Spécification du langage Visual Basic .NET

Ce document décrit le langage Visual Basic .NET. Conçu pour fournir une description complète de ce langage, il ne constitue pas une vue d'ensemble des concepts ou un manuel de référence de l'utilisateur. Pour obtenir une vue d'ensemble des concepts du langage, consultez *Fonctionnalités du langage Visual Basic*. Pour obtenir des informations de référence sur le langage, consultez *Référence du langage Visual Basic*.

ISTA.ma
Un portail au service
de la formation professionnelle

Le Portail http://www.ista.ma

Que vous soyez étudiants, stagiaires, professionnels de terrain, formateurs, ou que vous soyez tout simplement intéressé(e) par les questions relatives aux formations professionnelle, aux métiers, **http://www.ista.ma** vous propose un contenu mis à jour en permanence et richement illustré avec un suivi quotidien de l'actualité, et une variété de ressources documentaires, de supports de formation ,et de documents en ligne (supports de cours, mémoires, exposés, rapports de stage …) .

Le site propose aussi une multitude de conseils et des renseignements très utiles sur tout ce qui concerne la recherche d'un emploi ou d'un stage : offres d'emploi, offres de stage, comment rédiger sa lettre de motivation, comment faire son CV, comment se préparer à l'entretien d'embauche, etc.

Les forums **http://forum.ista.ma** sont mis à votre disposition, pour faire part de vos expériences, réagir à l'actualité, poser des questionnements, susciter des réponses.N'hésitez pas à interagir avec tout ceci et à apporter votre pierre à l'édifice.

Notre Concept

Le portail **http://www.ista.ma** est basé sur un concept de gratuité intégrale du contenu & un modèle collaboratif qui favorise la culture d'échange et le sens du partage entre les membres de la communauté ista.

Notre Mission

Diffusion du savoir & capitalisation des expériences.

Notre Devise

Partageons notre savoir

Notre Ambition

Devenir la plate-forme leader dans le domaine de la Formation Professionnelle.

Notre Défi

Convaincre de plus en plus de personnes pour rejoindre notre communauté et accepter de partager leur savoir avec les autres membres.

Web Project Manager

- Badr FERRASSI : http://www.ferrassi.com

- contactez : admin@ista.ma

1. Introduction

De Visual Basic 1.0, qui a simplifié de façon radicale l'écriture d'applications Windows, à Visual Basic 4.0, qui a contribué à imposer COM2 comme l'architecture objet Windows standard, le langage Visual Basic constitue un élément essentiel de la plate-forme Windows depuis plus d'une décennie.

L'application d'aujourd'hui n'est plus un simple exécutable autonome résidant sur le disque dur de l'utilisateur mais une application distribuée, publiée sur Internet par un serveur Web. Pour répondre à cette évolution, Microsoft ne se contente plus de vous proposer simplement un système d'exploitation, mais va plus loin en fournissant également des services Web XML. L'élément clé de l'initiative de Microsoft pour la création de ces services Web XML est le .NET Framework. Cette infrastructure a été conçue dès le départ pour permettre aux développeurs d'écrire et de déployer aisément des applications Web complexes.

Visual Basic .NET est un composant fondamental du .NET Framework et constitue une étape supplémentaire dans l'évolution du langage. Ce langage de programmation de haut niveau pour le .NET Framework procure le point d'entrée le plus simple à la plate-forme Microsoft .NET.

1.1 Principes de conception

Visual Basic .NET reflète les principes de conception suivants :

- Ce langage est un descendant indiscutable de Visual Basic. Un programmeur en Visual Basic sera immédiatement à l'aise avec lui.
- Sa syntaxe et sa sémantique sont simples, rapides à apprendre et faciles à comprendre. Dans ce langage, les fonctionnalités non intuitives sont évitées.
- Il offre aux développeurs les principales fonctionnalités du .NET Framework et respecte les conventions de cette infrastructure.
- Sa mise à niveau à partir de Visual Basic n'est pas très compliquée.
- Dans la mesure où le .NET Framework prend explicitement en charge plusieurs langages informatiques, il fonctionne parfaitement dans un environnement multilangage.
- Il assure une compatibilité optimale avec les précédentes versions de Visual Basic. Partout où cela est possible, Visual Basic .NET possède la même syntaxe, la même sémantique et le même comportement à l'exécution que ses prédécesseurs.

Ces principes viennent s'ajouter aux principes de conception d'origine de Visual Basic :

- Ce langage d'écriture d'applications est aussi sécurisé que possible. En règle générale, Visual Basic, dans la définition du langage, est un parfait compromis entre fiabilité, simplicité d'utilisation et efficacité.
- Il s'agit d'un langage extrêmement abordable.
- Il permet de développer rapidement des programmes, tout en garantissant une fiabilité maximale.
- Le code qu'il génère est efficace et prévisible.
- Il fonctionne aussi bien comme un langage fortement typé que comme un langage faiblement typé, pour accroître l'exactitude du code utilisateur dans le premier cas et pour accélérer la procédure de développement dans le second.

1.2 Notation grammaticale

Cette spécification décrit deux principales grammaires : une grammaire lexicale et une grammaire syntaxique. La grammaire lexicale définit la manière dont les caractères peuvent être combinés pour former des jetons, alors que la grammaire syntaxique détermine la façon dont les jetons peuvent être combinés pour former des programmes Visual Basic .NET. Il existe en outre plusieurs grammaires secondaires utilisées pour les opérations de prétraitement telles que la compilation conditionnelle.

> **Remarque** Les grammaires décrites dans cette spécification ont été conçues pour être lisibles par les utilisateurs, et non de façon formelle (c'est-à-dire utilisables par LEX ou YACC).

Toutes les grammaires utilisent une notation BNF (*Backus-Naur Form*) modifiée, qui consiste en un ensemble de productions composées de noms terminaux et non terminaux. Chaque nom non terminal est défini par une ou plusieurs productions. Dans une production, les noms non terminaux sont indiqués en *italique*, alors que les noms terminaux sont représentés en **gras**. Le texte sans mise en forme particulière et entouré des métasymboles que sont les crochets pointus représente des noms terminaux informels (par exemple, « < tous les caractères Unicode > »). Chaque grammaire débute par le nom non terminal *Début*.

La casse n'a pas d'importance dans les programmes Visual Basic .NET. Par souci de simplicité, tous les noms terminaux sont indiqués dans une casse standard, mais n'importe quelle casse peut être utilisée. Les noms terminaux qui sont des éléments imprimables du jeu de caractères ASCII sont représentés par leurs caractères ASCII correspondants. Visual Basic .NET n'est pas sensible à la chasse pour la mise en correspondance des noms terminaux, ce qui permet d'aligner des caractères Unicode à pleine chasse sur leurs équivalents Unicode à demi-chasse.

Un ensemble de productions commence par un nom non terminal, suivi de deux signes deux-points et d'un signe égal. La partie droite contient une production terminale ou non terminale. Une production non terminale peut présenter plusieurs productions séparées par le métasymbole barre verticale (|). Les éléments encadrés par des crochets droits ([]) sont

facultatifs. Le métasymbole plus (+) à la suite d'un élément indique que ce dernier peut avoir une ou plusieurs occurrences.

Des sauts de ligne et des retraits peuvent être ajoutés afin d'améliorer la lisibilité et ne font pas partie de la production.

2. Grammaire lexicale

La compilation d'un programme Visual Basic .NET commence par traduire le flux brut de caractères Unicode en une série de lignes logiques constituées d'un ensemble ordonné de jetons lexicaux. Une ligne logique s'étend à partir du début du flux ou d'un terminateur de ligne jusqu'au terminateur de ligne suivant qui n'est pas précédé d'un signe de continuité de ligne ou jusqu'à la fin du flux.

Début ::= [*LigneLogique*+]

LigneLogique ::= [*ÉlémentLigneLogique*+] [*Commentaire*] *TerminateurLigne*

ÉlémentLigneLogique ::= E*space* | *Jeton*

Jeton ::= *Identificateur* | *MotClé* | *Littéral* | *Séparateur* | *Opérateur*

2.1 Caractères et lignes

Tous les programmes Visual Basic .NET sont composés de caractères issus du jeu de caractères Unicode.

Caractère ::= < tout caractère Unicode à l'exception d'un *TerminateurLigne* >

2.2 Identificateurs

Les identificateurs Visual Basic .NET sont conformes à la norme Unicode 3.0, rapport 15, annexe 7, sauf qu'ils peuvent commencer par un caractère de soulignement (connecteur). Si un identificateur commence par un caractère de soulignement, il doit contenir au minimum un autre identificateur valide pour le distinguer d'une continuité de ligne.

Contrairement aux identificateurs avec séquence d'échappement, les identificateurs standard ne peuvent pas correspondre à des mots clés. Un *identificateur avec séquence d'échappement* est un identificateur délimité par des crochets droits. Les identificateurs avec séquence d'échappement suivent les mêmes règles que les identificateurs standard, excepté qu'ils peuvent correspondre à des mots clés et qu'ils n'ont pas toujours des caractères de type.

La longueur maximale d'un identificateur est de 16 383 caractères.

L'exemple suivant définit une classe nommée `class` avec une méthode **Shared** appelée `shared` qui prend un paramètre appelé `boolean` :

```
Class [class]
   Shared Sub [shared](ByVal [boolean] As Boolean)
      If [boolean] Then
            Console.WriteLine("true")
      Else
            Console.WriteLine("false")
      End If
   End Sub
End Class

Module Module1
   Sub Main()
      [class].[shared](True)
   End Sub
End Module
```

Les identificateurs ne respectent pas la casse, c'est pourquoi deux identificateurs sont considérés comme équivalents s'ils ne diffèrent que par la casse.

> **Remarque** Lors de la comparaison d'identificateurs, les mappages de casse un-à-un de la norme Unicode sont utilisés et tout mappage de casse spécifique aux paramètres régionaux est ignoré.

Identificateur ::=
IdentificateurSansSéquenceÉchappement [*CaractèreType*] |
IdentificateurAvecSéquenceÉchappement

IdentificateurSansSéquenceÉchappement ::= < *NomIdentificateur* mais pas *MotClé* >

IdentificateurAvecSéquenceÉchappement ::= **[** *NomIdentificateur* **]**

NomIdentificateur ::= DébutIdentificateur [*CaractèreIdentificateur*+]

DébutIdentificateur ::=
CaractèreAlpha |
CaractèreSoulignement CaractèreIdentificateur

CaractèreIdentificateur ::=
CaractèreSoulignement |
CaractèreAlpha |
CaractèreNumérique |
CaractèreCombinaison |
CaractèreMiseEnForme

CaractèreAlpha ::= < caractère alphabétique Unicode (classes Lu, Ll, Lt, Lm, Lo, Nl) >

CaractèreNumérique ::= < caractère chiffre décimal Unicode (classe Nd) >

CaractèreCombinaison ::= < caractère de combinaison Unicode (classes Mn, Mc) >

CaractèreMiseEnForme ::= < caractère de mise en forme Unicode (classe Cf) >

CaractèreSoulignement ::= < caractère de connexion Unicode (classe Pc) >

NomIdentificateur ::= *Identificateur* | *Mot clé*

2.3 Mots clés

Un mot clé est un mot doté d'une signification spéciale dans certaines constructions de langage. Tous les mots clés sont réservés.

> **Remarque** **Variant** et **Let** sont considérés comme des mots clés, même s'ils ne sont plus utilisés dans Visual Basic .NET.

MotClé ::= < membre de la table des mots clés >

AddHandler	**AddressOf**	**AndAlso**	**Alias**
And	**Ansi**	**As**	**Assembly**
Auto	**Boolean**	**ByRef**	**Byte**
ByVal	**Call**	**Case**	**Catch**
CBool	**CByte**	**CChar**	**CDate**
CDec	**CDbl**	**Char**	**CInt**
Class	**CLng**	**CObj**	**Const**
CShort	**CSng**	**CStr**	**CType**
Date	**Decimal**	**Declare**	**Default**
Delegate	**Dim**	**DirectCast**	**Do**
Double	**Each**	**Else**	**ElseIf**
End	**Enum**	**Erase**	**Error**
Event	**Exit**	**False**	**Finally**
For	**Friend**	**Function**	**Get**
GetType	**GoTo**	**Handles**	**If**
Implements	**Imports**	**In**	**Inherits**
Integer	**Interface**	**Is**	**Let**
Lib	**Like**	**Long**	**Loop**
Me	**Mod**	**Module**	**MustInherit**
MustOverride	**MyBase**	**MyClass**	**Namespace**
New	**Next**	**Not**	**Nothing**
NotInheritable	**NotOverridable**	**Object**	**On**
Option	**Optional**	**Or**	**OrElse**
Overloads	**Overridable**	**Overrides**	**ParamArray**
Preserve	**Private**	**Property**	**Protected**
Public	**RaiseEvent**	**ReadOnly**	**ReDim**
REM	**RemoveHandler**	**Resume**	**Return**
Select	**Set**	**Shadows**	**Shared**

Short	**Single**	**Static**	**Step**
Stop	**String**	**Structure**	**Sub**
SyncLock	**Then**	**Throw**	**To**
True	**Try**	**TypeOf**	**Unicode**
Until	**Variant**	**When**	**While**
With	**WithEvents**	**WriteOnly**	**Xor**

2.4 Littéraux

Un littéral est une représentation textuelle d'une valeur particulière d'un type. Les littéraux peuvent être de différents types : booléen, entier, en virgule flottante, de chaîne, de caractère et de date.

