Chapitre 3

Les variables plus complexes

3.1. Les énumérations

Définition

Ce type correspond à une constante entière, à laquelle vous donnez un nom de manière à la rendre plus explicite. Pour une couleur par exemple, la variable sera une énumération de type `couleur`, dont les valeurs seront par exemple `Vert`, `Jaune`, ou `Rouge`, etc.

Cependant, derrière ces noms se cache ni plus ni moins qu'une valeur entière. Les énumérations ne sont vraiment qu'un moyen pour rendre l'écriture plus agréable et plus compréhensive, quand il y a un nombre fini de valeurs possibles.

Déclarer une énumération

Voici comment déclarer une énumération :

```
Public Enum VetementHaut
  Tshirt
  Chemise
  Debardeur
End Enum
```

Lorsque vous ne spécifiez aucune valeur dans l'énumération, la valeur entière correspondante commence à 0, et va en augmentant. Dans ce cas, `Vert` vaut donc 0, `Jaune` vaut 1, et `Rouge` vaut 2. Cependant, vous pouvez spécifier des valeurs pour qu'elles soient plus pertinentes. Par exemple :

```
Public Enum VetementBas
  Pantalon = 1
  Jeans = 2
  Bermuda = 3
End Enum
```

Utiliser des énumérations

Les énumérations s'utilisent ensuite comme des variables normales. Après avoir déclaré le type comme décrit précédemment, il faut déclarer une variable de ce type comme ceci :

```
Dim haut As VetementHaut
```

Vous pouvez ensuite affecter une valeur à cette variable. Pour donner à la variable une valeur de l'énumération, il faut indiquer le type et la valeur en question, séparés par un point :

```
haut = VetementHaut.Chemise
```

Vous pouvez d'ailleurs initialiser la variable directement pendant la déclaration :

```
Dim bas As VetementBas = VetementBas.Pantalon
```

Les enregistrements

Définition

Un enregistrement (on peut également parler de "structure") est une variable contenant un ensemble de données de types différents, simples ou structurés. Ils permettent de regrouper un certain nombre d'informations liées à la même chose, ou encore de représenter une information formée de plusieurs composantes, par exemple des coordonnées géographiques.

C'est en fait un groupement de variables qui ne sont alors plus indépendantes car rassemblées autour d'un même élément englobant.

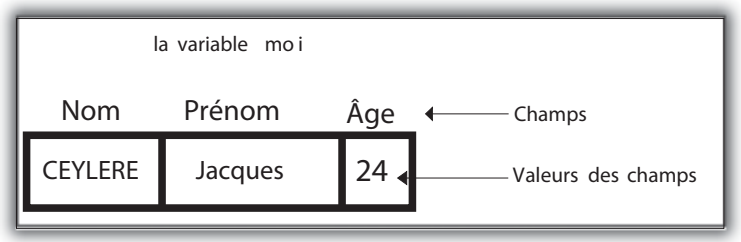

Figure 3.1 : *Représentation d'un enregistrement*

Déclarer un enregistrement

Pour cela, utilisez le mot-clé `Structure` et, dans le bloc de déclaration, indiquez les différentes composantes de l'enregistrement :

```
Public Structure Personne
  Public String nom
  Public String prenom
```

```
  Public Integer age
End Structure
```

Chaque composante de l'enregistrement est appelée "champ". Ici, il y en a trois : le nom, le prénom, et l'âge, de type chaîne de caractères pour les deux premiers, et entier pour le dernier.

Utilisation des enregistrements

L'utilisation des enregistrements se fait par la lecture et/ou l'écriture de ses différents champs.

Il faut d'abord déclarer la variable et l'instancier :

```
Dim moi As Personne
moi = New Personne()
```

La deuxième ligne instancie la variable, c'est-à-dire qu'elle crée en mémoire l'espace nécessaire à son utilisation. En effet, un enregistrement étant un regroupement de champs éventuellement de types différents et dont le nombre n'est pas directement limité, sa taille n'est pas fixe. L'instanciation sert donc à déterminer l'espace exact nécessaire pour que vous puissiez utiliser cette variable complètement.

Une fois qu'elle est instanciée, tout l'espace est créé, tous les champs sont accessibles par leur nom, précédé du nom de la variable et d'un point. Vous pouvez alors les lire, leur affecter des valeurs, comme n'importe quelle variable :

```
moi.nom = "CEYLERE"
moi.prenom = "Jacques"
moi.age = 24
MessageBox.Show("Je suis " + moi.nom + " " + moi.prenom)
MessageBox.Show("J'ai " + moi.age + " ans.")
```

Un enregistrement est un type structuré dont les champs peuvent être de différents types. C'est pourquoi il n'est pas aisé d'initialiser directement un enregistrement pendant la déclaration. Des mécanismes permettent de le faire lors de l'instanciation (nous y reviendrons). Pour l'instant, on supposer que, pour initialiser un enregistrement, il faut initialiser chacun de ses champs juste après sa création, ce qui peut s'avérer fastidieux si l'enregistrement est conséquent.

