

# Introduction à la Programmation Orientée Objet en VB.NET

par [Anoop Madusudanan Xavier Vlieghe](#) (Traducteur)

Date de publication : 10/11/2005

Dernière mise à jour : 28/01/2006

Cet article a pour but de vous initier aux mécanismes de la POO utilisés par le langage VB.NET.

Remerciements

Introduction

Mode d'emploi du code

Leçon 1 : Un peu de vocabulaire

- Espace de Noms

- Classes

- Objet

- Module

Leçon 2 : Portées de vos déclarations

Leçon 3 : Fonctions Partagées

Leçon 4 : La Surcharge

Leçon 5 : L'Héritage

Leçon 6 : La Rédéfinition

Leçon 7 : Le Polymorphisme d'héritage

Leçon 8 : Constructeurs et Destructeurs

Leçon 9 : Les Propriétés

Leçon 10 : Une petite application

Téléchargements

## Remerciements

Ce tutoriel est une traduction provenant d'un article du site [CodeProject](#).

L'auteur de cet article est [Anoop Madusudanan](#), je tiens à le remercier chaleureusement pour m'avoir autorisé à traduire son article.

Vous pouvez consulter son article original [ici](#).

Merci à [khany](#) et à [neo.51](#) pour leurs corrections et leurs conseils.

## Introduction

Cet article traite des fonctionnalités basiques de la POO en VB.Net. En effet, une des évolutions notables de VB.Net est d'être totalement orienté objet, contrairement aux versions précédentes de VB.

Cet article est découpé en dix leçons et le code source est fourni pour chacune d'entre elles, en fin d'article.

Cet article a pour but d'atteindre les objectifs suivants :

1. Aborder de manière exhaustive la POO en VB.NET,
2. Apprendre comment sont utilisées les techniques de la POO en VB.NET
3. Expliquer simplement les concepts suivants :
  - Création et utilisation de classes et d'objets en VB.NET,
  - Fonctions Partagées,
  - Encapsulation, Abstraction, Héritage et Polymorphisme,
  - Surcharge et Redéfinition,
  - Constructeurs et Destructeurs.

En étudiant ce tutoriel, vous serez en mesure d'appréhender facilement la plupart des codes .NET.

De même, des programmeurs Java/C++ peuvent l'utiliser pour aborder la POO avec VB.NET.

## Mode d'emploi du code

Pour chaque leçon, le code source est disponible sous forme d'un simple fichier .vb (sans projet associé) : vous aurez besoin du Framework SDK .NET pour compiler et exécuter les exercices dans cet article.

Vous pouvez le télécharger du site Web de Microsoft, [ici](#).

Le compilateur VB.NET (vbc.exe) réside normalement dans votre dossier FrameworkSDK\bin.

Pour compiler manuellement un fichier, passez par l'exécuteur de ligne de commande et tapez :

```
vbc filename.vb /out:"filename.exe" /r:"System.Windows.Forms.dll","System.dll"
```

## Leçon 1 : Un peu de vocabulaire

### Espace de Noms

En VB.NET, les classes et autres structures de données concernant un domaine bien défini sont incluses dans un espace de noms (ou NameSpace). Vous pouvez utiliser les classes appartenant à un espace de noms simplement en important cet espace : ceci se fait par le biais du mot-clé Imports.

Le Framework .NET fournit un ensemble très fournis de classes pré-définies, groupées ensemble dans divers espaces de noms.

Dans cette leçon, nous utiliserons le namespace System. Importons le namespace System (déjà disponible dans .NET).

```
Imports System
```

### Classes

Vous avez déjà entendu parler des classes ? Pour faire simple, une classe (Class) est la définition d'un objet de tous les jours. Par exemple, il est possible de définir une classe Humain afin de modéliser tous les êtres humains, et une nommée Chien pour représenter les chiens. Les classes peuvent également gérer des méthodes : ce sont (en général) des procédures concernant le fonctionnement de l'objet à modéliser.

Voici l'instruction de déclaration d'un espace de noms nommé Animaux :

```
Namespace Animaux
```

Chien est une classe appartenant à l'espace de noms Animaux:

```
Class Chien
```

Et Aboie est une procédure de cette classe :

```
Function Aboie()  
    Console.WriteLine ("Le chien aboie !")  
End Function  
End Class  
End Namespace
```

### Objet

Un objet est une instance de classe. Par exemple, Jimmy est un objet de type Chien. Pour illustrer ceci, nous allons créer un objet dans la section suivante.