Littéral ::=
  LittéralBooléen |
  LittéralNumérique |
  LittéralChaîne |
  LittéralCaractère |
  LittéralDate |
  Nothing

LittéralNumérique ::= *LittéralEntier* | *LittéralVirguleFlottante*

2.4.1 Littéraux booléens

True et **False** sont des littéraux de type booléen (**Boolean**) qui correspondent respectivement aux états vrai et faux.

LittéralBooléen ::= **True** | **False**

2.4.2 Littéraux entiers

Les littéraux entiers peuvent être de type décimal (base 10), hexadécimal (base 16) ou octal (base 8). Un littéral entier décimal est une chaîne de chiffres décimaux. Un littéral hexadécimal est un **&H** suivi d'une chaîne de chiffres hexadécimaux (**0**-**9**, **A**-**F**). Enfin, un littéral octal est un **&O** suivi d'une chaîne de chiffres octaux (**0**-**7**). Les littéraux décimaux représentent directement la valeur décimale du littéral intégral, tandis que les littéraux octaux et hexadécimaux représentent la valeur binaire du littéral entier (donc, **&H8000S** correspond à –32 768, et non à une exception de dépassement de capacité).

Le type d'un littéral est déterminé par sa valeur ou par le caractère de type qui le suit. Si aucun caractère de type n'est spécifié, les valeurs situées dans la plage de valeurs autorisées pour le type **Integer** prennent le type **Integer** et les valeurs situées en dehors de la plage du type

Integer prennent le type **Long**. Si le type d'un littéral entier ne correspond pas à la taille requise pour contenir le littéral entier, une erreur de compilation est générée.

LittéralEntier ::= *ValeurLittéralIntégral* [*CaractèreTypeIntégral*]

ValeurLittéralIntégral ::= *LittéralEntier* | *LittéralHex* | *LittéralOctal*

CaractèreTypeIntégral ::=
  CaractèreShort |
  CaractèreInteger |
  CaractèreLong |
  CaractèreTypeInteger |
  CaractèreTypeLong

CaractèreShort ::= **S**

CaractèreInteger ::= **I**

CaractèreLong ::= **L**

LittéralEntier ::= *Chiffre+*

LittéralHex ::= **& H** *ChiffreHex+*

LittéralOctal ::= **& O** *ChiffreOctal+*

Chiffre ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

ChiffreHex ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F**

ChiffreOctal ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

2.4.3 Littéraux en virgule flottante

Un littéral en virgule flottante est un littéral entier suivi d'une virgule décimale facultative (caractère virgule ASCII) et de la mantisse, puis d'un exposant de base 10 facultatif. Par défaut, il est du type **Double**. Si un caractère de type **Single**, **Double** ou **Decimal** est spécifié, le littéral présente ce type. Si le type d'un littéral en virgule flottante ne correspond pas à la taille requise pour contenir le littéral en virgule flottante, une erreur de compilation est générée.

LittéralVirguleFlottante ::=
  ValeurLittéralVirguleFlottante [*CaractèreTypeVirguleFlottante*] |
  LittéralEntier CaractèreTypeVirguleFlottante

CaractèreTypeVirguleFlottante ::=
  CaractèreSingle |
  CaractèreDouble |
  CaractèreDecimal |
  CaractèreTypeSingle |

CaractèreTypeDouble |
*Caractère*Type*Decimal*

CaractèreSingle ::= **F**

CaractèreDouble ::= **R**

CaractèreDecimal ::= **D**

ValeurLittéralVirguleFlottante ::=
LittéralEntier **.** *LittéralEntier* [*Exposant*] |
. *LittéralEntier* [*Exposant*] |
Exposant LittéralEntier

Exposant ::= **E** [*Signe*] *LittéralEntier*

Signe ::= **+** | **-**

2.4.4 Littéraux de chaîne

Un littéral de chaîne est une séquence de zéro caractère Unicode ou plus qui débute et se termine par un caractère ASCII guillemet double. Au sein d'une chaîne, une séquence de deux guillemets doubles correspond à une séquence d'échappement qui représente un seul guillemet double dans la chaîne. Une constante de chaîne est du type **String**.

L'exemple suivant illustre divers littéraux de chaîne :

```
Dim a As String = "hello, world"
Dim b As String = "hello    world"
Dim c As String = "Joe said ""Hello"" to me"
Dim d As String = "\\server\share\file.txt"
```

Chaque littéral de chaîne ne produit pas nécessairement une nouvelle instance de chaîne. Si un programme comprend deux littéraux de chaîne (ou plus) qui sont équivalents d'après l'opérateur d'égalité de chaînes à l'aide de la sémantique de comparaison binaire, ces littéraux de chaîne font référence à la même instance de chaîne. Par exemple, la sortie du programme suivant est `True` car les deux littéraux font référence à la même instance de chaîne.

```
Module Test
   Sub Main()
      Dim a As Object = "hello"
      Dim b As Object = "hello"
      Console.WriteLine(a Is b)
   End Sub
End Module
```

LittéralChaîne ::= " [*CaractèreString+*] "

CaractèreString ::= < *Caractère* à l'exception de " > | ""

2.4.5 Littéraux de caractère

Un littéral de caractère représente un caractère Unicode unique de type **Char**. Une séquence de deux guillemets doubles correspond à une séquence d'échappement qui représente un seul guillemet double dans la chaîne.

LittéralCaractère ::= **"** *CaractèreString* **" C**

2.4.6 Littéraux de date

Un littéral de date représente un point temporel particulier exprimé sous la forme d'une valeur de type **Date**. Le littéral peut spécifier à la fois une date et une heure, uniquement une date ou uniquement une heure. Si la valeur de la date est omise, la date du 1er janvier de l'an 1 (ap. J.-C.) est utilisée par défaut. Si la valeur de l'heure est omise, l'heure 12:00:00 est utilisée par défaut.

Pour éviter les problèmes d'interprétation de l'année dans une date, la valeur de l'année ne peut pas être exprimée sous la forme de deux chiffres. Pour indiquer une date comprise dans le premier siècle de notre ère, la date en question doit être précédée de zéros non significatifs.

Une heure peut être définie soit par une valeur au format 24 heures, soit par une valeur au format 12 heures ; si la mention AM ou PM est omise, le format 24 heures est pris en compte par défaut. Si une valeur d'heure n'indique pas les minutes, le littéral **0** est utilisé par défaut. Si une valeur d'heure n'indique pas les secondes, le littéral **0** est utilisé par défaut. Si ni les minutes ni les secondes ne sont indiquées, la mention AM ou PM doit absolument être spécifiée. Si la date spécifiée ne figure pas dans la plage de valeurs autorisées pour le type **Date**, une erreur de compilation se produit.

LittéralDate ::= **#** [*Espace+*] [*ValeurDate*] [*Espace+*] [*ValeurHeure*] [*Espace+*] **#**

ValeurDate ::= *ValeurMois SéparateurDate ValeurJour SéparateurDate ValeurAnnée*

SéparateurDate ::= **/** | **-**

ValeurHeure ::= *ValeurHeures* [**:** *ValeurMinutes*] [**:** *ValeurSecondes*] [*Espace+*] [*AMPM*]

ValeurMois ::= *LittéralEntier*

ValeurJour ::= *LittéralEntier*

ValeurAnnée ::= *LittéralEntier*

ValeurHeures ::= *LittéralEntier*

ValeurMinutes ::= *LittéralEntier*

ValeurSecondes ::= *LittéralEntier*

AMPM ::= **AM** | **PM**

2.4.7 Nothing

Nothing est un littéral spécial, qui est considéré comme ne présentant pas de type et qui peut être converti dans tous les types du système de types. Lorsqu'il est converti dans un type particulier, il est l'équivalent de la valeur par défaut de ce type.

Nothing ::= **Nothing**

2.5 Séparateurs

Les caractères ASCII suivants sont des séparateurs :

Séparateur ::= **(** | **)** | **!** | **#** | , | **.** | :

2.6 Caractères d'opérateur

Les caractères ASCII suivants représentent des opérateurs et peuvent être combinés pour représenter d'autres opérateurs :

Opérateur ::= **&** | ***** | **+** | **-** | / | \ | ^ | < | = | >

4. Concepts généraux

Ce chapitre traite d'un certain nombre de concepts que vous devez absolument connaître afin de comprendre la sémantique du langage Microsoft Visual Basic .NET. La plupart de ces concepts sont sans doute bien connus des programmeurs en Visual Basic ou en C/C++, mais les définitions précises qu'ils présentent dans Visual Basic .NET risquent de différer.

Voir aussi

Déclarations | Héritage | Implémentation | Polymorphisme | Types d'accès | Portée | Noms d'espaces de noms et de types | Résolution de noms | Attributs

4.1 Déclarations

Un programme Visual Basic .NET est composé d'entités nommées. La présentation de ces entités, qui représentent la « signification » du programme, s'effectue au moyen de déclarations.

Les *espaces de noms*, qui se situent au niveau supérieur, sont des entités qui organisent d'autres entités, comme les types et les espaces de noms imbriqués. Les *types* sont des entités qui décrivent des valeurs et définissent du code exécutable. Ils peuvent contenir des membres de type et des types imbriqués. Les *membres de type* sont des constantes, des variables, des méthodes, des propriétés, des événements, valeurs d'énumération et des constructeurs.

Une entité qui peut en contenir d'autres définit un *espace de déclaration*. D'autres entités sont introduites dans cet espace de déclaration soit par des déclarations, soit par héritage ; l'espace de déclaration conteneur est appelé *contexte de déclaration* de ces entités. La déclaration d'une entité dans un espace de déclaration définit à son tour un nouvel espace de déclaration qui peut comporter d'autres déclarations d'entité imbriquées, c'est pourquoi les déclarations figurant dans un programme forment une hiérarchie d'espaces de déclaration.

Sauf dans le cas de membres de type surchargé, les déclarations ne peuvent pas introduire des entités de même nom et de même nature dans le même contexte de déclaration. De plus, un espace de déclaration ne peut jamais contenir différents types d'entités de même nom (une variable et une méthode portant un nom identique, par exemple).

L'espace de déclaration d'un espace de noms est dit « ouvert », ce qui signifie que deux déclarations d'espace de noms portant le même nom complet font partie du même espace de déclaration. Dans l'exemple suivant, les deux déclarations d'espace de noms du haut appartiennent au même espace de déclaration, qui dans ce cas déclare deux classes dont les noms complets sont `Megacorp.Data.Customer` et `Megacorp.Data.Order` :

```
Namespace Megacorp.Data
   Class Customer
   End Class
End Namespace
Namespace Megacorp.Data
   Class Order
   End Class
End Namespace
```

Puisque les deux déclarations font partie du même espace de déclaration, une erreur de compilation aurait lieu si les deux contenaient une déclaration d'une classe homonyme.

L'espace de déclaration d'un bloc comprend les éventuels blocs imbriqués. Par conséquent, dans l'exemple suivant, les méthodes `F` et `G` sont erronées parce que le nom `i` est déclaré dans le bloc externe et ne peut plus l'être à nouveau dans le bloc interne. Cependant, la méthode `H` est valide car les deux `i` sont déclarés dans des blocs non imbriqués distincts.

```
Class A
   Sub F()
      Dim i As Integer = 0
      If True Then
         Dim i As Integer = 1 'Error; i already declared in outer block.
      End If
   End Sub

   Sub G()
```

```
      If True Then
         Dim i As Integer = 0 'Error; i already declared in outer block.
      End If
      Dim i As Integer = 1
   End Sub

   Sub H()
      If True Then
            Dim i As Integer = 0
      End If
      If True Then
            Dim i As Integer = 1
      End If
   End Sub

End Class
```

4.2 Portée

La portée du nom d'une entité est l'ensemble de tous les espaces de déclaration dans lesquels il est possible de faire référence à ce nom sans qualification. En général, la portée d'un nom d'entité correspond à l'intégralité de son contexte de déclaration ; la déclaration d'une entité peut toutefois contenir des déclarations imbriquées d'entités homonymes. Dans ce cas, l'entité imbriquée occulte l'entité externe, et l'accès à l'entité masquée n'est possible que par la qualification.

L'occultation par l'imbrication touche les espaces de noms ou les types imbriqués dans des espaces de noms, les types imbriqués dans d'autres types et les corps de membres de type. L'occultation par l'imbrication de déclarations s'effectue toujours implicitement ; elle ne nécessite aucune syntaxe explicite.