3.3. Les tableaux

Définition

Un tableau est une variable contenant un ensemble de données de même type. Ces données peuvent être de types simples (tableaux d'entiers ou de caractères) ou complexes (tableaux d'enregistrements, de chaînes de caractères ou tableaux de tableaux).

Un tableau peut avoir une, deux, trois ou n dimensions. À une dimension, c'est un ensemble de cases "sur une ligne".

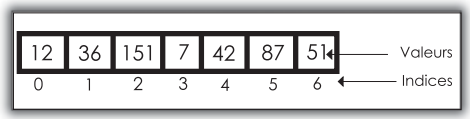

Figure 3.2 : *Tableau à une dimension*

À deux dimensions, on peut le représenter comme ayant des lignes et des colonnes. On l'appelle alors "matrice".

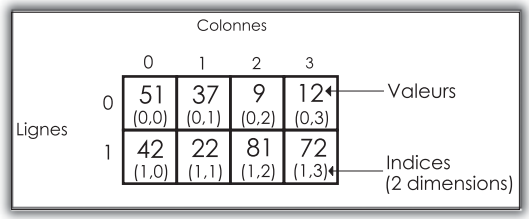

Figure 3.3 : *Tableau à deux dimensions (matrice)*

À trois dimensions, il aurait des lignes, des colonnes, plus une profondeur.

Déclarer un tableau

Pour déclarer un tableau, vous devez définir un certain nombre de choses. La première est le type des données qui seront à l'intérieur du tableau. En effet, un tableau d'entiers n'est pas pareil (en particulier il n'a pas la même taille) qu'un tableau de caractères. De plus, il faut préciser sa taille, c'est-à-dire le nombre d'éléments qu'il contiendra, et ce pour chacune des dimensions s'il y en a plusieurs. Visual Basic .NET

présente une particularité à ce niveau : il s'agit de définir, non pas la taille, mais l'indice du dernier élément, sachant que les indices commencent à 0.

```
' Déclaration d'un tableau simple de 9 chaînes de caractères
Dim courses(8) As String

' Déclaration d'une matrice de 4 X 3 entiers
Dim maMatrice(3, 2) As Integer
```

Utiliser les tableaux

Une fois qu'un tableau est déclaré, vous pouvez accéder à ses éléments grâce à leurs indices. C'est la position de la donnée dans le tableau. La numérotation commence à 0. En Visual Basic, le dernier élément a l'indice qui a été utilisé lors de la déclaration du tableau.

Il y a autant d'indices qu'il y a de dimensions.

```
courses(0) = "Pâtes"
courses(1) = "Jambon"
Courses(2) = "Beurre"
maMatrice(1)(1) = 42;
MessageBox.Show("En premier je dois prendre : " + courses(0))
MessageBox.Show("Il y a " + maMatrice(1)(1) + " à la case
✂ 2-2")
```

Un tableau est un type structure dont tous les éléments sont du même type. Il est possible d'initialiser le tableau à la déclaration, et ainsi de se retrouver avec un tableau dont les valeurs sont déjà définies. Il suffit de donner la liste des éléments entre accolades, séparés par des virgules :

```
Dim maListe(4) As Integer = { 0, 0, 0, 0, 0 }
```

Cela est pratique lorsque le tableau n'est pas trop grand, mais imaginez ce que cela donnerait avec un tableau de plusieurs centaines d'éléments, voire plusieurs milliers. Dans ce cas, il faut parcourir le tableau avec une boucle, dont nous décrirons le fonctionnement ultérieurement. Voici un avant-goût qui montre comment parcourir un tableau pour l'initialiser. Il faut pour cela utiliser une variable de parcours et donner l'indice de départ (0) et l'indice de fin, que l'on obtient grâce à la méthode `getUpperBound` :

```
Dim tableau(123) As Integer
Dim i As Integer
For i = 0 to tableau.GetUpperBound(0)
    tableau(i) = 23
Next
```

Ici, vous déclarez un tableau de cent vingt-quatre éléments valant tous 23.

Cas pratique : une bibliothèque multimédia

Vous avez maintenant les éléments de base permettant de construire une application intéressante. En combinant ces éléments, vous pouvez faire des programmes d'un plus grand intérêt. Par exemple, vous allez faire ici une bibliothèque multimédia pour gérer une liste de vidéos.

