : l'utilisateur a retiré la disquette !). L'exemple suivant tente de récupérer les attributs d'un fichier situé sur la disquette. Évidemment, une erreur interrompra l'exécution du code si le fichier n'existe pas ou le lecteur ne possède pas de disquette.

```
Dim stFichier As String, Attr As Long

stFichier = "a:\monFichier.txt"

Attr = GetAttr(stFichier)                    ‘ Peu générer une erreur
```

Évidemment, il ne s'agit pas là d'une erreur fatale à l'application. Pourtant, un message désagréable est affiché à l'utilisateur afin qu'il prenne la décision de stopper l'application ou d'en continuer l'exécution à ses risques dès que la plateforme .NET rencontre une erreur lors de l'exécution du code. Or, l'utilisateur ne possède pas la compétence pour décider de quelle erreur est fatale à l'exécution du programme et de celle qui ne l'est pas. C'est au programmeur que revient cette tâche.

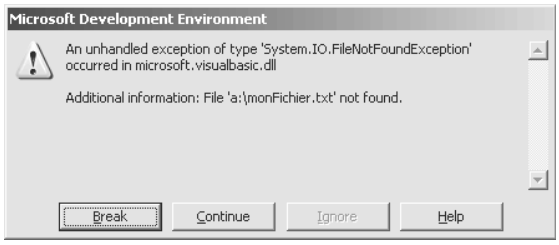


Figure 4.1 – Message d'erreur en mode conception

Figure 4.2 – Message d'erreur une fois l'application compilée

Visual Basic.NET possède deux jeux d'instructions lui permettant de gérer les erreurs. Le premier est intrinsèque au langage et est l'héritier des versions antérieures de Visual Basic. Un code écrit en utilisant ces instructions de traitement des erreurs sera donc compatible avec les versions antérieures du langage. Le second jeu d'instructions est natif à la plateforme .NET, est compatible avec l'ensemble des autres langages .NET et tire ses origines du C++ et du Java.

Gestion des erreurs intrinsèque au langage Visual Basic

Normalement, lorsqu'une erreur survient, l'exécution du code s'interrompt. Les instructions de gestion d'erreurs intrinsèques à Visual Basic permettent de désactiver ce comportement et de passer outre les instructions soulevant des erreurs. Ainsi, aucune erreur ne sera générée et, selon l'instruction précisée, Visual Basic continuera l'exécution du code à l'endroit précisé. Notez que les instructions de gestion des erreurs désactivent le soulèvement des erreurs pour la procédure ou la fonction en cours et n'ont pas de portée sur l'ensemble des modules du programme.

L'instruction On Error Resume

Visual Basic prévoit une instruction permettant au code de faire fit d'une erreur et de recommencer l'exécution de instruction ayant générée l'erreur jusqu'à ce que le problème soit réglé. Si, considérant l'exemple précédant, l'instruction `GetAttr` génère une erreur puisque le lecteur ne possède pas de disquette, l'instruction `On Error Resume` précisera à Visual Basic de ne pas en tenir compte et de ré-exécuter l'instruction.

```
Dim stFichier As String, Attr As Long

stFichier = "a:\monFichier.txt"

On Error Resume
Attr = GetAttr(stFichier)
```

L'instruction On Error Resume Next

L'instruction suivante permet au code de faire fît des erreurs pouvant survenir au sein du code. Si, considérant l'exemple précédant, l'instruction `GetAttr` génère une erreur puisque le lecteur ne possède pas de disquette, l'instruction `On Error Resume Next` précisera à Visual Basic de ne pas en tenir compte et de passer outre.

```
Dim stFichier As String, Attr As Long

stFichier = "a:\monFichier.txt"

On Error Resume Next
Attr = GetAttr(stFichier)
```

L'instruction On Error Goto *Ligne*

L'instruction `On Error Goto` *Ligne* permet de spécifier à Visual Basic de passer outre toute instruction erronée et de continuer l'exécution du code à une ligne précise. Cette ligne doit préalablement être déclarée comme suit :

```
NomLigne :
```

