# Programmation orientée objet dans Visual Basic

Les objets jouent un rôle central dans la programmation Visual Basic. Les formulaires et les contrôles sont des objets, ainsi que les bases de données. Si vous avez utilisé un peu Visual Basic ou si vous avez utilisé les exemples de ce fichier d'aide, vous avez déjà programmé des objets. Mais vous pouvez utiliser les objets différemment de ce que vous avez déjà appris jusqu'ici.

Dans les rubriques suivantes, vous pourrez constater qu'il est facile de créer vos objets personnels depuis les classes que vous définissez. Vous verrez également comment utiliser des objets pour simplifier le codage et augmenter la réutilisation du code.

**Dans cette section**

Introduction aux objets dans Visual Basic
: Fournit une introduction aux termes et aux concepts utilisés dans la programmation orientée objet.

Liaison anticipée et liaison tardive
: Décrit la liaison effectuée par le compilateur lorsqu'un objet est assigné à une variable objet et les différences entre des objets à liaison anticipée et à liaison tardive.

Membres partagés
: Décrit les membres pouvant être utilisés sans instances.

Création et utilisation d'objets
: Explique comment créer et utiliser des instances de classes.

Gestion de groupes d'objets
: Fournit des techniques de travail sur les tableaux et les collections d'objets

Obtention d'informations sur la classe au moment de l'exécution
: Explique comment déterminer la classe à laquelle appartient un objet.

Présentation des classes
: Présente des instructions détaillées sur la création d'objet et les problèmes de durée de vie.

Evénements et délégués
: Illustre la déclaration et l'utilisation des événements et des délégués.

Interfaces dans Visual Basic .NET
: Définit les interfaces et présente leur utilisation dans vos applications.

Héritage

Explique comment définir des classes servant de base à d'autres classes.

**Rubriques connexes**

~~Objets~~ Définit les objets et présente leur utilisation.

~~Programmation à l'aide de composants~~ :Explique comment utiliser Visual Basic pour contrôler des objets fournis par d'autres applications.

# Introduction aux objets dans Visual Basic

**Voir aussi**

Objets | Création et utilisation d'objets | Classes : modèles d'objets | Héritage | Propriétés et méthodes surchargées | Substitution de propriétés et de méthodes | Occultation

La plupart des opérations effectuées dans Visual Basic impliquent des objets. Si vous n'êtes pas familiarisé avec la programmation orientée objet, les termes et les concepts suivants vont vous aider à démarrer.

## Classes et objets

Les termes « classe » et « objet » sont tellement utilisés dans la programmation orientée objet qu'ils peuvent être facilement confondus. En règle générale, une classe est une représentation abstraite de quelque chose, tandis qu'un objet est un exemple utilisable de ce que représente la classe. La seule exception à cette règle concerne les membres de classe partagés, qui peuvent être utilisés dans les instances d'une classe et dans les variables d'objet déclarées comme le type de la classe.

## Champs, propriétés, méthodes et événements

Les classes se composent de champs, de propriétés, de méthodes et d'événements. Les propriétés et les champs sont des informations contenues dans un objet. Les champs sont similaires aux variables car ils peuvent être lus ou définis directement. Par exemple, si vous possédez un objet intitulé « Car », vous pouvez stocker sa couleur dans un champ nommé « Color ».

Les propriétés sont extraites et définies comme des champs mais sont implémentées via les procédures **Property Get** et **Property Set**, qui contrôlent davantage la façon dont les valeurs sont définies ou retournées. La couche d'adressage entre la valeur stockée et les procédures qui utilisent cette valeur vous aide à isoler vos données et vous permet de valider les valeurs avant qu'elles soient attribuées ou extraites.

Les méthodes représentent des actions que peut effectuer un objet. Par exemple, un objet « Car » peut avoir des méthodes « StartEngine », « Drive » et

« Stop ». Pour définir des méthodes, ajoutez des procédures (des fonctions ou des routines **Sub**) à votre classe.

Les événements vous informent lorsqu'un objet est reçu d'autres applications ou transmis à celles-ci. Les événements permettent aux objets d'effectuer des actions lorsqu'une occurrence spécifique se produit. Un exemple d'un événement de la classe « Car » est un événement « Check_Engine ». Dans la mesure où Microsoft Windows est un système d'exploitation piloté par événements, les événements peuvent provenir d'autres objets, applications ou actions de l'utilisateur (par exemple, lorsqu'il clique sur la souris ou appuie sur une touche).

## Encapsulation, héritage et polymorphisme

Les champs, les propriétés, les méthodes et les événements ne représentent que la moitié de l'équation de la programmation orientée objet. La programmation orientée objet réelle nécessite que les objets prennent en charge trois qualités : l'encapsulation, l'héritage et le polymorphisme.

L'encapsulation signifie qu'un groupe de propriétés, méthodes et autres membres liés est traité comme une unité ou un objet unique. Les objets peuvent contrôler la façon dont les propriétés sont modifiées ou la façon dont les méthodes sont exécutées. Par exemple, un objet peut valider des valeurs avant d'autoriser des modifications de propriété. L'encapsulation facilite également la modification ultérieure de votre implémentation en vous permettant de masquer les détails d'implémentation de vos objets ; cette méthode est connue sous le nom de masquage des données.

L'héritage décrit la possibilité de créer de nouvelles classes en fonction d'une classe existante. La nouvelle classe hérite de toutes les propriétés, des méthodes et des événements de la classe de base et peut être personnalisée avec des propriétés et des méthodes supplémentaires. Par exemple, vous pouvez créer une nouvelle classe intitulée « Truck » basée sur la classe « Car ». La classe « Truck » hérite de la propriété « Color » de la classe « Car » et peut comporter des propriétés supplémentaires telles que « FourWheelDrive ».

Le polymorphisme signifie que plusieurs classes peuvent être utilisées de manière interchangeable, même si chacune des classes implémente les mêmes propriétés et méthodes de manière différente. Le polymorphisme est primordial dans la programmation orientée objet car il vous permet d'utiliser des éléments portant le même nom, quel que soit le type d'objet utilisé actuellement. Pour une classe de base « Car » par exemple, le polymorphisme permet au programmeur de définir plusieurs méthodes « StartEngine » pour tout nombre de classes

dérivées. La méthode « StartEngine » d'une classe dérivée appelée « DieselCar » peut être totalement différente de la méthode portant le même nom dans la classe de base. D'autres procédures ou méthodes peuvent utiliser la méthode « StartEngine » des classes dérivées exactement de la même façon, quel que soit le type d'objet « Car » utilisé simultanément.

### Surcharge, substitution et occultation

La surcharge, la substitution et l'occultation sont des concepts similaires pouvant être facilement confondus. Même si ces trois techniques vous permettent de créer des membres portant le même nom, elles présentent d'importantes différences.

- Les membres surchargés sont utilisés pour fournir des versions différentes d'une propriété ou d'une méthode portant le même nom, mais qui acceptent divers nombres ou paramètres ou des paramètres de divers types de données.
- Les propriétés ou les méthodes substituées permettent de remplacer une propriété ou une méthode héritée qui n'est pas adaptée dans une classe dérivée. Les membres substitués doivent accepter le même type de données ainsi que le même nombre d'arguments. Les classes dérivées héritent des membres substitués.
- Les membres occultés permettent de remplacer localement un membre dont la portée est plus vaste. Tout type peut occulter un autre type. Par exemple, vous pouvez déclarer une propriété qui occulte une méthode héritée du même nom. Les membres occultés ne peuvent pas être hérités.

# Liaison anticipée et liaison tardive

Le compilateur Visual Basic exécute un processus appelé liaison quand un objet est assigné à une variable objet. Un objet est à liaison anticipée lorsque cet objet est assigné à une variable déclarée comme étant d'un type d'objet spécifique. Les objets à liaison anticipée permettent au compilateur d'allouer de la mémoire et d'effectuer d'autres optimisations avant l'exécution d'une application. Par exemple, le fragment de code suivant déclare une variable comme étant du type **FileStream** :

```
' Add Imports statements to the top of your file.
Imports System.IO
'...
'  Create a variable to hold a new object.
   Dim FS As FileStream
  ' Assign a new object to the variable.
   FS = New FileStream("C:\tmp.txt", FileMode.Open)
```

Etant donné que **FileStream** est un type d'objet spécifique, l'instance assignée à FS fait l'objet d'une liaison anticipée.

Par opposition, un objet est à liaison tardive lorsque cet objet est assigné à une variable déclarée comme étant du type **Object**. Les objets de ce type peuvent contenir des références à n'importe quel objet, mais ne présentent pas nombre des avantages des objets à liaison anticipée. Par exemple, le fragment de code suivant déclare une variable objet qui doit contenir un objet retourné par la fonction **CreateObject** :

```
' To use this example, you must have Microsoft Excel installed on your computer.
Option Strict Off ' Option Strict Off allows late binding.
...
Sub TestLateBinding()
   Dim xlApp As Object
   Dim xlBook As Object
   Dim xlSheet As Object
   xlApp = CreateObject("Excel.Application")
   'Late bind an instance of an Excel workbook.
   xlBook = xlApp.Workbooks.Add
   'Late bind an instance of an Excel worksheet.
   xlSheet = xlBook.Worksheets(1)
   xlSheet.Activate()
   xlSheet.Application.Visible = True ' Show the application.
   ' Place some text in the second row of the sheet.
   xlSheet.Cells(2, 2) = "This is column B row 2"
End Sub
```

Vous devez utiliser des objets à liaison anticipée chaque fois que cela est possible, car ils permettent au compilateur d'effectuer d'importantes optimisations, avec pour résultat des applications plus efficaces. Ils sont beaucoup plus rapides que les objets à liaison tardive et facilitent la lecture et la gestion du code en indiquant avec exactitude les types d'objets utilisés. La liaison anticipée présente également l'avantage d'activer des fonctionnalités utiles, telles que la saisie semi-automatique du code et l'Aide dynamique, car l'environnement de développement intégré (IDE, Integrated Development Environment) Visual Studio .NET peut déterminer avec précision le type d'objet utilisé lorsque vous modifiez le code. Elle réduit le nombre et la gravité des

erreurs d'exécution en permettant au compilateur de signaler les erreurs lors de la compilation d'un programme.

> **Remarque** La liaison tardive peut seulement être utilisée pour accéder aux membres de type déclarés comme **Public**. L'accès aux membres déclarés comme **Friend** ou **Protected Friend** entraîne une erreur d'exécution.

# Object, type de données

Les variables **Object** sont stockées sous la forme d'adresses 32 bits (4 octets) qui font référence à des objets. Vous pouvez assigner un type référence (chaîne, tableau, classe ou interface) à une variable déclarée comme un **Object**.

Une variable **Object** peut également faire référence à des données de tout type valeur (numérique, **Boolean**, **Char**, **Date**, structure ou énumération).

> **Remarque** Même si une variable déclarée avec le type **Object** est suffisamment flexible pour contenir une référence à un objet, l'appel à une méthode d'une instance utilisant une variable **Object** est toujours à liaison tardive (au moment de l'exécution). Pour forcer une liaison anticipée (au moment de la compilation), assignez la référence d'objet à une variable déclarée avec un nom de classe spécifique ou castée en un type de données spécifique.

Le type de données .NET équivalent est **System.Object**.

# CreateObject, fonction

Crée et retourne une référence à un objet COM. La fonction **CreateObject** ne peut être utilisée pour créer des instances de classes dans Visual Basic que si ces classes sont exposées explicitement en tant que composants COM.

```
Public Shared Function CreateObject( _
   ByVal ProgId As String, _
   Optional ByVal ServerName As String = "" _
) As Object
```

**Paramètres**

ProgId
: Requis. **String**. L'ID de programme de l'objet à créer.

ServerName
: Facultatif. **String**. Nom du serveur réseau où l'objet sera créé. Si ServerName est une chaîne vide (""), l'ordinateur local est utilisé.

**Exceptions/Erreurs**

| Type d'exception | Numéro de l'erreur | Condition |
|---|---|---|
| Exception | 429 | ProgId est introuvable ou non spécifié. |
| Exception | 462 | Le serveur est indisponible. |
| FileNotFoundException | 53 | Aucun objet du type spécifié n'existe. |

**Notes**

Pour créer une instance d'un composant COM, assignez l'objet retourné par la fonction **CreateObject** à une variable objet :

```
xlApp = CreateObject("Excel.Application")
```

Le type de la variable objet que vous utilisez pour stocker l'objet retourné peut affecter les performances de vos applications. La déclaration d'une variable objet avec la clause **As Object** crée une variable pouvant contenir une référence à n'importe quel type d'objet. Cependant, l'accès à l'objet par l'intermédiaire de cette variable est effectué par une liaison tardive, c'est-à-dire que la liaison est créée lors de l'exécution de votre programme. De nombreuses raisons militent contre la liaison tardive, parmi lesquelles le ralentissement de l'exécution de l'application. Pour créer une variable objet qui entraîne une liaison anticipée, c'est-à-dire une liaison qui survient au moment de la compilation du programme, ajoutez une référence à la bibliothèque de types de votre objet à partir de l'onglet **COM** de la boîte de dialogue **Ajouter une référence** dans le menu **Projet** et déclarez la variable objet avec le type spécifique de votre objet. Par exemple, vous pouvez déclarer et créer les références Microsoft Excel suivantes :

```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.WorkSheet
xlApp = CreateObject("Excel.Application")
```

La référence par une variable à liaison anticipée peut offrir de meilleures performances, mais ne peut contenir qu'une référence à la classe spécifiée dans la déclaration.

Un autre problème est que les objets COM utilisent un code non managé, un code sans l'avantage du Common Language Runtime. Le mélange de code managé Visual Basic .NET et de code non managé COM implique un certain degré de complexité. Quand vous ajoutez une référence à un objet COM, une recherche est effectuée pour un assembly d'interopérabilité prédéfini pour cette bibliothèque ; si un assembly est trouvé, il est alors utilisé. Si aucun assembly n'est trouvé, vous avez la possibilité de créer un assembly d'interopérabilité local qui contient les classes d'interopérabilité locales pour chaque classe de la bibliothèque COM.

Vous pouvez créer un objet sur un ordinateur réseau distant en passant le nom de l'ordinateur à l'argument ServerName de la fonction **CreateObject**. Ce nom est identique à une partie du nom de l'ordinateur partagé : pour un nom de partage « \\MyServer\Public », ServerName est « MyServer ».

> **Remarque** Consultez la documentation COM (voir Microsoft Developer Network) pour plus d'informations sur la manière de rendre visible une application à partir d'un ordinateur connecté à distance au réseau. Il sera nécessaire, le cas échéant, d'ajouter une clé de Registre pour votre application.

Le code suivant retourne le numéro de version d'une instance d'Excel s'exécutant sur un ordinateur distant appelé MyServer :

```
Dim xlApp As Object
' Replace string "\\MyServer" with name of the remote computer.
  xlApp = CreateObject("Excel.Application", "\\MyServer")
  MsgBox(xlApp.Version)
```

Si le nom du serveur distant est incorrect, ou si le serveur n'est pas disponible, une erreur se produit au moment de l'exécution.

> **Remarque** Utilisez **CreateObject** lorsqu'il n'existe aucune instance en cours de l'objet. S'il en existe une, une nouvelle instance est lancée et un objet du type spécifié est créé. Pour utiliser l'instance en cours ou pour lancer l'application en chargeant un fichier, utilisez la fonction **GetObject**. Si un objet s'est inscrit comme objet à instance unique, une seule instance de l'objet est créée, quel que soit le nombre d'exécutions de la fonction **CreateObject**.

### Exemple

Cet exemple utilise la fonction **CreateObject** pour définir une référence (xlApp) à Microsoft Excel. Il utilise la référence pour accéder à la propriété **Visible** de Microsoft Excel, puis utilise la méthode **Quit** de Microsoft Excel pour fermer cette application. Enfin, la référence elle-même est libérée.

```
Sub TestExcel()
  Dim xlApp As Excel.Application
  Dim xlBook As Excel.Workbook
  Dim xlSheet As Excel.Worksheet
  xlApp = CType(CreateObject("Excel.Application"), Excel.Application)
  xlBook = CType(xlApp.Workbooks.Add, Excel.Workbook)
  xlSheet = CType(xlBook.Worksheets(1), Excel.Worksheet)
  ' Place some text in the second row of the sheet.
  xlSheet.Activate()
  xlSheet.Cells(2, 2) = "This is column B row 2"
  ' Show the sheet.
  xlSheet.Application.Visible = True
  ' Save the sheet to C:\test.xls directory.
  xlSheet.SaveAs("C:\Test.xls")
  ' Optionally, you can call xlApp.Quit to close the work sheet.
End Sub
```

# GetObject, fonction

Retourne une référence à un objet fourni par un composant COM.

```
Public Function GetObject( _
   Optional ByVal PathName As String = Nothing, _
   Optional ByVal Class As String = Nothing _
) As Object
```

### Paramètres

PathName

Facultatif ; **String**. Chemin d'accès complet et nom du fichier contenant l'objet à extraire. Si l'argument pathname est omis, l'argument class est requis.

Class

Facultatif ; **String**. Chaîne représentant la classe de l'objet.

L'argument Class emploie la syntaxe et les paramètres suivants :

```
appname.objecttype
```

**Paramètres Class**

appname
Requis ; **String**. Nom de l'application qui fournit l'objet.
objecttype
Requis ; **String**. Type ou classe de l'objet à créer.