### Module

Il est possible d'utiliser des modules pour écrire des fonctions ou procédures publiques : Un Module est donc un groupe de fonctions. À la différence d'une fonction appartenant à une classe, une fonction publique d'un module peut être appelée à tout endroit du code. VB implémente les fonctions (Function) et les procédures (Sub). La seule différence entre une fonction et une procédure est que contrairement à une fonction, une procédure ne retourne pas de valeur.

```
Public Module modMain
```

L'exécution va démarrer à partir de la procédure Main() :

```
Sub Main()  
    'Appelle la fonction ci-dessous :  
    MaFonction()  
End sub
```

MaFonction : une petite fonction afin d'utiliser la classe Chien :

```
Function MaFonction()  
    'Voici comment déclarer une variable nommée Jimmy, de type chien Chien :  
    'Nous spécifions Animaux.Chien car la classe Chien fait partie de l'espace de Nom Animaux  
(cf. plus haut)  
    Dim Jimmy as Animaux.Chien  
  
    'Crée l'objet. Contrairement à VB6, le mot clé Set n'est plus utilisé !  
    Jimmy = New Animaux.Chien()  
  
    'Voici une autre manière de créer un objet :  
    'Dim Jimmy as New Chien  
  
    'Appel de la fonction principale de Jimmy  
    Jimmy.Aboie()  
End Function  
End module
```

## Leçon 2 : Portées de vos déclarations

Les principales portées sont Public, Private, Friend et Protected. Une classe peut contenir des attributs ou des méthodes, qui peuvent être de chacun des 4 types pré-cités :

- Ceux qui sont Public sont accessibles suite à la création de l'objet.
- Ceux qui sont Private ou Protected ne sont accessibles qu'à l'intérieur même du module de classe.
- Les membres Protected sont similaires aux Private, mais ils ont une particularité en cas d'héritage. Ce point sera abordé plus tard dans la leçon correspondante.
- Les membres Friend ne sont accessibles qu'à l'intérieur du projet, et pas par des éléments extérieurs au projet en cours.

Continuons sur notre exemple et importons l'espace de noms System :

```
Imports System
```

L'instruction de début de notre espace de noms : Animaux

(Tout le code qui suit en fera partie)

```
Namespace Animaux
```

Chien est une classe appartenant à l'espace de noms Animaux :

```
Public Class Chien
    'Voici une variable public
    Public AgeDuChien as Integer
End Class
```

Aboie est une fonction publique (Public) de notre classe Chien :

```
Public Function Aboie()
    Console.WriteLine ("Le chien aboie !")
End Function
```

Et Marche en est une autre, déclarée en privée (Private) :

```
Private Function Marche()
    Console.WriteLine ("Le chien marche")
End Function
End Class
End Namespace
```

Passons maintenant à notre module :

```
Public Module modMain
```

L'exécution va démarrer à partir de la procédure Main() :

```
Sub Main()  
    'Appelle notre fonction (cf. ci-dessus)  
    MaFonction()  
End sub  
  
    'MaFonction: Appelée à partir de la procédure Main()  
Function MaFonction()  
    Dim Jimmy as Animaux.Chien  
    Jimmy = New Animaux.Chien()  
  
    'Ce qui suit fonctionnera, car Aboie et AgeDuChien sont Public  
    Jimmy.Aboie()  
    Jimmy.AgeDuChien = 10  
  
    'Par contre, l'appel de la fonction Marche() échouera, car il se situe en dehors  
    'du module de classe Chien  
    'Donc le code qui suit est incorrect : décommentez-le  
    'et essayez de compiler, vous obtiendrez une erreur !  
    'Jimmy.Marche()  
End Function  
End Module
```

### **Vocabulaire :**

#### **L'encapsulation**

*La gestion de données et des procédures associées à une notion dans une classe s'appelle l'encapsulation.*

#### **Data Hiding ou Abstraction**

*Habituellement, dans une classe, les variables gérant les données, comme AgeDuChien, sont déclarée en tant que Private. Ce sont les procédures Property qui sont utilisée pour accéder à ces données. La protection des données d'un objet en dehors du module de classe est appelée Abstraction, ou bien encore Data Hiding. Ceci sert à prévenir des modifications "accidentelles" des données d'un objet via des fonctions externes au module de classe.*

## Leçon 3 : Fonctions Partagées

Dans une classe, les membres partagés (propriétés et méthodes) peuvent être appelés directement, sans passer par l'instanciation d'un objet (comme précédemment décrit). Le mot-clé Shared indique en effet que la propriété ou méthode ne s'appuie pas sur un objet spécifique mais bien sur la classe elle-même, et qu'elle peut être appelée directement à partir de cette classe, même si celle-ci n'a aucune instance en cours.