L'exemple suivant comprend deux méthodes. Dans la méthode `F`, la variable d'instance `i` est occultée par la variable locale `i` et, dans la méthode `G`, `i` fait toujours référence à la variable d'instance.

```
Class A
   Private i As Integer = 0
   Sub F()
      Dim i As Integer = 1
   End Sub
   Sub G()
      i = 1
   End Sub
End Class
```

Lorsqu'un nom d'une portée interne masque un nom d'une portée externe, il masque toutes les occurrences surchargées de ce nom. Dans l'exemple suivant, l'appel `F(1)` invoque l'instance `F` déclarée dans `Inner` car toutes les occurrences externes de `F` sont masquées par la déclaration interne. C'est pour cette raison en outre que l'appel `F("Hello")` provoque une erreur.

```
Class Outer
```

```
   Overloads Shared Sub F(i As Integer)
   End Sub
   Overloads Shared Sub F(s As String)
   End Sub

   Class Inner
      Sub G()
            F(1) ' Invokes Outer.Inner.F.
            F("Hello") ' Error.
      End Sub
      Shared Sub F(l As Long)
      End Sub
   End Class
```

End Class

4.3 Héritage

Une relation d'héritage est une relation par laquelle un type (le type *dérivé*) découle d'un autre type (le type de *base)*, de telle sorte que l'espace de déclaration du type dérivé contient implicitement tous les membres de type non constructeur accessibles à partir du type de base. Dans l'exemple suivant, la classe `A` est la classe de base directe de la classe `B`, et la classe `B` est dérivée de la classe `A`.

```
Class A
End Class
Class B
   Inherits A
End Class
```

Puisque la classe `A` ne spécifie pas explicitement une classe de base directe, sa classe de base directe implicite est **Object**.

L'héritage obéit à des principes importants :

- L'héritage est transitif. Si le type *C* est dérivé du type *B*, lequel est dérivé du type *A*, le type *C* hérite des membres de type déclarés dans le type *B* ainsi que des membres de type déclarés dans le type *A*.
- Un type dérivé étend son type de base direct, mais il ne peut pas le restreindre. Un type dérivé peut ajouter de nouveaux membres de type et peut occulter des membres de type hérités, mais il ne peut pas supprimer la définition d'un membre de type hérité.
- Étant donné qu'une instance d'un type contient tous les membres de type de son type de base, une conversion a toujours lieu d'un type dérivé vers son type de base.
- Le système de types étant unifié dans Visual Basic .NET, tous les types doivent présenter un type de base, excepté le type **Object**. Par conséquent, **Object** est le type de base ultime de tous les types, et tous les types peuvent y être convertis.

4.4 Implémentation

Il existe une relation d'implémentation lorsqu'un type déclare qu'il implémente une interface et qu'il implémente tous les membres de type de l'interface. Un type qui implémente une interface particulière peut être converti dans cette interface : il prend donc en charge certaines méthodes sans qu'il existe de relation d'héritage.

> **Remarque** Il est possible de déclarer des variables d'interfaces, mais seule une valeur appartenant à une classe qui implémente l'interface en question peut être assignée à ces variables.

Un type qui implémente une interface avec des membres de type hérités plusieurs fois doit toujours implémenter ces méthodes, bien qu'elles ne soient pas directement accessibles à partir de l'interface dérivée implémentée. Même les classes **MustInherit** doivent fournir des implémentations de tous les membres des interfaces implémentées ; elles peuvent toutefois différer l'implémentation de ces méthodes en les déclarant comme **MustOverride**.

4.5 Polymorphisme

Le polymorphisme offre la possibilité de modifier l'implémentation d'une méthode **Overridable**. Il permet à une méthode de base unique d'effectuer différentes actions en fonction du type au moment de l'exécution de l'instance qui appelle la méthode. Par opposition, l'implémentation d'une méthode non substituable est invariable ; l'implémentation demeure identique que la méthode soit appelée sur une instance de la classe dans laquelle elle est déclarée ou sur une instance d'une classe dérivée. Quand une méthode non substituable est appelée, le type au moment de la compilation de l'instance est le facteur déterminant.

Une méthode substituable peut également être **MustOverride**, ce qui signifie qu'elle ne fournit pas de corps de méthode et doit être substituée. Les méthodes **MustOverride** ne sont autorisées que dans les classes **MustInherit**.

L'insertion du modificateur **Shared** ou **Overridable** dans la déclaration d'une méthode **MustOverride** entraîne une erreur.

Dans l'exemple suivant, la classe `Shape` définit la notion abstraite d'un objet de forme géométrique qui peut se peindre lui-même :

```
MustInherit Public Class Shape
   Public MustOverride Sub Paint(g As Graphics, r As Rectangle)
End Class

Public Class Ellipse
   Inherits Shape
   Public Overrides Sub Paint(g As Graphics, r As Rectangle)
      g.drawEllipse(r)
   End Sub
End Class

Public Class Box
   Inherits Shape
```

```
   Public Overrides Sub Paint(g As Graphics, r As Rectangle)
      g.drawRect(r)
   End Sub
End Class
```

La méthode `Paint` est de type **MustOverride** car elle ne présente pas d'implémentation par défaut significative. Les classes `Ellipse` et `Box` sont des implémentations de `Shape` concrètes. Étant donné que ces classes ne sont pas **MustInherit**, elles doivent absolument se substituer à la méthode `Paint` et fournir la véritable implémentation.

4.6 Types d'accès

Une déclaration définit les types d'accès de l'entité qu'elle déclare. Le type d'accès d'une entité ne modifie pas la portée du nom de l'entité. Le *domaine d'accessibilité* d'une déclaration est l'ensemble de tous les espaces de déclaration dans lesquels l'entité déclarée est accessible.

Les cinq types d'accès sont **Public, Protected, Friend, Protected Friend** et **Private**. **Public** est le type d'accès le plus permissif, et les quatre autres types d'accès en sont des sous-ensembles. **Private** est le type d'accès le moins permissif, et les quatre autres types d'accès en sont des sur-ensembles.

Le type d'accès d'une déclaration est déterminé par un modificateur d'accès facultatif, qui peut être **Public**, **Protected**, **Friend**, **Private** ou la combinaison de **Protected** et **Friend**. Si aucun modificateur d'accès n'est spécifié, le type d'accès par défaut comme les types d'accès autorisés dépendent du contexte de déclaration.

- Les entités déclarées avec le modificateur **Public** ont un accès **Public**. Il n'existe aucune restriction quant à l'utilisation des entités **Public**.
- Les entités déclarées avec le modificateur **Protected** ont un accès **Protected**. L'accès **Protected** peut être spécifié uniquement sur des membres de classes (à la fois des membres de types réguliers et des classes imbriquées). Un membre **Protected** est accessible à une classe dérivée, à condition que le membre ne soit pas un membre d'instance ou que l'accès s'effectue via une instance de la classe dérivée. L'accès **Protected** n'est pas un sur-ensemble de l'accès **Friend**.
- Les entités déclarées avec le modificateur **Friend** ont un accès **Friend**. Une entité avec accès **Friend** n'est accessible qu'à l'intérieur du programme qui contient sa déclaration.
- Les entités déclarées avec les modificateurs **Protected Friend** disposent de l'accès défini par l'union des accès **Protected** et **Friend**.
- Les entités déclarées avec le modificateur **Private** ont un accès **Private**. Une entité **Private** n'est accessible qu'à l'intérieur de son contexte de déclaration, y compris les entités imbriquées éventuelles.

L'accessibilité stipulée par une déclaration ne dépend pas de l'accessibilité du contexte de déclaration. Ainsi, un type déclaré avec l'accès **Private** peut contenir un membre de type disposant de l'accès **Public**.

Le code suivant illustre divers domaines d'accessibilité :

```
Public Class A
   Public Shared X As Integer
   Friend Shared Y As Integer
   Private Shared Z As Integer
End Class

Friend Class B
   Public Shared X As Integer
   Friend Shared Y As Integer
   Private Shared Z As Integer

   Public Class C
      Public Shared X As Integer
      Friend Shared Y As Integer
      Private Shared Z As Integer
   End Class

   Private Class D
      Public Shared X As Integer
      Friend Shared Y As Integer
      Private Shared Z As Integer
   End Class
End Class
```

Les classes et membres possèdent les domaines d'accessibilité suivants :

- Le domaine d'accessibilité de `A` et de `A.X` est illimité.
- Le domaine d'accessibilité de `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X` et `B.C.Y` est le texte du programme conteneur.
- Le domaine d'accessibilité de `A.Y` est le texte du programme de `A`.
- Le domaine d'accessibilité de `B.Z` et de `B.D` est le texte du programme de `B`, y compris le texte du programme de `B.C` et de `B.D`.
- Le domaine d'accessibilité de `B.C.Z` est le texte du programme de `B.C`.
- Le domaine d'accessibilité de `B.D.X`, `B.D.Y` et `B.D.Z` est le texte du programme de `B.D`.

Comme l'illustre cet exemple, le domaine d'accessibilité d'un membre n'est jamais plus étendu que celui d'un type contenant. Par exemple, bien que tous les membres de `X` aient une accessibilité déclarée **Public**, tous sauf `A.X` possèdent des domaines d'accessibilité qui sont restreints par un type contenant.

ModificateurAccès ::= **Public** | **Protected** | **Friend** | **Private**

4.7 Noms d'espaces de noms et de types

Nombre de constructions de langage nécessitent la définition d'un espace de noms ou d'un type ; ceux-ci peuvent être spécifiés au moyen d'une forme qualifiée du nom de l'espace de noms ou du nom du type. Un nom qualifié se compose d'une série d'identificateurs séparés par des points ; l'identificateur situé à droite du point est résolu dans l'espace de déclaration défini par l'identificateur situé à gauche du point.

Le nom complet d'un espace de noms ou d'un type est un nom qualifié contenant le nom de tous les types et espaces de noms contenants. En d'autres termes, le nom complet d'un espace de noms ou d'un type est *N.T*, où *T* est le nom de l'entité et *N* est le nom complet de son entité contenante.

L'exemple ci-dessous montre plusieurs déclarations d'espace de noms et de type, accompagnées de leurs noms complets correspondants dans les commentaires inline.

```
Class A ' A
End Class
Namespace X ' X
   Class B ' X.B
      Class C ' X.B.C
      End Class
   End Class
   Namespace Y ' X.Y
      Class D ' X.Y.D
      End Class
   End Namespace
End Namespace
Namespace X.Y ' X.Y
   Class E ' X.Y.E
   End Class
End Namespace
```

Dans la mesure où il est possible que d'autres langages introduisent des types et des espaces de noms correspondant à des mots clés dans Visual Basic .NET, ce dernier reconnaît les mots clés comme faisant partie d'un nom qualifié pour autant qu'ils suivent un point. Les mots clés employés de cette façon sont traités comme des identificateurs.

IdentificateurQualifié ::=
  Identificateur |
  IdentificateurQualifié **.** *IdentificateurOuMotClé*

4.7.1 Résolution de noms

Prenons comme exemple un espace de noms ou un type portant le nom qualifié *N.R*. La procédure ci-dessous explique comment déterminer l'espace de noms ou de type auquel le nom qualifié fait référence :

1. Résolvez *N*, qui peut être un nom qualifié ou non qualifié.

2. Si la résolution de *N* échoue ou ne donne ni un espace de noms, ni un type, une erreur de compilation se produit. Si *R* correspond au nom d'un espace de noms ou d'un type accessible dans *N*, arrêtez.

3. Si *N* contient un ou plusieurs modules standard et si *R* correspond au nom d'un type accessible dans un et un seul module standard précis, arrêtez. Si *R* correspond au nom de types accessibles dans plusieurs modules standard, une erreur de compilation se produit.

4. Dans les autres cas, une erreur de compilation se produit.

Dans le cas d'un nom non qualifié *R*, la procédure ci-dessous décrit comment déterminer le nom d'espace de nom ou de type auquel ce nom non qualifié fait référence :

1. Pour chaque type imbriqué contenant la référence au nom, en allant du type le plus interne au type le plus externe, si *R* correspond au nom d'un type imbriqué dans le type en cours, arrêtez.

2. Pour chaque espace de noms imbriqué contenant la référence au nom, en allant de l'espace de noms le plus interne à l'espace de noms le plus externe (global), procédez comme suit :

    a. Si *R* correspond au nom d'un type ou d'un espace de noms imbriqué situé dans l'espace de noms en cours, arrêtez.

    b. Si l'espace de noms contient un ou plusieurs modules standard et si *R* correspond au nom d'un type imbriqué dans un et un seul module standard précis, arrêtez. Si *R* correspond au nom de types situés dans plusieurs modules standard, une erreur de compilation se produit.

3. Si le fichier source dispose d'un ou de plusieurs alias d'importation et si *R* correspond au nom de l'un d'eux, arrêtez.

4. Si le fichier source contenant la référence au nom possède une ou plusieurs importations :

    a. Si *R* correspond au nom d'un membre de type ou d'un espace de noms situé dans une et une seule importation, arrêtez. Si *R* correspond au nom de membres de type ou d'un espace de noms dans plusieurs importations, une erreur de compilation se produit.

    b. Si les importations contiennent un ou plusieurs modules standard et si *R* correspond au nom d'un type imbriqué dans un et un seul module standard précis, arrêtez. Si *R* correspond au nom de types situés dans plusieurs modules standard, une erreur de compilation se produit.

5. Si l'environnement de compilation définit un ou plusieurs alias d'importation et si *R* correspond au nom de l'un d'eux, arrêtez.

6. Si l'environnement de compilation définit une ou plusieurs importations :

    a. Si *R* correspond au nom d'un membre de type ou d'un espace de noms situé dans une et une seule importation, arrêtez. Si *R* correspond au nom de membres de type ou d'espaces de noms dans plusieurs importations, une erreur de compilation se produit.

    b. Autrement, si les importations contiennent un ou plusieurs modules standard et si *R* correspond au nom d'un type imbriqué dans un et un seul module standard

précis, arrêtez. Si *R* correspond au nom de types situés dans plusieurs modules standard, une erreur de compilation se produit.