**Exceptions/Erreurs**

| Type d'exception | Numéro de l'erreur | Condition |
|---|---|---|
| Exception | 429 | Aucun objet avec le chemin d'accès spécifié n'existe. |
| FileNotFoundException | 432 | Aucun objet du type spécifié n'existe. |

**Notes**

Utilisez la fonction **GetObject** pour charger une instance d'un composant COM à partir d'un fichier. Exemple :

```
Dim CADObject As Object
CADObject = GetObject("C:\CAD\SCHEMA.CAD")
```

Lorsque ce code est exécuté, l'application associée à l'argument pathname spécifié est lancée et l'objet contenu dans le fichier indiqué est activé.

Si PathName est une chaîne de longueur nulle (""), la fonction **GetObject** retourne une nouvelle instance d'objet du type spécifié. Si l'argument PathName est omis, **GetObject** retourne un objet actif du type spécifié. S'il n'existe aucun objet du type spécifié, une erreur se produit.

Certaines applications permettent d'activer un sous-objet associé à un fichier. Il suffit d'ajouter un point d'exclamation (**!**) à la fin du nom de fichier et d'indiquer à la suite une chaîne identifiant la partie du fichier à activer. Pour plus d'informations sur la façon de créer cette chaîne, consultez la documentation de l'application qui a créé l'objet.

Dans une application de dessin, par exemple, un fichier peut contenir un dessin constitué de plusieurs couches. Le code suivant vous permet d'activer un plan d'un dessin nommé SCHEMA.CAD :

```
LayerObject = GetObject("C:\CAD\SCHEMA.CAD!Layer3")
```

Si vous ne spécifiez pas l'argument Class de l'objet, Automation se base sur le nom de fichier que vous fournissez pour déterminer l'application à démarrer et l'objet à activer. Cependant, certains fichiers peuvent prendre en charge plusieurs classes d'objets. Par exemple, un dessin peut comporter un objet Application, un objet Drawing et un objet Toolbar, chacun faisant partie du même fichier. Pour spécifier l'objet à activer parmi ceux du fichier, utilisez l'argument facultatif Class. Exemple :

```
Dim MyObject As Object
MyObject = GetObject("C:\DRAWINGS\SAMPLE.DRW", "FIGMENT.DRAWING")
```

Dans l'exemple, FIGMENT est le nom d'une application de dessin et DRAWING est l'un des types d'objets pris en charge par cette application.

Une fois l'objet activé, vous le référencez dans le code à l'aide de la variable objet que vous avez définie. Dans l'exemple précédent, vous accédez aux propriétés et aux méthodes du nouvel objet en utilisant la variable objet MyObject. Exemple :

```
MyObject.Line (9, 90)
MyObject.InsertText (9, 100, "Hello, world.")
MyObject.SaveAs ("C:\DRAWINGS\SAMPLE.DRW")
```

> **Remarque** Utilisez la fonction **GetObject** lorsqu'il existe une instance en cours de l'objet, ou si vous souhaitez créer l'objet avec un fichier déjà chargé. S'il n'existe aucune instance en cours et si vous ne souhaitez pas démarrer l'objet en chargeant un fichier, utilisez la fonction **CreateObject**. Si un objet s'est inscrit comme objet à instance unique Active-X, une seule instance de l'objet est créée quel que soit le nombre d'exécutions de la fonction **CreateObject**. Avec un objet à instance unique, la fonction **GetObject** retourne toujours la même instance lorsqu'elle est appelée à l'aide de la syntaxe de chaîne de longueur nulle (""). En outre, elle produit une erreur si l'argument PathName est omis. Vous ne pouvez pas utiliser **GetObject** pour obtenir une référence à une classe créée à l'aide de Visual Basic.

**Exemple**

Cet exemple utilise la fonction **CreateObject** pour obtenir une référence à une feuille de calcul Microsoft Excel spécifique (MyXL). Il utilise la propriété Application de la feuille de calcul pour rendre Microsoft Excel visible, pour fermer l'application, etc. En utilisant deux appels API, la procédure Sub

DetectExcel recherche Microsoft Excel et, si ce programme est en cours d'exécution, l'entre dans le tableau des objets en exécution. Le premier appel à la fonction **GetObject** génère une erreur si Microsoft Excel n'est pas déjà en exécution. Dans notre exemple, l'erreur a pour conséquence d'attribuer la valeur True à l'indicateur ExcelWasNotRunning. Le deuxième appel à la fonction **GetObject** indique le fichier à ouvrir. Si Microsoft Excel n'est pas déjà en exécution, le deuxième appel lance l'application et retourne une référence à la feuille de calcul représentée par le fichier spécifié test.xls. Ce fichier doit se trouver à l'emplacement spécifié ; dans le cas contraire, l'erreur Erreur Automation Visual Basic est générée. L'exemple de code rend ensuite Microsoft Excel et la fenêtre contenant la feuille de calcul spécifiée visibles.

**Option Strict Off** est requis car cet exemple utilise une liaison tardive au cours de laquelle des objets sont assignés à des variables objet génériques. Vous pouvez spécifier **Option Strict On** et déclarer des objets de types d'objet spécifiques si vous ajoutez une référence à la bibliothèque de types Excel à partir de l'onglet **COM** de la boîte de dialogue **Ajouter une référence** dans le menu **Projet** de Visual Studio .NET.

```
' Add Option Strict Off to the top of your program.
Option Strict Off
 ' Declare necessary API routines:
  Declare Function FindWindow Lib "user32" Alias _
    "FindWindowA" (ByVal lpClassName As String, _
    ByVal lpWindowName As Long) As Long

  Declare Function SendMessage Lib "user32" Alias _
  "SendMessageA" (ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long

Sub GetExcel()
  Dim MyXL As Object   ' Variable to hold reference
  ' to Microsoft Excel.
  Dim ExcelWasNotRunning As Boolean   ' Flag for final release.

  ' Test to see if there is a copy of Microsoft Excel already running.
  On Error Resume Next   ' Defer error trapping.
  ' Getobject function called without the first argument returns a
  ' reference to an instance of the application.
  ' If the application is not running, an error occurs.
  MyXL = GetObject(, "Excel.Application")
  If Err().Number <> 0 Then ExcelWasNotRunning = True
  Err().Clear() ' Clear Err object in case error occurred.

  ' Check for Microsoft Excel. If Microsoft Excel is running,
  ' enter it into the Running Object table.
```

```
  DetectExcel()

  ' Set the object variable to reference the file you want to see.
  MyXL = GetObject("c:\vb\TEST.XLS")

  ' Show Microsoft Excel through its Application property. Then
  ' show the actual window containing the file using the Windows
  ' collection of the MyXL object reference.
  MyXL.Application.Visible = True
  MyXL.Parent.Windows(1).Visible = True
  '  Do manipulations of your  file here.
  '
End Sub

Sub DetectExcel()
  ' Procedure dectects a running Excel and registers it.
  Const WM_USER = 1024
  Dim hWnd As Long
  ' If Excel is running this API call returns its handle.
  hWnd = FindWindow("XLMAIN", 0)
  If hWnd = 0 Then   ' 0 means Excel not running.
    Exit Sub
  Else
    ' Excel is running so use the SendMessage API
    ' function to enter it in the Running Object Table.
    SendMessage(hWnd, WM_USER + 18, 0, 0)
  End If
End Sub
```

Quand vous appelez la fonction GetExcel, un contrôle est effectué pour déterminer si Excel est déjà en exécution. Dans le cas contraire, une instance d'Excel est créée.

# Membres partagés

Les membres partagés sont des propriétés, des procédures et des champs qui sont partagés par toutes les instances d'une classe. Ils sont appelés membres statiques dans certains langages de programmation.

Les champs et propriétés partagés sont utiles lorsque certaines informations font partie d'une classe mais ne sont pas spécifiques à une instance particulière d'une classe. Des champs et propriétés normaux existent indépendamment pour chaque instance d'une classe. Modifier la valeur d'un champ ou d'une propriété associés à une instance particulière n'a aucune incidence sur la valeur des champs ou propriétés d'autres instances de la classe. Par contre, si vous modifiez la valeur d'un champ et d'une propriété partagés associés à une instance d'une classe, vous modifiez également la valeur associée à toutes les instances de la classe. De cette façon, les champs et propriétés partagés se comportent comme des variables globales accessibles uniquement à partir d'instances d'une classe. Sans champs et propriétés statiques, vous devriez recourir à des variables de module pour obtenir le même résultat. Cependant, l'emploi de variables de module peut rendre les classes difficiles à comprendre et à gérer. De plus, utiliser ces variables de cette façon enfreint le concept d'encapsulation représenté par les classes.

Les procédures partagées sont des méthodes de classe qui ne sont pas associées à une instance spécifique d'une classe. Par exemple, la méthode **Cos** définie dans la class **Math** est une méthode partagée. Il est possible d'appeler une procédure partagée en l'appelant en tant que méthode d'un objet ou directement à partir de la classe. Dès lors, les procédures partagées constituent une exception à la règle selon laquelle vous devez créer une instance d'une classe avant de pouvoir l'utiliser.

Des instances de la classe ne sont pas passées implicitement aux procédures partagées. Pour cette raison, les méthodes partagées ne peuvent pas contenir de référence non qualifiée à des données membres non partagées.

## Membres partagés - exemple

L'exemple suivant crée une instance de champ, un champ partagé et une méthode partagée pour illustrer le fonctionnement des membres partagés dans le code :

```
Public Class ShareClass
```

```
  Public InstanceValue As String
  Public Shared SharedValue As String
  Public Shared Sub ShareMethod()
    MsgBox("This is a shared method.")
  End Sub
End Class
Sub TestShared()
  Dim Shared1 As New ShareClass()   ' Create an instance of the class.
  Dim Shared2 As New ShareClass()   ' Create an instance of the class.
  Shared1.SharedValue = "Share Value 1"    ' Set the value of a shared field.
  Shared2.SharedValue = "Share Value 2"   ' Overwrite the first value.
  MsgBox("The value of the shared field in the first instance" & _
    "is: " & Shared1.SharedValue)
  MsgBox("The value of the shared field in the second instance" & _
    "is: " & Shared2.SharedValue)
  ' Call a method on the class without creating an instance.
  ShareClass.ShareMethod()
End Sub
```

Lorsque vous exécutez la procédure TestShared, deux instances de la classe sont créées, et le champ partagé SharedValue est modifié pour les deux instances. Quand le champ partagé de la deuxième instance de la classe est modifié, la valeur assignée au champ partagé de la première instance de la classe est remplacée car les deux instances font référence au même champ.

Par souci de clarté, cet exemple utilise des variables objet pour accéder aux membres partagés, mais il est préférable en programmation d'accéder aux membres partagés directement au moyen du nom de classe ; par exemple, ShareClass.SharedValue = "Share Value 2". L'utilisation de la syntaxe des classes pour les membres partagés réduit la confusion, car elle permet de distinguer clairement là où des membres partagés sont employés et là où des instances membres le sont.

# Création et utilisation d'objets

Les objets sont des combinaisons de code et de données qui peuvent être traitées comme des entités uniques. Quasiment toutes vos actions dans Visual Basic .NET reposent sur des objets, qui vont des contrôles que vous utilisez aux formulaires dans lesquels vous les insérez. Cette section traite de l'origine des objets et explique comment les utiliser.

**Dans cette section**

Objets issus de Visual Basic et d'autres sources
: Explique comment utiliser des objets à partir de Microsoft Word, Microsoft Excel et d'autres applications.

Définition et récupération de propriétés
: Explique comment modifier et obtenir des valeurs de propriétés.

Exécution d'actions à l'aide de méthodes
: Explique ce que sont les méthodes et comment les utiliser.

Exécution de plusieurs actions sur un objet
: Traite de l'utilisation de l'instruction **With...End With** pour simplifier l'accès à une propriété.

Traitement des formulaires en tant qu'objets
: Montre comment utiliser les formulaires comme des objets ordinaires.

Utilisation du mot clé New
: Montre comment créer des instances de formulaires, des classes définies dans des modules de classe et des collections.

Gestion des ressources
: Explique comment diminuer les besoins en mémoire et en ressources système grâce à la libération des références aux objets.

Passage d'objets aux procédures
: Explique comment passer des objets sous forme d'arguments aux procédures.

# Objets issus de Visual Basic et d'autres sources

Les objets utilisés dans Visual Basic .NET sont issus de sources internes et externes. Des exemples d'objets internes comprennent des objets intrinsèques et des classes de votre projet ; des exemples d'objets externes comprennent des assemblys et des objets COM.

## Objets internes

Les objets intrinsèques (ou intégrés) sont des objets fournis par Visual Basic .NET. Ils comprennent les types scalaires primitifs tels que **Integer** et **Double** ainsi que les types **Array** et **String**. Vous ne devez pas créer de références à des objets internes avant de les utiliser dans votre projet.

D'autres objets internes sont des instances de classes dans votre projet en cours. Vous pouvez utiliser ces classes dès que vous en avez besoin dans votre projet et les rendre disponibles à d'autres applications lorsque vous créez un assembly.

## Objets externes

Les objets externes sont issus d'autres projets ou assemblys qui ne sont pas disponibles par défaut dans votre projet. Vous devez créer des références de projet à des objets externes avant de les utiliser dans votre projet.

Les assemblys sont la source la plus ordinaire des objets pour les applications Visual Basic .NET. Le .NET Framework comprend des assemblys qui contiennent les objets les plus fréquemment utilisés. Certains objets du .NET Framework sont traités comme des objets intrinsèques, mais la plupart des assemblys doivent être importés explicitement via l'instruction **Imports** avant de pouvoir être utilisés. Tout langage conforme CLS (Common Language Specification), tel que Visual Basic .NET ou Visual C#, peut créer et utiliser des assemblys. Pour plus de détails, consultez Assemblys.

Auparavant, les composants COM constituaient une source traditionnelle d'objets pour les programmeurs Visual Basic, mais à l'heure actuelle, les

assemblys .NET sont mieux adaptés aux nouveaux objets. Vous pouvez toujours utiliser des composants COM existants dans vos applications mais vous devez accéder aux objets COM via les classes d'interopérabilité .NET. L'accès à une bibliothèque COM exige l'utilisation d'un assembly d'interopérabilité contenant des classes d'interopérabilité pour chacune des classes COM définies dans la bibliothèque COM.

Outre l'accès aux classes .NET et COM natives, il est également possible d'appeler des fonctions définies dans des bibliothèques de liaisons dynamiques (DLL, Dynamic-Link Libraries). Visual Basic .NET vous permet d'appeler des fonctions dans des DLL après qu'elles ont été déclarées par une instruction **Declare**. Visual Basic .NET ajoute des fonctionnalités à l'instruction **Declare** en vous permettant d'utiliser l'attribut **DllImport** pour spécifier des valeurs par défaut pour **CallingConvention**, **ExactSpelling** et **SetLastError**, par exemple. Les paramètres des instructions **Declare** peuvent contenir l'attribut **MarshalAs**, qui prend en charge la conversion de paramètres de façon différente par rapport aux versions antérieures de Visual Basic.

# Définition et récupération de propriétés

Lorsque vous utilisez des formulaires et des contrôles dans Visual Basic .NET, vous pouvez définir leurs propriétés par programme au moment de l'exécution ou en mode design à l'aide de la fenêtre **Propriétés**. Les propriétés de la plupart des autres objets, issus par exemple d'un assembly ou d'un objet, ne peuvent être définies que par programme.

Les propriétés que vous pouvez définir et lire sont dites en lecture-écriture. Les propriétés que vous pouvez lire mais pas modifier sont dites en lecture seule. Les propriétés que vous pouvez écrire mais pas lire sont dites en écriture seule.

Vous définissez la valeur d'une propriété lorsque vous voulez modifier le comportement ou l'aspect d'un objet. Par exemple, vous modifiez la propriété **Text** d'un contrôle de zone de texte pour modifier le contenu de cette zone.

Vous extrayez la valeur d'une propriété lorsque vous voulez connaître l'état d'un objet avant que le code n'accomplisse d'autres actions telles qu'assigner la valeur à un autre objet. Par exemple, vous pouvez retourner la propriété **Text** d'un contrôle de zone de texte pour déterminer le contenu de celle-ci avant d'exécuter un code susceptible de modifier sa valeur.

**Pour définir la valeur des propriétés**

- Utilisez la syntaxe suivante :

  object.property = expression

Les exemples d'instructions suivants montrent comment définir les propriétés :

```
TextBox1.Top = 200   ' Sets the Top property to 200 twips.
TextBox1.Visible = True   ' Displays the text box.
TextBox1.Text = "hello"   ' Displays 'hello' in the text box.
```

> **Remarque** Vous pouvez également définir une propriété en la passant à des paramètres **ByRef**, auquel cas elle est modifiée par le résultat retourné par le paramètre **ByRef**.

**Pour extraire les valeurs de propriétés**

- Utilisez la syntaxe suivante :

  variable = object.property

Vous pouvez également extraire la valeur d'une propriété dans une expression plus complexe sans assigner la propriété à une variable. Le code suivant modifie la propriété **Top** d'un contrôle de case d'option :

```
Protected Sub RadioButton1_CheckedChanged(ByVal sender As Object, _
    ByVal e As System.EventArgs)
  ' [statements]
  RadioButton1.Top += 20
  ' [statements]
End Sub
```

# Exécution d'actions à l'aide de méthodes

Les méthodes sont des procédures associées à des objets. A la différence des champs et des propriétés, qui représentent les informations que peut enregistrer un objet, les méthodes représentent les actions qu'un objet peut accomplir. Les méthodes peuvent affecter la valeur des propriétés. Par exemple, pour faire une analogie avec un poste de radio, vous pouvez utiliser une méthode SetVolume pour modifier la valeur d'une propriété Volume. Pareillement, les éléments des zones de liste, dans Visual Basic .NET, possèdent une propriété **List** que vous pouvez modifier à l'aide des méthodes **Clear** et **Add**.