Une propriété Shared contient quant à elle une valeur commune à tous les objets d'une classe.

Notre exemple démarre de manière habituelle, avec l'import de l'espace de noms System, l'instruction de début de notre espace de noms, Animaux, et une classe lui appartenant, Chien :

```
Imports System
Namespace Animaux
Class Chien
```

Et Aboie est maintenant une fonction Public et Shared de cette classe :

```
Public Shared Function Aboie()
    Console.WriteLine ("Le chien aboie !")
End Function
```

Marche, quant à elle, est une fonction Public, mais non partagée.

```
Public Function Marche()
    Console.WriteLine ("Le chien marche ...")
End Function
End Class
End Namespace
```

Démarrons notre module, l'exécution se fera à partir de la procédure Main() :

```
Public Module modMain
Sub Main()
    'Nous pouvons faire appel à la méthode Aboie() directement,
    'Sans passer par la création d'un objet de type Chien,
    'puisque qu'elle est partagée
    Animaux.Chien.Aboie()

    'Par contre, nous ne pourrions utiliser la méthode Marche()
    'que suite à la création d'un objet, puisque celle-ci
    'n'est pas partagée.
    Dim Jimmy as Animaux.Chien
    Jimmy = New Animaux.Chien()
    Jimmy.Marche()

End sub
End Module
```

Réfléchissons un instant! La procédure WriteLine() que nous utilisons depuis le début est une méthode partagée de la classe Console.

De même, nous pouvons déclarer la procédure Main() elle-même comme une méthode partagée dans une classe :

c'est-à-dire Shared Sub Main().

Essayez de déplacer cette procédure Main() du module dans la classe Chien ci-dessus et testez le résultat !

## Leçon 4 : La Surcharge

La surcharge est une technique simple à utiliser, qui permet d'utiliser le même nom de fonction avec des paramètres de différents types. Voyons ce que ça donne sur la classe Addition dans l'exemple qui suit.

Comme d'habitude, nous avons besoin de l'espace de noms System :

```
Imports System
Class Addition
```

Ensuite, ajoutons 2 fonctions Add(). La première additionne 2 entiers ... (Convert.ToString est l'équivalent de la fonction CStr de VB6)

```
Overloads Public Sub Add(A as Integer, B as Integer)
    Console.WriteLine ("Adding Integers: " + Convert.ToString(a + b))
End Sub
```

... et la deuxième concatène 2 chaînes de caractères :

```
Overloads Public Sub Add(A as String, B as String)
    Console.WriteLine ("Adding Strings: " + a + b)
End Sub
```

Et les 2 ont le même noms ? Ceci est possible uniquement car nous avons utilisé le mot-clé Overloads dans leur déclaration, pour justement spécifier la surcharge !

Ici, voyons ce que ça donne avec la procédure Main() suivante, incluse dans le module de classe Addition (et donc déclarée en tant que partagée, cf paragraphe précédent) :

```
Shared Sub Main()
    Dim monCalcul as Addition
    'Crée l'objet
    monCalcul = New Addition

    'Appel de la première fonction ...
    monCalcul.Add(10, 20)

    ' ... et appel de la seconde :
    monCalcul.Add("Bonjour", " comment allez-vous ?")
End Sub
End Class
```

### Compléments :

#### Signature

*C'est ainsi que l'on appelle chaque séquence distincte de paramètres, c'est-à-dire chaque manière de faire appel à la méthode surchargée.*

## Leçon 5 : L'Héritage

L'héritage est un mécanisme par lequel une classe dérivée (ou classe fille) hérite de toutes les caractéristiques de sa classe de base (ou classe mère). En bref, il est possible de créer via héritage vos classes à partir d'une classe existante : Il suffit pour cela d'utiliser le mot-clé Inherits.