7. Dans les autres cas, une erreur de compilation se produit.

> **Remarque** Ce processus de résolution implique notamment que, lors de la résolution de noms de types ou d'espaces de noms, les membres de type n'occultent pas les types ou les espaces de noms.

4.8 Attributs

Le langage Visual Basic .NET permet au programmeur de spécifier des informations déclaratives au sujet des entités définies dans le programme. Par exemple, l'apposition des modificateurs **Public**, **Protected**, **Friend**, **Protected Friend** ou **Private** à une méthode de classe spécifie le type d'accès de cette dernière.

Visual Basic .NET permet également aux programmeurs d'inventer de nouveaux types d'informations déclaratives et de les spécifier pour diverses entités de programme. Ces nouvelles sortes d'informations déclaratives sont définies par la déclaration de classes d'attributs, qui peuvent comporter des paramètres positionnels et des initialiseurs de variable/propriété. Ainsi, une infrastructure peut définir un attribut **Help** pouvant être placé sur des éléments de programme, tels que classes et méthodes, pour assurer le mappage de ces éléments sur de la documentation, comme l'illustre l'exemple ci-dessous :

```
Imports System
<AttributeUsage(AttributeTargets.All)> Public Class HelpAttribute
   Inherits Attribute

   Public Sub New(urlValue As String)
      Me.urlValue = urlValue
   End Sub
   Public Topic As String = Nothing
   Private urlValue As String

   Public ReadOnly Property Url() As String
      Get
         Return urlValue
      End Get
   End Property
End Class
```

Cet exemple définit une classe d'attributs nommée **HelpAttribute** (ou **Help** en abrégé), qui présente un paramètre positionnel (`url`) et un argument nommé (`Topic`). Les paramètres positionnels sont définis par les paramètres formels des constructeurs **Public** de la classe d'attributs, tandis que les paramètres nommés sont définis par des propriétés en lecture et en écriture **Public** de la classe d'attributs.

L'exemple suivant illustre plusieurs utilisations de l'attribut **:**

```
<Help("http://www.mycompany.com/.../Class1.htm")> Public Class Class1
   <Help("http://www.mycompany.com/.../Class1.htm", Topic := "F")> Public
Sub F()
   End Sub
End Class
```

Les nouvelles informations déclaratives sont spécifiées sur les entités au moyen de blocs d'attributs. Le .NET Framework permet de récupérer au moment de l'exécution des attributs individuels spécifiés sur des déclarations.

L'exemple suivant vérifie si la classe `Class1` possède un attribut **Help** et, si c'est le cas, écrit les valeurs `Topic` et `Url` associées.

```
Imports System
Class Test
   Shared Sub Main()
      Dim type As Type = GetType(Class1)
      Dim arr As Object() =
type.GetCustomAttributes(GetType(HelpAttribute), true)
      If arr.Length = 0 Then
         Console.WriteLine("Class1 has no Help attribute.")
      Else
         Dim ha As HelpAttribute = CType(arr(0), HelpAttribute)
         Console.WriteLine("Url = " & ha.Url & "Topic = " & ha.Topic)
      End If
   End Sub
End Class
```

5. Fichiers sources et espaces de noms

Un programme Visual Basic .NET se compose d'un ou de plusieurs fichiers sources. Lors de sa compilation, tous les fichiers sources sont traités simultanément ; dès lors, ceux-ci peuvent être dépendants les uns des autres, voire même de façon circulaire, sans que cela n'exige de déclaration anticipée. Par conséquent, en règle générale, l'ordre dans lequel les déclarations apparaissent dans le texte du programme n'a pas d'importance.

Un fichier source se compose d'un ensemble facultatif d'attributs, de directives d'option et de directives d'importation, suivi d'un corps. Le corps du fichier source joue le rôle d'une déclaration d'espace de noms implicite pour l'espace de noms global, ce qui signifie que toutes les déclarations situées au niveau supérieur d'un fichier source sont placées dans l'espace de noms global.

Source ::=
  [*DirectiveOption*+]
  [*DirectiveImports*+]
  [*Attributs*]
  [*CorpsEspaceNoms*]

5.1 Options de compilation

Quatre options de compilation influencent la sémantique du langage, mais seules trois de ces options – la sémantique de type strict, la sémantique de déclaration explicite et la sémantique

de comparaison – peuvent être précisées soit par l'environnement de compilation, soit par des instructions **Option**. La quatrième option, contrôles de dépassement sur les entiers, peut uniquement être définie par l'environnement de compilation. Une instruction **Option** s'applique uniquement au fichier source dans laquelle elle apparaît, et une seule instruction **Option** de chaque type peut figurer dans un fichier source.

DirectiveOption ::= *DirectiveOptionExplicit* | *DirectiveOptionStrict* | *DirectiveOptionCompare*

5.2 Instruction Imports

Les instructions **Imports** importent les noms des entités dans un fichier source, ce qui permet aux noms d'être référencés sans qualification.

Au sein des déclarations de membre d'un fichier source qui contient une instruction **Imports**, les types contenus dans l'espace de noms donné peuvent être référencés directement, comme l'illustre l'exemple suivant :

```
Imports N1.N2
Namespace N1.N2

   Class A
   End Class
End Namespace
Namespace N3

   Class B
      Inherits A
   End Class
End Namespace
```

Dans le fichier source de cet exemple, les membres de type de l'espace de noms `N1.N2` sont directement disponibles et, donc, la classe `N3.B` est dérivée de la classe `N1.N2.A`.

Les instructions **Imports** suivent les éventuelles instructions **Option** mais précèdent les éventuelles déclarations de type. L'environnement de compilation peut également définir des instructions **Imports** implicites.

Les instructions **Imports** rendent les noms disponibles dans un fichier source, mais ne déclarent rien dans l'espace de déclaration de l'espace de noms global. La portée des noms importés par une instruction **Imports** s'étend sur les déclarations de membre d'espace de noms contenues dans le fichier source. La portée d'une instruction **Imports** n'inclut pas spécifiquement d'autres instructions **Imports**, ni même d'autres fichiers sources. Les instructions **Imports** ne peuvent pas se référencer mutuellement.

> **Remarque** Les noms de type ou d'espace de noms figurant dans une instruction **Imports** sont toujours traités comme des noms complets. Ainsi, l'identificateur le plus à gauche dans un nom de type ou d'espace de noms est toujours résolu en l'espace de noms global et le reste de la résolution se poursuit d'après les règles normales de résolution de noms. Il s'agit là de la seule partie du langage qui applique une telle règle, laquelle garantit qu'un nom

ne peut pas être complètement « masqué » de la qualification. Sans cette règle, si un nom de l'espace de noms global était masqué dans un fichier source particulier, il serait impossible de spécifier des noms de cet espace de noms sous une forme qualifiée.

DirectiveImports ::= **Imports** *ClausesImports TerminateurLigne*

ClausesImports ::=
ClauseImports |
ClausesImports , *ClauseImports*

ClauseImports ::= ClauseAliasImports | *ClauseImportsNormale*

5.3 Espaces de noms

Les programmes Visual Basic .NET sont organisés au moyen d'espaces de noms, tant du point de vue de leur organisation interne que de la manière dont leurs éléments sont exposés à d'autres programmes. Les directives d'importation sont conçues pour permettre l'utilisation des espaces de noms.

Contrairement à d'autres entités, les espaces de noms sont ouverts. Ils peuvent être déclarés plusieurs fois dans le même programme et même dans de nombreux programmes, chaque déclaration fournissant des membres au même espace de noms. Dans l'exemple suivant, les deux déclarations d'espace de noms appartiennent au même espace de déclaration, qui déclare deux classes dont les noms complets sont `N1.N2.A` et `N1.N2.B` :

```
Namespace N1.N2
   Class A
   End Class
End Namespace
Namespace N1.N2
   Class B
   End Class
End Namespace
```

Puisque les deux déclarations font partie du même espace de déclaration, une erreur aurait lieu si les deux contenaient une déclaration d'un membre homonyme.

Il existe un espace de noms global dépourvu de nom, et dont les types et les espaces de noms imbriqués sont toujours accessibles sans qualification. La portée d'un membre d'espace de noms déclaré dans l'espace de noms global est le texte du programme tout entier. Par ailleurs, la portée d'un espace de noms ou d'un type déclaré dans un espace de noms dont le nom complet est **N** est le texte de programme de chaque espace de noms dont le nom complet de l'espace de noms correspondant commence par **N** ou est **N** lui-même.

Namespace, instruction

Déclare le nom d'un espace de noms.

```
Namespace { name | name.name }
   [ componenttypes ]
End Namespace
```

Éléments

name
Requis. Nom univoque d'identification de l'espace de noms.

componenttypes
Facultatif. Éléments qui composent l'espace de noms. Ceux-ci incluent (sans s'y limiter) les énumérations, structures, interfaces, classes, modules, délégués et d'autres espaces de noms.

End Namespace
Met fin à un bloc **Namespace**.

Notes

Les espaces de noms servent de système d'organisation : une façon de présenter des composants de programme qui sont exposés à d'autres programmes et applications.

Les espaces de noms sont toujours de type **Public** ; par conséquent, la déclaration d'un espace de noms ne peut pas comporter de modificateurs d'accès. Cependant, les composants contenus dans l'espace de noms peuvent avoir un accès **Public** ou **Friend**. S'il n'est pas indiqué, le type d'accès par défaut est **Friend**.

Exemple

L'exemple suivant déclare deux espaces de noms.

```
Namespace N1   ' Declares a namespace named N1.
   Namespace N2   ' Declares a namespace named N2 within N1.
      Class A   ' Declares a class within N1.N2.
      ' Add class statements here.
      End Class
   End Namespace
End Namespace
```

L'exemple suivant déclare plusieurs espaces de noms imbriqués sur une seule ligne, et est équivalent à l'exemple précédent.

```
Namespace N1.N2 ' Declares two namespaces; N1 and N2.
   Class A   ' Declares a class within N1.N2.
      ' Add class statements here.
   End Class
End Namespace
```

Espace de noms

Les espaces de noms permettent d'organiser les objets définis dans un assembly. Les assemblys peuvent contenir plusieurs espaces de noms, qui peuvent à leur tour contenir d'autres espaces de noms. Les espaces de noms permettent d'éviter les ambiguïtés et de simplifier les références lors de l'utilisation de grands groupes d'objets, tels que les bibliothèques de classes.

Par exemple, Visual Studio .NET définit la classe **ListBox** dans l'espace de noms **System.Windows.Forms**. Le fragment de code suivant montre comment déclarer une variable en utilisant le nom complet de cette classe :

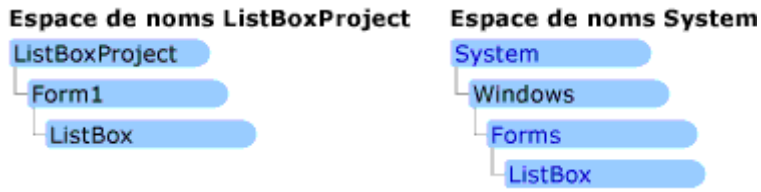
```
Dim LBox As System.Windows.Forms.ListBox
```

Les espaces de noms Visual Studio .NET permettent de résoudre le problème de l'*encombrement des espaces de noms*, par exemple lorsque le développeur d'une bibliothèque de classes est gêné par l'utilisation de noms similaires dans une autre bibliothèque. Ces conflits avec des composants existants sont parfois appelés *collisions de noms*.

Par exemple, si vous créez une nouvelle classe nommée `ListBox`, vous pouvez l'utiliser dans votre projet sans qualification. Cependant, si vous souhaitez utiliser la classe **ListBox** de Visual Studio .NET dans le même projet, vous devez utiliser un nom complet pour rendre la référence unique. Si la référence n'est pas unique, Visual Basic .NET génère une erreur en indiquant que le nom est ambigu. Le fragment de code suivant montre comment déclarer ces objets :

```
'   Define a new object based on your ListBox class.
Dim LBC as New ListBox
'   Define a new Windows.Forms ListBox control.
Dim MyLB as New System.Windows.Forms.ListBox
```

L'illustration suivante montre deux hiérarchies d'espaces de noms, chacune contenant un objet nommé `ListBox`.



Par défaut, tous les fichiers exécutables créés à l'aide de Visual Basic .NET contiennent un espace de noms portant le même nom que votre projet. Par exemple, si vous définissez un objet dans un projet nommé `ListBoxProject`, le fichier exécutable, **ListBoxProject.exe**, contient un espace de noms appelé `ListBoxProject`.

Plusieurs assemblys peuvent utiliser le même espace de noms. Visual Basic .NET les considère comme un ensemble unique de noms. Par exemple, vous pouvez définir des classes pour un espace de noms appelé `SomeNameSpace` dans un assembly nommé `Assemb1`, et définir des classes supplémentaires pour le même espace de noms à partir d'un assembly nommé `Assemb2`.