Lorsque vous utilisez un argument dans le code, vous écrivez l'instruction en tenant compte du nombre d'arguments requis par la méthode et du fait qu'elle retourne ou non une valeur. En général, vous utilisez les méthodes comme vous utilisez des sous-routines ou des appels de fonction. Mais vous pouvez aussi appeler des méthodes comme vous appelez des procédures de module, à cette exception près que vous pouvez qualifier les méthodes avec une expression spécifiant l'instance d'objet dont la méthode doit être appelée. En l'absence de qualification, la variable **Me** est implicitement utilisée comme instance.

**Pour utiliser une méthode ne requérant pas d'arguments**

- Utilisez la syntaxe suivante :

  object.method()

Dans l'exemple suivant, la méthode **Refresh** repeint la zone d'image :

```
PictureBox1.Refresh()  ' Forces a repaint of the control
```

> **Remarque** Certaines méthodes, par exemple **Refresh**, n'ont pas d'arguments et ne retournent pas de valeurs.

**Pour utiliser une méthode requérant plusieurs arguments**

- Placez les arguments entre parenthèses en les séparant par des virgules. Dans l'exemple suivant, la méthode **MsgBox** utilise des arguments spécifiant le message à afficher et le style du message :
- MsgBox("Database update complete", _

```
    MsgBoxStyle.OKOnly Or MsgBoxStyle.Exclamation, _
    "My Application")
```

**Pour utiliser une méthode retournant une valeur**

- Assignez la valeur de retour à une variable ou utilisez directement l'appel de méthode comme un paramètre d'un autre appel. Le code suivant stocke la valeur de retour :

```
Dim Response As MsgBoxResult
Response = MsgBox("Do you want to exit?", _
            MessageBoxButtons.YesNo Or MsgBoxStyle.Question, _
            "My Application")
```

Cet exemple utilise la valeur retournée de la méthode Len comme un argument de MsgBox.

```
Dim MyStr As String = "Some String"
'   Displays "String length is : 11
MsgBox("String length is : " & Len(MyStr))
```

# Exécution de plusieurs actions sur un objet

Il arrive fréquemment d'avoir à exécuter plusieurs actions sur le même objet. Par exemple, vous pouvez avoir besoin de définir plusieurs propriétés ou d'exécuter plusieurs méthodes pour le même objet.

**Pour définir plusieurs propriétés pour le même objet**

- Vous pouvez écrire plusieurs instructions en vous servant de la même variable d'objet, comme dans le code suivant :

```
Private Sub UpdateForm()

  Button1.Text = "OK"

  Button1.Visible = True
```

```
  Button1.Top = 24

  Button1.Left = 100

  Button1.Enabled = True

  Button1.Refresh()

End Sub
```

Toutefois, vous pouvez faciliter l'écriture et la lecture de ce code en vous servant de l'instruction **With...End With**, comme dans le code suivant :

```
Private Sub UpdateForm2()

  With Button1

    .Text = "OK"

    .Visible = True

    .Top = 24

    .Left = 100

    .Enabled = True

    .Refresh()

  End With

End Sub
```

Vous pouvez également imbriquer des instructions **With...End With** les unes à l'intérieur des autres, comme dans le code suivant :

```
Sub SetupForm()
  Dim AnotherForm As New Form1()
  With AnotherForm
    .Show()  ' Show the new form.
    .Top = 250
    .Left = 250
    .ForeColor = Color.LightBlue
    .BackColor = Color.DarkBlue
    With AnotherForm.Textbox1
      .BackColor = Color.Thistle   ' Change the background.
      .Text = "Some Text"   ' Place some text in the text box.
    End With
  End With
```

```
End Sub
```

Toutefois, à l'intérieur de l'instruction **With**, la syntaxe fait référence à l'objet imbriqué ; les propriétés de l'objet de l'instruction **With** extérieure ne sont pas définies.

# Traitement des formulaires en tant qu'objets

Les formulaires sont les objets graphiques qui constituent l'interface utilisateur de vos applications. Dans Visual Basic, les classes définissent la manière dont les formulaires sont affichés et ce qu'ils peuvent faire. Lorsqu'un formulaire est affiché au moment de l'exécution, Visual Basic .NET crée une instance de la classe **Form** que vous pouvez utiliser comme n'importe quel autre objet. Vous pouvez ajouter des méthodes et des propriétés personnalisées à des formulaires et y accéder à partir d'autres formulaires ou d'autres classes de votre application.

**Pour créer une nouvelle méthode pour un formulaire**

- Ajoutez une procédure déclarée **Public**, comme dans le code suivant :

```
' Create a custom method on a form.
Public Sub PrintMyJob()
    '.Add code for your method here.
End Sub
```

**Pour ajouter un nouveau champ à un formulaire**

- Déclarez une variable publique dans le module de formulaire, comme dans le code suivant : Public IDNumber As Integer

**Pour accéder aux méthodes d'un autre formulaire**

. Créez une instance du formulaire aux méthodes duquel vous voulez accéder. Lorsque vous référencez un nom de formulaire, vous référencez en réalité la classe à laquelle appartient ce formulaire et non l'objet lui-même.

**Remarque** Visual Basic 6 fournissait, pour chaque classe, une variable globale implicite portant le même nom que la classe Form. Visual Basic .NET ne fournit pas de déclaration de variable implicite.

. Assignez le formulaire à une variable d'objet. La variable d'objet référence une nouvelle instance de la classe Form.

L'exemple suivant appelle correctement la procédure PrintMyJob :

```
Dim newForm1 As New Form1
newForm1.PrintMyJob
```

Dans l'exemple précédent, le nouveau formulaire n'est pas affiché. Il n'est pas nécessaire d'afficher un objet de formulaire pour utiliser ses méthodes. Pour afficher le nouveau formulaire, vous devez ajouter le code suivant :

```
newForm1.show
```

# Utilisation du mot clé New

Pour créer une instance d'une classe, utilisez le mot clé **New**. A la différence de **Integer** et **Double** qui sont des types valeur, les objets sont des types référence, et vous devez donc les créer explicitement avant de pouvoir les utiliser. Considérons par exemple les deux lignes de code suivantes :

```
Dim Button1 As System.Windows.Forms.Button()

Dim Button2 As New System.Windows.Forms.Button()
```

La première instruction déclare une variable d'objet qui peut contenir une référence à un objet Button. Toutefois, la variable Button1 contient la valeur **Nothing** jusqu'à ce que vous lui assigniez un objet de type **Button**. La seconde instruction définit également une variable pouvant contenir un objet Button, mais le mot clé **New** crée un objet Button et l'assigne à la variable Button2.

Les formulaires et les contrôles étant en réalité des classes, vous pouvez utiliser le mot clé **New** pour créer de nouvelles instances de ces éléments en fonction de vos besoins.

**Pour créer de nouvelles instances d'une classe à l'aide du mot clé New**

. Ouvrez un nouveau projet Windows Forms, puis placez un bouton de commande et plusieurs autres contrôles dans un formulaire appelé Form1.
. Ajoutez le code suivant à la procédure d'événement **Click** du bouton de commande :

```
. Dim f As New Form1
  f.Show
```

. Exécutez l'application et cliquez plusieurs fois sur le bouton de commande.
. Mettez de côté le formulaire de premier plan. Etant donné qu'un formulaire est une classe dotée d'une interface visible, vous pouvez voir les autres copies. Chacune de ces copies contient les mêmes contrôles et aux mêmes positions que ceux du formulaire d'origine au moment du design.

Vous pouvez utiliser le mot clé **New** pour créer des objets à partir des classes. La procédure suivante vous montre comment.

**Pour voir comment le mot clé New crée des instances d'une classe**

. Ouvrez un nouveau projet Windows Forms, puis placez un bouton de commande dans un formulaire appelé Form1.
. Dans le menu **Projet**, sélectionnez **Ajouter une classe** pour ajouter une classe au projet.
. Donnez à la classe le nom ShowMe.vb.
. Ajoutez la procédure suivante à ShowMe :

```
. Public Class ShowMe
.   Sub ShowFrm()
.     Dim frmNew As Form1
.     frmNew = New Form1
.     frmNew.Show()
.     frmNew.WindowState = 1
.   End Sub
  End Class
```

. Ajoutez le code suivant à l'événement **Button1_Click** du bouton de votre formulaire :

```
. Protected Sub Button1_Click(ByVal sender As System.Object, _
.     ByVal e As System.EventArgs) Handles Button1.Click
.   Dim clsNew As New ShowMe()
.   clsNew.ShowFrm()
  End Sub
```

n . Pour utiliser l'exemple, exécutez l'application et cliquez plusieurs fois sur le bouton de commande. Une icône de formulaire réduite apparaît dans

votre barre des tâches chaque fois qu'est créée une nouvelle instance de la classe ShowMe.

# Gestion des ressources

Tous les objets consomment des ressources système, comme de la mémoire, des handles de fichier et des connexions de base de données. Le Common Language Runtime (CLR) gère automatiquement les ressources ; vous ne devez généralement pas vous préoccuper de libérer des objets inutiles. Toutefois, la compréhension du fonctionnement de la gestion des ressources peut vous aider à concevoir plus efficacement vos applications.

## Garbage collection

Le CLR utilise un système appelé garbage collection pour gérer les ressources allouées. Le garbage collector (également appelé ramasse-miettes) libère les ressources d'un objet lorsque le code d'exécution de votre application ne peut plus atteindre cet objet. Les sections suivantes décrivent certaines modifications de la gestion des ressources dans Visual Basic .NET.

### Assignation d'objets à Nothing

**Nothing** est un mot clé utilisé par Visual Basic .NET pour indiquer qu'une variable objet ne contient pas de référence à un objet. Les versions antérieures de Microsoft Visual Basic vous encourageaient à assigner des objets non utilisés à **Nothing** pour dissocier la variable d'objet de l'objet et libérer des ressources. Vous pouvez toujours assigner des objets non utilisés à **Nothing** mais, en raison de la façon dont Visual Basic .NET gère les ressources, ce processus ne garantit pas une libération immédiate des objets. En règle générale, vous devez uniquement assigner des objets à **Nothing** pour des objets à longue durée de vie, tels que des membres partagés ou des variables globales.

### Dispose

Certains objets prennent en charge la méthode **Dispose** dont le but est de libérer plus rapidement des ressources système. Les classes qui prennent en charge cette méthode doivent implémenter l'interface **IDisposable**. La méthode **Dispose** doit être appelée explicitement lorsque vous voulez libérer des ressources d'objet. Par exemple :

```
ThisObject.Dispose
```

### Finalize

La méthode **Finalize**, prise en charge par certaines classes, s'exécute automatiquement lorsqu'un objet est libéré et peut être utilisée pour effectuer d'autres tâches de nettoyage. Elle est similaire à la méthode **Class_Terminate()** utilisée dans les versions antérieures de Microsoft Visual Basic. Contrairement à la méthode **Dispose**, le CLR appelle automatiquement la méthode **Finalize** peu de temps après qu'un objet n'est plus nécessaire.

# Durée de vie d'un objet : création et destruction des objets

Les objets entament leur existence au moment où une instance d'une classe est créée à l'aide du mot clé **New**. Les nouveaux objets nécessitent souvent l'exécution de tâches d'initialisation avant leur première utilisation. Parmi les tâches d'initialisation courantes, citons l'ouverture de fichiers, la connexion à une base de données et la lecture de valeurs de clés de Registre. Microsoft Visual Basic .NET contrôle l'initialisation des nouveaux objets à l'aide de procédures appelées constructeurs.

Les objets cessent d'exister dès que leur portée se termine et qu'ils sont libérés par le Common Language Runtime (CLR). Visual Basic .NET contrôle la libération des ressources système à l'aide de procédures nommées destructeurs. Ensemble, les constructeurs (méthodes spéciales qui permettent de contrôler l'initialisation) et les destructeurs prennent en charge la création de bibliothèques de classes robustes et prévisibles.

## Sub New et Sub Finalize

Les procédures **Sub New** et **Sub Finalize** de Visual Basic .NET initialisent et détruisent les objets ; elles remplacent les méthodes **Class_Initialize** et **Class_Terminate** utilisées dans les précédentes versions de Visual Basic. A la différence de **Class_Initialize**, le constructeur **Sub New** ne peut s'exécuter qu'une seule fois lors de la création d'une classe et ne peut pas être appelé explicitement ailleurs que dans la première ligne de code d'un autre constructeur de la même classe ou d'une classe dérivée. De plus, le code de la méthode **Sub New** s'exécute toujours avant toute autre portion de code d'une classe. Visual Basic .NET crée implicitement un constructeur **Sub New** au moment de l'exécution si vous ne définissez pas explicitement une procédure **Sub New** pour une classe.

Avant de libérer des objets, le CLR appelle automatiquement la méthode **Finalize** pour les objets qui définissent une procédure **Sub Finalize**. La méthode **Finalize** peut contenir du code qui doit s'exécuter juste avant la destruction d'un objet, comme pour la fermeture de fichiers et l'enregistrement d'informations d'état.

L'exécution de la méthode **Sub Finalize** réduit légèrement les performances, c'est pourquoi vous ne devez exécuter cette méthode que si vous devez libérer des objets explicitement.

Le destructeur **Finalize** est une méthode protégée qui peut être appelée uniquement à partir de la classe à laquelle elle appartient ou de classes dérivées. Le système appelle automatiquement **Finalize** lors de la destruction d'un objet, c'est pourquoi il ne faut pas appeler explicitement **Finalize** depuis l'extérieur de l'implémentation **Finalize** d'une classe dérivée. Contrairement à **Class_Terminate**, qui s'exécutait dès qu'un objet était défini à Nothing, il existe généralement un délai entre le moment où la portée d'un objet se termine et le moment où Visual Basic .NET appelle le destructeur **Finalize**. Visual Basic .NET autorise l'emploi d'un deuxième type de destructeur, appelé **Dispose**, qui peut être appelé explicitement à tout moment pour libérer immédiatement des ressources.

## Interface IDisposable

Les instances de classe contrôlent souvent les ressources non managées par le CLR, comme les handles Windows et les connexions aux bases de données. En complément des opérations garbage collection, les classes peuvent représenter un mécanisme de gestion active des ressources système si elles implémentent l'interface **IDisposable**. L'interface **IDisposable** possède une méthode, **Dispose**, que les clients doivent appeler lorsqu'ils ont fini d'utiliser un objet. Vous pouvez employer l'implémentation de **Dispose** pour libérer des ressources et effectuer des tâches comme la fermeture de fichiers et de connexions à une base de données. Contrairement au destructeur **Finalize**, la méthode **Dispose** n'est pas appelée automatiquement. Les clients d'une classe doivent appeler explicitement **Dispose** pour pouvoir libérer des ressources.

### ~~Garbage collection et destructeur Finalize~~

Le .NET Framework utilise un système appelé garbage collection par traçage des références qui libère de façon périodique les ressources non utilisées. Les versions antérieures de Visual Basic gèrent les ressources à l'aide d'un système différent, appelé décompte de références. Bien que ces deux systèmes exécutent la même fonction automatiquement, un certain nombre de différences importantes les distinguent.

Le CLR détruit régulièrement les objets quand le système détermine que ces objets sont devenus inutiles. Les objets sont libérés plus vite en cas de pénurie des ressources système et moins vite dans le cas contraire. Le délai qui sépare le

moment où la portée d'un objet se termine et le moment où le CLR libère celui-ci implique qu'il est désormais impossible de déterminer avec précision l'instant où l'objet sera détruit, alors que cela était possible dans les précédentes versions de Visual Basic. Dans une telle situation, l'on dit que les objets ont une durée de vie non déterminable. Dans la plupart des cas, la durée de vie non déterminable ne modifie en rien la manière dont vous écrivez les applications, à condition de ne pas oublier que le destructeur **Finalize** ne s'exécutera peut-être pas immédiatement au moment où la portée d'un objet s'achève.

Une autre différence entre les deux systèmes d'opérations garbage collection concerne l'utilisation de **Nothing**. Pour tirer parti du décompte de références, les programmeurs qui utilisent une version antérieure de Visual Basic assignent parfois **Nothing** aux variables objet afin de libérer les références que comportent ces variables. Si la variable détient la dernière référence à l'objet, les ressources de ce dernier sont libérées immédiatement. Dans Visual Basic .NET, si l'emploi de cette procédure présente sans doute toujours un intérêt dans certaines situations, il n'entraîne jamais la libération immédiate des ressources de l'objet référencé. La seule situation qui nécessite l'attribution de **Nothing** à une variable est lorsque cette variable possède une durée de vie qui est longue par rapport à la période requise pour la détection des objets orphelins par le garbage collector.