Voici un exemple simple. Commençons par importer l'espace de noms habituel :

```
Imports System
```

Voici notre classe de base :

```
Class Humain
    'Quelque chose que la plupart des êtres humains font :
    Public Sub Marche()
        Console.WriteLine ("Je marche ...")
    End Sub
End Class
```

Et maintenant, créons une classe fille nommée Developpeur :

```
Class Developpeur
    Inherits Humain
    'Nous avons donc déjà accès à la méthode Marche() définie ci-dessus

    'Et voici une autre qui illustre ce que certains programmeurs font parfois :
    Public Sub PiqueDuCode()
        Console.WriteLine ("Je pompe du code ...")
    End Sub
End Class
```

Ce qui nous permet de coder la procédure MainClass() suivante :

```
Class MainClass
    'Notre procédure principale :
    Shared Sub Main()
        Dim Tom as Developpeur
        Tom = New Developpeur

        'Cet appel est valide puisque Developpeur a accès à cette fonction
        'héritée de la classe Humain :
        Tom.Marche()

        'Celui-ci également puisque Tom est une instance de la classe Developpeur
        Tom.PiqueDuCode()
    End Sub
End Class
```

### Compléments :

#### MustInherit

Ce mot clé indique qu'une classe ne peut être instanciée, et qu'elle ne peut donc être utilisée **que** comme classe de base. Par exemple, si vous déclarez la classe Humain en tant que "MustInherit Humain", alors vous ne pourrez pas créer d'objets de type Humain à partir de cette classe, mais seulement à partir de classes dérivées.

### ***NotInheritable***

*A l'inverse, ce mot-clé indique que cette classe ne peut pas être héritée, c'est-à-dire servir de classe de base. Par exemple, si vous déclarez la classe Humain en tant que "NotInheritable Humain", aucune classe dérivée ne pourra lui être créée.*

## Leçon 6 : La Rédéfinition

Par défaut, une classe dérivée hérite des méthodes de sa classe de base. Mais si une propriété ou une méthode doit avoir un comportement différent dans la classe fille, elle doit y être redéfinie !

En fait, il suffit de définir une nouvelle implémentation de la méthode dans la classe dérivée :

Le mot-clé `Overridable` est utilisé pour préciser qu'une méthode peut être redéfinie,

et le mot-clé `Overrides` est quant à lui utilisé pour indiquer quelles méthodes sont redéfinies.

Étudions ceci sur un exemple, à partir de notre classe `Humain` :

```
Imports System

Class Humain
    'Parle() est redéfinissable :
    Overridable Public Sub Parle()
        Console.WriteLine ("Je parle")
    End Sub
End Class
```

Maintenant, créons une classe dérivée de `Humain` :

Un Indien (Indien) est un humain (`Humain`) :

```
Class Indien
    Inherits Humain
    'Notre Indien parle l'Hindi, la langue nationale en Inde
    'la méthode Parle() qui suit redéfinit la méthode Parle() de la classe de base (Humain)
    Overrides Public Sub Parle()
        Console.WriteLine ("Je parle Hindou")
    End Sub
End Class
```

Important : Évidemment, un appel de `Parle()` à partir d'un objet d'une classe dérivée va faire appel à la méthode `Parle()` de cette même classe.

Si vous souhaitez faire référence à la méthode `Parle()` de la classe de base, il vous suffit alors d'utiliser le mot-clé `MyBase`, comme ceci :

```
Mybase.Parle()
```

Voici une classe pour tester nos exemples :

```
Class MainClass
    'Notre procédure principale :
    Shared Sub Main()
        'Tom est un humain
        Dim Tom as Humain
        Tom = New Humain
    End Sub
End Class
```

```
'Tony est humain, et plus précisément Indien
Dim Tony as Indien
Tony = New Indien

'Voici un appel à la méthode Parle() de la classe Humain :
Tom.Parle()

'Et un appel de la méthode Parle() de la classe Indien
Tony.Parle()
End Sub
End Class
```

### **Compléments :**

#### ***MustOverride***

*Ce mot clé indique qu'une méthode doit être redéfinie pour chaque classe fille. Par exemple, si dans la classe Humain, vous déclarez la méthode Parle en tant que "MustOverride Parle", chacune des classes filles devra redéfinir cette méthode.*

