Surcharge, Redéfinition et Occultation avec VB.NET

par Mohamed Elhadi Benzeghiba

Date de publication : 11/07/2007

Dernière mise à jour :

L'objectif de ce tutorial est d'expliquer les notions de surcharge, de redéfinition et d'occultation apparues avec Visual Basic .NET, afin d'éviter aux développeurs toute confusion entre les mots clés servant à mettre en oeuvre ces mécanismes.

I - Introduction
II - Prerequis
III - Définitions
    III-A - L'héritage
    III-B - Le polymorphisme
    III-C - Signature de paramètres d'une méthode
IV - La surcharge de méthodes
V - La redéfinition de méthodes
VI - L'occultation
VII - Différences entre redéfinition et occultation
VIII - Glossaire des mots clés
IX - Bibliographie

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

I - Introduction

.NET est la première version de Visual Basic à introduire la notion d'héritage entre classes. Cela a conduit à l'apparition de nouveaux mots clés dont certains se ressemblent par leur syntaxe et d'autres par leur fonctionnement. De ce fait, les développeurs peuvent se trouver facilement perdus.

Ainsi, si le concept de surcharge et le rôle du mot clé Overloads sont faciles à comprendre, la confusion entre Overloads et deux mots clés introduits par la redéfinition, Overridable et Overrides, est probable. Même pour les développeurs qui arrivent à passer ce stade sans incident, l'apprentissage de l'occultation et de son mot clé Shadows risque de tout chambouler ! Pour présenter les choses de manière claire, ce tutorial commence par expliquer les notions d'héritage et de polymorphisme nécessaires pour comprendre le reste du document. Ensuite, il aborde les mécanismes de surcharge, de redéfinition et d'occultation en mettant l'accent plus sur le concept que sur les mots clés. Les explications seront appuyées par des exemples simples et chaque paragraphe sera clôturé par une liste des points essentiels à retenir. Beaucoup de livres et articles traitent des concepts abordés dans ce document, le présent tutorial a pour but de faciliter l'assimilation et d'éviter les confusions habituelles. Pour cela, nous allons volontairement éviter certains détails, par exemple, certaines règles spécifiques aux constructeurs, aux évènements et aux membres partagés ne seront pas évoquées.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

II - Prerequis

La connaissance des bases du langage VB.NET, comme par exemple la syntaxe de déclaration d'une procédure ou d'une fonction, est nécessaire pour bien comprendre ce tutorial. La connaissance des concepts de la programmation orientée objet est particulièrement souhaitable. Par conséquent, si vous ne connaissez pas le langage VB.NET et que vous cherchez à le prendre en main, ce tutorial n'est sûrement pas pour vous.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

III - Définitions

Avec sa version .NET, Visual Basic est devenu un véritable langage orienté objet, puisque cette version a introduit les constructeurs de classes, l'héritage entre classes, le polymorphisme et bien d'autres concepts. Même si ce tutorial suppose que vous connaissez les bases de l'orienté objet et la syntaxe correspondante dans Visual Basic .NET, nous allons revenir ici sur quelques définitions qui nous seront utiles par la suite.

III-A - L'héritage

L'héritage entre classes consiste en la possibilité de créer une classe, dite classe dérivée, qui englobe toutes les propriétés, les méthodes et les évènements définis dans une autre classe, dite classe de base, sans avoir à les re-saisir. Bien entendu, le principal avantage de l'héritage n'est pas d'éviter la saisie mais surtout d'établir un lien de spécialisation entre les classes. Par exemple, on peut créer une classe Employe en la faisant hériter d'une classe Personne, puisque tout employé est une personne. Ainsi, la classe Employe contiendra automatiquement toutes les caractéristiques de la classe Personne comme le nom et le prénom, et pourra définir des membres spécifiques à un employé comme le salaire par exemple.