# New

Le mot clé **New** introduit une clause **New**, qui crée une nouvelle instance d'objet. La clause **New** doit spécifier une classe définie à partir de laquelle l'instance peut être créée. Vous pouvez utiliser **New** dans une instruction de déclaration ou une instruction d'assignation. Lorsque l'instruction est exécutée, elle appelle le constructeur de la classe spécifiée, en passant les arguments que vous avez fournis :

```
Dim Obj As Object
Obj = New SomeClass("String required by constructor")
' ...
Dim MyLabel As New Label()
```

Puisque les tableaux sont des classes, **New** peut créer une nouvelle instance de tableau :

```
Dim MyArray As Integer()
MyArray = New Integer() {0, 1, 2, 3}
```

Le mot clé **New** est utilisé dans le contexte suivant :

Dim, instruction

# Nothing

Le mot clé **Nothing** représente la valeur par défaut d'un type de données. L'assignation de **Nothing** à une variable la définit à sa valeur par défaut pour son type déclaré. Si ce type contient des membres de variable, ils ont tous leurs valeurs par défaut. L'exemple suivant illustre ce comportement :

```
Public Structure MyStruct
  Public Name As String
  Public Number As Short
End Structure
Dim S As MyStruct, I As Integer, B As Boolean
S = Nothing   ' Sets S.Name to Nothing, S.Number to 0.
I = Nothing   ' Sets I to 0.
B = Nothing   ' Sets B to False.
```

Si la variable est de type référence (une variable objet), **Nothing** signifie que la variable n'est pas associée à un objet. Par exemple :

```
Dim MyObject As Object
MyObject = Nothing   ' No object currently referred to.
```

Lorsque vous assignez **Nothing** à une variable objet, cette dernière ne fait plus référence à une instance d'objet. Si la variable avait fait référence à une instance au préalable, l'assignation de **Nothing** à la variable ne met pas fin à l'instance. L'instance se termine, et les ressources mémoire et système qui lui sont associées sont libérées uniquement lorsque le garbage collector détecte l'absence de toute référence active restante.

# Passage d'objets aux procédures

Visual Basic .NET vous permet de passer des objets en tant qu'arguments à des procédures comme vous passez d'autres types d'arguments. Les procédures suivantes vous montrent comment.

**Pour passer une nouvelle instance d'un formulaire à une procédure**

. Ouvrez un projet, créez un formulaire et ajoutez un bouton de commande appelé Button1.
. Copiez le code suivant dans l'événement Button1_click :

```
. Protected Sub Button1_Click(ByVal sender As System.Object, _
.         ByVal e As System.EventArgs)
.     Dim newForm As New Form1()
.     newForm.Show()
.     CenterForm(newForm)
. End Sub
.
n. Sub CenterForm(ByVal TheForm As Form)
n.     ' Centers the form on the screen.
n.     Dim RecForm As rectangle = Screen.GetBounds(TheForm)
n.     TheForm.Left = CInt((RecForm.Width - TheForm.Width) / 2)
n.     TheForm.Top = CInt((RecForm.Height - TheForm.Height) / 2)
  End Sub
```

Vous pouvez également passer un objet par référence, puis, dans la procédure, définir l'argument comme un nouvel objet.

**Pour passer une référence d'objet à une procédure d'un autre formulaire**

. Ouvrez un projet et créez un formulaire appelé Form1.
. Ajoutez un second formulaire appelé Form2.
. Placez un contrôle de zone d'image dans chaque formulaire.
. Donnez à la zone d'image du premier formulaire le nom PictureBox1.
. Donnez à la zone d'image du deuxième formulaire le nom PictureBox2.
. Assignez une image à PictureBox2 en cliquant sur la propriété **Image** dans la fenêtre **Propriétés**. Vous pouvez utiliser n'importe quelle petite image ; vous trouverez des fichiers .bmp et .jpg dans votre répertoire Windows.
. Ajoutez le code suivant à Form2 :

```
. Public Sub GetPicture(ByVal x As PictureBox)
.     Dim objX As PictureBox
.     ' Assign the passed-in picture box to an object variable.
```

```
    objX = x
    ' Assign the value of the Picture property to the Form1 picture box.
    objX.Image = PictureBox2.Image
End Sub
```

- Ajoutez le code suivant à l'événement **Form1_Click** de form1 :

```
Protected Sub Form1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs)
  Dim newForm2 As New Form2()
  newForm2.GetPicture(PictureBox1)
End Sub
```

- Exécutez l'application et cliquez sur Form1. L'image de Form2 apparaît dans la zone d'image de Form1.

  La procédure d'événement **Form1_Click** appelle la procédure GetPicture de Form2 et lui passe la zone d'image vide. La procédure GetPicture de Form2 assigne la propriété **Image** de la zone d'image de Form2 à la zone d'image vide de Form1, et l'image de Form2 est affichée dans Form1.

# Gestion de groupes d'objets

Pour de nombreuses applications, vous voulez être en mesure de créer et de gérer des groupes d'objets apparentés. Il existe deux manières de grouper des objets : en créant des tableaux d'objets ou des collections.

Les tableaux sont des structures assez rigides. Si vous souhaitez modifier la taille du tableau au moment de l'exécution, vous devez utiliser l'instruction **ReDim** pour le redéclarer. Tous les membres du tableau doivent être du même type. En revanche, les tableaux vous permettent de traiter tous les objets les uns après les autres et de posséder des membres vides. Les tableaux sont donc plus utiles pour la création et l'utilisation d'un nombre fixe d'objets fortement typés.

Les collections offrent plus de souplesse pour utiliser des groupes d'objets. Dans la mesure où une collection est une classe, vous devez déclarer une nouvelle collection avant de pouvoir lui ajouter des membres. Contrairement aux tableaux, le groupe d'objets utilisé peut augmenter ou diminuer de façon dynamique si les besoins du programme varient. Vous pouvez assigner une clé à un objet que vous insérez dans une collection afin de pourvoir l'extraire et le manipuler en fonction de cette clé.

**Dans cette section**

Création de tableaux d'objets
: Explique comment déclarer un tableau et instancier ses membres.

Collections dans Visual Basic .NET
: Propose une vue d'ensemble des collections, présente la classe **Collection**, les collections de base zéro et de base un, les valeurs d'index et de clé et explique comment ajouter et supprimer des éléments.

Création de collections d'objets
: Explique comment créer une collection et utiliser **For...Each...Next** pour traiter ses membres.

La classe Collection de Visual Basic .NET
: Présente la classe **Collection**, les types d'éléments pouvant être ajoutés à une collection et explique comment accéder aux membres après les avoir ajoutés.

Propriétés et méthodes de la classe Collection
: Présente les propriétés (**Count** et **Item**) et les méthodes (**Add** et **Remove**) permettant de manipuler des éléments dans un objet **Collection**.

Ajout, suppression et extraction d'éléments d'une collection

Explique comment ajouter, extraire et supprimer des éléments d'une collection.

# Création de tableaux d'objets

Vous déclarez et vous utilisez des tableaux d'un type d'objet comme un tableau d'un type de données. Les membres de ce tableau peuvent être extraits par leur index et peuvent être manipulés comme tout objet de ce type. Les tableaux peuvent posséder des fonctionnalités intégrées pour effectuer des recherches et des tris accessibles via la variable tableau. Pour plus d'informations sur ces méthodes, consultez Array, classe.

**Pour créer un tableau d'objets**

. Déclarez le tableau comme indiqué dans l'exemple de code suivant. Dans la mesure où les tableaux sont de base zéro, ils contiennent un membre supplémentaire par rapport à la limite supérieure déclarée.

```
Dim x(10) As Widget   ' Contains 11 members, from x(0) to x(10).
```

. Créez une instance de chaque membre du tableau ou assignez à chacun des membres une référence à un objet déjà existant. Voici un exemple de chacune de ces approches :

```
. ' Instantiates each member of an array by using a loop.
. Dim q As Integer
. For q = 0 to 10
.    x(q) = New Widget()
. Next
.
. ' Assigns a member of an array a reference to an existing object.
e. Dim myWidget As New Widget()
e. x(0) = myWidget
   x(1) = myWidget
```

Notez que vous pouvez assigner, à divers membres du tableau, des références à un même objet.

# Collections dans Visual Basic .NET

En règle générale, une collection permet de grouper et de gérer des objets apparentés. Cette définition simple décrit toutefois une réalité un peu plus complexe.

Par exemple, chaque formulaire possède une collection **Controls**. Il s'agit d'un objet qui représente tous les contrôles de ce formulaire. Il permet d'obtenir une référence à un contrôle de la collection en fonction de son index et de parcourir les membres de la collection par l'intermédiaire d'instructions **For Each...Next**.

Visual Basic propose également une classe **Collection**, qui vous permet de définir et de créer vos propres collections. A l'instar de la collection **Controls** d'un formulaire, la classe **Collection** offre également une fonctionnalité intégrée qui vous permet de parcourir les membres par l'intermédiaire de **For Each...Next** et de référencer les membres par leur index. Dans la mesure où elles sont toutes deux des collections, pourquoi le code suivant, issu d'une application Windows Forms, génère-t-il une erreur de compilation ?

```
Dim myControlsCollection As Collection
myControlsCollection = Controls   ' This line generates a compiler error.
```

Que se passe-t-il ? La collection **Controls** est une collection ; la variable myControlsCollection est déclarée en tant que **Collection** ; pourquoi ne pouvez-vous pas assigner une référence à **Controls** à la variable myControlsCollection ?

En fait, la classe **Collection** et la collection **Controls** ne sont pas polymorphes : elles ne sont pas interchangeables car elles représentent des types distincts avec des implémentations différentes. Elles ne possèdent pas les mêmes méthodes, ne stockent pas des références d'objets de la même manière ou n'utilisent pas les mêmes types de valeurs d'index.

Le nom de la classe **Collection** peut donc sembler étrange car elle ne représente que l'une des nombreuses implémentations de collection possibles. Cette rubrique présente certaines de ces différences d'implémentation.

## Collections de base zéro et de base un

Une collection peut être soit de base zéro, soit de base un, selon son index de départ. En effet, dans le premier cas, l'index du premier élément de la collection est zéro, tandis que dans le deuxième cas, l'index commence par un. La collection **Controls** décrite ci-dessus est un exemple de collection de base zéro. Une instance de l'objet **Collection**, décrit également ci-dessus, représente un exemple d'une collection de base un.

L'utilisation des collections de base un est plus intuitive, car l'index est compris entre un et la valeur de **Count**, où la propriété **Count** retourne le nombre d'éléments dans une collection. En revanche, l'index d'une collection de base zéro est compris entre zéro et une valeur inférieure d'une unité à la propriété **Count**.

Le .NET Framework normalise les collections comme étant des collections de base zéro. La classe Collection de Visual Basic sert principalement à assurer une compatibilité avec des versions précédentes.

## Valeurs d'index et de clé

Comme les instances de la classe **Collection** de Visual Basic, de nombreuses collections dans Visual Basic vous permettent d'accéder à un élément par l'intermédiaire d'un index numérique ou d'une clé **String**. Vous pouvez également ajouter des éléments aux objets **Collection** sans spécifier de clé.

En revanche, certaines collections (par exemple, **System.Collections.ArrayList**) autorisent uniquement un index numérique. Ces collections permettent d'accéder à leurs membres uniquement par l'index et ne permettent pas de leur associer une clé.

## Ajout et suppression d'éléments

Une autre différence entre les collections est la possibilité d'y ajouter des éléments ainsi que la façon dont ces éléments sont ajoutés, le cas échéant. Puisque l'objet **Collection** de Visual Basic est un outil de programmation à caractère général, il est plus flexible que les autres collections. Sa méthode **Add** permet d'ajouter des éléments dans la collection et sa méthode **Remove** permet d'en supprimer.

Par contre, certaines collections spécialisées ne permettent pas aux développeurs d'ajouter ou d'enlever des membres en utilisant du code. Par exemple, la propriété **CheckedListBox.CheckedItems** retourne des références à des éléments en fonction de leur index, mais ne permet pas d'ajouter ou de supprimer des éléments de cette collection dans le code. Pour ajouter ou enlever des éléments, vous devez activer ou désactiver les cases correspondantes dans l'interface visuelle. En effet, cette collection ne possède pas de méthode **Add** ou **Remove**.

# Création de collections d'objets

Contrairement aux tableaux, les collections sont des objets. Pour créer une collection d'objets, vous devez premièrement créer l'objet **Collection**. Dès que cet objet est créé, vous pouvez ajouter, enlever ou manipuler des éléments. Cet exemple illustre la façon de créer et de remplir un objet **Collection**.

**Pour créer une collection**

. Déclarez et créez une instance d'une variable **Collection** comme indiqué dans l'exemple de code suivant :

```
Dim myCollection As New Collection()
```

. Utilisez la méthode **Add** pour ajouter des membres à votre collection. Dans cet exemple, vous créez quatre chaînes et vous les ajoutez à votre collection. Une valeur de type **chaîne** unique peut éventuellement être ajoutée en tant que clé pour les membres de votre collection. Cette valeur est passée à la collection en tant que deuxième argument de la méthode **Add**.

```
. Dim w, x, y, z As String
. w = "Wow!"
. x = "It's"
. y = "A"
. z = "Collection"
. myCollection.Add(w, "This")
. myCollection.Add(x, "Is")
. myCollection.Add(y, "A")
  myCollection.Add(z, "Key")
```

. Si vous souhaitez enlever un membre de votre collection, vous pouvez utiliser la méthode **Remove** en fonction de l'index du membre ou de la clé facultative. Voici quelques exemples :

```
. myCollection.Remove(1)   ' Removes the first member of the collection.
  myCollection.Remove("Is")   ' Removes the member with the key "Is".
```

Notez que lorsqu'un membre est enlevé d'une collection, les valeurs d'index sont renumérotées depuis la valeur un jusqu'à la valeur de **Count**.

L'instruction **For Each...Next** permet de parcourir et de traiter les membres d'une collection, comme le montre l'exemple ci-dessous.

**Pour utiliser l'instruction For Each...Next afin de traiter les membres de votre collection**

. Déclarez une variable du type que vous souhaitez traiter dans votre collection. Par exemple, si vous voulez parcourir toutes les chaînes, vous devez déclarer une variable de type **String**, comme indiqué ci-dessous :

```
. ' Code from the previous example goes here.
  Dim aString As String
```

. Utilisez l'instruction **For Each...Next** pour examiner chacun des membres de votre collection. Dans cet exemple, vous recherchez une chaîne particulière qui s'affiche dans un message lorsqu'elle est trouvée.

```
. For Each aString in myCollection
.   If aString = "Collection" Then
.     MsgBox(aString)   ' Displays "Collection".
.   End If
  Next
```

# La classe Collection de Visual Basic .NET

Une collection permet de grouper un ensemble d'éléments apparentés. Il existe plusieurs types de collections. Les collections sont utilisées dans Visual Basic afin d'assurer le suivi de divers éléments, tels que tous les contrôles d'un formulaire (par exemple, la collection **Controls**). Par ailleurs, les utilisateurs peuvent créer leurs propres collections afin d'organiser et de manipuler des objets.

Les collections représentent une méthode efficace pour assurer le suivi des objets que votre application doit créer et détruire de façon dynamique. Le fragment de code suivant illustre la façon d'utiliser la méthode **Add** d'un objet collection afin de conserver une liste des objets Widget créés par l'utilisateur.

```
' Declares and instantiates the Collection object.
Public widgetCollection As New Collection
' Creates a new Widget and adds it to the widgetCollection.
Private Sub MakeAWidget()
  Dim tempWidget As New Widget()
```

```
  widgetCollection.Add(tempWidget)
End Sub
```

La collection widgetCollection de cet exemple organise et expose tous les objets Widget créés par l'intermédiaire de la méthode MakeAWidget. Vous pouvez extraire des références d'objet de chaque objet Widget par l'index de la collection. La taille de la collection est automatiquement ajustée lorsqu'un nouvel objet Widget est ajouté. Les instructions **For Each... Next** vous permettent d'itérer sur la collection. Si vous souhaitez assigner une clé à l'objet Widget pour extraire ce dernier, vous pouvez affecter une chaîne de texte en tant que second paramètre de la méthode **Add**, conformément à ce qui est décrit plus loin dans cette section.

Le mot clé **New** dans la déclaration de la variable widgetCollection entraîne la création d'un objet **Collection** lors de la première référence à la variable dans le code. Puisque **Collection** est une classe et non un type de données, vous devez en créer une instance et conserver une référence à cette instance dans une variable.

Comme tout autre objet, un objet **Collection** est marqué pour une opération garbage collection lorsque la dernière variable contenant une référence à cet objet est définie à **Nothing** ou devient hors de portée. Toutes les références d'objet sont libérées lors de cette opération. C'est la raison pour laquelle la variable widgetCollection est déclarée dans la classe parent afin d'exister pendant toute la durée de vie du programme.

## A l'intérieur d'un objet Collection

Un objet **Collection** stocke chaque élément d'un **Objet**. Il est donc possible d'ajouter à un objet **Collection** les mêmes éléments que ceux pouvant être stockés dans une variable objet. Vous pouvez par exemple ajouter des types de données, des objets et des tableaux standard, ainsi que des structures et des instances de classe définies par l'utilisateur.

Puisque l'objet **Collection** stocke chaque élément comme un objet, **Object** est le type de référence retourné par la propriété **Item**. Ceci engendre le problème lié à la façon d'accéder ultérieurement aux membres de la collection. Si **Option Strict** est défini sur **Off**, vous pouvez convertir implicitement une référence depuis **Collection** vers le type approprié, comme l'illustre cet exemple.

```
Option Strict Off
Dim myString As String = "This is my String"
Dim aString As string
Dim myCollection As New Collection()
```

```
myCollection.Add(myString)
aString = myCollection.Item(1)   ' Collection item converted to string.
```

En revanche, si **Option Strict** est défini sur **On**, vous devez effectuer un cast depuis **Object** vers le type que contient la collection. Pour obtenir une référence issue de **Item** de cette façon, vous pouvez utiliser la fonction **CType** pour effectuer explicitement la conversion, comme indiqué ci-dessous :

```
Option Strict On
Dim myString As String = "This is my String"
Dim aString As String
Dim myCollection As New Collection()
myCollection.Add(myString)
aString = CType(myCollection.Item(1), String)
```

Vous pouvez également créer votre propre collection fortement typée. Pour plus d'informations, consultez Procédure pas à pas : création de votre propre classe de collection.