#### ***NotOverridable***

*A l'inverse, ce mot-clé indique que cette procédure ne peut être redéfinie dans une classe fille. Par exemple, si dans la classe Humain, vous déclarez la méthode Parle en tant que "NotOverridable Parle", aucune classe fille ne pourra redéfinir cette méthode.*

## Leçon 7 : Le Polymorphisme d'héritage

Le polymorphisme est un mécanisme via lequel un objet peut prendre plus d'une forme. Par exemple, si vous avez une classe de base nommée Humain, un objet de type Humain peut être utilisé pour contenir un objet de n'importe laquelle de ses classes dérivées. Quand vous appelez une méthode à partir de votre objet, le système déterminera automatiquement le type de l'objet afin d'appeler la fonction appropriée.

Imaginons par exemple une méthode nommée Parle() dans la classe de base Humain. Créons ensuite une classe fille et redéfinissons-y la méthode Parle(). Puis créez un objet de la classe fille et affectez-lui la variable de la classe mère : maintenant, si vous faites appel à la méthode Parle() à partir de cette variable, ce sera celle définie dans votre classe fille qui sera appelée.

Au contraire, si vous affectez à cette variable un objet de la classe mère, alors ce sera la méthode Parle() de la classe mère qui sera appelée.

Ceci est déterminé lors de la compilation, c'est ce qu'on appelle une liaison tardive.

Reprenons l'exemple du paragraphe précédent : la différence se situe dans la procédure Shared Sub Main(), de la classe MainClass. Étudions le code pour voir ce qui a changé :

```

Import Systems

Class Humain
    'Parle() peut être redéfinie :
    Overridable Public Sub Parle()
        Console.WriteLine ("Je parle")
    End Sub
End Class

```

Maintenant, créons une classe dérivée de Humain :

Un Indien (Indien) est un humain (Humain) :

```

Class Indien
    Inherits Humain

    'Notre Indien parle l'Hindi, la langue nationale en Inde
    'la méthode Parle() qui suit redéfinit la méthode Parle() de la classe de base (Humain)
    Overrides Public Sub Parle()
        Console.WriteLine ("Je parle Hindou")
    End Sub
End Class

```

Étudions soigneusement le code suivant :

```

Class MainClass
    'Notre fonction principale :
    Shared Sub Main()
        'Tom est un humain (classe de base)
        Dim Tom as Humain

        'Affectons-lui une variable de la classe fille Indien
    End Sub
End Class

```

```
Tom = New Indian
'Cette affectation est correcte, car la classe Indien
'est fille de la classe Humain.

'Puis faisons appel à la méthode Parle comme ceci :
Tom.Parle()
End Sub
End Class
```

Sur le dernier appel (Tom.Parle()), quelle procédure va être exécutée ? Celle de la classe Indien, ou bien celle de la classe Humain ?

Cette question est pertinente, car Tom a été déclaré en tant qu'Humain, mais un objet de type Indien lui a été affecté.

La réponse est : c'est la méthode Parle() de la classe Indien va être appelée.

Ceci parce que, comme la plupart des langages orientés objet, Vb.NET peut automatiquement détecter le type de l'objet assigné à la variable appartenant à la classe de base : c'est ce que l'on appelle le Polymorphisme.

## Leçon 8 : Constructeurs et Destructeurs

Un constructeur (Constructor) est une procédure spéciale qui est automatiquement exécutée quand un objet est créé. Une telle procédure se nomme New(). Comme indiqué dans la leçon 4, les constructeurs peuvent faire l'objet de surcharge mais contrairement aux autres méthodes, le mot clé Overloads n'est pas requis.

Par opposition, un destructeur (Destructor) est une procédure qui est automatiquement exécutée à la mort de l'objet. Une telle procédure se nomme Finalize(). En VB6, ces procédures étaient déjà disponibles, elles se nommaient Class\_Initialize() et Class\_Terminate().