Dans Visual Basic .NET, la mise en oeuvre de l'héritage passe par l'utilisation du mot clé Inherits, comme le montre l'exemple suivant : Déclaration de la classe de base :

```
Public Class Personne
Public titre As String
Public nom As String
Public prenom As String
Public DateDeNaissance As Date
End Class
```

Pour la classe dérivée on utilise le mot clé Inherits et on définit uniquement les membres spécifiques à un employé :

```
Public Class Employe
Inherits Personne
Public Salaire As Double
End Class
```

Comme la classe Employe englobe automatiquement tous les membres de la classe Personne, le code suivant fonctionne sans problème :

```
Module Execution
Sub main()
Dim e As New Employe
'On peut faire référence aux membres
'définis dans la classe Personne
Console.Out.WriteLine(e.nom & " " & e.prenom)
End Sub
End Module
```

Remarque : *vous êtes supposés savoir qu'il est déconseillé d'utiliser des champs publics dans les classes, et qu'il faut utiliser des champs Private exposés, si nécessaire, via des propriétés. Dans ce tutorial, cette règle ne sera pas respectée dans le but de condenser le code.*

III-B - Le polymorphisme

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Le polymorphisme est la possibilité de référencer un objet du type de la classe dérivée avec une variable dont le type est la classe de base, comme dans l'exemple suivant où, à la variable p de type Personne, on affecte un objet Employe, ce code est logiquement correct puisque un employé est en fait une personne :

```
Dim p As Personne
p = New Employe
```

Le polymorphisme garantie aussi autre chose, si une méthode définie dans la classe de base est redéfinie dans une classe dérivée, l'appel à cette méthode à travers une variable de type classe de base va dépendre du type de l'objet pointé par la variable et non pas du type de la variable. Pour clarifier ces propos, considérons l'exemple suivant :

La classe Personne définit une méthode affiche() qui affiche les renseignements de la personne, la classe Employe qui dérive de Personne redéfinit la méthode affiche() afin d'afficher aussi le salaire de l'employé. Dans ce cas quel sera le résultat de cette portion de code

```
Dim p As Personne
p = New Employe
'Cette instruction affiche aussi le salaire
'car la méthode appelé est celle de la classe
'employe
p.affiche()
```

Dans la dernière instruction, est-ce que c'est la méthode définie dans la classe Personne qui sera appelée ou bien celle définie dans la classe Employe ?

Autrement dit, est-ce que le salaire sera affiché ou non ?

Le polymorphisme garantit aussi que, si une méthode appelée soit celle définie dans la classe de l'objet désigné par la variable et non pas celle de la classe de la variable utilisée pour appeler la méthode. Dans notre cas, la variable p utilisée pour appeler la méthode est de type Personne, mais comme elle pointe vers un objet Employe (à cause de l'instruction : p = new Employe) alors la méthode appelée est celle définie dans la classe Employe, et par conséquent le salaire sera affiché.

III-C - Signature de paramètres d'une méthode

La signature de paramètres d'une méthode est définie par le nombre, l'ordre et le type de ses paramètres. Par exemple, les deux méthodes suivantes ont la même signature :

```
Sub methode1(ByVal par1 As Integer, ByVal par2 As String, ByRef par3 As String)
...
End Sub
Sub methode1(ByVal p1 As Integer, ByVal p2 As String, ByVal p3 As String)
...
End Sub
```

[1] Nous consacrerons par la suite toute une section sur la redéfinition de méthodes

Remarquez que le nom des paramètres et leur ordre de passage (byval ou byref) n'ont pas d'influence sur la signature. Pour que deux signatures de paramètres soit différentes, il ne faut pas que la différence soit due uniquement à un paramètre optionnel ou uniquement aux types de retour pour les fonctions. Dans les deux exemples suivants le compilateur va générer une erreur. Rappelons qu'il n'est pas possible de déclarer dans la même classe deux méthodes avec le même nom et la même signature de paramètres :

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

```
'Erreur les deux signatures diffèrent par un paramètre optionnel
'ce qui n'est pas suffisant
Sub methode1(ByVal param1 As Integer)
...
End Sub
Sub methode1(ByVal p1 As Integer, Optional ByVal p3 As String = "")
...
End Sub
```

```
'Erreur les deux signatures diffèrent uniquement par leur type de retour
'ce qui n'est pas suffisant
Function fonction1(ByVal param1 As Integer) As Integer
'...
End Function
Function fonction1(ByVal p1 As Integer) As Date
'...
End Function
```

Maintenant que vous avez appris ce que sont l'héritage, le polymorphisme et la signature de méthode, nous pouvons aborder la surcharge, la redéfinition et l'occultation sans risquer aucune ambiguïté.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

IV - La surcharge de méthodes

Visual Basic .NET autorise la surcharge de méthodes, c'est-à-dire qu'il permet de déclarer dans une même classe plusieurs méthodes ayant le même nom mais des signatures différentes.