# Propriétés et méthodes de la classe Collection

Chaque objet **Collection** comprend des propriétés et des méthodes permettant d'insérer, de supprimer et d'extraire des éléments d'une collection. Le tableau suivant décrit ces éléments.

| Pour obtenir des informations sur | Consultez |
|---|---|
| L'ajout d'éléments à la collection | Add, méthode |
| Le retour du nombre d'éléments dans la collection | Count, propriété |
| Le retour d'un élément par index ou par clé | Item, propriété |
| La suppression d'un élément de la collection, par index ou par clé | Remove, méthode |

Ces propriétés et ces méthodes ne proposent que les services de base pour les collections. Par exemple, la méthode **Add** ne peut pas contrôler le type d'objet ajouté à une collection, alors que vous pourriez vouloir procéder à cette vérification pour garantir que la collection ne comporte qu'un seul type d'objet. Il est possible d'inclure des fonctionnalités plus robustes (ainsi que des propriétés, des méthodes et des événements supplémentaires) en créant votre classe de collection personnelle. Pour plus d'informations, consultez Procédure pas à pas : création de votre propre classe de collection.

Les services de base (ajout, suppression et extraction d'une collection) dépendent des clés et des index. Une clé est une valeur de type **String**. Cette valeur peut être un nom, un numéro de licence d'un pilote, un numéro de sécurité sociale ou un entier converti en une chaîne. La méthode **Add** permet d'associer une clé à un élément, comme décrit dans la section ~~Ajout, suppression et extraction d'éléments d'une collection~~.

Un index de la classe Collection est un entier compris entre un (1) et le nombre d'éléments de la collection. Vous pouvez contrôler la valeur initiale d'index d'un élément, en utilisant les paramètres before et after ; cette valeur peut toutefois changer lors de l'ajout ou de la suppression d'autres éléments. Pour plus d'informations, consultez la méthode **Add**.

Vous pouvez utiliser l'index pour procéder à une itération sur les éléments d'une collection. Par exemple, le code suivant illustre deux possibilités pour octroyer une augmentation de 10 % à tous les employés de la collection d'objets Employee, si la variable employeesCollection contient une référence à un objet **Collection** :

```
Option Strict On
Dim counter As Integer
Dim emp As Employee
For counter = 1 To employeesCollection.Count
  emp = CType(employeesCollection(counter), Employee)
  emp.Rate *= 1.1
Next

 ' Alternative code follows.
Dim emp As Employee
For Each emp In employeesCollection
  emp.Rate *= 1.1
Next
```

> **Remarque** Si vous tentez d'utiliser **For Each** avec une collection comprenant des éléments d'un type différent de celui qui est itéré, une erreur **ArgumentException** se produit.

# Ajout, suppression et extraction d'éléments d'une collection

La classe **Collection** comprend une fonctionnalité intégrée qui vous permet d'ajouter, de supprimer et d'extraire des éléments. L'utilisation de cette fonctionnalité est décrite ci-dessous.

### Ajout d'éléments à une collection

La méthode **Add** permet d'ajouter un élément à une collection. La syntaxe est la suivante :

```
object.Add(Item, Key)
```

Par exemple, pour ajouter un objet ordre de travail (work order) à une collection d'ordres de travail en utilisant la propriété ID de l'ordre de travail en tant que clé, vous pouvez écrire la syntaxe suivante :

```
workordersCollection.Add(woNew, woNew.ID)
```

Dans ce cas, la propriété ID est une chaîne. Si la propriété est un nombre (par exemple, un entier de type **Long**), vous devez utiliser sa fonction **.ToString** pour le convertir en une valeur de type **String** requise pour les clés :

```
workordersCollection.Add(woNew, woNew.ID.ToString)
```

L'utilisation d'une clé est facultative. Si vous ne souhaitez pas associer de clé à l'objet dans votre collection, vous pouvez l'ajouter sans clé, comme illustré ci-dessous :

```
workordersCollection.Add(woNew)
```

Vous pouvez utiliser les arguments before et after pour gérer une collection ordonnée d'objets. Le membre à ajouter est placé dans la collection avant ou après le membre identifié par l'argument before ou after, respectivement. Par exemple, si la valeur de before est égale à 1, un élément est inséré au début de

la collection car les objets **Collection** sont de base un, comme l'indique l'exemple suivant :

```
workordersCollection.Add(woNew, woNew.ID, 1)
```

De même, l'argument after ajoute un élément après l'index spécifié. L'exemple suivant ajoute un élément en tant que troisième élément :

```
workordersCollection.Add(woNew, woNew.ID,,2)
```

Vous pouvez spécifier une valeur soit pour before, soit pour after, mais pas pour ces deux arguments.

## Suppression d'éléments d'une collection

La méthode **Remove** permet de supprimer un élément d'une collection. La syntaxe est la suivante :

```
object.Remove(index | key)
```

L'argument index peut être l'emplacement de l'élément à supprimer ou la clé de l'élément. Si la clé du troisième élément d'une collection est « W017493 », vous pouvez utiliser l'une des deux instructions suivantes pour la supprimer :

```
workordersCollection.Remove(3)
workordersCollection.Remove("W017493")
```

## Extraction d'éléments d'une collection

La propriété **Item** permet d'extraire des éléments spécifiques d'une collection. La syntaxe est la suivante :

```
variable = object.Item(index)
```

Comme pour la méthode **Remove**, l'index peut être l'emplacement de l'élément dans la collection ou la clé de l'élément. En utilisant le même exemple que pour la méthode **Remove**, vous pouvez utiliser l'une des deux instructions suivantes pour extraire le troisième élément de la collection :

```
woCurrent = workordersCollection.Item(3)
woCurrent = workordersCollection.Item("W017493")
```

> **Remarque** Si vous utilisez des nombres entiers en tant que clés, vous devez utiliser la fonction **.ToString** pour les convertir en

chaînes avant de les passer à la propriété **Add**, **Remove** ou **Item**. Un objet **Collection** suppose toujours qu'un nombre entier est un index.

## Item est la propriété par défaut

Dans la mesure où la propriété **Item** est la propriété par défaut d'un objet **Collection**, vous pouvez l'omettre lorsque vous accédez à un élément d'une collection. Il est donc également possible d'écrire les exemples de code précédents comme suit :

```
woCurrent = workordersCollection(3)
woCurrent = workordersCollection("W017493")
```

Vous pouvez également utiliser le qualificateur d'accès au membre ! pour accéder à un membre de la collection par sa clé sans assigner de guillemets doubles ou de parenthèses à la clé. Il est donc également possible d'écrire l'exemple précédent comme suit :

```
woCurrent = workordersCollection!W017493
```

**Remarque** Les objets Collection mettent automatiquement à jour leurs nombres d'index numériques lorsque vous ajoutez ou supprimez des éléments ; l'index numérique d'un élément donné varie donc fréquemment. Par conséquent, n'enregistrez pas une valeur d'index numérique qui procédera ultérieurement à l'extraction du même élément dans votre programme. Vous devez utiliser des clés à cette fin.

# Obtention d'informations sur la classe au moment de l'exécution

Dans certaines situations, vous pouvez avoir besoin de travailler avec des variables définies avec le type **Object** plutôt qu'avec des types d'objets plus spécifiques. Les variables définies comme **Object** peuvent contenir un objet de n'importe quel autre type de données (par exemple, Integer, Double ou Object).

**Dans cette section**

~~Détermination du type Object~~

Explique comment déterminer la classe à laquelle appartient un objet.

~~Appel d'une propriété ou méthode à l'aide d'un nom de chaîne~~

Décrit pas à pas le processus permettant de spécifier les arguments par leur nom plutôt que par leur position lorsque vous appelez une méthode.

# Détermination du type Object

Les variables objets génériques (c'est-à-dire les variables que vous déclarez comme étant de type **Object**) peuvent contenir les objets de n'importe quelle classe. Si vous utilisez des variables de type **Object**, vous pouvez être amené à effectuer différentes actions selon la classe de l'objet, par exemple lorsque certains objets ne prennent pas en charge une propriété ou méthode particulière. Visual Basic .NET fournit deux moyens permettant de déterminer le type d'objet stocké dans une variable objet : la fonction **TypeName** et l'opérateur **TypeOf...Is**.

La fonction **TypeName** retourne une chaîne et représente le meilleur choix lorsque vous devez stocker ou afficher le nom de classe d'un objet, comme illustré dans le fragment de code suivant :

```
MsgBox(TypeName(Ctrl))
```

L'opérateur **TypeOf…Is** représente le meilleur choix lorsqu'il s'agit de tester le type d'un objet, car il est beaucoup plus rapide qu'une comparaison de chaîne équivalente à l'aide de la fonction **TypeName**. Le fragment de code suivant utilise **TypeOf…Is** dans une instruction **If...Then...Else** :

```
If TypeOf Ctrl Is Button Then
  MsgBox("The control is a button.")
End If
```

Néanmoins, il convient d'apporter certaines précisions. L'opérateur **TypeOf…Is** retourne **True** si l'objet est d'un type spécifique ou s'il est dérivé d'un type spécifique. Quasiment toutes vos actions dans Visual Basic .NET reposent sur l'utilisation d'objets, lesquels rassemblent des éléments qui ne sont pas forcément perçus comme des objets, par exemple les données de type **String** et **Integer**. Ces objets sont dérivés et héritent des méthodes de **System.Object**. Lorsque l'opérateur **TypeOf…Is** est passé sous forme de donnée de type **Integer** et qu'il est évalué à l'aide du type **Object**, il retourne **True**. L'exemple suivant indique que le paramètre InParam est à la fois du type **Object** et **Integer** :

```
Sub CheckType(ByVal InParam)
' Both If statements evaluate to True when an
' Integer is passed to this procedure.
  If TypeOf InParam Is Object Then
```

```
    MsgBox("InParam is an Object")
  End If
  If TypeOf InParam Is Integer Then
    MsgBox("InParam is an Integer")
  End If
End Sub
```

L'exemple suivant utilise à la fois **TypeOf...Is** et **TypeName** pour déterminer le type d'objet passé dans l'argument Ctrl. La procédure TestObject appelle ShowType à l'aide de trois sortes de contrôles différents.

**Pour exécuter l'exemple**

- Créez un nouveau projet Windows Forms et ajoutez un contrôle **Button**, un contrôle **CheckBox** et un contrôle **RadioButton** au formulaire.
- A partir du bouton situé sur votre formulaire, appelez la procédure TestObject.
- Ajoutez le code suivant à la section **Declarations** de votre formulaire :

```
. Sub ShowType(ByVal Ctrl As Object)
.   'Use the TypeName function to display the class name as text.
.   MsgBox(TypeName(Ctrl))
.   'Use the TypeOf function to determine the object's type.
.   If TypeOf Ctrl Is Button Then
.     MsgBox("The control is a button.")
.   ElseIf TypeOf Ctrl Is CheckBox Then
.     MsgBox("The control is a check box.")
.   Else
.     MsgBox("The object is some other type of control.")
.   End If
. End Sub
n .
. Protected Sub TestObject()
. 'Test the ShowType procedure with three kinds of objects.
.   ShowType(Me.Button1)
.   ShowType(Me.CheckBox1)
.   ShowType(Me.RadioButton1)
  End Sub
```

# Appel d'une propriété ou méthode à l'aide d'un nom de chaîne

Dans la plupart des cas, vous pouvez découvrir les propriétés et méthodes d'un objet au moment du design et écrire du code pour les gérer. Toutefois, dans quelques cas vous ne connaîtrez peut-être pas les propriétés et méthodes d'un objet à l'avance ou vous voudrez tout simplement offrir à l'utilisateur la possibilité de spécifier les propriétés et d'exécuter les méthodes au moment de l'exécution.

Prenons l'exemple d'une application cliente qui évalue des expressions entrées par l'utilisateur en passant un opérateur au composant COM. Supposons que vous voulez constamment ajouter de nouvelles fonctions au composant requérant de nouveaux opérateurs. Si vous utilisez les techniques d'accès standard, vous devrez recompiler et redistribuer l'application cliente avant qu'elle puisse utiliser les nouveaux opérateurs. Pour éviter ce problème, vous pouvez appeler la fonction **CallByName** pour passer les nouveaux opérateurs en tant que chaînes sans modifier l'application.

La fonction **CallByName** vous permet d'utiliser une chaîne pour spécifier une propriété ou une méthode au moment de l'exécution. La signature de la fonction **CallByName** ressemble à ceci :

```
Result = CallByName(Object, ProcedureName, CallType, Arguments())
```

Le premier argument, Object, prend le nom de l'objet sur lequel vous voulez agir. L'argument ProcedureName accepte une chaîne contenant le nom de la procédure de méthode ou de propriété à appeler. L'argument CallType accepte une constante représentant le type de procédure à appeler : une méthode (Microsoft.VisualBasic.CallType.Method), une lecture de propriété (Microsoft.VisualBasic.CallType.Get) ou une définition de propriété (Microsoft.VisualBasic.CallType.Set). L'argument Arguments, facultatif, accepte un tableau de type **Object** contenant n'importe quel argument de la procédure.

Vous pouvez utiliser la fonction **CallByName** avec des classes de votre solution en cours, mais elle est plus souvent employée pour accéder à des objets COM ou des objets d'assemblys .NET.

Supposons que vous ajoutez une référence à un assembly contenant une classe appelée MathClass dotée d'une nouvelle fonction appelée SquareRoot, comme illustré dans le code suivant :

```
Class MathClass
  Function SquareRoot(ByVal X As Double) As Double
    Return Math.Sqrt(X)
  End Function
  Function InverseSine(ByVal X As Double) As Double
```

```
    Return Math.Atan(X / Math.Sqrt(-X * X + 1))
  End Function
  Function Acos(ByVal X As Double) As Double
    Return Math.Atan(-X / Math.Sqrt(-X * X + 1)) + 2 * Math.Atan(1)
  End Function
End Class
```

Votre application peut utiliser des contrôles de zone de texte pour garantir que la méthode et ses arguments seront appelés. Par exemple, si TextBox1 contient l'expression à évaluer et si TextBox2 est utilisé pour entrer le nom de la fonction, vous pouvez utiliser le code suivant pour appeler la fonction SquareRoot pour l'expression contenue dans TextBox1 :

```
Private Sub CallMath()
  Dim Math As New MathClass()
  Me.TextBox1.Text = CStr(CallByName(Math, Me.TextBox2.Text, _
    Microsoft.VisualBasic.CallType.Method, TextBox1.Text))
End Sub
```

Si vous entrez « 64 » dans TextBox1, « SquareRoot » dans TextBox2 et que vous appelez ensuite la procédure CallMath, la racine carrée du nombre contenu dans TextBox1 est évaluée. Le code de cet exemple appelle la fonction SquareRoot (qui accepte une chaîne contenant l'expression à évaluer comme un argument obligatoire) et retourne « 8 » dans TextBox1 (la racine carrée de 64). Evidemment, si l'utilisateur entre une chaîne non valide dans TextBox2, si la chaîne contient le nom d'une propriété et non celui d'une méthode ou si la méthode possède un argument obligatoire supplémentaire, il se produit une erreur d'exécution. Vous devez ajouter un code robuste de gestion des erreurs lorsque vous utilisez la fonction **CallByName** afin d'anticiper les erreurs de ce type.

> **Remarque** **CallByName** peut être utile dans certaines situations, mais pesez bien ses avantages par rapport à ses inconvénients en matière de performances — l'appel d'une procédure à l'aide de cette fonction est un peu plus lent qu'un appel à liaison tardive. Si la fonction est appelée de façon répétitive, comme dans une boucle, **CallByName** peut avoir un impact négatif sur les performances.

# Présentation des classes

Les classes revêtent de l'importance dans la programmation orientée objet car elles permettent de regrouper des éléments en une seule entité, mais aussi d'en contrôler la visibilité et l'accessibilité par d'autres procédures. En outre, une classe peut hériter du code défini dans d'autres classes et le réutiliser.

**Dans cette section**

Classes : modèles d'objets
: Propose une vue d'ensemble de l'encapsulation, de l'héritage et des membres partagés.

Procédure pas à pas : définition de classes
: Décrit étape par étape le processus de création d'une classe.

Classes et modules standard
: Explique les différences qui distinguent les classes des modules standard.

Durée de vie d'un objet : création et destruction des objets
: Analyse la création et la suppression des instances de classe.

Utilisation de constructeurs et de destructeurs
: Traite de l'initialisation de nouvelles instances de classes et de la suppression des ressources devenues inutiles.

Propriétés, champs et méthodes de classe
: Décrit les champs, les propriétés et les méthodes qui constituent une classe.

# Classes : modèles d'objets

Les classes sont des représentations symboliques d'objets ; elles décrivent les propriétés, les champs, les méthodes et les événements qui constituent des objets de la même façon que les plans sont des modèles décrivant les éléments qui composent un bâtiment. Tout comme un plan peut être employé pour construire plusieurs bâtiments, une classe unique peut servir à créer autant d'objets que cela est nécessaire. Et tout comme un plan définit les parties d'un bâtiment qui sont accessibles aux personnes utilisant celui-ci, les classes peuvent elles aussi contrôler l'accès des utilisateurs aux composants des objets par l'encapsulation.