Noms complets

Les noms complets sont des références d'objet qui sont préfixées à l'aide du nom de l'espace de noms où l'objet est défini. Vous pouvez utiliser les objets définis dans d'autres projets si vous créez une référence à la classe (en cliquant sur **Ajouter une référence** dans le menu **Projet**) et si vous utilisez le nom complet de l'objet dans votre code. Le fragment de code suivant montre comment utiliser le nom complet d'un objet provenant de l'espace de noms d'un autre projet :

```
Dim LBC As New ListBoxProject.Form1.ListBox()
```

Les noms complets permettent d'éviter les conflits entre les noms, car le compilateur peut toujours déterminer l'objet en cours d'utilisation. Cependant, ces noms peuvent devenir longs

et encombrants. Pour contourner ce problème, utilisez l'instruction **Imports** pour définir un *alias* (nom abrégé que vous pouvez utiliser à la place d'un nom complet). Par exemple, le fragment de code suivant crée les alias de deux noms complets, puis utilise ces alias pour définir deux objets :

```
Imports LBControl = System.Windows.Forms.ListBox

Imports MyListBox = ListBoxProject.Form1.ListBox

Dim LBC As LBControl

Dim MyLB As MyListBox
```

Si vous utilisez l'instruction **Imports** sans alias, vous pouvez vous servir de tous les noms de l'espace de noms sans qualification, à condition qu'ils soient uniques à l'intérieur du projet. Si votre projet contient des instructions **Imports** pour les espaces de noms contenant des éléments qui portent le même nom, vous devez fournir un nom complet lorsque vous l'utilisez. Supposons, par exemple, que votre projet contienne les deux instructions **Imports** suivantes :

```
Imports MyProj1 ' This namespace contains a class called Class1.
Imports MyProj2 ' This namespace also contains a class called Class1.
```

Si vous essayez d'utiliser `Class1` sans fournir son nom complet, Visual Basic .NET génère une erreur en indiquant que le nom `Class1` est ambigu.

Instructions au niveau de l'espace de noms

Dans un espace de noms, vous pouvez définir des éléments tels que des modules, des interfaces, des classes, des délégués, des énumérations, des structures et d'autres espaces de noms. Vous ne pouvez pas définir des éléments tels que des propriétés, des procédures, des variables et des événements au niveau de l'espace de noms ; ces éléments doivent être déclarés dans des conteneurs tels que des modules, des structures ou des classes.

6. Types

Les deux catégories fondamentales de types dans Visual Basic .NET sont les *types valeur* et les *types référence*. Les types primitifs, les énumérations et les structures sont des types valeur. Les classes, les chaînes, les modules standard, les interfaces, les tableaux et les délégués sont des types référence.

Tous les types sont soit des types valeur, soit des types référence, à une exception près. Le type racine **Object**, qui est un alias de **System.Object**, est un type spécial dans la mesure où il ne constitue ni un type référence, ni un type valeur, et ne peut pas être instancié. Une variable de type **Object** peut donc contenir soit un type valeur, soit un type référence.

NomType ::= *IdentificateurQualifié* | **Object** | *NomTypePrimitif* | *NomTypeTableau*

6.1 Types valeur et types référence

Bien que les types valeur et les types référence puissent être similaires sur le plan de la syntaxe de déclaration, ils diffèrent du point de vue de leur sémantique.

Les types référence sont stockés dans le tas runtime ; ils ne sont accessibles que par une référence à cet emplacement de stockage. Ceci permet au « garbage collector » d'assurer un suivi des références en suspens à une instance particulière et de libérer l'instance lorsqu'il ne reste plus aucune référence. Une variable d'un type référence contient toujours une référence à une valeur de ce type ou une référence null. Une référence null fait référence à rien du tout. Aucune opération effectuée avec une référence null n'est valide, à l'exception de l'assignation. L'assignation d'un type référence à une variable crée une copie de la référence, et non une copie de la valeur référencée.

Les types valeur sont stockés directement dans la pile, soit au sein d'un tableau, soit au sein d'un autre type. Lorsque l'emplacement qui contient une instance d'un type valeur est détruit, cette instance l'est également. L'accès aux types valeur s'effectue toujours directement : il n'est pas possible de créer une référence à un type valeur. L'interdiction d'une telle référence rend impossible tout référence à une instance de classe valeur ayant été détruite. Une variable d'un type valeur contient toujours une valeur de ce type. À la différence de la valeur d'un type référence, la valeur d'un type valeur ne peut pas être une référence null et ne peut pas référencer un objet d'un type plus dérivé. L'assignation d'un type valeur à une variable crée une copie de la valeur assignée.

L'exemple suivant illustre cette différence :

```
Imports System
Class Class1
   Public Value As Integer = 0
End Class

Module Test
   Sub Main()
      Dim val1 As Integer = 0
      Dim val2 As Integer = val1
      val2 = 123
      Dim ref1 As New Class1()
      Dim ref2 As Class1 = ref1
      ref2.Value = 123
      Console.WriteLine("Values: " & val1 & ", " & val2)
      Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)
   End Sub
End Module
```

Ce programme produit la sortie suivante :

```
Values: 0, 123
Refs: 123, 123
```

L'assignation à la variable locale `val2` n'influe pas sur la variable locale `val1` car toutes deux possèdent un type valeur (le type **Integer**) et chaque variable locale ayant un type valeur possède son propre emplacement de stockage. Par contre, l'assignation `ref2.Value = 123;` affecte l'objet auquel font référence tant `ref1` que `ref2`.

6.2 Types primitifs

Les types primitifs sont identifiés au moyen de mots clés, qui constituent des alias de types prédéfinis dans l'espace de noms **System**. Un type primitif ne se distingue en rien du type structure dont il constitue l'alias : ainsi, l'insertion du mot réservé **Byte** revient exactement à écrire **System.Byte.**

Puisqu'un type primitif représente l'alias d'un type régulier, tous les types primitifs possèdent des membres. Par exemple, **Integer** comprend les membres déclarés dans **System.Int32.** Les littéraux peuvent être traités comme des instances de leurs types correspondants.

Les types primitifs se distinguent d'autres types structure en ce qu'ils autorisent certains opérations supplémentaires :

- Tous les types primitifs permettent la création de valeurs par l'écriture de littéraux. Par exemple, `123I` est un littéral de type **Integer**.
- Il est possible de déclarer des constantes des types primitifs.
- Lorsque les opérandes d'une expression constituent tous des constantes de type primitif, il est possible pour le compilateur d'évaluer l'expression au moment de la compilation. Une telle expression est appelée expression de constante.

Visual Basic .NET définit les types primitifs suivants :

- Les types valeur intégrale **Byte** (entier non signé sur 1 octet), **Short** (entier signé sur 2 octets), **Integer** (entier signé sur 4 octets) et **Long** (entier signé sur 8 octets). Ces types sont mappés respectivement sur **System.Byte**, **System.Int16**, **System.Int32** et **System.Int64**. La valeur par défaut d'un type intégral équivaut au littéral **0**.
- Les types valeur en virgule flottante **Single** (virgule flottante sur 4 octets) et **Double** (virgule flottante sur 8 octets). Ces types sont mappés respectivement sur **System.Single** et **System.Double**. La valeur par défaut d'un type en virgule flottante équivaut au littéral **0**. Le type **Decimal** (valeur décimale sur 16 octets), qui est mappé sur **System.Decimal**. La valeur par défaut du type **Decimal** équivaut au littéral **0D**.
- Le type valeur **Boolean**, qui représente une valeur vrai/faux et est généralement le résultat d'une opération relationnelle ou logique. Le littéral est de type **System.Boolean**. La valeur par défaut du type **Boolean** équivaut au littéral **False**.
- Le type valeur **Date**, qui représente une date et/ou une heure et est mappé sur **System.DateTime**. La valeur par défaut du type **Date** équivaut au littéral **# 01/01/0001 12:00:00AM #**.
- Le type valeur **Char**, qui représente un seul caractère Unicode et est mappé sur **System.Char**. La valeur par défaut du type **Char** équivaut à l'expression de constante **ChrW(**0**)**.
- Le type valeur **String**, qui représente une séquence de caractères Unicode et est mappé sur **System.String**. La valeur par défaut du type **String** est une référence null.

NomTypePrimitif ::= *NomTypeNumérique* | **Boolean** | **Date** | **Char**

NomTypeNumérique ::= *NomTypeIntégral* | *NomTypeVirguleFlottante* | **Decimal**

NomTypeIntégral ::= **Byte** | **Short** | **Integer** | **Long**

NomTypeVirguleFlottante ::= **Single** | **Double**

6.3 Énumérations

Les énumérations sont des types qui héritent de **System.Enum** et qui représentent symboliquement un ensemble de valeurs de l'un des types intégraux primitifs. La valeur par défaut d'un type énumération *E* correspond au résultat de l'expression **CType(**0, *E***)**.

Le type sous-jacent d'une énumération doit être un type intégral qui peut représenter toutes les valeurs d'énumérateur définies dans l'énumération. Si un type sous-jacent est spécifié, il doit s'agir de **Byte**, **Short**, **Integer** ou **Long**. Si aucun type sous-jacent n'est défini explicitement, le type par défaut est **Integer**.

L'exemple suivant déclare une énumération dont le type sous-jacent est **Long** :

```
Enum Color As Long
   Red
   Green
   Blue
End Enum
```

Un développeur peut choisir d'utiliser le type sous-jacent **Long**, comme dans l'exemple, pour permettre l'utilisation de valeurs comprises dans la plage de valeurs de **Long** mais pas dans la plage d'Integer ou bien pour se réserver cette option pour l'avenir.

DéclarationÉnum ::=
[*Attributs*] [*ModificateurÉnum+*] **Enum** *Identificateur*
[**As** *NomTypeIntégral*] *TerminateurLigne*
  DéclarationMembreÉnum+
  End Enum *TerminateurLigne*

ModificateurÉnum ::= ModificateurAccès | **Shadows**

6.4 Structures

Les structures sont semblables aux classes dans la mesure où elles représentent des structures de données pouvant contenir des données membres et des fonctions membres. Toutefois, à l'inverse des classes, les structures sont des types valeur et ne doivent pas être allouées à un tas. Une variable d'un type structure contient directement les données de la structure, alors qu'une variable d'un type classe contient une référence aux données, ces dernières étant

connues sous le nom d'objet. Une structure hérite de **System.ValueType** et ne peut pas être l'objet d'un héritage.

Dans le cas des classes, il est possible de voir deux variables référencer le même objet et donc de voir des opérations sur une variable particulière affecter l'objet référencé par l'autre variable. Avec les structures, les variables disposent chacune de leur propre copie des données, c'est pourquoi les opérations portant sur l'une d'elles ne peuvent en aucun cas affecter l'autre, conformément à l'exemple suivant :

```
Structure Point
   Public x, y As Integer
   Public Sub New(x As Integer, y As Integer)
      Me.x = x
      Me.y = y
   End Sub
End Structure
```

En partant de la déclaration ci-dessus, le fragment de code suivant génère la valeur `10` :

```
Point a = new Point(10, 10)
Point b = a
a.x = 100
Console.WriteLine(b.x)
```

L'assignation de `a` à `b` crée une copie de la valeur, et `b` n'est donc pas affecté par l'assignation à `a.x`. Si, en revanche, `Point` avait été déclaré comme une classe, la sortie aurait été `100` puisque `a` et `b` auraient référencé le même objet.

Enfin, les structures n'étant pas des types référence, il est impossible que les valeurs d'un type structure soient null.

DéclarationStructure ::=
[*Attributs*] [*ModificateurStructure+*] **Structure** *Identificateur TerminateurLigne*
  [*ClauseImplementsType+*]
  [*DéclarationMembreStructure+*]
  End Structure *TerminateurLigne*

ModificateurStructure ::= ModificateurAccès | **Shadows**

6.5 Classes

Une classe est une structure de données qui peut contenir des données membres (des constantes, des variables et des événements), des fonctions membres (des méthodes, des propriétés, des indexeurs, des opérateurs et des constructeurs) ainsi que des types imbriqués. Les types classe prennent en charge l'héritage, mécanisme par lequel une classe dérivée peut étendre et spécialiser une classe de base.

Les classes sont des types référence.