Lorsque vous écrivez plusieurs versions d'une même méthode dans une classe, s'il n y a aucune méthode portant le même nom dans les classes parentes de cette classe 2 alors vous n'avez besoin d'aucun mot clé particulier pour déclarer vos méthodes, comme dans le cas suivant 3 pour la fonction additionne :

```
Public Class Addition
'Plusieurs version de la fonction additionne
'aucun mot clé particulier n'est nécessaire
'puisque la classe Object ne possède pas une méthode additionne
Function additionne(ByVal p1 As Double, ByVal p2 As Double) As Double
Return p1 + p2
End Function
Function additionne(ByVal p1 As Double, ByVal p2 As Double, _
ByVal p3 As Double) As Double
Return p1 + p2 + p3
End Function
End Class
```

En revanche, si la méthode que l'on souhaite surcharger a déjà été définie dans l'une des classes de base, alors chaque version surchargée de la méthode dans la classe en cours doit être précédée par le mot clé Overloads.

Dans l'exemple suivant, nous allons écrire la classe AdditionAvancee qui hérite de la classe Addition précédente. Dans cette nouvelle classe, nous allons définir deux nouvelles version de la méthode Additionne, qui prennent respectivement quatre et cinq paramètres. Comme la méthode Additionne a déjà été définie dans la classe de base Addition, alors les deux nouvelles versions de la méthode Additionne doivent être précédées par le mot clé Overloas :

```
Public Class AdditionAvancee
Inherits Addition
Overloads Function additionne(ByVal p1 As Double, ByVal p2 As Double, _
ByVal p3 As Double, ByVal p4 As Double) As Double
Return p1 + p2 + p3 + p4
End Function
Overloads Function additionne(ByVal p1 As Double, ByVal p2 As Double, _
ByVal p3 As Double, ByVal p4 As Double, ByVal p5 As Double) As Double
Return p1 + p2 + p3 + p4 + p5
End Function
End Class
```

[2] Rappelons que toutes les classes que vous définissez possèderont une classe de base, car même si vous n'utilisez pas le mot clé Inherits, votre classe héritera implicitement de la classe System.Object

[3] On aurait pu écrire une seule méthode additionne qui calcule la somme d'un nombre quelconque de paramètres en utilisant un seul paramètre ParamArray. Or, ce que nous cherchons ici c'est juste un exemple pour illustrer les choses. Ce sera le cas pour plusieurs autres exemples à venir.

Dans la classe fille, nous pouvons définir une méthode qui a le même nom et la même signature qu'une méthode de la classe de base en utilisant le mot clé Overloads. Cependant, comme elles ont la même signature ce ne sera pas une surcharge mais une occultation, nous reviendrons un peu plus loin sur ce concept.

A retenir

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

- Nous pouvons définir plusieurs versions d'une même méthode à condition qu'elles aient des signatures différentes. Dans la terminologie orientée objet cela s'appelle la surcharge de méthodes.
- Lorsque nous définissons une méthode surchargée pour la première fois, aucun mot clé n'est nécessaire. Les méthodes seront écrites normalement.
- Lorsque nous définissons des versions de surcharge pour une méthode qui a été définie dans l'une des classes de base alors, chaque version doit être précédée par le mot clé Overloads.
- La surcharge peut s'appliquer aussi aux propriétés, dans ce cas les règles sont les mêmes que pour les méthodes.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

V - La redéfinition de méthodes

Comme vous le savez déjà, une classe dérivée hérite de toutes les méthodes et propriétés de sa classe de base. Or, il se pourrait qu'une méthode nécessite d'être adaptée à la classe dérivée et non pas reprise en l'état.

Considérons à nouveau les deux classes Personne et Employe :

```
Public Class Personne
Public titre As String
Public nom As String
Public prenom As String
Public Sub affiche()
Console.Out.Write(titre & " " & nom & " " & prenom)
End Sub
End Class
```

Voici le code de la classe Employe :

```
Public Class Employe
Inherits Personne
Public Salaire As Double
End Class
```

Dans le cas précédent, la classe Employe va hériter de la méthode affiche définie dans la classe Personne. Or cette méthode n'affiche pas le salaire de l'employé et donc elle est en quelque sorte incomplète pour la classe Employe qui a tout intérêt de l'adapter à ses besoins en la redéfinissant. Nous allons voir comment se fait la redéfinition de méthode.