## Classes et objets

Les termes « classe » et « objet » sont parfois employés indifféremment, mais en réalité, les classes décrivent la structure des objets, alors que ces derniers sont des instances utilisables des classes. Chaque instance est une copie exacte mais distincte de sa classe. Dans la mesure où un objet est une « instance » d'une classe, l'action de créer un objet se nomme instanciation.

Si l'on reprend l'analogie avec un plan de construction, une classe est un plan, et un objet est un bâtiment basé sur ce plan. La plupart du temps, la modification des données d'un objet particulier n'entraîne pas une modification des données des autres objets (les membres partagés, qui sont des membres de classe déclarés avec le modificateur **Shared**, font toutefois exception à cette règle puisqu'ils existent indépendamment des instances spécifiques d'une classe).

## Encapsulation

L'encapsulation est la capacité à contenir un groupe d'éléments associés et à en contrôler l'accès. Les classes constituent l'un des moyens les plus courants d'encapsuler des éléments. Dans l'exemple ci-dessous, la classe `BankAccount` encapsule les méthodes, les champs et les propriétés qui décrivent un compte bancaire. Sans encapsulation, il aurait été nécessaire de déclarer des procédures et des variables distinctes pour stocker et gérer les informations relatives au compte bancaire, sans parler de la difficulté de manipuler plus d'un seul compte bancaire à la fois. L'encapsulation vous permet d'utiliser les données et les procédures de la classe `BankAccount` en tant qu'unité. Vous pouvez manipuler plusieurs comptes bancaires à la fois sans qu'il n'y ait confusion car chaque compte est représenté par une instance unique de la classe.

L'encapsulation permet en outre de contrôler la manière dont les données et les procédures sont utilisées. Vous pouvez recourir à des modificateurs d'accès, tels que **Private** ou **Protected**, pour empêcher les procédures externes d'exécuter des méthodes de classe ou de lire et de modifier les données figurant dans les propriétés et les champs. Il est essentiel de déclarer les informations internes d'une classe comme **Private** pour empêcher qu'elles ne soient utilisées par un élément extérieur à la classe ; cette technique est appelée masquage des données. Dans la classe BankAccount, les informations sur le client, telles que le solde du compte, sont protégées de cette manière. L'une des règles fondamentales de l'encapsulation est que des données de classe doivent être modifiées ou extraites uniquement par des méthodes ou des procédures **Property**. Masquer les informations d'implémentation des classes vous permet d'empêcher que celles-ci ne soient utilisées de façon non désirée et de modifier ces éléments ultérieurement sans risquer des problèmes de compatibilité. Par exemple, les versions ultérieures de la classe BankAccount ci-dessous peuvent modifier le type de données du champ AccountBalance sans risquer d'entraver le fonctionnement des applications pour lesquelles ce champ doit absolument présenter un type de données spécifique.

## Héritage

A l'instar des structures Visual Basic .NET, les classes vous permettent de définir des types de données qui encapsulent un groupe d'éléments apparentés. Contrairement aux structures, toutefois, les classes Visual Basic .NET peuvent hériter des caractéristiques d'autres classes et étendre ces caractéristiques. Les classes qui servent de base à de nouvelles classes sont appelées classes de base. Les classes dérivées des classes de base sont appelées classes dérivées. Les classes dérivées héritent de tous les champs, propriétés, méthodes et événements définis dans leur classe de base. Cela signifie que vous pouvez développer et déboguer une classe une seule fois et la réutiliser ensuite comme la classe de base d'autres classes.

L'exemple suivant définit une classe de base qui représente un compte bancaire générique, et une classe spécifique qui hérite des propriétés de la classe de base mais qui est personnalisée pour décrire un compte de chèques :

```
Class BankAccount
  Private AccountNumber As String
  Private AccountBalance As Decimal
  Private HoldOnAccount As Boolean = False
  Public Sub PostInterest()
    ' Add code to calculate the interest for this account.
  End Sub
```

```
  ReadOnly Property Balance() As Decimal
    Get
      Return AccountBalance 'Returns the available balance.
    End Get
  End Property
End Class

Class CheckingAccount
  Inherits BankAccount
  Sub ProcessCheck()
    ' Add code to process a check drawn on this account.
  End Sub
End Class
```

Pour plus d'informations sur l'héritage, consultez Eléments fondamentaux de l'héritage.

## Membres partagés

Par défaut, les données de classe sont spécifiques à chaque instance de la classe, mais il peut arriver qu'une donnée unique doive être partagée entre tous les objets créés à partir d'une classe. Dans ce cas, vous pouvez utiliser le modificateur **Shared** pour faire en sorte qu'une variable partage la même valeur dans toutes les instances d'une classe (les membres partagés sont parfois appelés « membres statiques » dans d'autres langages de programmation). Vous pouvez appeler des méthodes partagées directement à l'aide d'un nom de classe sans créer au préalable une instance de la classe.

Pour plus d'informations sur les membres partagés, consultez Membres partagés.

## Occultation

Les classes dérivées peuvent utiliser le mot clé **Shadows** pour déclarer un membre avec le même nom que le membre hérité. Les membres occultés ne doivent pas forcément être du même type de données que le membre à occulter. Par exemple, une propriété peut occulter une variable de type **Integer**.

Pour plus d'informations sur les membres partagés, consultez Occultation.

# Procédure pas à pas : définition de classes

Cette procédure pas à pas montre comment utiliser des modules de classe pour définir des classes, à partir desquelles vous pouvez ensuite créer des objets. Elle montre également comment créer des propriétés et des méthodes pour la nouvelle classe et illustre l'initialisation des objets.

**Pour définir une classe**

. Ouvrez un nouveau projet d'application Windows en cliquant sur **Nouveau** dans le menu **Fichier**, puis sur **Projet**. La boîte de dialogue **Nouveau projet** s'affiche.
. Sélectionnez **Application Windows** dans la liste des modèles de projet Visual Basic. Le nouveau projet s'affiche.
. Ajoutez une nouvelle classe au projet en cliquant sur **Ajouter une classe** dans le menu **Projet**. La boîte de dialogue **Ajouter un nouvel élément** s'affiche.
. Nommez le nouveau module StateInfo.vb, puis cliquez sur **Ouvrir**. Le code de la nouvelle classe s'affiche :
. Public Class StateInfo
End Class

> **Remarque** Vous pouvez utiliser l'éditeur de code de Visual Studio .NET pour ajouter une classe à votre formulaire de démarrage en tapant le mot clé **Class**, suivi du nom de la nouvelle classe. L'éditeur de code fournit une instruction **End Class** correspondante.

. Pour simplifier l'accès aux classes du Registre, ajoutez une instruction **Imports** en haut du code source contenant l'instruction de classe :

```
Imports Microsoft.Win32
```

. Définissez trois champs privés pour la classe en insérant le code suivant entre les instructions **Class** et **End Class** :
. Private pVal As String ' Holds the LastFile property value.
. Private KeyName As String ' Holds the name of the registry key.
Private SubKeyName As String ' Holds the name of the subkey.

Ces trois champs sont déclarés **Private** et peuvent être utilisés uniquement à partir de la classe. Vous pouvez rendre les champs

accessibles à partir de l'extérieur d'une classe en utilisant des modificateurs qui fournissent un accès plus grand, tel que l'accès **Public**.

. Définissez une propriété pour la classe en ajoutant le code suivant :

```
. Public Property LastFile() As String
.   Get ' Retrieves the property value.
.     Return pVal
.   End Get
.   Set(ByVal Value As String)
.     pVal = Value
.   End Set
  End Property
```

. Définissez des méthodes pour la classe en ajoutant le code suivant :

```
. Sub SaveStateInfo()
. '  Save the current value of the LastFile property to the registry.
.   Dim aKey As RegistryKey
. '  Create a key if it does not exist.
.   aKey = Registry.CurrentUser.CreateSubKey(KeyName)
. '  Save the property value to the registry.
.   aKey.SetValue(SubKeyName, pVal)
. End Sub
n.
n. Sub GetStateInfo()
n. '  Restore the property value LastFile from the
n. '  value stored in the registry.
r .   Dim aKey As Object
r .   Dim myRegKey As RegistryKey = Registry.CurrentUser
r .   Try
  . '  This call goes to the Catch block if the registry key is not set.
  .   myRegKey = myRegKey.OpenSubKey(KeyName)
  .     Dim oValue As Object = myRegKey.GetValue(SubKeyName)
) .       pVal = CStr(oValue)
) .   Catch
) .     pVal = "" ' Set to default if key value cannot be read.
) .   End Try
  End Sub
```

n. Définissez un constructeur paramétré pour la nouvelle classe en ajoutant une procédure intitulée **Sub New** :

```
. Sub New(ByVal RegistryKeyName As String, _
.       ByVal RegistrySubKeyName As String)
. '  Save the names of the registry key and subkey in two fields.
.   KeyName = RegistryKeyName
.   SubKeyName = RegistrySubKeyName
. '  Restore the value of the LastFile property from the registry.
.   MyClass.GetStateInfo()
  End Sub
```

Le constructeur **Sub New** est appelé automatiquement lorsqu'un objet basé sur cette classe est créé. Ce constructeur définit la valeur des champs qui contiennent les noms des clés et des sous-clés du Registre et appelle une procédure qui restaure la valeur de la propriété LastFile à partir du Registre.

- Définissez un destructeur **Finalize** pour la classe en ajoutant une procédure appelée Finalize :

```
Protected Overrides Sub Finalize()
  ' Save the value of the LastFile property to the registry
  ' when the class loses scope.
  SaveStateInfo()
  MyBase.Finalize() ' Call Finalize on the base class.
End Sub
```

Le destructeur **Finalize** enregistre la valeur stockée dans la propriété dans le Registre après la fin de la portée de la classe.

**Pour créer un bouton permettant de tester la classe**

- Modifiez le formulaire de démarrage en mode design en cliquant avec le bouton droit sur son nom dans l'Explorateur de solutions, puis en cliquant sur Concepteur de vues. Par défaut, le formulaire de démarrage des projets d'application Windows est intitulé Form1.vb. Le formulaire principal s'affiche.
- Ajoutez un bouton au formulaire principal et double-cliquez dessus pour afficher le code du gestionnaire d'événements **Button1_Click**. Ajoutez le code suivant pour appeler la procédure de test :

```
  ' Create an instance of the class and pass the registry key
  ' names to the Sub New constructor.
  Dim SI As New StateInfo("Software\StateInfo", "PropertyValue")
  ' Display the value of the property if it has been restored.
  If Len(SI.LastFile) > 1 Then
    MsgBox("The value of the property LastFile is: " _
        & SI.LastFile)
  Else
    MsgBox("The LastFile property has not been set.")
  End If
  SI.LastFile = "C:\BinaryFile.txt" ' Set the property.
  ' Ask the object to save the property value to the registry.
  SI.SaveStateInfo()
```

**Pour exécuter l'application**

- Exécutez l'application en appuyant sur F5. L'application apparaît.

. Cliquez sur le bouton situé sur votre formulaire pour appeler la procédure de test. La première fois que vous effectuez cette opération, un message s'affiche indiquant que la propriété LastFile n'a pas été définie.
. Cliquez sur **OK** pour fermer le message. La procédure **Button1_Click** définit la valeur de la propriété LastFile et appelle la méthode **SaveState**. Même si **SaveState** n'était pas explicitement appelé à partir de **Button1_Click**, il serait automatiquement appelé par la méthode **Finalize** après la fermeture du formulaire de démarrage.
. Cliquez une seconde fois sur le bouton. Le message « The value of the property LastFile is C:\BinaryFile.txt » s'affiche.
. Fermez le formulaire principal, puis redémarrez le programme en appuyant sur F5. Un objet basé sur la classe est créé, et son constructeur **Sub New** appelle la procédure **GetStateInfo** qui restaure la valeur de la propriété à partir du Registre. Le message « The value of the property LastFile is C:\BinaryFile.txt » s'affiche à nouveau lorsque vous cliquez sur le bouton.

# Classes et modules standard

Une classe et un module sont tous deux des types référence qui encapsulent des éléments définis en leur sein, mais ils se distinguent par la manière dont ces éléments sont accessibles à partir d'autres procédures.

La principale différence qu'il existe entre les classes et les modules est que les classes peuvent être instanciées, mais pas les modules standard. Dans la mesure où il n'existe jamais plus d'une seule copie des données d'un module standard, quand une partie de votre programme transforme une variable publique en module standard, toute autre partie du programme reçoit la même valeur si, par la suite, elle lit cette variable. En revanche, les données de classe existent séparément pour chaque objet instancié. Une autre différence est que les classes peuvent implémenter des interfaces, contrairement aux modules standard.

Les classes et les modules donnent également une portée différente à leurs membres. Les membres définis dans une classe ont comme portée une instance spécifique de la classe et n'existent que pendant la durée de vie de l'objet. Par conséquent, les membres d'une classe ne sont accessibles depuis l'extérieur de la classe qu'au moyen de noms qualifiés complets, tels que Object.Member. Les membres déclarés dans un module standard, en revanche, sont partagés par

défaut et ont comme portée l'espace de déclaration de l'espace de noms contenant du module standard. Dès lors, les variables publiques d'un module standard sont effectivement des variables globales puisqu'elles sont visibles à n'importe quel emplacement du projet et qu'elles existent pendant toute la durée de vie du programme. Contrairement aux membres de classe, les membres des modules standard sont implicitement partagés et ne peuvent pas utiliser le mot clé Shared.

# Durée de vie d'un objet : création et destruction des objets

Les objets entament leur existence au moment où une instance d'une classe est créée à l'aide du mot clé **New**. Les nouveaux objets nécessitent souvent l'exécution de tâches d'initialisation avant leur première utilisation. Parmi les tâches d'initialisation courantes, citons l'ouverture de fichiers, la connexion à une base de données et la lecture de valeurs de clés de Registre. Microsoft Visual Basic .NET contrôle l'initialisation des nouveaux objets à l'aide de procédures appelées constructeurs.

Les objets cessent d'exister dès que leur portée se termine et qu'ils sont libérés par le Common Language Runtime (CLR). Visual Basic .NET contrôle la libération des ressources système à l'aide de procédures nommées destructeurs. Ensemble, les constructeurs (méthodes spéciales qui permettent de contrôler l'initialisation) et les destructeurs prennent en charge la création de bibliothèques de classes robustes et prévisibles.

### Sub New et Sub Finalize

Les procédures **Sub New** et **Sub Finalize** de Visual Basic .NET initialisent et détruisent les objets ; elles remplacent les méthodes **Class_Initialize** et **Class_Terminate** utilisées dans les précédentes versions de Visual Basic. A la différence de **Class_Initialize**, le constructeur **Sub New** ne peut s'exécuter qu'une seule fois lors de la création d'une classe et ne peut pas être appelé explicitement ailleurs que dans la première ligne de code d'un autre constructeur de la même classe ou d'une classe dérivée. De plus, le code de la méthode **Sub New** s'exécute toujours avant toute autre portion de code d'une classe. Visual Basic .NET crée implicitement un constructeur **Sub New** au moment de

l'exécution si vous ne définissez pas explicitement une procédure **Sub New** pour une classe.

Avant de libérer des objets, le CLR appelle automatiquement la méthode **Finalize** pour les objets qui définissent une procédure **Sub Finalize**. La méthode **Finalize** peut contenir du code qui doit s'exécuter juste avant la destruction d'un objet, comme pour la fermeture de fichiers et l'enregistrement d'informations d'état. L'exécution de la méthode **Sub Finalize** réduit légèrement les performances, c'est pourquoi vous ne devez exécuter cette méthode que si vous devez libérer des objets explicitement.

Le destructeur **Finalize** est une méthode protégée qui peut être appelée uniquement à partir de la classe à laquelle elle appartient ou de classes dérivées. Le système appelle automatiquement **Finalize** lors de la destruction d'un objet, c'est pourquoi il ne faut pas appeler explicitement **Finalize** depuis l'extérieur de l'implémentation **Finalize** d'une classe dérivée. Contrairement à **Class_Terminate**, qui s'exécutait dès qu'un objet était défini à Nothing, il existe généralement un délai entre le moment où la portée d'un objet se termine et le moment où Visual Basic .NET appelle le destructeur **Finalize**. Visual Basic .NET autorise l'emploi d'un deuxième type de destructeur, appelé **Dispose**, qui peut être appelé explicitement à tout moment pour libérer immédiatement des ressources.

## Interface IDisposable

Les instances de classe contrôlent souvent les ressources non managées par le CLR, comme les handles Windows et les connexions aux bases de données. En complément des opérations garbage collection, les classes peuvent représenter un mécanisme de gestion active des ressources système si elles implémentent l'interface **IDisposable**. L'interface **IDisposable** possède une méthode, **Dispose**, que les clients doivent appeler lorsqu'ils ont fini d'utiliser un objet. Vous pouvez employer l'implémentation de **Dispose** pour libérer des ressources et effectuer des tâches comme la fermeture de fichiers et de connexions à une base de données. Contrairement au destructeur **Finalize**, la méthode **Dispose** n'est pas appelée automatiquement. Les clients d'une classe doivent appeler explicitement **Dispose** pour pouvoir libérer des ressources.

### Garbage collection et destructeur Finalize

Le .NET Framework utilise un système appelé garbage collection par traçage des références qui libère de façon périodique les ressources non utilisées. Les versions antérieures de Visual Basic gèrent les ressources à l'aide d'un système

différent, appelé décompte de références. Bien que ces deux systèmes exécutent la même fonction automatiquement, un certain nombre de différences importantes les distinguent.