L'exemple suivant montre une classe contenant chaque sorte de membre :

```
Imports System
Class AClass
```

```
   Public Sub New()
      Console.WriteLine("Constructor")
   End Sub

   Public Const MyConst As Integer = 12
   Public MyVariable As Integer = 34

   Public Sub MyMethod()
      Console.WriteLine("MyClass.MyMethod")
   End Sub

   Public Property MyProperty() As Integer
      Get
         Return MyVariable
      End Get
      Set (ByVal Value As Integer)
         MyVariable = value
      End Set
   End Property

   Default Public Property Item(index As Integer) As Integer
      Get
         Return 0
      End Get
      Set (ByVal Value As Integer)
         Console.WriteLine("item(" & index & ") = " & value)
      End Set
   End Property
   Public Event MyEvent()
   Friend Class MyNestedClass
   End Class
End Class
```

L'exemple suivant illustre l'utilisation de ces membres **:**

```
Module Test

   Dim WithEvents aInstance As AClass

   Sub Main()

      ' This shows constructor usage.
      Dim a As New AClass()
      Dim b As New AClass()

      ' This shows constant usage.
      Console.WriteLine("MyConst = " & AClass.MyConst)

      ' This shows variable usage.
      a.MyVariable += 1
      Console.WriteLine("a.MyVariable = " & a.MyVariable)

      ' This shows method usage.
      a.MyMethod()

      ' This shows property usage.
      a.MyProperty += 1
      Console.WriteLine("a.MyProperty = " & a.MyProperty)
      a(1) = 2

      ' This shows event usage.
      aInstance = a
```

```
   End Sub

   Sub MyHandler Handles aInstance.MyEvent
      Console.WriteLine("Test.MyHandler")
   End Sub
End Module
```

DéclarationClasse ::=
[*Attributs*] [*ModificateurClasse+*] **Class** *Identificateur TerminateurLigne*
[*BaseClasse*]
[*ClauseImplementsType+*]
[*DéclarationMembreClasse+*]
End Class *TerminateurLigne*

6.6 Modules standard

Un module standard est un type référence dont les membres sont implicitement **Shared** et dont la portée est l'espace de déclaration de l'espace de noms contenant le module standard, et non pas uniquement la déclaration du module lui-même. Il se peut que des modules standard ne soient jamais instanciés. Un membre d'un module standard possède deux noms complets, l'un avec le nom du module et l'autre sans. Un membre doté d'un nom particulier peut être défini dans plusieurs modules standard d'un même espace de noms ; les références non qualifiées de ce nom à l'extérieur de l'un de ces modules sont ambiguës. Un module ne peut être déclaré que dans un espace de noms et ne peut pas être imbriqué dans un autre type. Il n'est pas correct de déclarer une variable d'un type module standard. Les modules standard ne peuvent pas implémenter d'interfaces, ils sont implicitement dérivés d'**Object** et ils possèdent uniquement des constructeurs **Shared**.

DéclarationModule ::=
[*Attributs*] [*ModificateurAccès+*] **Module** *Identificateur TerminateurLigne*
[*DéclarationMembreModule+*]
End Module *TerminateurLigne*

6.8 Tableaux

Un tableau est un type référence qui contient plusieurs variables accessibles par l'intermédiaire d'*indices* correspondant un à un à l'ordre des variables dans le tableau (dans Visual Basic .NET, qui diffère en cela des versions précédentes de Visual Basic, la numérotation de l'index de tableau commence toujours par zéro). Les variables contenues dans un tableau, également appelées *éléments* du tableau, doivent toutes être de même type. Ce type est appelé *type des éléments* du tableau. Les éléments d'un tableau n'existent qu'à partir du moment où une instance de tableau est créée et cessent d'exister quand cette instance est détruite. Chacun de ces éléments est initialisé à la valeur par défaut de son type. Le type **System.Array** est le type de base de tous les types tableau et ne peut pas être instancié. Tous

les types tableau héritent des membres déclarés par le type **System.Array** et est convertible dans ce type (et dans le type **Object**).

Un tableau possède un *rang* qui détermine le nombre d'indices associés à chaque élément du tableau. Le rang d'un tableau équivaut aussi au nombre de *dimensions* que celui-ci possède. Un tableau de rang un, par exemple, est appelé tableau à une dimension, alors qu'un tableau de rang supérieur à un est qualifié de multidimensionnel.

L'exemple suivant crée un tableau à une dimension de valeurs entières, initialise les éléments du tableau et imprime chacun d'eux :

```
Module Test
   Sub Main()
      Dim arr(5) As Integer
      Dim i As Integer
      For i = 0 To arr.Length - 1
         arr(i) = i * i
      Next i
      Dim i As Integer
      For i = 0 To arr.Length - 1
         Console.WriteLine("arr(" & i & ") = " & arr(i))
      Next i
   End Sub
End Module
```

Le programme génère la sortie suivante :

```
arr(0) = 0
arr(1) = 1
arr(2) = 4
arr(3) = 9
arr(4) = 16
```

Chaque dimension d'un tableau possède une longueur. Les longueurs de dimension ne font pas partie du type du tableau, mais elles sont établies lors de la création d'une instance du type tableau, au moment de l'exécution. La longueur d'une dimension détermine la plage valide d'indices de cette dimension : pour une dimension de longueur *N*, les indices peuvent aller de zéro à *N – 1*. Une dimension de longueur zéro ne présente pas d'indices valides. Le nombre total d'éléments du tableau est le produit des longueurs de toutes les dimensions du tableau. Si le tableau comporte une dimension de longueur zéro, l'on dit qu'il est vide. Le type des éléments d'un tableau peut être n'importe quel type.

Les types tableau sont spécifiés par l'ajout d'un modificateur à un nom de type existant. Le modificateur se compose d'une parenthèse ouvrante, d'un ensemble de zéro virgule ou plus et d'une parenthèse fermante. Le type modifié est le type des éléments du tableau, et le nombre de dimensions correspond au nombre de virgules plus une. Si vous spécifiez deux modificateurs ou plus, le type des éléments du tableau est un tableau. Les modificateurs sont lus de gauche à droite, le modificateur étant situé le plus à gauche représentant le tableau le plus externe. Dans l'exemple suivant

```
Module Test
   Dim arr As Integer(,)(,,)()
End Module
```

le type des éléments d'`arr` est un tableau à deux dimensions de tableaux à trois dimensions de tableaux à une dimension de type `Integer`.

Une variable peut également être déclarée comme un type tableau en plaçant un modificateur de type tableau ou un modificateur d'initialisation de tableau sur le nom de la variable. Dans

ce cas, le type des éléments du tableau est le type indiqué dans la déclaration, et les dimensions du tableau sont déterminées par le modificateur du nom de variable. Par souci de clarté, une déclaration ne peut pas contenir un modificateur de type tableau qui soit défini à la fois sur un nom de variable et sur un nom de type.

L'exemple suivant montre diverses déclarations de variables locales utilisant des types tableau dont le type des éléments est **Integer** :

```
Module Test
   Sub Main()
      Dim a1() As Integer   ' Declares single-dimensional array of type
Integer.
      Dim a2(,) As Integer   ' Declares 2-dimensional array of type
Integer.
      Dim a3(,,) As Integer   ' Declares 3-dimensional array of type
Integer.
   End Sub
End Module
```

NomTypeTableau ::= *NomType ModificateurTypeTableau*

ModificateurTypeTableau ::= **(** [*ListeRang*] **)**

ListeRang ::=
, |
ListeRang ,

ModificateurTypeTableau ::=
ModificateurTypeTableau |
ModificateurInitialisationTableau

Types valeur et types référence

Un type de données est un *type valeur* s'il contient des données dans l'espace qui lui est alloué en mémoire. Un *type référence* contient un pointeur vers un autre emplacement en mémoire contenant les données.

Les types valeur sont :

- tous les types de données numériques ;
- **Boolean**, **Char** et **Date** ;
- toutes les structures, même si leurs membres sont des types référence ;
- les énumérations, dans la mesure où leur type sous-jacent est toujours **Byte**, **Short**, **Integer** ou **Long**.

Les types référence sont :

- **String ;**

- tous les tableaux, même si leurs éléments sont des types valeur ;
- les types classe comme **Form**.

Vous pouvez assigner au choix un type référence ou un type valeur à une variable de type de données **Object**. Une variable **Object** contient toujours un pointeur vers les données, jamais les données elles-mêmes. Toutefois, si vous assignez un type valeur à une variable **Object**, elle se comporte comme si elle contenait ses propres données.

Vous pouvez savoir si une variable **Object** correspond à un type référence ou à un type valeur en la passant à la méthode **IsReference** de la classe **Information** dans l'espace de noms **Microsoft.VisualBasic**. **Microsoft.VisualBasic.Information.IsReference** retourne **True** si le contenu de la variable **Object** représente un type référence.

Liste des types de données

Voir aussi

Boolean, type de données | Byte, type de données | Char, type de données | Date, type de données | Decimal, type de données | Double, type de données | Integer, type de données | Long, type de données | Object, type de données | Short, type de données | Single, type de données | String, type de données | Type de données défini par l'utilisateur | Fonctions de conversion de types de données | Utilisation efficace des types de données | Int, Fix, fonctions | StrConv, fonction

Le tableau suivant illustre les types de données Visual Basic .NET, leurs types du Common Language Runtime pris en charge, leur allocation de stockage nominal et leur plage de valeurs.

Type Visual Basic	Structure de type Common Language Runtime	Allocation de stockage nominal	Plage de valeurs
Boolean	**System.Boolean**	2 octets	**True** ou **False**.
Byte	**System.Byte**	1 octet	0 à 255 (non signé).
Char	**System.Char**	2 octets	0 à 65 535 (non signé).
Date	**System.DateTime**	8 octets	0:00:00 le 1er janvier 0001 à 23:59:59 le 31 décembre 9999.
Decimal	**System.Decimal**	16 octets	0 à +/- 79 228 162 514 264 337 593 543 950 335 sans décimale ; 0 à +/- 7,9228162514264337593543950335 avec 28 décimales ; le plus petit nombre différent de zéro étant +/- 0,0000000000000000000000000001 (+/- 1E-28).
Double (virgule flottante à double	**System.Double**	8 octets	-1,79769313486231E+308 à -4,94065645841247E-324 pour les valeurs négatives ; 4,94065645841247E-324 à 1,79769313486231E+308 pour les

précision)			valeurs positives.
Integer	**System.Int32**	4 octets	-2 147 483 648 à 2 147 483 647.
Long (entier de type Long)	**System.Int64**	8 octets	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807.
Object	**System.Object** (classe)	4 octets	N'importe quel type peut être stocké dans une variable de type **Object**.
Short	**System.Int16**	2 octets	-32 768 à 32 767.
Single (virgule flottante en simple précision)	**System.Single**	4 octets	-3,402823E+38 à -1,401298E-45 pour les valeurs négatives ; 1,401298E-45 à 3,402823E+38 pour les valeurs positives.
String (longueur variable)	**System.String** (classe)	Dépend de la plate-forme d'implémentation	0 à environ 2 milliards de caractères Unicode.
User-Defined Type (structure)	(hérite de **System.ValueType**)	Dépend de la plate-forme d'implémentation	Chaque membre de la structure présente une plage déterminée par son type de données et qui est indépendante des plages des autres membres.

Remarque Pour les chaînes contenant du texte, utilisez la fonction **StrConv** pour effectuer une conversion d'un format de texte en un autre.

Consommation de mémoire

Lorsque vous déclarez un type de données élémentaire, il est risqué de supposer que sa consommation de mémoire est identique à son allocation de stockage nominal. Le Common Language Runtime assigne le stockage en fonction des caractéristiques de la plate-forme sur laquelle s'exécute votre application. Dans certains cas, il compresse vos éléments déclarés aussi étroitement que possible ; dans d'autres, il aligne leurs adresses mémoire sur les limites matérielles naturelles. De même, l'assignation de stockage est différente sur une plate-forme 64 bits que sur une plate-forme 32 bits.

Les mêmes considérations sont d'application pour chacun des membres d'un type de données composite (par exemple, une structure ou un tableau). Par ailleurs, la configuration requise pour la mémoire est plus importante pour certains types composites. Par exemple, un tableau utilise davantage de mémoire pour le tableau mais également pour chaque dimension. Sur une plate-forme 32 bits, cette charge mémoire est actuellement de 12 octets, auxquels il convient d'ajouter 8 octets supplémentaires pour chaque dimension. Sur une plate-forme 64 bits, la configuration requise est doublée. Il ne suffit pas d'additionner les allocations de stockage nominal des composants.

Un **Object** se rapportant à un type de données élémentaire ou composite utilise 4 octets de plus que les données contenues dans le type de données.

Utilisation efficace des types de données

Les variables non déclarées et celles qui sont déclarées sans type de données prennent le type de données **Object**. Vous pouvez ainsi écrire plus facilement et rapidement des programmes, mais leur exécution sera plus lente.

Le processus qui consiste à spécifier des types de données pour toutes les variables est appelé *typage fort*. Le typage fort présente plusieurs avantages :

- Il permet la prise en charge par IntelliSense™ de vos variables. Il vous permet de voir leurs propriétés et leurs autres membres à mesure que vous tapez le code.
- Il permet au compilateur d'effectuer la vérification des types. Il intercepte les instructions susceptibles de provoquer, au moment de l'exécution, des erreurs telles qu'un dépassement de capacité. Il intercepte également les appels de méthodes dans des objets qui ne les prennent pas en charge.
- Il permet ainsi à votre code de s'exécuter plus rapidement.

Pour les variables qui ne contiennent jamais de fractions, les types de données intégraux sont plus efficaces que les types non intégraux. Dans Visual Basic .NET, **Integer** est le type numérique le plus efficace.

Faites appel à l'instruction **Dim** pour déclarer une variable d'un type spécifique. Vous pouvez simultanément spécifier son accessibilité à l'aide d'un mot clé **Public**, **Protected**, **Friend** ou **Private**, comme dans l'exemple suivant :

```
Private X As Double   ' X is a double-precision floating point.
Protected S As String   ' S is a character string.
```

Conversions de type

L'opération consistant à remplacer une valeur d'un certain type de données par une valeur d'un autre type est appelée *conversion*. Les conversions sont soit étendues, soit restrictives, en fonction de la capacité de données des types concernés. Elles sont également implicites ou explicites, en fonction de la syntaxe du code source.