On peut définir une vidéo par son type d'affichage, à savoir monochrome ou couleur. Pour cela, recourez à une énumération possédant ces deux valeurs :

```
Public Enum Affichage
  Monochrome
  Couleur
End Enum
```

Une vidéo possède un certain nombre d'attributs qui lui sont propres, par exemple son nom, sa longueur, etc. Pour représenter une vidéo, vous pouvez donc utiliser un enregistrement dont les champs correspondent aux différents attributs.

```
Public Structure Video
  Public String nom
  Public Affichage type
  Public Integer longueur
End Structure
```

Vous avez défini les éléments fondamentaux de la bibliothèque. Comme une bibliothèque avec une seule vidéo a peu d'intérêt, vous allez créer un tableau de vidéos pour qu'elle puisse en contenir plusieurs. Il suffit d'utiliser les méthodes vues précédemment :

```
Dim bibliotheque(5) As Video
```

Voila, vous avez décrit tous les types dont vous avez besoin pour gérer la liste de vidéos. Vous pouvez maintenant décrire chacune d'entre elles, puis les mettre dans la bibliothèque, que vous consulterez avant de visionner un film :

```
Dim temps_modernes As Video
temps_modernes = new Video()
temps_modernes.nom = "Les temps modernes"
temps_modernes.type = Monochrome
```

```
temps_modernes.longueur = 89
bibliotheque(0) = temps_modernes
Dim braveheart As Video
braveheart  = new Video()
braveheart.nom = "Braveheart"
braveheart.type = Couleur
braveheart.longueur = 165
bibliotheque(1) = braveheart
```

Vous avez décrit et ajouté deux films à la bibliothèque. Selon vos besoins, ajoutez de nouvelles vidéos, éventuellement d'autres attributs pour les décrire, etc.

Vous connaissez maintenant les différents types de variables, simples et structurés, qui vous permettront de décrire vos données et de les utiliser. Vous verrez dans les chapitres suivants comment en faire un usage plus intéressant et un peu moins simpliste…

Chapitre 4

Contrôler un programme

Un programme n'est pas qu'une suite directe de lectures et d'affectations de variables, même s'il y a beaucoup de cela. De plus, si l'on devait faire tout le reste à la main, cela limiterait l'intérêt d'un ordinateur et de sa puissance de calcul. C'est pourquoi vous allez voir dans ce chapitre comment structurer un programme et le contrôler, c'est-à-dire lui donner un certain comportement selon différents critères. Il sera question de conditions, de boucles, de fonctions, de procédures, etc.

4.1. Imposer des conditions

Nous allons traiter ici des instructions conditionnelles. Elles permettent d'exécuter une série d'instructions selon le résultat d'un test.

La plus utilisée d'entre elles est le Si… Alors… Sinon, dit aussi `If… Then… Else`. Elle permet d'exécuter un bloc d'instructions selon le résultat d'un test booléen, dont la réponse est vraie ou fausse. Sa syntaxe générale est :

```
If (condition) Then
  InstructionsThen
Else
  InstructionsElse
End If
```

Cette expression signifie que si `condition` est vrai, alors `InstructionsThen` sera exécuté, et dans le cas contraire, le programme exécutera `InstructionsElse`. `condition` peut être de multiples natures, toutes valables à partir du moment où le résultat est une valeur booléenne. Ce peut être :

- une variable booléenne : `If (il_pleut)` ;
- une expression comparative : `If (3 + 2 == 5)` ;
- une fonction booléenne : `If (il_fait_beau()).`

La partie `Else` de la condition est optionnelle. Si vous souhaitez exécuter des instructions dans des conditions particulières, et ne rien faire dans le cas contraire, il vous suffit de retirer la partie `Else`. Cela donne :

```
If (condition) Then
  InstructionsThen
End If
```

Ici, si `condition` est vrai, le programme exécutera `InstructionsThen`, mais dans le cas contraire, il continuera son

exécution normalement. Les parenthèses autour de la condition ne sont pas obligatoires, mais il est fortement conseillé de prendre l'habitude de les mettre. En effet, si la condition est assez longue, ou composée de plusieurs conditions, elles favorisent la compréhension.

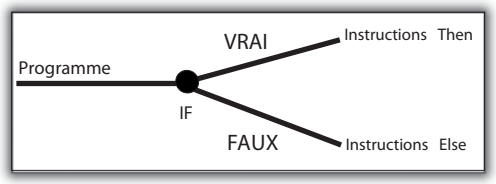

Figure 4.1 : *Représentation de la condition If… Then… Else*

La clause `ElseIf` existe dans l'écriture des instructions conditionnelles pour imposer une condition dans le cas où la première serait fausse. Exemple :

```
Dim il_fait_beau As Boolean
Dim il_pleut As Boolean
If (il_fait_beau) Then
  MessageBox.Show("Il faut prendre une casquette.")
ElseIf (il_pleut) Then
  MessageBox.Show("Il faut prendre un parapluie.")
EndIf
```

Ces lignes stipulent qu'il faut prendre une casquette s'il fait beau, et s'il pleut, que l'on doit prendre un parapluie. Le fait d'imposer la seconde condition fera qu'il n'y aura rien de spécial en cas de grisaille. Si l'on avait utilisé un `Else` sans préciser la condition sur la pluie, le programme aurait dit de prendre un parapluie dans tous les cas, sauf par beau temps.