Le CLR détruit régulièrement les objets quand le système détermine que ces objets sont devenus inutiles. Les objets sont libérés plus vite en cas de pénurie des ressources système et moins vite dans le cas contraire. Le délai qui sépare le moment où la portée d'un objet se termine et le moment où le CLR libère celui-ci implique qu'il est désormais impossible de déterminer avec précision l'instant où l'objet sera détruit, alors que cela était possible dans les précédentes versions de Visual Basic. Dans une telle situation, l'on dit que les objets ont une durée de vie non déterminable. Dans la plupart des cas, la durée de vie non déterminable ne modifie en rien la manière dont vous écrivez les applications, à condition de ne pas oublier que le destructeur **Finalize** ne s'exécutera peut-être pas immédiatement au moment où la portée d'un objet s'achève.

Une autre différence entre les deux systèmes d'opérations garbage collection concerne l'utilisation de **Nothing**. Pour tirer parti du décompte de références, les programmeurs qui utilisent une version antérieure de Visual Basic assignent parfois **Nothing** aux variables objet afin de libérer les références que comportent ces variables. Si la variable détient la dernière référence à l'objet, les ressources de ce dernier sont libérées immédiatement. Dans Visual Basic .NET, si l'emploi de cette procédure présente sans doute toujours un intérêt dans certaines situations, il n'entraîne jamais la libération immédiate des ressources de l'objet référencé. La seule situation qui nécessite l'attribution de **Nothing** à une variable est lorsque cette variable possède une durée de vie qui est longue par rapport à la période requise pour la détection des objets orphelins par le garbage collector.

# Utilisation de constructeurs et de destructeurs

**Voir aussi**

Durée de vie d'un objet : création et destruction des objets | Méthodes et destructeurs Finalize | IDisposable.Dispose, méthode

Les constructeurs et les destructeurs régissent la création et la destruction des objets.

Pour créer un constructeur d'une classe, créez une procédure nommée **Sub New** à n'importe quel endroit de la définition de la classe. Pour créer un constructeur paramétré, spécifiez les noms et les types de données des arguments de la procédure **Sub New**, de la même façon que vous définiriez les arguments de toute autre procédure, comme dans le code ci-dessous :

```
Sub New(ByVal sString As String)
```

Les constructeurs sont souvent surchargés, comme l'illustre le code suivant :

```
Sub New(ByVal sString As String, iInt As Integer)
```

Quand vous définissez une classe dérivée d'une autre classe, la première ligne d'un constructeur doit constituer un appel au constructeur de la classe de base, à moins que la classe de base ne possède un constructeur accessible n'acceptant pas de paramètre. Un appel à la classe de base contenant le constructeur ci-dessus, par exemple, serait MyBase.New(sString). Sinon, MyBase.New est facultatif, et le runtime Visual Basic .NET l'appelle implicitement.

A la suite du code qui appelle le constructeur de l'objet parent, vous pouvez éventuellement ajouter du code d'initialisation supplémentaire à la procédure **Sub New**. Cette dernière peut accepter des arguments lorsqu'elle est appelée en tant que constructeur paramétré. Ces paramètres sont passés à partir de la procédure qui appelle le constructeur, comme dans Dim AnObject As New ThisClass(X).

L'exemple de code suivant illustre l'utilisation de **Dispose** et de **Finalize** pour libérer des ressources :

```
Class BaseType
   Implements IDisposable
   Public Sub Dispose() Implements IDisposable.Dispose
' Place cleanup code here.
   End Sub
End Class

Class ResourceWrapper
   Inherits BaseType
   Implements IDisposable

   Private handle As IntPtr ' Pointer to an external resource.
   Private otherRes As OtherResource ' Other resource you happen to use.
```

```
  Private disposed As Boolean = False

  Public Sub New()
    handle = ' Resource allocated by call to unmanaged API.
    otherRes = New OtherResource() ' Object that wraps a resource.
  End Sub

  ' Free your own state.
  Private Sub freeState()
     If Not disposed Then
        CloseHandle(handle)
        disposed = True
     End If
  End Sub

  ' Free your own state, call Dispose on all held states, and leave
  ' the finalization queue.
  Public Overrides Sub Dispose() Implements IDisposable.Dispose
     freeState()
     otherRes.Dispose()
     MyBase.Dispose() ' If base type implements Dispose, call it.
     ' Calling GC.SuppressFinalize in Dispose is an important
     ' optimization because it ensures that resources are
     ' released promptly.
     GC.SuppressFinalize(Me)
  End Sub

  ' Free your own state (NOT other state you hold) and give the
  ' parent class a chance to finalize.
  Overloads Overrides Sub Finalize()
     freeState()
     MyBase.Finalize()
  End Sub

  ' Whenever you do something with this class, check to see if the
  ' state is disposed. If so, throw this exception.
  Public Sub SimpleDoStuff()
     If disposed Then
        Throw ObjectDisposedException("ResWrapper")
     End If
  End Sub
End Class
```

L'exemple suivant montre un modèle de design courant d'utilisation du destructeur **Dispose** :

```
Dim con As Connection, rs As RecordSet
Try
  con = New Connection("xyz")
```

```
  rs = New RecordSet(con, "MyTable")
  ' Use the connection if successful.
Finally
' Call the Dispose method when done with any created objects.
If Not con Is Nothing Then
    con.Dispose()
  End If
  If Not rs Is Nothing Then
    rs.Dispose()
  End If
End Try
```

L'exemple suivant crée un objet à l'aide d'un constructeur paramétré et appelle des destructeurs une fois que l'objet a perdu son utilité :

```
Sub TestConstructorsAndDestructors()
  Dim X As Integer = 6
  Dim AnObject As New ThisClass(X)
  ' Place statements here that use the object.
  AnObject.DoSomething()
  ' Test the parameterized constructor.
  MsgBox("The value of ThisProperty after being initialized " & _
       " by the constructor is " & AnObject.ThisProperty)
  ' Run the Dispose method when you are done with the object.
  AnObject.Dispose()
End Sub

  Public Class BaseClass
    Sub New()
      MsgBox("The base class is initializing with Sub New.")
    End Sub

    Protected Overrides Sub Finalize()
      MsgBox("The base class is destroying objects with Sub Finalize.")
      ' Place final cleanup tasks here.
      MyBase.Finalize()
    End Sub
  End Class

  Public Class ThisClass
    Inherits BaseClass
    Implements IDisposable ' Implements the Dispose method of IDisposable.
    Private m_PropertyValue  As Integer

    Sub New(ByVal SomeValue As Integer)
      MyBase.New() ' Call MyBase.New if this is a derived class.
      MsgBox("Sub New is initializing the class ThisClass.")
      ' Place initialization statements here.
      m_PropertyValue  = SomeValue
```

```
    End Sub

    Property ThisProperty() As Integer
      Get
        ThisProperty = m_PropertyValue
      End Get
      Set(ByVal Value As Integer)
        m_PropertyValue  = Value
      End Set
    End Property

    Sub DoSomething()
      ' Place code here that does some task.
    End Sub

    Protected Overrides Sub Finalize()
      MsgBox("Finalize is destroying objects in ThisClass.")
      ' Place final cleanup tasks here.
      MyBase.Finalize()
    End Sub

    Overridable Sub Dispose() Implements IDisposable.Dispose
      MsgBox("The ThisClass is running Dispose.")
    End Sub
  End Class
```

Lorsque vous exécutez cet exemple, la classe ThisClass appelle le constructeur **Sub New** de la classe BaseClass. Au terme de l'exécution du constructeur de la classe de base, la classe ThisClass exécute les instructions restantes de **Sub New** qui initialisent une valeur pour la propriété ThisProperty.

Quand la classe n'est plus nécessaire, le destructeur **Dispose** est appelé dans ThisClass.

Si vous avez créé au départ l'instance de ThisClass à partir d'un formulaire, rien ne semble se produire tant que vous ne fermez pas le formulaire. A ce moment, le destructeur **Finalize** s'exécute dans la classe ThisClass et enfin dans la classe BaseClass.

# Propriétés, champs et méthodes de classe

En général, les champs et les propriétés représentent des informations au sujet d'un objet, tandis que les méthodes représentent des actions qu'un objet peut effectuer.

Les rubriques suivantes décrivent les mécanismes d'ajout de propriétés, de champs et de méthodes aux classes, et traitent de questions relatives à ces éléments.

**Dans cette section**

Ajout de champs et de propriétés à une classe
: Fournit des informations sur la déclaration de champs et de propriétés.

Propriétés et procédures Property
: Explique le fonctionnement des procédures **Property** et la manière d'implémenter des types de propriété courants.

Procédures Property et champs
: Explique quand il faut utiliser des champs pour stocker des données dans une classe et quand les propriétés se révèlent une meilleure option.

Méthodes de classe
: Analyse les procédures publiques que vous ajoutez à une classe.

Propriétés et méthodes
: Vous aide à déterminer si la fonctionnalité que vous désirez est mieux implémentée sous la forme d'une propriété ou d'une méthode.

Propriétés par défaut
: Traite de propriétés utilisées quand aucune propriété spécifique n'est nommée.

Propriétés et méthodes surchargées
: Explique la définition de plusieurs propriétés ou méthodes de même nom mais utilisant des types de données différents.

Substitution de propriétés et de méthodes
: Explique comment redéfinir une méthode ou une propriété héritée.

Gestion d'objet avec des collections
: Fournit des informations sur l'utilisation de collections pour stocker et extraire des objets.

# Evénements et délégués

Les événements et les délégués sont étroitement associés dans Visual Basic .NET. Un événement est un message envoyé par un objet qui annonce que quelque chose d'important s'est produit. Les événements sont implémentés à l'aide de délégués, qui sont une forme de pointeur fonction orienté objet permettant à une fonction d'être appelée indirectement par l'intermédiaire d'une référence à la fonction.

**Dans cette section**

Evénements et gestionnaires d'événements
: Propose une vue d'ensemble des événements, des gestionnaires d'événements et de la terminologie associée.

Ajout d'événements à une classe
: Décrit le processus de définition des événements.

Ecriture de gestionnaires d'événements
: Montre comment écrire des gestionnaires d'événements à l'aide de la clause **Handles** ou de l'instruction **AddHandler**.

WithEvents et la clause Handles
: Indique comment utiliser le mot clé **WithEvents** et la clause **Handles** pour associer un événement à un gestionnaire d'événements.

AddHandler et RemoveHandler
: Indique comment utiliser les instructions **AddHandler** et **RemoveHandler** pour associer de manière dynamique un événement à un gestionnaire d'événements.

Exemple de gestionnaire d'événements
: Définit une classe qui déclenche un événement lorsque vous appelez la méthode **CauseEvent**.

Délégués et opérateur AddressOf
: Offre une introduction aux délégués et à leur utilisation.

Exemple de délégué
: Crée une procédure de tri dans une classe qui utilise le tri alphabétique standard dans la plupart des zones de liste mais qui est capable, au moment de l'exécution, de passer à la procédure de tri personnalisée. Pour ce faire, vous faites appel à des délégués pour passer la procédure de tri personnalisée à la classe de tri au moment de l'exécution.

Procédure pas à pas : déclaration et déclenchement des événements
: Vous guide pas à pas dans le processus de déclaration et de déclenchement des événements pour une classe.

Procédure pas à pas : gestion des événements

Montre comment écrire une procédure gestionnaire d'événements.

**Rubriques connexes**

~~Gestion et déclenchement des événements~~

Propose une vue d'ensemble du modèle d'événement dans le .NET Framework.

~~Gestion des événements dans les Windows Forms~~

Décrit comment utiliser des événements associés aux objets Windows Forms.

# Evénements et gestionnaires d'événements

Il peut être tentant de comparer un projet Visual Studio à une série de procédures qui s'exécutent dans l'ordre. En réalité, la plupart des programmes sont pilotés par des événements, c'est-à-dire que le déroulement de l'exécution dépend d'occurrences externes que l'on appelle événements.

Un événement est un signal qui informe l'application que quelque chose d'important s'est produit. Par exemple, lorsqu'un utilisateur clique sur un contrôle ou un formulaire, le formulaire peut déclencher un événement **Click** et appeler une procédure qui gère l'événement. Les événements séparent également les tâches pour communiquer. Supposons, par exemple, que votre application exécute une tâche de tri séparément de l'application principale. Si un utilisateur annule le tri, votre application peut envoyer un événement d'annulation demandant au processus de tri de s'interrompre.

## ~~Termes et concepts relatifs aux événements~~

Cette section décrit les termes et concepts utilisés avec les événements dans Visual Basic .NET.

### Déclaration d'événements

Vous déclarez des événements dans le cadre de classes, structures, modules et interfaces à l'aide du mot clé **Event**, comme dans l'exemple suivant :

```
Event AnEvent(ByVal EventNumber As Integer)
```

### Déclenchement d'événements

Un événement est comparable à un message annonçant que quelque chose d'important s'est produit. L'acte de diffusion du message est appelé déclenchement de l'événement. Dans Visual Basic .NET, vous déclenchez des événements à l'aide de l'instruction **RaiseEvent**, comme dans l'exemple suivant :

```
RaiseEvent AnEvent(EventNumber)
```

Les événements doivent être déclenchés dans la portée dans laquelle ils sont déclarés. Par exemple, une classe dérivée ne peut pas déclencher d'événements hérités à partir d'une classe de base.

## Emetteurs d'événements

Tout objet capable de déclencher un événement est un émetteur d'événements, également appelé source d'événements. Les formulaires, contrôles et objets définis par l'utilisateur sont des exemples d'émetteurs d'événements.

## Gestionnaires d'événements

Les gestionnaires d'événements sont des procédures appelées lorsqu'un événement correspondant se produit. Vous pouvez utiliser n'importe quelle sous-routine valide comme gestionnaire d'événements. Il vous est cependant impossible d'utiliser une fonction en tant que gestionnaire d'événements parce qu'une fonction ne peut retourner une valeur à la source des événements.

Visual Basic utilise pour les gestionnaires d'événements une convention d'affectation de noms standard qui associe le nom de l'émetteur de l'événement, un trait de soulignement et le nom de l'événement. Par exemple, l'événement click d'un bouton appelé `button1` aurait pour nom `Sub button1_Click`.

> **Remarque** Il est recommandé (mais non obligatoire) d'utiliser cette convention d'affectation de noms lorsque vous définissez des gestionnaires d'événements pour vos propres événements ; vous pouvez utiliser n'importe quel nom de sous-routine valide.

## Association d'événements aux gestionnaires d'événements

Avant de pouvoir utiliser un gestionnaire d'événements, vous devez au préalable l'associer à un événement à l'aide de l'instruction **Handles** ou **AddHandler**.

L'instruction **WithEvents** et la clause **Handles** assurent une méthode déclarative de spécification des gestionnaires d'événements. Les événements déclenchés par un objet déclaré par **WithEvents** peuvent être gérés par n'importe quelle sous-routine avec une clause **Handles** qui nomme ces événements. Même si la clause **Handles** constitue la méthode standard d'association d'un événement à un gestionnaire d'événements, son action est restreinte à l'association d'événements aux gestionnaires d'événements au moment de la compilation.

Les instructions **AddHandler** et **RemoveHandler** sont plus souples à utiliser que la clause **Handles**. Elles vous permettent de connecter et déconnecter dynamiquement les événements par rapport à au moins un gestionnaire d'événements au moment de l'exécution et ne vous obligent pas à déclarer des variables d'objets à l'aide de **WithEvents**.

Dans certains cas, par exemple dans celui d'événements associés à des formulaires ou contrôles, Visual Basic .NET choisit automatiquement un gestionnaire d'événements vide et l'associe à un événement. Par exemple, lorsque vous double-cliquez sur un bouton de commande dans un formulaire en mode design, Visual Basic .NET crée un gestionnaire d'événements vide et une variable **WithEvents** pour le bouton de commande, comme dans le code suivant :

```
Friend WithEvents Button1 As System.Windows.Forms.Button
Protected Sub Button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button1.Click
End Sub
```

# Ajout d'événements à une classe

Vous ajoutez des événements à une classe en les déclarant avec l'instruction **Events**. La déclaration comprend le nom de l'événement et les arguments qu'il utilise.

**Pour ajouter un événement à une classe**

- Dans la section Declarations du module de classe qui définit la classe, utilisez l'instruction **Event** pour déclarer l'événement avec tous les arguments que vous souhaitez qu'il comporte. Par exemple :
- 
```
Public Event PercentDone(ByVal Percent As Single, _
  ByRef Cancel As Boolean)
```

Les événements ne peuvent avoir de valeurs de retour, d'arguments facultatifs ou d'arguments **ParamArray**.

L'ajout d'un événement à une classe permet de spécifier qu'un objet de cette classe peut déclencher un événement spécifique. Pour provoquer l'occurrence effective d'un événement, vous devez utiliser l'instruction **RaiseEvent**. Vous pouvez utiliser le mot clé **Handles** ou l'instruction **AddHandler** pour associer l'événement à une procédure de gestionnaire d'événements. Les événements doivent être déclenchés dans la portée dans laquelle ils sont déclarés. Par exemple, une classe dérivée ne peut pas déclencher d'événements hérités à partir d'une classe de base.

# Ecriture de gestionnaires d'événements

**Voir aussi**

Handles | AddHandler, instruction | Evénements et gestionnaires d'événements | Délégués et opérateur AddressOf | AddHandler et RemoveHandler | Ecriture de gestionnaires d'événements

La façon dont vous construisez un gestionnaire d'événements dépend de celle dont vous voulez l'associer aux événements. La méthode standard de création d'un gestionnaire d'événements consiste à utiliser le mot clé **Handles** avec le mot clé **WithEvents**. Visual Basic .NET offre une deuxième méthode de gestion des événements : l'instruction **AddHandler**. **AddHandler** et **RemoveHandler** permettent de démarrer et d'interrompre dynamiquement la gestion des événements pour un événement spécifique. Vous pouvez utiliser l'une ou l'autre de ces méthodes, mais pas à la fois **WithEvents** et **AddHandler** avec le même événement.