Tout d'abord, il faut savoir que c'est le concepteur de la classe de base qui décide quelles méthodes peuvent être redéfinies. Par défaut, les méthodes ne peuvent pas être redéfinies. Elles doivent être uniquement héritées en l'état ou occultées (le chapitre suivant est consacrée à l'occultation). Pour déclarer une méthode qui peut être redéfinie il faut faire précéder sa déclaration par le mot clé Overridable. Par exemple, voici la déclaration de la méthode affiche() de la classe Personne pour qu'elle soit redéfinissable [4] :

```
Public Overridable Sub affiche()
Console.Out.Write(titre & " " & nom & " " & prenom)
End Sub
```

Maintenant que nous avons rendu la méthode affiche() virtuelle, nous allons voir comme la redéfinir dans la classe dérivée.

Pour cela, il suffit de déclarer dans la classe dérivée une méthode qui a le même nom et la même signature [5] que la méthode virtuelle en faisant précéder sa déclaration par le mot clé Overrides. Comme dans l'exemple suivant pour la classe Employe :

[4] Les méthodes qui peuvent être redéfinies sont appelées également méthodes virtuelles.

[5] Une méthode de la classe dérivée ne peut redéfinir que la méthode ayant le même nom et la même signature qu'elle dans la classe de base.

```
Public Class Employe
Inherits Personne
Public salaire As Double
'La classe Personne possède maintenant sa propre méthode affiche
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

```
'qui affiche aussi le salaire de l'employé
Public Overrides Sub affiche()
Console.Out.Write(titre & " " & nom & " " & prenom & " " & salaire)
End Sub
End Class
```

Ce qui est important à savoir c'est que la redéfinition de méthodes est totalement compatible avec le polymorphisme. Ainsi, si on appelle la méthode affiche() à partir d'une variable de type Personne, ce n'est pas la méthode définie dans la classe Personne qui sera toujours appelée, car le comportement polymorphique veut que la méthode appelée ne soit pas celle du type de la variable mais du type pointé par la variable. Encore un exemple pour illustrer ces propos :

```
Module Execution
Sub main()
Dim p As New Personne
p.titre = "M."
p.nom = "Dubois"
p.prenom = "Dupont"
'L'instruction suivante va appeler la méthode affiche
'de la classe Personne
p.affiche()
Console.Out.WriteLine()
Dim e = New Employe
e.titre = "Mme"
e.nom = "Dupuis"
e.prenom = "Doloris"
e.salaire = 10000
p = e 'La variable p pointe désormais vers un objet Employe
p.affiche()
End Sub
End Module
```

Dans cet exemple, la première instruction p.affiche() va faire appel à la méthode définie dans la classe Personne, non pas parce que la variable p est de type Personne mais parce qu'elle pointe vers un objet de type Personne. La deuxième instruction p.affiche() va faire appel à la méthode définie dans la classe Employe qui affiche aussi le salaire car la variable p pointe vers un objet Employe.

Lors de la redéfinition d'une méthode, l'utilisation du mot clé MyBase pour appeler la méthode de la classe de base peut souvent nous rendre un grand service. Dans l'exemple précédent la méthode affiche() de la classe Employe aurait pu être écrite comme suit :

```
Public Overrides Sub affiche()
MyBase.affiche()
Console.Out.Write(" " & salaire)
End Sub
```

Autre chose importante à signaler, les méthodes déclarées Overrides (c'est-à-dire qui redéfinissent une méthode de leur classe de base) sont automatiquement virtuelles sans qu'il y ait besoin d'utiliser le mot clé Overridable. Par exemple, une classe Manager qui hérite de la classe Employe peut redéfinir la méthode affiche même si cette méthode n'a pas été déclarée Overridable dans la classe Employe. Pour faire en sorte qu'une méthode redéfinie ne soit plus virtuelle il faut utiliser explicitement le mot clé NotOverridable. Voici le code complet et commenté de la classe Manager :

```
Public Class Manager
Inherits Employe
Public PrimeResponsabilite As Double
'La classe Manager peut redéfinir la méthode affiche de la classe Employe
'mais en la déclarant NotOverridable les classes qui hériteront de la classe
'Manager ne pourront plus redéfinir cette méthode !
Public NotOverridable Overrides Sub affiche()
MyBase.affiche()
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

```
Console.Out.Write(" " & PrimeResponsabilite)
End Sub
End Class
```