Si, considérant l'exemple précédant, l'instruction `GetAttr` génère une erreur puisque le lecteur ne possède pas de disquette, l'instruction `On Error Goto` précisera à Visual Basic de ne pas en tenir compte et poursuivre l'instruction à la ligne `ErrFichier`.

```
  Dim stFichier As String, Attr As Long

  stFichier = "a:\monFichier.txt"

  On Error Goto ErrFichier
  Attr = GetAttr(stFichier)
  Exit Sub

ErrFichier :
  MsgBox ("Le fichier ne peut être localisé")
```

L'instruction On Error Goto 0

L'instruction `On Error Goto 0` permet d'annuler tout traitement des erreurs. Ainsi, si une erreur survient après cette instruction, le code sera interrompu et un message d'erreur sera affiché à l'utilisateur. Simplement, l'instruction `On Error Goto 0` ramène le comportement initial de Visual Basic. Ce comportement est majoritairement souhaitable puisqu'il nous permet de connaître les erreurs non-gérées que pourrait générer code script afin de rendre celui-ci le plus parfait que possible.

```
  Dim stFichier As String, Attr As Long

  stFichier = "a:\monFichier.txt"

  On Error Resume Next
  Attr = GetAttr(stFichier)

  On Error Goto 0
```

L'objet Err

Maintenant, avec de telles instructions, il est désormais possible d'exécuter un code au grand complet sans qu'aucune erreur ne soit jamais générée. Youppi ! Comme il a été précisé précédemment, les erreurs ne doivent pas être perçues comme des événements tragiques mais plutôt comme des messages qu'il suffit de traiter adéquatement. Ainsi, il ne suffit pas de spécifier à Visual Basic de passer outre les erreurs mais il demeure plus sage de vérifier si une erreur est survenue et de réagir en conséquence.

L'objet `Err` représente l'erreur produite par la dernière instruction exécutée et vous permet d'en connaître la source, le numéro, la description, etc. En testant adéquatement l'objet `Err` à l'aide d'une simple structure `If`, il est possible de savoir si l'instruction précédemment exécutée a générée une erreur ou non.

```
Dim stFichier As String, Attr As Long

stFichier = "a:\monFichier.txt"

On Error Resume Next
Attr = GetAttr(stFichier)

If Err.Number Then
     MsgBox ("Le fichier ne peut être localisé")

End If

On Error Goto 0
```

Notez que l'exploit est réalisable seulement si l'instruction `On Error` a été activé sinon le code s'interrompra sur la ligne de code générant une erreur.

Ainsi, si la propriété `Number` de l'objet `Err` retourne zéro (0), cela signifie qu'aucune erreur n'a été générée par la dernière instruction exécutée. Sinon, la propriété `Err.Number` possédera comme valeur le numéro de l'erreur correspondante. L'objet `Err` expose les membres importants suivants :

Membre	Description
Clear	Méthode procédant à la suppression de toute information concernant la dernière erreur survenue.
Description	Retourne le nom de la description associé à la dernière erreur survenue.
Number	Retourne le numéro de la dernière erreur survenue. Retourne zéro (0) si la dernière instruction n'a générée aucune erreur.
Raise	Méthode soulevant intentionnellement l'erreur spécifiée par son numéro.
Source	Retourne le nom de la librairie ayant générée la dernière erreur survenue.

Gestion structurée des erreurs par le bloc Try… End Try

L'ensemble des langages .NET supportent une forme de gestion des erreurs tirée directement du C++ et du Java appelée **gestion structurée des erreurs**. Cette technique de gestion des erreurs s'opère à l'aide d'une structure de contrôle spécifique – le bloc `Try` – et possède la syntaxe suivante :

```
Try
        <Instructions>
Catch <DéfinitionErreur>
        <Instructions>
[Finally]
        <Instructions>
End Try
```

Les instructions placées à l'intérieur du bloc `Try` seront tentées mais ne provoqueront pas immédiatement l'arrêt du programme si une erreur est rencontrée au sein de ces instructions. On utilise donc généralement un bloc `Try` pour y placer les instructions pouvant potentiellement générer des erreurs.