Lorsque qu'une condition porte sur une variable avec plusieurs possibilités, il est possible d'écrire une série d'expressions conditionnelles correspondant à ces différentes possibilités. On parle alors d'"imbrication" :

```
Dim age As Integer
If (age < 16) Then
  MessageBox.Show("Vous êtes trop jeune pour travailler.")
Else
  If (age > 64) Then
    MessageBox.Show("Vous ne pouvez plus travailler.")
  Else
    MessageBox.Show("Vous avez l'âge légal pour travailler")
  EndIf
EndIf
```

On pourrait, dans ce cas, utiliser un `ElseIf`, mais cet exemple sert à montrer le principe d'imbrication. Attention, un excès d'instructions conditionnelles imbriquées peut poser des problèmes de compréhension. De plus, en termes de performances, ce n'est généralement pas la meilleure solution. À utiliser donc avec prudence !

4.2. Faire les bons choix

L'instruction Select

Après le `If… Then… Else`, qui permet d'exécuter des instructions selon une condition particulière, vous allez découvrir le `Select`, qui permet de séparer des instructions selon la valeur d'une variable donnée. Sa syntaxe est la suivante :

```
Select Case variable
  Case Valeur1
    Instructions1
  Case Valeur2
    Instructions2
  …
  Default
    InstructionsDefault
End Select
```

Cette expression signifie que, selon les valeurs de `variable`, certaines instructions seront exécutées. En particulier, si `variable` vaut `Valeur1`, le programme exécutera `Instructions1`, si `variable` vaut `Valeur2`, `Instructions2` sera exécuté, et si `variable` vaut une valeur non prise en charge dans l'expression, `InstructionsDefault` sera exécuté.

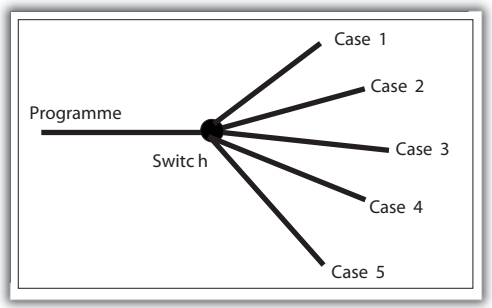

Figure 4.2 : *Représentation de l'instruction Select*

N'importe quel type simple peut être utilisé comme variable à partir du moment où le type de la variable est cohérent avec les valeurs des différents cas. Ceci provoque une erreur :

```
Dim age As Integer
Select Case age
  Case "mineur"
  Case "majeur"
End Select
```

La variable est de type entier alors que les valeurs proposées sont des chaînes de caractères. La comparaison ne pouvant être faite, cette expression est fausse et provoque une erreur.

Les énumérations sont souvent combinées avec les `Select`. En effet, du fait qu'elles ont un nombre fini de valeurs, généralement significatives, elles se prêtent bien à cette instruction. Exemple :

```
Public Enum Jour
  Lundi
  Mardi
  Mercredi
  Jeudi
  Vendredi
  Samedi
  Dimanche
End Enum

Dim jour As Jour
Select Case jour
  Case Jour.Dimanche
    MessageBox.Show("C'est le week-end et les magasins
    ✂ sont fermés.")
  Case Jour.Samedi
    MessageBox.Show("C'est le week-end mais les magasins
    ✂ sont ouverts.")
  Default
    MessageBox.Show("C'est la semaine. Vous devez aller
    ✂ travailler.")
End Select
```

Nous avons vu les principales instructions conditionnelles, qui permettent de faire telle ou telle chose dans telle ou telle situation, ce qui les rend plus intéressants.

4.3. Répéter des opérations

À présent, vous allez apprendre à "boucler", c'est-à-dire à répéter des opérations un certain nombre de fois, sans avoir à intervenir.

Les boucles représentent l'un des mécanismes les plus importants de la programmation, si ce n'est le plus important. En effet, elles permettent de tirer le meilleur parti de l'ordinateur et de sa puissance de calcul. Imaginez que vous voulez envoyer un même e-mail à un grand nombre de personnes. Ce serait une tâche horrible à exécuter à la main. En revanche, si vous écrivez le modèle et faites une boucle qui l'envoie à tous les destinataires de votre Carnet d'adresses, cela n'aura pris que le temps d'écrire le programme. C'est ce qui rend les boucles aussi importantes en programmation.

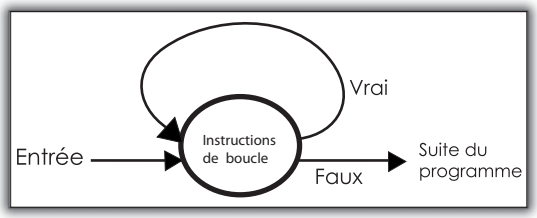

Figure 4.3 : *Fonctionnement d'une boucle*

Les boucles, ou instructions répétitives, peuvent être de deux types :

- Les boucles non déterministes : le nombre d'itérations (tours de boucle) est non défini à l'avance. Ce sont les boucles Tant que… Faire.
- Les boucles déterministes : le nombre d'itérations est connu car on spécifie un état de départ et un état d'arrêt. Ce sont les boucles Pour… Faire.