A retenir

- Pour rendre une méthode redéfinissable il faut faire précéder sa déclaration par le mot clés
- Overridable. Cette méthode sera appelée alors méthode virtuelle.
- Pour redéfinir une méthode virtuelle, il suffit de définir une méthode dans la classe dérivée qui a le même nom et la même signature que la méthode virtuelle et dont la déclaration est précédée par le mot clé Overrides.
- Les méthodes qui redéfinissent des méthodes virtuelles sont automatiquement virtuelles à leur tour. Pour faire en sorte qu'une telle méthode ne soit pas virtuelle il faut précéder sa déclaration par le mot clé NotOverridable.
- L'appel à une méthode redéfinie est complètement polymorphe.
- La redéfinition peut s'appliquer aussi aux propriétés, dans ce cas les règles sont les mêmes que pour les méthodes.

Et si on redéfinit une méthode surchargée ?

A ce stade, et avant de poursuivre il est important de comprendre exactement la différence entre surcharge et redéfinition et le rôle de chacun des mots clés Overloads, Overridable et Overrides.

Dans une classe dérivée, on peut à la fois redéfinir une méthode virtuelle (avec une méthode qui a le même nom et la même signature) et surcharger cette méthode (en proposant d'autres méthodes ayant le même nom mais des signatures différentes).

Ce cas de figure est en fait très simple, il n y a absolument pas de règles spécifiques, ce sont les règles précédentes qui s'appliquent. Ainsi :

- Dans la classe dérivée, toutes les versions surchargées de la méthode doivent être précédées par le mot clé Overloads.

- La méthode ayant la même signature que la méthode virtuelle doit être précédée par le mot clé Overrides (puisqu'il s'agit d'une redéfinition). Par conséquent, cette dernière méthode sera précédée à la fois par le mot clé Overloads et le mot clé Overrides.

En voici un exemple de la classe Employe qui redéfinie la méthode affiche et lui propose une nouvelle version surchargée :

```
Public Class Employe
Inherits Personne
Public salaire As Double
'Méthode de redéfinition et de surcharge
'Précédée à la fois par Overloads et Overrides
Public Overloads Overrides Sub affiche()
Console.Out.Write(titre & " " & nom & " " & prenom & " " & salaire)
End Sub
'Méthode de surcharge seulement précédée par Overloads
Public Overloads Sub affiche(ByVal prefixe As String)
Console.Out.Write(prefixe & " ")
MyClass.affiche()
End Sub
End Class
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

VI - L'occultation

Nous avons vu dans la section précédente comment redéfinir une méthode de la classe de base et comment celle-ci doit être déclarée virtuelle, avec le mot clé Overridable, pour pouvoir être redéfinie dans les classes dérivées.

Or, il se pourrait que l'on ait besoin de redéfinir une méthode qui n'a pas été déclarée virtuelle dans la classe de base. Dans ce cas, si nous avons accès au code source de la classe de base le mieux serait de modifier la classe de base afin de déclarer la méthode avec le mot clé Overridable. En revanche, si la classe de base se trouve dans une DLL alors il n y a pas moyen de rendre la méthode virtuelle et le seul moyen de la remplacer dans la classe dérivée est l'occultation (Shadowing en Anglais).

Pour indiquer qu'une méthode occulte la méthode portant le même nom de la classe de base, il suffit de faire précéder la déclaration de cette méthode par le mot clé Shadows 6. Comme dans l'exemple suivant :

```
Public Class Personne
Public titre As String
Public nom As String
Public prenom As String
Public Sub affiche()
Console.Out.Write(titre & " " & nom & " " & prenom)
End Sub
Public Sub affiche(ByVal prefixe As String)
Console.Out.Write(prefixe & " ")
Console.Out.Write(titre & " " & nom & " " & prenom)
End Sub
End Class
```