Si une erreur est soulevée à l'intérieur du bloc `Try`, l'exécution du programme se poursuivra au bloc `Catch` correspondant à l'erreur survenue. Le ou les blocs `Catch` doivent en effet correspondre à des erreurs potentielles et le code s'y trouvant sera exécuté le cas échéant. Si aucun bloc `Catch` ne correspond à l'erreur réellement soulevée, le logiciel s'interrompra en affichant un message *« Erreur non-gérée »*.

Le bloc optionnel `Finally` permet d'y stocker des instructions qui seront exécutées dans tous les cas, qu'une erreur soit survenue ou non.

Le code de l'exemple suivant tente d'accéder à un fichier situé sur le lecteur de disquettes. Ce code fonctionne donc à merveille à la condition qu'une disquette soit située dans le lecteur et que le fichier spécifié existe :

```
Dim Attr As System.Int32

Try
      Attr = GetAttr("a:\exemple.txt")

Catch Ex As System.IO.FileNotFoundException
      MsgBox("Le fichier est introuvable")

Catch Ex As System.IO.FileLoadException
      MsgBox("Le fichier ne peut pas être chargé")

Finally
      Beep()

End Try
```

Les blocs `Catch` peuvent se voir imposer des conditions supplémentaires à leur exécution au simple fait qu'ils correspondent à une exception spécifique. Il est possible d'ajouter une **clause When** afin que le code du bloc `Catch` ne s'exécute seulement lorsque la condition est rencontrée :

```
Try
      <Instructions>
Catch <DéfinitionErreur> When <Condition>
      <Instructions>
[Finally]
      <Instructions>
End Try
```

L'exemple suivant tente d'ouvrir un fichier par trois reprises et quitte la boucle lorsque les trois essais infructueux sont écoulés ou, évidemment, lorsque le fichier peut être ouvert sans encombre.

```
Dim Attr As System.Int32
Dim N As System.Short = 0

Do
  Try
      Attr = GetAttr("a:\exemple.txt")
      Exit Do

  Catch Ex As System.IO.FileNotFoundException When N < 3
      MsgBox("Le fichier est introuvable")
      N += 1

  Catch Ex As System.IO.FileNotFoundException When N >= 3
      Exit Do

  End Try
Loop
```

Il est possible de récupérer toutes les erreurs qui pourraient être rencontrées sans égard à leur code d'erreur en utilisant le mot-clé `Catch` sans argument ou en y utilisant un argument de type `Exception` comme le démontre l'exemple suivant :

```
Try
      Attr = GetAttr("a:\exemple.txt")

Catch Ex As Exception
      MsgBox Ex.Message
End Try
```

L'instruction Throw

L'instruction `Trow` permet au code de soulever intentionnellement une erreur. Quoique cette fonctionnalité puisse sembler légèrement inutile à point-ci, elle prend toute son importance lors de la conception de classes, de composants réutilisables et de contrôles personnalisés qui seront abordés dans des chapitres subséquents. Cependant, cette instruction fait partie intégrale du langage *Visual Basic.NET* et c'est pourquoi nous la présentons tout de même.

Pour soulever une exception, `Throw` nécessite d'abord qu'une instance de l'exception désirée soit créée puis paramétrée à l'aide d'un code semblable au suivant :

```
Dim Excp As New Exception

Excp.Message = "Erreur d'authentification"
```

Ensuite, l'exception ainsi créée peut être soulevée à l'aide de `Throw`. L'exception soulevée sera récupérée par un bloc `Try` correspondant.

```
Dim Excp As New Exception

Excp.Message = "Erreur d'authentification"

Throw Excp
```

Le chapitre 8 *(Développement orienté objets)* couvre en détails l'utilisation de l'instruction `Throw`.