La boucle Tant que... Faire

La boucle Tant que… Faire est non déterministe, c'est-à-dire qu'il n'est pas indiqué dans la boucle elle-même le nombre d'itérations qui seront faites. Voici sa syntaxe :

```
While (condition_continuite)
  Instructions
End While
```

La boucle commence avec le mot-clé `While` et se termine au niveau du `End While`. `Instructions` correspond aux instructions qui seront exécutées lorsque l'on passera dans cette boucle. Comme dans un `If`, `condition_continuite` est un test booléen qui fera que l'on passe ou pas dans la boucle. Si sa valeur est vraie, on passera (ou repassera) dedans, et si elle est fausse, on la quittera et le programme continuera son exécution normalement. Comme dans le `If`, `condition_continuite` peut être :

- une variable booléenne : `If (il_pleut)` ;
- une expression comparative : `If (3 + 2 == 5)` ;
- une fonction booléenne : `If (il_fait_beau()).`

Voici un exemple de boucle basique, à savoir un compteur :

```
Dim compteur As Integer = 0
While (compteur < 100)
  compteur = compteur + 1
  MessageBox.Show("Compteur = " + compteur)
End While
```

Cet exemple compte de 0 à 99 et affiche la valeur courante dans une boîte de dialogue. Examinons les principaux éléments de cette boucle.

Remarquez l'importance de l'état de départ. En effet, s'il était inconnu à l'entrée dans la boucle, on ne pourrait être sûr de la bonne exécution de cette dernière. C'est pourquoi on a initialisé `compteur`. De cette manière, on est sûr que la condition de continuité au départ sera respectée et que la boucle sera donc exécutée.

L'autre élément important est la modification de la variable de boucle : `compteur`. La continuité en dépend directement. L'une des instructions de boucle modifie `compteur`. Elle est indispensable au bon fonctionnement de la boucle. C'est même ici l'instruction la plus importante. Imaginez ce qu'il se passerait si `compteur` n'est pas modifié :

1. `compteur` vaut 0. On entre dans la boucle.
2. On affiche sa valeur. On revient au début de la boucle.
3. `compteur` vaut toujours 0. On entre de nouveau dans la boucle.
4. On réaffiche sa valeur, qui est toujours la même d'ailleurs. Et l'on revient au début.
5. `compteur` vaut encore 0. On entre une nouvelle fois dans la boucle, et ainsi de suite, sans fin.

On vient de créer une boucle infinie qui affiche à chaque fois 0 dans une boîte de dialogue. C'est pourquoi il est indispensable de bien modifier la variable de boucle pour garantir une sortie. Cependant, il ne faut pas la modifier n'importe comment non plus, car cela pourrait provoquer une sortie inopinée. Si vous utilisez la variable de boucle à mauvais escient, vous pouvez provoquer des comportements non voulus du programme. Exemple :

```
Dim compteur As Integer = 0
While (compteur < 100)
  compteur = compteur + 1
  If (compteur == 14) Then
    compteur = 25
End If
  MessageBox.Show("Compteur = " + compteur)
End While
```

En ajoutant une instruction, on provoque un trou entre 14 et 25. Ici, cela est détectable. Mais des modifications peuvent être moins évidentes que celle-ci à repérer. La règle d'or est de ne jamais utiliser une variable de boucle pour autre chose que compter le nombre d'itérations et de toujours modifier sa valeur à chaque tour pour éviter une boucle infinie.

La boucle Faire... Tant que

Faire… Tant que, ou `Do… Loop While`, est comparable à Tant que… Faire, à ceci près que la condition de continuité est appliquée à la fin de la boucle, et non au début. Voici la syntaxe :

```
Do
  Instructions
Loop While (condition_continuite)
```

Si, une fois que `Instructions` est exécuté, `condition_continuite` est vrai, il y a un nouveau tour de boucle ; sinon, la boucle est terminée et le programme continue son exécution normalement. On retrouve exactement les mêmes éléments que dans la boucle Tant que… Faire. Nous n'y reviendrons pas. Pour bien comprendre la différence, considérez ces deux exemples :

```
Dim compteur As Integer = 0
While (compteur == 1)
  compteur = compteur + 1
MessageBox.Show("Compteur = " + compteur)
End While

Dim compteur As Integer = 0
```

```
Do
  Compteur = compteur + 1
  MessageBox.Show("Compteur = " + compteur)
Loop While (compteur == 1)
```

Dans les deux programmes, toutes les composantes sont les mêmes : une boucle, avec la même condition de continuité, les mêmes instructions de boucle et les mêmes états initiaux.

Dans le premier cas, `compteur` vaut 0 lorsqu'on entre dans la boucle. La première condition n'étant pas vérifiée, le programme ne passe pas dans la boucle.