Démarrons l'étude d'un exemple, avec la classe Chien :

```
Imports System

Class Chien
    'La variable age
    Private Age as Integer
```

Voici un constructeur :

```
Public Sub New()
    Console.WriteLine ("Un chien est né")
    Age = 0
End Sub
```

Et grâce à la surcharge, un 2° exemple de constructeur (avec paramètre cette fois-ci) :

```
Public Sub New(val as Integer)
    Console.WriteLine ("Nouveau chien dont l'âge est " + Convert.ToString(val))
    Age = val
End Sub
```

Voici le code du destructeur, suivi de la procédure habituelle Main() :

```
Overrides Protected Sub Finalize()
    Console.WriteLine ("Paf le chien ... (désolé)")
End Sub

'La fonction principale :
Shared Sub Main()
    Dim Jimmy, Jacky as Chien

    'Création des objets
    'Ceci va appeler le constructeur n°1
    Jimmy = New Chien

    'Et ceci le constructeur n°2
    Jacky = New Chien (10)
End Sub

End Class
```

La destruction intervient automatiquement en fin de portée ou de programme, grâce au Garbage Collector. Il est possible de faire le ménage vous-même en détruisant vos objets.

Ex :

```
Jimmy = Nothing
```

## Leçon 9 : Les Propriétés

Pour gérer les données de vos objets, vous avez le choix entre des champs ou des propriétés. Mais alors que les champs sont de simples variables publiques, les propriétés utilisent des procédures pour encadrer la lecture et la mise à jour des données (cf. la définition d'encapsulation). Ce sont les mots-clé Get (lecture) et Set (mise à jour) qui sont utilisées pour la déclaration de ces procédures.

Étudions l'exemple suivant :

```
Imports System

Public Class Chien
    'Utilisons une variable privée pour l'âge :
    Private mAgeDuChien as Integer
```

Et voici nos procédures Property :

```
Public Property Age() As Integer
    'Lecture de l'âge :
    Get
        Console.WriteLine ("Getting Property")
        Return mAgeDuChien
    End Get

    'MAJ de l'âge :
    Set(ByVal Value As Integer)
        Console.WriteLine ("Setting Property")
        mAgeDuChien = Value
    End Set

End Property
End Class
```

Voyons comment utiliser ces procédures :

```
Class MainClass
    'Début du programme :
    Shared Sub Main()

        'Créons un objet :
        Dim Jimmy as Chien
        Jimmy = New Chien

        'Nous ne pouvons accéder directement à la variable mAgeDuChien,
        'nous devons utiliser la procédure Age().
        'Affectons-lui une valeur. La procédure Set Age sera mise à contribution :
        Jimmy.Age = 30

        'Récupérons maintenant cette valeur : c'est au tour de la procédure Get Age de travailler :
        Dim curAge = Jimmy.Age()

    End Sub
End Class
```

## Leçon 10 : Une petite application

Passons maintenant à l'étude d'un programme basique. Importons tout d'abord les espaces de noms requis :

```
Imports System
Imports System.ComponentModel
Imports System.Windows.Forms
Imports System.Drawing
'Nous allons créer une classe dérivée de la classe System.Windows.Forms.Form
'En effet, Forms est un espace de noms inclus dans l'espace de noms Windows,
'lui-même inclus dans System, et Form est une classe de Forms.
Public Class SimpleForm

Inherits System.Windows.Forms.Form

'Notre constructeur :
Public Sub New()
'On fait appel à la procédure de la classe mère :
MyBase.New()
```

Affectons la propriété Text de cette classe. Nous avons hérité cette propriété de la classe mère :

```
Me.Text = "Bonjour, comment allez-vous ?"
End Sub
End Class
```

```
Public Class MainClass
Shared Sub Main()
'Créons un objet à partir de cette classe SimpleForm :
Dim sf as SimpleForm
sf = New SimpleForm

'Et passons cet objet en argument à la procédure Run() pour démarrer :
System.Windows.Forms.Application.Run(sf)
End Sub
End Class
```

Nous avons fait le tour...

Maintenant, vous êtes apte à lire et appréhender la plupart des sources VB.Net relatives à la POO, et certainement capable de faire un bon usage des mécanismes de la POO dans vos propres programmes.

## Téléchargements

Les sources : [OOPS\\_In\\_VBNET\\_src.zip](#) (FTP) ou [OOPS\\_In\\_VBNET\\_src.zip](#) (HTTP)

Le tutoriel au format PDF (23 pages, 80 Ko) : [POO\\_VB\\_NET.pdf](#) (FTP) ou [POO\\_VB\\_NET.pdf](#) (HTTP)