```
Public Class Employe
Inherits Personne
Public salaire As Double
'Cette méthode occulte les deux occurrences de la méthode affiche de la
'classe de Personne
Public Shadows Sub affiche()
Console.Out.Write(titre & " " & nom & " " & prenom & " " & salaire)
End Sub
End Class
```

Dans l'exemple précédent, la méthode affiche() de la classe Personne n'est pas virtuelle et donc pour la remplacer dans la classe Employe nous sommes obligé de la déclarer avec le mot clé Shadows.

6 A vrai dire le mot clé Shadows n'est pas obligatoire, cependant, si vous l'omettez vous obtiendrez un message d'avertissement (warning) du compilateur.

Il faut savoir que dans le cas d'une occultation, la méthode déclarée avec le mot clé Shadows occulte toutes les méthodes portants le même nom dans la classe de base (quelque soit leurs signatures), contrairement à la redéfinition où la méthode déclarée avec Overrides qui ne redéfinit que la méthode ayant le même nom et la même signature. Par exemple, dans le code suivant la deuxième instruction va générer une erreur de compilation :

```
Dim e As New Employe
e.affiche("Employe :") 'cette instruction va générer une erreur de compilation
```

Mais pourquoi cette instruction ne se compile pas alors que normalement la classe Employe devait hériter la méthode affiche*(ByVal prefixe As String)* de la classe Personne. La raison est que la méthode affiche() de la classe Employe occulte toutes les méthodes portants le nom affiche dans la classe Personne, elle les occulte. Cela veut dire qu'elle les supprime de la classe Employe et qu'elle les remplace. Par conséquent, la classe Employe ne contient pas la méthode affiche*(ByVal prefixe As String)*

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Autre chose très importante à connaître sur l'occultation, c'est que l'appel à une méthode occultée ne respecte pas le principe du polymorphisme. Autrement dit, quand on fait appel à une méthode occultée, la méthode appelée dépend du type de la variable utilisée pour l'appel et non pas du type de l'objet pointé par cette variable. Cela vous semble flou, examinons le résultat produit par le code suivant :