Dans cette section

Conversions étendues et restrictives
: Le fait que le type de destination puisse ou non contenir les données détermine la classification de la conversion.

Conversions implicites et explicites
: Le fait que Visual Basic l'effectue automatiquement ou non détermine également la classification de la conversion.

Modification des valeurs lors de la conversion
: Explique en quoi la conversion dans un autre type de données peut modifier la représentation de la valeur ou des données.

Conversions entre des chaînes et d'autres types
: Décrit la conversion entre des chaînes et des valeurs numériques, **Boolean** ou date/heure.

Conversions des tableaux

Vous guide pas à pas dans le processus de conversion entre des tableaux de types de données différents.

Conversions étendues et restrictives

Le fait que le résultat se situe ou non dans la plage du type de données de destination constitue une caractéristique importante de la conversion de type. Dans une *conversion étendue*, la valeur est convertie dans un type de données capable d'accueillir toutes les valeurs possibles des données d'origine. Dans une *conversion restrictive*, la valeur est convertie dans un type de données susceptible de ne pas pouvoir accueillir certaines valeurs possibles des données.

Conversions étendues

Le tableau suivant répertorie les conversions étendues standard.

Type de données	Extension aux types de données
Byte	**Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**
Short	**Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**
Integer	**Integer**, **Long**, **Decimal**, **Single**, **Double**
Long	**Long**, **Decimal**, **Single**, **Double**
Decimal	**Decimal**, **Single**, **Double**
Single	**Single**, **Double**
Double	**Double**
Tout type énuméré	Son type entier sous-jacent et tout type auquel cela l'étend.
Char	**Char**, **String**
Tout type	**Object**
Tout type dérivé	Tout type de base dont il est dérivé.
Tout type	Toute interface qu'il implémente.
Nothing	Tout type de données ou type d'objet.

Les conversions entre **Integer** et **Single**, entre **Long** et **Single** ou **Double**, ou entre **Decimal** et **Single** ou **Double**, peuvent entraîner une perte de précision, mais jamais une perte de magnitude. Elles n'entraînent donc jamais de perte d'information.

Il peut sembler surprenant qu'une conversion entre un type dérivé et l'un de ses types de base soit étendue. Cela s'explique par le fait que le type dérivé contient tous les membres du type de base ; il se qualifie donc comme une instance du type de base. Dans le sens opposé, le type de base ne contient pas les membres définis par le type dérivé.

Les conversions étendues n'échouent jamais et peuvent toujours être effectuées implicitement.

Conversions restrictives

Les conversions restrictives standard sont les suivantes :

- Conversions inverses des conversions étendues répertoriées dans le tableau précédent.

- Conversions entre un type **Boolean** et tout type numérique (dans l'un et l'autre sens).
- Conversions entre un type numérique et tout type énuméré.
- Conversions dans l'un et l'autre sens entre un tableau **Char()** et un type **String**.
- Conversions dans l'un et l'autre sens entre un type **String** et tout type numérique, **Boolean** ou **Date**.
- Conversions entre un type de données ou un type d'objet et un type dérivé de lui.

Les conversions restrictives ne réussissent pas toujours ; elles peuvent donc connaître des échecs au moment de l'exécution. Une erreur se produit si le type de données de destination ne peut pas accueillir la valeur convertie. Par exemple, une conversion numérique peut provoquer un dépassement de capacité. En général, le compilateur n'autorise pas les conversions restrictives implicites.

Utilisez une conversion restrictive lorsque vous savez que la valeur source peut être convertie sans erreur dans le type de données de destination. Par exemple, si vous avez un type **String** dont vous savez qu'il contient soit « True », soit « False », vous pouvez utiliser le mot clé **CBool** pour le convertir en **Boolean**.

Conversions implicites et explicites

Une conversion *implicite* ne requiert pas de syntaxe particulière dans le code source. Dans l'exemple suivant, Visual Basic .NET convertit implicitement la valeur de `K` en virgule flottante à simple précision avant de l'assigner à `Q`.

```
Dim K As Integer
Dim Q As Double
' ...
K = 432   ' Integer widens to Double, so Option Strict can be On.
Q = K
```

Une conversion *explicite* fait appel à un mot clé de conversion de type. Visual Basic .NET fournit plusieurs de ces mots clés qui forcent la conversion d'une expression entre parenthèses vers le type de données souhaité. Ces mots clés agissent comme des fonctions, mais le compilateur génère le code en ligne, si bien que l'exécution est légèrement plus rapide qu'avec un appel de fonction.

Dans l'exemple ci-dessous qui fait suite à l'exemple précédent, le mot clé **CInt** reconvertit la valeur de `Q` en un entier avant de l'assigner à `K` :

```
Q = Math.Sqrt(Q)   ' Q had been assigned the value 432 from K.
K = CInt(Q)   ' K now has the value 21 (rounded square root of 432).
```

Le tableau suivant répertorie les mots clés de conversion disponibles.

Mot clé de conversion de type	Convertit une expression vers le type de données	Types de données autorisés d'expression à convertir
CBool	**Boolean**	Tout type numérique (y compris **Byte** et les types énumérés),

		String, **Object**
CByte	**Byte**	Tout type numérique, type énuméré, **Boolean**, **String**, **Object**
CChar	**Char**	**String**, **Object**
CDate	**Date**	**String**, **Object**
CDbl	**Double**	Tout type numérique (y compris **Byte** et les types énumérés), **Boolean**, **String**, **Object**
CDec	**Decimal**	Tout type numérique (y compris **Byte** et les types énumérés), **Boolean**, **String**, **Object**
CInt	**Integer**	Tout type numérique (y compris **Byte** et les types énumérés), **Boolean**, **String**, **Object**
CLng	**Long**	Tout type numérique (y compris **Byte** et les types énumérés), **Boolean**, **String**, **Object**
CObj	**Object**	Tout type
CShort	**Short**	Tout type numérique (y compris **Byte** et les types énumérés), **Boolean**, **String**, **Object**
CSng	**Single**	Tout type numérique (y compris **Byte** et les types énumérés), **Boolean**, **String**, **Object**
CStr	**String**	Tout type numérique (y compris **Byte**), **Boolean**, **Char**, tableau **Char()**, **Date**, **Object**
CType	Type spécifié après la virgule (**,**)	Lors d'une conversion vers un type élémentaire (y compris un tableau d'un type élémentaire), les mêmes types que ceux autorisés pour le mot clé de conversion correspondant Lors d'une conversion vers un type composite, les interfaces qu'il implémente et les classes à partir desquelles il hérite

Le mot clé **CType** opère sur deux arguments. Le premier correspond à l'expression à convertir, le second au type de données de destination ou à la classe d'objet. L'exemple suivant illustre l'utilisation du mot clé **CType** :

```
K = CType(Q, Integer)  ' Uses CType keyword instead of CInt.
F = CType(W, Label)    ' Coerces W to the specific object class Label.
```

Vous pouvez utiliser le mot clé **CType** pour convertir des valeurs vers des types de données composites ou élémentaires. Vous pouvez également l'utiliser pour forcer la conversion d'une classe d'objet vers le type de l'une de ses interfaces, comme dans l'exemple suivant :

```
' Assume you have defined a structure called Customer.
U = CType(S, Customer)   ' Converts S to a Customer structure.
' Assume class CZone implements interface IZone.
H = CType(CZone, IZone)   ' Coerces CZone to its interface IZone.
```

CType peut également convertir des types tableau, comme dans l'exemple suivant :

```
Dim V() As ClassV   ' Array of ClassV objects.
Dim ObArray() As Object   ' Array of objects to be assigned.
' Some object array is assigned to ObArray.
If TypeOf ObArray Is ClassV()   ' Check for run-time type compatibility.
V = CType(ObArray, ClassV())   ' ObArray can be converted to ClassV.
End If
```

> **Remarque** Les valeurs utilisées avec un mot clé de conversion doivent être acceptées par le type de données de destination, sans quoi une erreur se produit. Par exemple, si vous tentez de convertir un type **Long** en **Integer**, la valeur du type **Long** doit se situer dans la plage acceptée par le type de données **Integer**.

Pour désigner le processus de conversion explicite, on parle également de *casting* d'une expression vers un certain type de données ou une classe d'objet.

Modification des valeurs lors de la conversion

La conversion à partir d'un type valeur enregistre une copie de la valeur source dans la destination de la conversion. Toutefois, cette copie n'est pas une image fidèle de la valeur source. Le type de données de destination enregistre différemment les valeurs ; il peut même arriver que la valeur représentée change en fonction du type de conversion effectuée.

Modification lors des conversions étendues et restrictives

Les conversions restrictives changent la copie de destination de la valeur source en entraînant éventuellement une perte d'information. Par exemple, une valeur fractionnaire est arrondie lorsqu'elle est convertie en type intégral, et un type numérique converti en **Boolean** se trouve réduit à la valeur **True** ou **False**.

Les conversions étendues préservent la valeur mais peuvent modifier sa représentation. Cela se produit notamment si vous convertissez un type intégral en **Decimal** ou un type **Char** en **String**.

La valeur source d'origine n'est pas modifiée à la suite d'une conversion.

Modification lors des conversions de type référence

Une conversion à partir d'un type référence ne copie que le pointeur vers la valeur. La valeur elle-même n'est ni copiée ni modifiée. Seul est modifié le type de données de la variable contenant le pointeur. Dans l'exemple suivant, le type de données est converti de la classe dérivée vers sa classe de base, mais l'objet vers lequel pointent les deux variables reste inchangé :

```
' Assume class CSquare inherits from class CShape.
Dim Shape As CShape
Dim Square As CSquare = New CSquare
' ...
Shape = Square   ' Widening conversion from derived class to base class.
```

Conversion entre des chaînes et d'autres types

Liste des types de données | Fonctions de conversion de type | Format, fonction | Val, fonction | Introduction aux applications internationales dans Visual Basic et Visual C#

Vous pouvez convertir une valeur numérique, **Boolean** ou date/heure vers le type **String**. Vous pouvez également opérer la conversion inverse, c'est-à-dire d'une chaîne vers un type de données numérique, **Boolean** ou **Date**, à condition que le contenu de la chaîne soit une valeur reconnue par le type de données de destination. Si tel n'est pas le cas, une erreur d'exécution se produit.

Ces différentes conversions (dans l'un et l'autre sens) sont des conversions restrictives. Utilisez les mots clés de conversion de type (**CStr**, **CShort**, **CInt**, **CLng**, **CDec**, **CSng**, **CDbl**, **CBool**, **CDate** et **CType**). Les fonctions **Format** et **Val** vous permettent de mieux contrôler les conversions entre les chaînes et les nombres.

Conversion de nombres en chaînes

Vous pouvez utiliser la fonction **Format** pour convertir un nombre en chaîne mise en forme, laquelle peut inclure en plus des chiffres appropriés des symboles de mise en forme tels qu'un symbole monétaire (comme **$**), des *symboles de groupement des chiffres* (comme l'**espace**) et un séparateur décimal (comme **,**). La fonction **Format** utilise automatiquement les symboles adaptés aux **Options régionales** spécifiées dans le **Panneau de configuration** Windows.

Notez que l'opérateur de concaténation (**&**) peut convertir implicitement un nombre en chaîne, comme le montre l'exemple suivant :

```
Str = "The total count is " & Count    ' Count is converted to String.
```

Conversion de chaînes en nombres

Vous pouvez utiliser la fonction **Val** pour convertir explicitement les chiffres d'une chaîne en nombre. La fonction **Val** lit la chaîne jusqu'à ce qu'elle rencontre un caractère autre qu'un chiffre, un espace, une tabulation, un saut de ligne ou un point. Les séquences « &O » et « &H » altèrent la base du système numérique et interrompent la lecture. Tant qu'elle n'arrête pas sa lecture, la fonction **Val** convertit tous les caractères appropriés en une valeur numérique. Par exemple, l'instruction suivante retourne la valeur 141.8 :

```
Val(" 14 1.8 miles")
```

Lorsque Visual Basic convertit une chaîne en valeur numérique, il utilise les **Options régionales** spécifiées dans le **Panneau de configuration** Windows pour interpréter le séparateur de milliers, le séparateur décimal et le symbole monétaire. Cela signifie qu'une conversion peut réussir avec certains paramètres mais pas avec d'autres. Par exemple, la valeur `"$14.20"` est acceptée pour le paramètre régional Anglais (États-Unis), mais pas pour un paramètre régional français.

Conversion des tableaux

Vous pouvez convertir un type tableau vers un autre type, pourvu que les conditions suivantes soient remplies :

- Les rangs des deux tableaux doivent être identiques ; autrement dit, les deux tableaux doivent avoir le même nombre de dimensions. Toutefois, il n'est pas obligatoire que la longueur des dimensions respectives soit identique.
- Les types de données des éléments des deux tableaux doivent être des types référence. Vous ne pouvez pas convertir un tableau de type **Integer** en tableau **Long** ou même **Object**, parce que cette opération fait intervenir au moins un type valeur.
- Une conversion, qu'elle soit restrictive ou étendue, doit être possible entre les types d'éléments des deux tableaux. Cette condition n'est pas vérifiée, par exemple, dans le cas d'une conversion entre un tableau **String** et un tableau d'une classe dérivée de **System.Enum**. Ces deux types n'ont rien en commun, et toute conversion entre eux est totalement exclue.