Dans le deuxième cas, `compteur` vaut également 0. Le programme passe dans la boucle, change la valeur de `compteur`, et affiche "Compteur = 1". La condition de continuité est alors vérifiée. Le programme repasse donc dans la boucle. Le programme augmente `compteur` et affiche "Compteur = 2". La condition de continuité n'est alors plus vérifiée, et le programme sort de la boucle.

Dans le second cas, on effectue deux passages, alors que dans le premier, on n'entre même pas dans la boucle. C'est ce qui fait la différence entre les deux, alors que les conditions de continuité et les instructions de boucle sont les mêmes. À peu de choses près, ces boucles sont identiques, mais il y a néanmoins quelques subtilités à connaître pour éviter les petits désagréments.

La boucle Pour… Faire

Pour… Faire est une boucle déterministe. Elle ressemble à Faire… Tant que et à Tant que… Faire dans le sens où elle possède également une condition de continuité et des instructions de boucle qui seront exécutées lors du passage dans ladite boucle. La différence ici est que l'on impose un intervalle de boucle, c'est-à-dire que l'on donne à l'avance la valeur de départ de la variable de boucle, ainsi que sa valeur de fin. Il faut donc connaître au préalable le nombre de fois que l'on va boucler. Voici la syntaxe :

```
For variable_de_boucle = valeur_de_depart to valeur_de_fin
  Instructions
Next
```

D'après cette construction, on voit que l'on ne peut utiliser que des variables de boucle de type numérique. En effet, contrairement aux cas

précédents, il n'y a pas de condition de continuité booléenne qui permettrait d'utiliser n'importe quel type d'expression pour gérer les passages dans la boucle. Celle-ci s'exécutera jusqu'à ce que la variable de boucle atteigne sa valeur finale. Il n'est pas nécessaire de la modifier. C'est l'instruction `Next` qui se charge de l'augmenter. De cette manière, on ne risque pas de rencontrer le problème de boucle infinie. C'est pourquoi il est recommandé d'utiliser tant que possible la boucle `For`, dont le comportement est moins hasardeux, justement à cause de sa nature déterministe. Voici l'équivalent de l'exemple précédent si l'on utilise un `For` :

```
Dim compteur As Integer
For compteur = 0 to 100
  MessageBox.Show("Compteur = " + compteur)
Next
```

Attention, toutes les valeurs entre les bornes (bornes comprises) seront utilisées. En d'autres termes, le programme fait ici 101 tours de boucle, de 0 à 100 compris. Dans les autres boucles, étant donné que la condition d'arrêt est `(compteur < 100)`, le programme n'affiche pas "Compteur = 100", alors que, cette fois, il l'affiche. Par défaut, la variable de boucle augmente par pas de 1. Cependant, il est possible de l'augmenter d'un pas plus grand grâce au mot-clé `Step`, qui permet de spécifier une valeur de pas :

```
Dim compteur As Integer
For compteur = 0 to 100 Step 2
  MessageBox.Show("Compteur = " + compteur)
Next
```

Ici, la variable de boucle augmente par pas de 2, et le programme affiche les nombres pairs entre 0 et 100. On peut aussi faire des boucles décroissantes :

```
Dim compteur As Integer
For compteur = 100 to 0 Step -1
  MessageBox.Show("Compteur = " + compteur)
Next
```

Dans cet exemple, l'affichage des valeurs se fait dans l'ordre décroissant, de 100 à 0.

La boucle `For` est certes moins permissive et moins flexible à cause de l'absence de condition de continuité booléenne, mais elle permet d'éviter des erreurs d'utilisation à cause d'une condition mal écrite. Il est recommandé de privilégier son utilisation par rapport aux autres boucles.

4.4. Bien découper un programme

Les programmes examinés jusqu'à présent se limitent à des suites d'instructions qui s'exécutent au fur et à mesure. Cependant, on a fréquemment besoin de réutiliser des lignes de code qui ont déjà été exécutées auparavant, c'est-à-dire de refaire un même traitement. Il serait dommage de récrire ces lignes encore et encore. En ce sens, nous allons voir ici comment structurer un programme et créer du code réutilisable à plusieurs endroits. Pour y parvenir, nous utiliserons les fonctions et les procédures, et tout ce qu'il y a autour, à savoir les paramètres, les valeurs de retour, etc.

Les fonctions et procédures

Les fonctions et les procédures sont les éléments primordiaux des programmes, lorsqu'ils commencent à devenir un peu conséquents. Elles permettent de regrouper une ou plusieurs instructions. C'est en quelque sorte un sous-programme, à ceci près qu'il n'est pas autonome et qu'il est utilisé par le programme principal à un certain moment. On dit que le programme appelle une fonction ou une procédure.