```
Sub main()
Dim p As New Personne
p.titre = "M."
p.nom = "Dubois"
p.prenom = "Dupont"
'L'instruction suivante va appeler la méthode affiche de la classe
Personne
p.affiche()
Console.Out.WriteLine()
Dim e = New Employe
e.titre = "Mme"
e.nom = "Dupuis"
e.prenom = "Doloris"
e.salaire = 10000
p = e 'La variable p pointe désormais vers un objet Employe
p.affiche()
End Sub
```

Dans le code précédent, à la dernière instruction p.affiche() la variable p qui pointe vers un objet Employe. Mais comme la méthode affiche est occultée dans la classe Employe alors l'appel p.affiche() n'est pas polymorphique. Ce qui veut dire que la méthode affiche() appelée est celle de la classe de la variable p (donc Personne) et non pas celle de l'objet pointé par la variable p (Employe). Donc, si vous exécutez le code précédent, la dernière instruction n'affichera pas le salaire de Mme Dupuis.

Dernière chose à connaître sur l'occultation, c'est qu'elle permet d'anticiper les changements dans la classe de base (dans le cas où celle-ci ne nous appartient pas) et de mettre la classe dérivée à l'abri dans le cas où on ajoute à la classe de base un membre qui a le même nom qu'un membre déjà présent dans la classe dérivée.

A retenir

- L'occultation sert à remplacer des méthodes dans la classe dérivée qui n'ont pas été déclarées virtuelles dans la classe de base.
- Pour déclarer une méthode qui occulte tous les membres ayant le même nom dans la classe de base, il faut faire précéder la déclaration de la méthode par le mot clé Shadows.
- Les membres occultés ne sont pas hérités par la classe dérivée.
- L'appel des méthodes occultées ne respecte pas le principe du polymorphisme.
- Si votre besoin est d'adapter le comportement d'une méthode à la classe dérivée, alors la redéfinition est toujours préférable à l'occultation, tant que la redéfinition est possible.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

VII - Différences entre redéfinition et occultation

Le tableau suivant 7 présente les principales différences entre la redéfinition et l'occultation :

Point de comparaison	Redéfinition	Occultation
Objectif	Permet d'adapter les méthodes de la classe de base à la classe dérivée tout en respectant le polymorphisme	Offre la possibilité de remplacer des méthodes non virtuelles de la classe de base. Permet de protéger la classe dérivée contre des changements futurs dans la classe de base (au cas où la classe de base ajoute une méthode qui le même nom qu'une méthode de la classe dérivée)
Elément de redéfinition (élément que l'on utilise pour redéfinir un autre élément)	Méthodes et propriétés	N'importe quel membre de la classe
Elément redéfini	Uniquement la propriété ou la méthode ayant le même nom et la même signature que l'élément redéfinissant	Tous les membres ayant le même nom que l'élément redéfinissant
Accessibilité (Public, private, ...)	La méthode de la classe dérivée ne peut pas modifier l'accessibilité de la méthode de la classe de base	L'élément de la classe dérivée peut avoir une accessibilité différente de celle de l'élément occulté
Possibilité de lecture et d'écriture de l'élément (élément déclaré ReadOnly et WriteOnly)	La classe de base peut indiquer les éléments redéfinissables avec : Overridable et MustOverride et les éléments qui ne peuvent être redéfinis avec NotOverridable	Ces attributs peuvent être changés lors d'une occultation
Contrôle de la redéfinition	Dans la classe de base : Overridable, MustOverride Dans la classe dérivée : Overrides	La classe de base ne peut pas forcer ou annuler l'occultation de ces éléments
Mots clés utilisés	C'est la nouvelle version définie dans la classe dérivée qui sera héritée. Les méthodes redéfinies ne seront pas héritées	Dans la classe dérivée : Shadows
Héritage des éléments utilisés pour la redéfinition par les classes qui héritent de la classe dérivée	C'est la nouvelle version définie dans la classe dérivée qui sera héritée. Les méthodes redéfinit ne seront pas héritées	Les éléments déclarés avec Shadows dans la classe dérivée seront hérités, les éléments occultés de la classe de base ne seront pas hérités

[7] Ce tableau est largement inspiré de l'article MSDN « Differences Between Shadowing and Overriding » que vous trouverez à l'adresse suivante :

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

http://msdn2.microsoft.com/en-us/library/ms172785.aspx

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

VIII - Glossaire des mots clés

Inherits : Mot clé utilisé pour spécifier la classe de base à partir de laquelle la classe en cours va être dérivée. Si ce mot clé n'est pas utilisé, la classe en cours héritera de la classe System.Object

MustInherit : Mot clé utilisé avant la déclaration d'une classe pour indiquer que cette classe ne peut pas être instanciée, elle sert seulement à être héritée par d'autres classes.

MustOverride : Mot clé utilisé avant la déclaration d'une méthode pour indiquer que celle-ci doit être redéfinie dans toutes les classes dérivées de la classe en cours. Les classes qui possèdent au moins une méthode avec MustOverride doivent être déclarées avec le mot clé MustInherit.

NotInheritable : Mot clé précédant la déclaration d'une classe qui indique que l'on ne peut pas créer des classes dérivées à partir de cette classe.

NotOverridable : Mot clé utilisé avant la déclaration d'une méthode pour indiquer que celle-ci ne peut pas être redéfinie dans les classes dérivées de la classe en cours. Par défaut, les méthodes ne sont pas virtuelles, ce mot clé n'est utile que pour les méthodes déclarées avec Overrides pour qu'elles cessent d'être virtuelles dans les classes dérivées.

Overloads : Mot clé utilisé avant la déclaration d'une méthode pour indiquer que celle-ci surcharge les méthodes ayant le même nom dans la classe de base (ou la classe en cours quoique optionnel dans ce cas).

Overridable : Mot clé utilisé avant la déclaration d'une méthode pour indiquer que celle-ci est virtuelle et donc peut être redéfinie dans les classes dérivées.

Overrides : Mot clé utilisé avant la déclaration d'une méthode pour indiquer qu'elle redéfinit la méthode portant le même nom et ayant la même signature dans la classe de base. Les méthodes déclarées avec ce mot clé sont automatiquement virtuelles.

Shadows : Mot clé utilisé avant la déclaration d'un membre pour occulter tous les membres de la classe de base ayant le même nom.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

IX - Bibliographie

Article MSDN : "Shadowing in Visual Basic" **http://msdn2.microsoft.com/en-us/library/c4swkw24.aspx**

Article MSDN : "Differences Between Shadowing and Overriding" **http://msdn2.microsoft.com/en-us/library/ms172785.aspx**

Livre : "Manuel de référence Microsoft Visual Basic .NET" de Francesco Balena, éditions Microsoft press.

Vos remarques et suggestions sont les bienvenues

L'objectif de ce tutorial est de rendre les concepts abordés clairs pour vous. Si à ce stade, il y a toujours des choses confuses, c'est que ce tutorial a vraiment besoin d'être amélioré. Merci d'envoyer vos remarques et suggestion à l'adresse suivante : ***ebenzeghiba@halkorb.com***

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.