Les conversions entre deux types tableau sont soit étendues, soit restrictives, selon qu'elles élargissent ou rétrécissent leurs éléments respectifs.

Conversion vers un tableau de type Object

Lorsque vous déclarez un tableau **Object**, son type d'élément reste **Object** tant qu'il n'est pas initialisé. Toutefois, si par la suite vous l'initialisez pour une classe particulière, il prend le type de cette classe. Dans l'exemple suivant, `MyArray` une fois initialisé devient un tableau `Student`. Comme aucune conversion n'existe entre **String** et `Student`, sa dernière instruction se solde par un échec :

```
' Assume Student has already been defined as a class.
Dim MyArray() As Object   ' MyArray is an Object array at this point.
Dim Names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
' ...
MyArray = New Student(3) {}   ' MyArray is now of type Student.
MyArray = Names   ' Fails at compile time.
```

Structures constituées à partir de vos propres types de données

Vous pouvez combiner des éléments de données de différents types pour constituer une *structure*. Une structure associe un ou plusieurs *membres* les uns avec les autres et avec la structure elle-même. Lorsque vous déclarez une structure, elle devient un type de données composite et vous pouvez déclarer des variables de ce type.

Une structure est une généralisation du type défini par l'utilisateur (UDT, *user-defined type*) des précédentes versions de Visual Basic. En plus des champs, les structures peuvent exposer des propriétés, des méthodes et des événements. Une structure peut implémenter une ou plusieurs interfaces et vous pouvez déclarer une accessibilité propre à chaque champ.

Les structures sont utiles lorsque vous voulez qu'une même variable contienne plusieurs éléments d'information connexes. Par exemple, vous pouvez conserver ensemble le nom, le poste téléphonique et le salaire d'un employé. Vous pouvez enregistrer ces informations dans plusieurs variables ou définir une structure et l'utiliser comme une variable d'employé unique. Les avantages des structures deviennent évidents lorsque vous avez de nombreux employés et par conséquent de nombreuses instances de la variable.

Dans cette section

Déclaration de structure

Vous débutez une déclaration de structure par l'instruction **Structure** et la terminez par une instruction **End Structure**. Entre ces deux instructions, vous devez déclarer au moins un membre. Les membres peuvent avoir n'importe quel type de données.

Pour conserver le nom, le poste téléphonique et le salaire d'un employé dans une même variable, déclarez une structure de la façon suivante :

```
Structure Employee
   Public GivenName As String   ' This employee's given name.
   Public FamilyName As String   ' This employee's family name.
   Public Extension As Long   ' This employee's telephone extension.
   Private Salary As Decimal   ' This employee's annual salary.
End Structure
```

Vous pouvez spécifier l'accessibilité de la structure à l'aide du mot clé **Public**, **Protected**, **Friend** ou **Private** ou garder la valeur par défaut, **Public**. Vous devez déclarer chaque membre et spécifier pour lui une accessibilité. Si vous utilisez l'instruction **Dim** sans mot clé, l'accessibilité prend la valeur par défaut, **Public**.

> **Remarque** Vous ne pouvez pas initialiser l'un des membres d'une structure dans la déclaration de structure. Lorsque vous déclarez une variable pour être d'un type structure, vous assignez des valeurs aux membres en y accédant par l'intermédiaire de la variable.

Variables de structure

Structure, instruction

Une fois que vous avez créé une structure, vous pouvez déclarer des variables du niveau procédure ou du niveau module avec ce type. Par exemple, vous pouvez créer une structure enregistrant des informations au sujet d'un système informatique, comme dans le code suivant :

```
Private Structure SystemInfo
```

```
   Private CPU As String
   Private Memory As Long
   Private PurchaseDate As Date
End Structure
```

Vous pouvez ensuite déclarer des variables de ce type, de la façon suivante.

```
Dim MySystem, YourSystem As SystemInfo
```

> **Remarque** Dans les classes et les modules, les structures déclarées à l'aide de l'instruction **Dim** ont par défaut un accès public. Si vous souhaitez qu'une structure soit privée, veillez à la déclarer à l'aide du mot clé **Private**.

Accès aux valeurs des structures

Pour assigner et extraire des valeurs à partir des éléments d'une variable de structure, vous utilisez la même syntaxe que pour définir et récupérer les propriétés d'un objet. Par exemple, si vous utilisez les variables précédemment déclarées comme `SystemInfo`, vous pouvez avoir accès à leurs éléments de la façon suivante :

```
MySystem.CPU = "486"
If YourSystem.PurchaseDate < #1/1/1992# Then TooOld = True
```

Vous pouvez également assigner une variable à une autre si toutes deux ont le même type structure. Vous copiez ainsi tous les éléments d'une structure vers les éléments correspondants de l'autre structure, comme dans l'exemple suivant :

```
YourSystem = MySystem
```

Si un élément de structure est un type référence (par exemple, **String**, **Object** ou un tableau), le pointeur vers les données est copié. Si `SystemInfo` avait inclus une variable d'objet, l'exemple précédent aurait copié le pointeur à partir de `MySystem` vers `YourSystem`, et une modification vers les données de l'objet serait appliquée par l'intermédiaire d'une structure lors de son accès par l'autre structure.

Structures et autres éléments de programmation

Voir aussi

Structures constituées à partir de vos propres types de données | Variables de structure | Structures et classes | Function, procédures | Arguments de procédure | Structure, instruction

Vous pouvez utiliser des structures avec des tableaux, des objets, des procédures, mais aussi les unes avec les autres. Les interactions utilisent la même syntaxe que ces éléments pris individuellement.

> **Remarque** Vous ne pouvez pas initialiser l'un des membres d'une structure dans la déclaration de structure. Vous pouvez assigner des valeurs uniquement aux membres d'une variable qui a été déclarée comme un type structure.

Structures et tableaux

Une structure peut contenir un tableau en tant qu'élément, comme le montre l'exemple suivant :

```
Private Structure SystemInfo
   Private CPU As String
   Private Memory As Long
   Private DiskDrives() As String   ' Array of changeable size.
   Private PurchaseDate As Date
End Structure
```

Pour accéder aux valeurs d'un tableau dans une structure, procédez comme pour accéder à une propriété d'un objet, comme le montre l'exemple suivant :

```
Dim MySystem As SystemInfo
ReDim MySystem.DiskDrives(3)
MySystem.DiskDrives(0) = "1.44 MB"
```

Vous pouvez également déclarer un tableau de structures, comme dans l'exemple suivant :

```
Dim AllSystems(100) As SystemInfo
```

Vous appliquez les mêmes règles pour accéder aux composants de cette architecture de données, comme le montre cet exemple :

```
AllSystems(5).CPU = "386SX"
AllSystems(5).DiskDrives(2) = "100M SCSI"
```

Structures et objets

Une structure peut contenir un objet en tant qu'élément, comme le montre l'exemple suivant :

```
Private Structure AccountPack
   Private frmInput as Form
   Private dbPayRollAccount as Database
End Structure
```

Dans une déclaration de ce type, utilisez une classe d'objet spécifique de préférence à **Object**.

Structures et procédures

Vous pouvez passer une structure en tant qu'argument de procédure, comme le montre l'exemple suivant :

```
Sub FillSystem(ByRef SomeSystem As SystemInfo)
   SomeSystem.CPU = lstCPU.Text
   SomeSystem.Memory = lstCPU.Amount
   SomeSystem.PurchaseDate = Now
End Sub
```

Dans l'exemple précédent, la structure est passée par référence, ce qui permet à la procédure de modifier ses éléments de sorte que les modifications prennent effet dans le code appelant. Pour protéger une structure contre de telles modifications, passez-la par valeur.

Vous pouvez retourner une structure à partir d'une procédure de fonction, comme le montre l'exemple suivant :

```
Dim AllSystems(100) As SystemInfo
' ...
Function FindByDate(ByVal SearchDate As Date) As SystemInfo
   Dim I As Integer
   For I = 1 To 100
      If AllSystems(I).PurchaseDate = SearchDate Then Return AllSystems(I)
   Next I
   ' Process error: system with desired purchase date not found.
End Function
```

Structures dans d'autres structures

Les structures peuvent contenir d'autres structures, comme le montre l'exemple suivant :

```
Structure DriveInfo
   Dim Type As String
   Dim Size As Long
End Structure
' ...
Private Structure SystemInfo
   Private CPU As String
   Private Memory As Long
   Private DiskDrives() As DriveInfo   ' Array of changeable size.
   Private PurchaseDate As Date
End Structure
' ...
Dim AllSystems(100) As SystemInfo
AllSystems(1).DiskDrives(0).Type = "Floppy"
```

Vous pouvez également utiliser cette technique pour encapsuler une structure définie dans un module à l'intérieur d'une autre structure définie dans un module différent.

Les structures peuvent s'imbriquer dans d'autres structures sans limite de profondeur.

Structures et classes

Visual Basic .NET unifie la syntaxe des structures et des classes, en conséquence de quoi les deux types d'entités prennent en charge les mêmes fonctionnalités. Toutefois, il traite également des principales différences existant entre les classes et les structures.

Similitudes

Les structures et les classes sont identiques à plusieurs égards :

- elles ont des membres comprenant des constructeurs, des méthodes, des propriétés, des champs, des constantes, des énumérations et des événements ;
- elles peuvent implémenter des interfaces ;
- elles peuvent avoir des constructeurs partagés, avec ou sans paramètres.

Différences

Les structures et les classes diffèrent à plusieurs égards :

- les structures sont des types valeur, les classes des types référence ;
- les structures utilisent l'allocation de la pile, les classes l'allocation de tas ;
- tous les membres de structure sont **Public** par défaut, les variables et les constantes de classe sont **Private** par défaut, tandis que les autres membres de classe sont **Public** par défaut. Ce comportement pour les membres de classe assure la compatibilité avec le système de valeurs par défaut Visual Basic 6.0 ;

- les membres de structure ne peuvent pas être déclarés comme **Protected**, contrairement aux membres de classe ;
- les procédures de structure ne peuvent pas gérer des événements, contrairement aux procédures de classe ;
- les déclarations de variable de structure ne peuvent pas spécifier d'initialiseur, le mot clé **New** ou les tailles initiales des tableaux, contrairement aux déclarations de variable de classe ;
- les structures héritent implicitement de la classe **ValueType** et ne peuvent pas hériter d'un autre type, les classes peuvent hériter d'une ou plusieurs classes distinctes de **ValueType** ;
- il est possible d'hériter des classes, mais pas des structures ;
- les structures n'étant jamais arrêtées, le Common Language Runtime (CLR) n'appelle jamais la méthode **Finalize** sur une structure, les classes sont arrêtées par le « garbage collector », qui appelle **Finalize** sur une classe lorsqu'il ne détecte aucune référence active restante ;
- les structures peuvent avoir des constructeurs non partagés uniquement lorsqu'ils acceptent des paramètres, les classes peuvent en accepter avec ou sans paramètre.

Chaque structure possède un constructeur public implicite sans paramètres. Ce constructeur initialise toutes les données membres de la structure en leur attribuant leurs valeurs par défaut. Vous ne pouvez pas modifier ce comportement.

Instances et variables

Puisque les structures sont des types valeur, chaque variable de structure est définitivement liée à une instance de structure individuelle. En revanche, les classes sont des types référence, et une variable d'objet peut faire référence à plusieurs instances de classe. Cette distinction affecte votre utilisation de structures et de classes de diverses façons :

- Une variable de structure comprend implicitement une initialisation des membres à l'aide du constructeur sans paramètre de la structure. En conséquence, `Dim S As Struct1` équivaut à `Dim S As Struct1 = New Struct1()`.
- Lorsque vous assignez une variable de structure à une autre ou que vous passez une instance de structure à un argument de procédure, les valeurs actuelles de tous les membres de variable sont copiés à la nouvelle structure. Lorsque vous assignez une variable d'objet à une autre ou que vous passez une variable d'objet à une procédure, seul le pointeur de référence est copié.
- Vous pouvez assigner la valeur **Nothing** à une variable de structure, mais l'instance continue à être associée à la variable. Vous pouvez toujours appeler ses méthodes et accéder à ses données membres, même si les membres de variable sont réinitialisés par l'assignation. En revanche, si vous attribuez la valeur **Nothing** à une variable d'objet, vous la dissociez d'une instance de classe et vous ne pouvez pas accéder aux membres par l'intermédiaire de la variable jusqu'à ce que vous lui assigniez une autre instance.
- Une variable d'objet peut avoir diverses instances de classe qui lui sont assignées à des moments différents, et plusieurs variables d'objet peuvent faire référence

simultanément à la même instance de classe. Les modifications apportées aux valeurs des membres de classe affectent ces membres lorsque vous y accédez par l'intermédiaire d'une autre variable pointant vers la même instance. Cependant, les membres de structure sont isolés dans leur propre instance. Les modifications apportées à leurs valeurs ne sont pas réfléchies dans d'autres variables de structure, même dans d'autres instances de la même déclaration **Structure**.

- Un test d'égalité de deux structures doit être effectué sur un membre après l'autre. Deux variables d'objet peuvent être comparées par la méthode **Equals**. **Equals** détermine si les deux variables pointent vers la même instance