Une procédure est série d'instructions regroupées, généralement utilisées pour l'affichage ou la sauvegarde, car elles n'ont pas de valeur de retour. On utilise le mot-clé `Sub` pour la déclarer, suivi du nom de la procédure, de paramètres entre parenthèses (que nous verrons plus tard) et de `End Sub` en fin de déclaration :

```
Public Sub HelloWorld()
  MessageBox.Show("Bonjour")
  MessageBox.Show("le")
  MessageBox.Show("monde")
End Sub
```

Le nom de la procédure sert à l'appeler à partir d'un autre endroit du programme. `HelloWorld` regroupe trois instructions, qui affichent chacune un mot de la phrase "Bonjour le monde" dans une boîte de dialogue. On l'appelle dans le programme tout simplement en indiquant son nom et des paramètres, s'il y en a :

```
HelloWorld()
```

À l'endroit de votre programme où vous inscrivez ce nom, c'est comme si vous récriviez les instructions regroupées dans la procédure. Vous le pouvez l'appeler autant de fois que nécessaire dans le programme.

Écrire des procédures permet de gagner en compréhension et en temps, car vous n'avez pas besoin de récrire toutes les instructions à chaque fois. Ici, en un appel, vous avez récrit l'équivalent de trois instructions, mais les procédures peuvent être beaucoup plus longues.

Une fonction est une série d'instructions regroupées, comme dans une procédure, mais elle permet de renvoyer une valeur de retour. Cette valeur peut être utilisée par le programme appelant comme si c'était une valeur normale. Voici comment déclarer une fonction :

```
Public Function HelloWorld()
  Dim resultat As String = "Bonjour le monde"
Return (resultat)
End Function
```

`HelloWorld` renvoie tout simplement la chaîne de caractères "Bonjour le monde". Vous n'êtes pas obligé de préciser le type de la valeur de retour mais vous pouvez le faire en ajoutant `As` et le type après le nom et les paramètres. Il est recommandé de le préciser, car le code est plus clair. Cela aide à la compréhension et permet de détecter plus facilement les erreurs.

```
Public Function HelloWorld() As String
  Dim resultat As String = "Bonjour le monde"
Return (resultat)
End Function
```

`Return` est le mot-clé qui sert à spécifier que l'on sort de la fonction en retournant la valeur de retour. On peut également affecter la valeur de retour au nom de la fonction, comme ceci :

```
Public Function HelloWorld() As String
  Dim resultat As String = "Bonjour le monde"
HelloWorld = resultat
End Function
```

Les deux fonctions sont exactement les mêmes et, bien que ce ne soit pas d'une importance primordiale, mieux vaut le savoir pour comprendre des programmes écrits par d'autres.

Vous pouvez appeler une fonction comme une procédure. Dans ce cas, la valeur de retour n'a pas d'effet, mais les instructions de la fonction sont quand même exécutées :

```
HelloWorld()
```

La fonction se contente de renvoyer une chaîne de caractères. Mais vous pouvez utiliser cette valeur de retour dans n'importe quelle partie du programme qui attend une valeur du même type. Exemple :

```
MessageBox.Show(HelloWorld())
```

Par cet appel, le programme affiche une boîte de dialogue contenant le message "Bonjour le monde", car `Show` attend une chaîne de caractères, et justement, `HelloWorld` en renvoie une.

Les paramètres

Lorsque l'on déclare ou que l'on appelle des fonctions ou des procédures, leur nom est toujours suivi de parenthèses vides ou qui contiennent des paramètres. Les paramètres sont des variables, dont on récupère la valeur lors de l'appel de la fonction ou de la procédure. Il peut y en avoir un, plusieurs, ou aucun. Ils permettent de donner de la flexibilité à une fonction ou à une procédure. En effet, étant donné que leur valeur exacte n'est donnée que lors de l'appel, celle-ci peut varier et apporter du changement. À la déclaration, il faut spécifier leur type et leur nom. Voici un exemple de procédure avec des paramètres :

```
Public Sub QuiSuisJe(ByVal nom As String, ByVal prenom As
✂ String)
  MessageBox.Show("Je suis " + nom + " " + prenom)
End Sub
```

Le principe est le même pour les fonctions :

```
Public Function Moi(ByVal nom As String, ByVal prenom As
✂ String) As String
  Return ("Je suis " + nom + " " + prenom)
End Function
```

Par contre, lorsque l'on appelle des fonctions ou des procédures qui ont des paramètres, il faut mettre entre parenthèses les valeurs de ces paramètres. On peut le faire soit par le biais de variable, ou d'expression constante, voire de fonction. En effet, une fonction peut être un paramètre d'une autre fonction, car elle retourne une valeur ; mais il faut que les types correspondent.

```
Dim mon_nom As String = "Martin"
Dim mon_prenom As String = "Pierre"
QuiSuisJe(mon_nom, mon_prenom)
MessageBox.Show(Moi("Martin", "Pierre"))
```

Le mot-clé `ByVal` qui figure dans la déclaration des paramètres indique qu'ils sont transmis par valeur. Cela signifie qu'à chaque fois qu'un paramètre est fourni, une copie des données est faite avant qu'elles soient transmises à la fonction ou à la procédure. Cela garantit qu'un changement sur la valeur transmise ne sera pas répercuté sur la variable originale. Il s'agit du comportement par défaut, ce qui rend `ByVal` optionnel. Modifions un peu la procédure `QuiSuisJe` :

```
Public Sub QuiSuisJe(ByVal nom As String, ByVal prenom As
✂ String)
  MessageBox.Show("Je suis " + nom + " " + prenom)
  nom = "DOE"
  prenom = "John"
End Sub
```

Après avoir affiché le nom et le prénom, on les modifie. On reprend le programme précédent, avec une légère modification :

```
Dim mon_nom As String = "Martin"
Dim mon_prenom As String = "Pierre"
QuiSuisJe(mon_nom, mon_prenom)
MessageBox.Show(Moi(mon_nom, mon_prenom))
```

Au lieu d'écrire des valeurs constantes dans la dernière instruction, on indique `mon_nom` et `mon_prenom`. Le programme affiche deux fois une boîte de dialogue dans laquelle figure le texte "Je suis Martin Pierre", bien que les paramètres aient été modifiés dans la deuxième procédure `QuiSuisJe`. Cela est dû au fait que ces paramètres ont été passés par valeur. La procédure a donc travaillé sur une copie des variables `mon_nom` et `mon_prenom` et leurs valeurs originales n'ont donc pas été modifiées.

Le passage de paramètres par valeur s'oppose au passage de paramètres par référence. Dans ce cas, la procédure ou la fonction ne travaille plus avec une copie de la variable passée en paramètre, mais directement avec celle-ci, plus précisément avec son adresse mémoire. En d'autres termes, si une modification est faite pendant la procédure ou la fonction, elle est conservée hors de celle-ci. À manipuler avec une prudence donc ! Pour spécifier un passage par référence, on utilise `ByRef` au lieu de `ByVal`. Appliquons cela à la fonction `QuiSuisJe` :

```
Public Sub QuiSuisJe(ByRef nom As String, ByRef prenom As
✂ String)
  MessageBox.Show("Je suis " + nom + " " + prenom)
  nom = "DOE"
  prenom = "John"
End Sub
```

Si maintenant nous exécutons le même bout de programme que précédemment, on obtient une première boîte de dialogue dans laquelle figure le texte "Je suis Martin Pierre", puis une deuxième contenant "Je suis DOE John". Le fait d'avoir passé les paramètres par référence et de les avoir modifiés dans la procédure provoque leur modification réelle dans le reste du programme. Encore une fois, manipulez le passage de paramètres par référence avec précaution.

Les variables de fonction et la portée

Dans les fonctions et les procédures vues précédemment, les comportements sont relativement simplistes, mais les opérations peuvent être beaucoup complexes. Vous pouvez par exemple utiliser des variables dont la "durée de vie" est limitée aux procédures ou fonctions dans lesquelles elles apparaissent. On dit de ces variables qu'elles sont "locales". Vous pourrez les utiliser seulement dans la fonction ou la procédure en question. Elles se déclarent et s'utilisent comme des variables normales :

```
Public Function Addition(ByVal nombre1 As Integer, ByVal
✂ nombre2 As Integer) As Integer
  Dim resultat As Integer
  Return (resultat)
End Function
```

Ici, `resultat` est une variable locale de la fonction `addition`. Elle ne peut être utilisée ailleurs dans le programme.

```
Dim somme As Integer
somme = Addition(3, 5)
MessageBox.Show("Somme = " + somme)
MessageBox.Show("Resultat = " + resultat)
```

La troisième ligne affiche une boîte de dialogue contenant "Somme = 8". On peut penser que `resultat` existe car cette variable est créée dans la fonction `addition`. Or, `resultat` est une variable locale de la fonction `Addition`. Elle n'existe donc pas pour le reste du programme. De ce fait, l'utiliser hors de `Addition` provoque une erreur et il est impossible d'exécuter la seconde instruction `Show`.

Cela nous permet d'introduire la notion de portée. La portée correspond à l'accessibilité d'une variable. La portée d'une variable locale d'une fonction ou d'une procédure est simplement cette fonction ou cette procédure. Au-delà, la variable n'est pas utilisable. Il en est de même

pour les boucles. Une variable déclarée dans une boucle n'est utilisable que dans cette boucle :

```
Dim compteur As Integer
For compteur = 1 to 100
  Dim sauvegarde As Integer
  Sauvegarde = compteur
Next
MessageBox.Show("Sauvegarde = " + sauvegarde)
```

La variable `sauvegarde` étant déclarée et utilisée dans la boucle `For`, sa portée se limite à cette boucle. C'est pourquoi l'instruction `Show` de ce bout de programme provoque une erreur. `sauvegarde` étant utilisée hors de sa portée, elle n'est pas accessible. C'est comme si elle n'existait pas, on ne peut pas s'en servir. En raison de la portée des variables, faites bien attention à l'endroit où vous les déclarez pour être sûr qu'elles existent au moment où vous vous en servez. Bien que ce principe puisse paraître subtil au premier abord, vous arriverez vite le maîtriser.