Gestion des événements à l'aide de WithEvents

Le mot clé **WithEvents** vous permet de créer des variables d'objets au niveau de la classe ou du module qui sont utilisables avec la clause **Handles** dans les gestionnaires d'événements.

**Pour gérer les événements à l'aide de WithEvents et de la clause Handles**

. Dans la section Declarations du module destiné à gérer l'événement, utilisez le mot clé **WithEvents** pour déclarer une variable d'objet pour la source de vos événements, conformément à l'exemple suivant :

```
Public WithEvents ClassInst As Class1
```

. Dans l'Editeur de code, choisissez la variable **WithEvents** que vous venez de déclarer dans la liste déroulante **Nom de la classe** située à gauche.
. Choisissez l'événement que vous souhaitez gérer dans la liste déroulante **Nom de la méthode** située à droite. L'Editeur de code crée la procédure gestionnaire d'événements vide avec une clause **Handles**.

> **Remarque** Cette étape est facultative. Vous pouvez créer manuellement la procédure gestionnaire d'événements à

condition que la procédure créée soit une sous-routine, contienne la liste d'arguments correspondant à l'événement géré et comporte une clause **Handles** spécifiant l'événement géré.

. Ajoutez le code de gestion des événements à la procédure gestionnaire d'événements à l'aide des arguments fournis. Le code suivant en est un exemple :

```
. Public Sub ClassInst_AnEvent(ByVal EventNumber As System.Integer) _
.     Handles ClassInst.AnEvent
.   MessageBox.Show("Received event number: " & CStr(EventNumber))
End Sub
```

## Gestion des événements à l'aide de AddHandler

Vous pouvez utiliser l'instruction **AddHandler** pour connecter dynamiquement les événements par rapport aux procédures gestionnaires d'événements.

**Pour gérer des événements à l'aide de AddHandler**

. Déclarez une variable d'objet de la classe qui est la source des événements que vous souhaitez gérer. Contrairement à une variable **WithEvents**, il peut s'agir d'une variable locale dans une procédure ; par exemple :

```
Dim CI As New Class1()
```

. Utilisez l'instruction **AddHandler** pour spécifier le nom de l'émetteur d'événements et l'instruction **AddressOf** pour fournir le nom de votre gestionnaire d'événements ; par exemple :

```
AddHandler CI.AnEvent, AddressOf EHandler
```

N'importe quelle procédure peut servir de gestionnaire d'événements si elle prend en charge les bons arguments pour l'événement géré.

. Ajoutez le code au gestionnaire d'événements, comme dans l'exemple suivant :

```
. Public Sub EHandler(ByVal EventNumber As Integer)
.   MessageBox.Show("Received event number " & CStr(EventNumber))
End Sub
```

## Utilisation de RemoveHandler pour arrêter la gestion des événements

Vous pouvez utiliser l'instruction **RemoveHandler** pour déconnecter dynamiquement les événements à partir des procédures gestionnaires d'événements.

**Pour arrêter la gestion des événements en utilisant RemoveHandler**

. Utilisez l'instruction **RemoveHandler** pour spécifier le nom de l'émetteur d'événements et l'instruction **AddressOf** pour fournir le nom de votre gestionnaire d'événements. La syntaxe des instructions **RemoveHandler** correspondra toujours étroitement à l'instruction **AddHandler** utilisée pour démarrer la gestion des événements ; par exemple :

```
RemoveHandler CI.AnEvent, AddressOf EHandler
```

## Gestion d'événements hérités d'une classe de base

Les classes dérivées, classes qui héritent les caractéristiques d'une classe de base, peuvent gérer des événements déclenchés par leur classe de base à l'aide de l'instruction **Handles MyBase**.

**Pour gérer les événements d'une classe de base**

- Déclarez un gestionnaire d'événements dans la classe dérivée en ajoutant une instruction **Handles MyBase.<event name>** à la ligne de déclaration de votre procédure gestionnaire d'événements, où **event name** est le nom de l'événement dans la classe de base gérée. Par exemple :

```
Public Class Class1
  Public Event SE(ByVal i As Integer)
  ' Place methods and properties here.
End Class

Public Class Class2
  Inherits Class1
  Sub EventHandler(ByVal x As Integer) Handles MyBase.SE
  ' Place code to handle events from Class1 here.
  End Sub
End Class
```

# WithEvents et la clause Handles

**Voir aussi**

~~AddHandler et RemoveHandler~~ | Handles | WithEvents

L'instruction **WithEvents** et la clause **Handles** assurent une méthode déclarative de spécification des gestionnaires d'événements. Les événements déclenchés par un objet déclaré par le mot clé **WithEvents** peuvent être gérés par n'importe quelle procédure avec une instruction **Handles** pour ces événements, comme le montre l'exemple suivant :

```
Dim WithEvents EClass As New EventClass() ' Declare a WithEvents variable.
Sub TestEvents()
  EClass.RaiseEvents()
End Sub
' Declare an event handler that handles multiple events.
Sub EClass_EventHandler() Handles Eclass.XEvent, Eclass.YEvent
  MsgBox("Received Event.")
End Sub

Class EventClass
  Public Event XEvent()
  Public Event YEvent()
  Sub RaiseEvents() 'Raises two events handled by EClass_EventHandler.
    RaiseEvent XEvent()
    RaiseEvent YEvent()
  End Sub
End Class
```

L'instruction **WithEvents** et la clause **Handles** représentent souvent la meilleure solution pour les gestionnaires d'événements, parce que la syntaxe déclarative qu'ils utilisent simplifie beaucoup le codage, la lecture et le débogage de la gestion des événements. Il faut cependant garder à l'esprit les restrictions liées à l'utilisation des variables **WithEvents** :

- Vous ne pouvez pas utiliser une variable **WithEvents** comme variable d'objet générique. Cela signifie que vous ne pouvez pas la déclarer en tant que **Object** ; vous devez spécifier le nom de la classe lorsque vous déclarez la variable.
- Vous ne pouvez pas utiliser **WithEvents** pour gérer de manière déclarative des événements partagés, puisque ces derniers ne sont pas liés à une instance qui peut être assignée à une variable **WithEvents**.
- Vous ne pouvez pas créer de tableaux de variables **WithEvents**.
- Les variables **WithEvents** autorisent un seul gestionnaire d'événements à gérer un ou plusieurs types d'événements ou un ou plusieurs gestionnaires d'événements à gérer le même type d'événements.

# AddHandler et RemoveHandler

**Voir aussi**

AddHandler, instruction | Evénements et gestionnaires d'événements | WithEvents et la clause Handles | Ecriture de gestionnaires d'événements

L'instruction **AddHandler** est comparable à la clause **Handles** dans la mesure où toutes deux vous permettent de spécifier un gestionnaire d'événements destiné à gérer un événement. Cependant, **AddHandler** et **RemoveHandler** offrent une plus grande souplesse que la clause **Handles** ; ils vous permettent d'ajouter, de supprimer et de changer dynamiquement le gestionnaire d'événement associé à un événement. En outre, contrairement à **Handles**, **AddHandler** permet d'associer plusieurs gestionnaires d'événements à un seul événement.

**AddHandler** prend en compte deux arguments : le nom d'un événement issu d'un émetteur d'événements tel qu'un contrôle et une expression correspondant à un délégué. Comme l'instruction **AddressOf** retourne toujours une référence au délégué, vous n'êtes pas obligé de spécifier explicitement la classe déléguée lorsque vous utilisez **AddHandler**. L'exemple ci-dessous associe un gestionnaire d'événements à un événement déclenché par un objet :

```
AddHandler MyObject.Event1, AddressOf Me.MyEventHandler
```

**RemoveHandler**, qui déconnecte un événement d'un gestionnaire d'événements, utilise la même syntaxe que **AddHandler**. Par exemple :

```
RemoveHandler MyObject.Event1, AddressOf Me.MyEventHandler
```

# Exemple de gestionnaire d'événements

**Voir aussi**

Evénements et gestionnaires d'événements | Délégués et opérateur AddressOf | AddHandler et RemoveHandler | Ecriture de gestionnaires d'événements

L'exemple suivant définit une classe qui déclenche un événement lorsque vous appelez la méthode CauseEvent. Cet événement est géré par une procédure gestionnaire d'événements appelée EventHandler.

Pour exécuter cet exemple, ajoutez le code ci-dessous à la classe Form d'un projet d'application Windows Visual Basic .NET et appelez la procédure TestEvents avec un argument de type nombre entier.

```
Public Class Class1
  ' Declare an event for this class.
  Public Event Event1(ByVal EventNumber As Integer)
  ' Define a method that raises an event.
  Sub CauseEvent(ByVal EventNumber As Integer)
    RaiseEvent Event1(EventNumber)
  End Sub
End Class

Protected Sub TestEvents(ByVal EventNumber As Integer)
  Dim MyObject As New Class1()
  AddHandler MyObject.Event1, AddressOf Me.EventHandler
  ' Cause the object to raise an event.
  MyObject.CauseEvent(EventNumber)
End Sub

Sub EventHandler(ByVal EventNumber As Integer)
  MessageBox.Show("Received event number " & CStr(EventNumber))
End Sub
```

# Délégués et opérateur AddressOf

**Voir aussi**

Delegate, instruction | AddressOf, opérateur | ~~Evénements et gestionnaires d'événements~~ | ~~AddHandler et RemoveHandler~~ | ~~Ecriture de gestionnaires d'événements~~ | ~~Applications multithread~~

Les délégués sont des objets que vous pouvez utiliser pour appeler les méthodes d'autres objets. Ils sont parfois décrits comme des pointeurs fonction de type sécurisé parce qu'ils sont comparables aux pointeurs fonction utilisés dans d'autres langages de programmation. Cependant, contrairement aux pointeurs fonction, les délégués Visual Basic .NET constituent un type référence basé sur la classe **System.Delegate** et peuvent référencer à la fois les méthodes

partagées, méthodes qui peuvent être appelées sans une instance spécifique de classe, et les méthodes d'instance.

Les délégués sont utiles lorsque vous avez besoin d'un intermédiaire entre une procédure appelante et la procédure appelée. Vous pouvez, par exemple, souhaiter un objet qui déclenche des événements pour pouvoir appeler différents gestionnaires d'événements dans des circonstances différentes. Malheureusement, l'objet qui déclenche les événements ne peut pas identifier à l'avance le gestionnaire d'événements qui gère un événement spécifique. Visual Basic .NET vous permet d'associer dynamiquement les gestionnaires d'événements aux événements en créant un délégué lorsque vous utilisez l'instruction **AddHandler**. Au moment de l'exécution, le délégué transmet les appels au gestionnaire d'événements approprié.

Même s'il vous est tout à fait possible de créer vos propres délégués, c'est Visual Basic .NET qui se charge le plus souvent de créer le délégué et de gérer tous les détails à votre place. Par exemple, une instruction **Event** définit implicitement une classe déléguée appelée <EventName>EventHandler en tant que classe imbriquée de la classe contenant l'instruction **Event**, ce avec la même signature que l'événement. L'instruction **AddressOf** crée implicitement une instance de délégué. Par exemple, les deux lignes de code suivantes sont équivalentes :

```
AddHandler Button1.Click, AddressOf Me.Button1_Click
  ' The previous line of code is shorthand for the next line of code.
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
```

Vous pouvez utiliser la méthode abrégée de création de délégués à chaque fois que le compilateur peut déterminer le type du délégué en fonction du contexte.

## Déclaration d'événements utilisant un type délégué existant

Dans certaines situations, vous pouvez choisir de déclarer un événement pour utiliser un type délégué existant comme son délégué sous-jacent. La syntaxe ci-dessous vous indique comment procéder :

```
Event AnEvent As DelegateType
```

Cela peut vous être utile lorsque vous voulez router plusieurs événements vers le même gestionnaire.

# Exemple de délégué

**Voir aussi**

Delegate, instruction | ~~Délégués et opérateur AddressOf~~ | ~~Evénements et gestionnaires d'événements~~ | ~~AddHandler et RemoveHandler~~ | ~~Ecriture de gestionnaires d'événements~~ | ~~Applications multithread~~

Vous pouvez utiliser des délégués pour d'autres tâches non liées à des événements, par exemple pour le traitement dissocié des threads ou pour des procédures impliquant d'appeler différentes versions de fonctions au moment de la compilation.

Supposons, par exemple, que vous ayez une application de gestion de petites annonces contenant une zone de liste affichant les noms de voitures. Les annonces sont triées par titre, lequel correspond normalement à la marque de la voiture. Un problème peut néanmoins se poser lorsque, pour certaines voitures, la marque est précédée de l'année de mise en circulation. Ce cas représente un vrai problème, car la fonctionnalité de tri intégrée de la zone de liste base son tri sur les codes de caractères ; elle commence donc par trier en tête de liste toutes les annonces dont l'intitulé commence par des dates pour ensuite traiter les annonces dont l'intitulé commence par la marque.

Pour résoudre ce problème, vous pouvez créer une procédure de tri dans une classe qui utilise le tri alphabétique standard dans la plupart des zones de liste mais qui est capable, au moment de l'exécution, de passer à la procédure de tri personnalisée définie pour les petites annonces concernant les voitures. Pour ce faire, vous faites appel à des délégués pour passer la procédure de tri personnalisée à la classe de tri au moment de l'exécution.

Chaque classe déléguée définit un constructeur auquel la spécification d'une méthode objet est passée. Le code ci-dessous fournit un exemple de paramètre pour un constructeur délégué de ce type :

```
AddressOf [<expression>.]<method name>
```

Le type de moment de compilation de l'expression doit correspondre à une classe ou à une interface avec une méthode portant le nom spécifié dont la signature correspond à la signature de la classe déléguée. La méthode peut être soit une méthode (classe) partagée, soit une méthode d'instance.

Vous appelez la méthode associée à une instance de délégué en appelant sa méthode intégrée **Invoke**, comme le montre l'exemple suivant :

```
Delegate Sub MySubDelegate(ByVal x As Integer)
Protected Sub Test()
  Dim c2 As New class2()
  ' Test the delegate.
  c2.DelegateTest()
End Sub

Class class1
  Sub Sub1(ByVal x As Integer)
    MessageBox.Show("The value of x is: " & CStr(x))
  End Sub
End Class

Class class2
  Sub DelegateTest()
    Dim c1 As Class1
    Dim msd As MySubDelegate
    c1 = New Class1()
    ' Create an instance of the delegate.
    msd = AddressOf c1.Sub1
    msd.Invoke(10) ' Call the method.
  End Sub
End Class
```

# Interfaces dans Visual Basic .NET

Les interfaces représentent les propriétés, méthodes et événements qui peuvent être implémentés par les classes. Les interfaces vous permettent de définir des fonctionnalités sous forme de petits groupes de propriétés, méthodes et événements étroitement liés ; les risques d'incompatibilité s'en trouvent limités parce que vous avez la possibilité de développer des implémentations avancées sans détruire le code existant. Vous pouvez ajouter de nouvelles fonctionnalités à tout moment en développant des interfaces et implémentations supplémentaires.

Les versions de Visual Basic antérieures à Visual Basic .NET étaient capables d'utiliser des interfaces mais pas de les créer directement. Visual Basic .NET introduit l'instruction **Interface**, qui permet de définir de vraies interfaces en tant qu'entités distinctes des classes et de les implémenter avec une version améliorée du mot clé **Implements**.

**Dans cette section**

Vue d'ensemble des interfaces
: Offre une vue d'ensemble des interfaces et de la façon dont Visual Basic .NET les implémente.

Définition des interfaces
: Décrit comment définir les interfaces avec les instructions **Interface** et **End Interface**.

Implements, mot clé
: Décrit comment indiquer qu'une section de code implémente une interface spécifique.

Exemples d'implémentation d'interfaces
: Fournit trois exemples d'implémentation d'interface.

Quand utiliser des interfaces
: Explique dans quels cas il convient d'utiliser des interfaces au lieu d'une hiérarchie d'héritage.

Création et implémentation d'une interface
: Indique les étapes intervenant dans la définition et l'implémentation d'une interface et en fournit des exemples.

Procédure pas à pas : création et implémentation d'interfaces
: Fournit une procédure détaillée qui vous fait découvrir le processus de définition et d'implémentation de votre propre interface.

# Vue d'ensemble des interfaces

Tout comme les classes, les interfaces définissent un ensemble de propriétés, méthodes et événements. Cependant, contrairement aux classes, les interfaces n'assurent pas l'implémentation. Elles sont implémentées par les classes et définies en tant qu'entités distinctes des classes.

Une interface représente un contrat, dans le sens où une classe qui implémente une interface doit implémenter tous les aspects de cette interface exactement telle qu'elle a été définie.

Avec les interfaces, vous pouvez définir les fonctionnalités en tant que petits groupes de membres étroitement liés. Vous pouvez développer des implémentations avancées pour vos interfaces sans endommager le code existant, ce qui limite les problèmes de compatibilité. Vous pouvez également ajouter de nouvelles fonctionnalités à tout moment en développant des interfaces et implémentations supplémentaires.

Même si les implémentations d'interfaces peuvent évoluer, les interfaces elles-mêmes ne peuvent pas être modifiées après publication. En effet, toute modification apportée à une interface publiée risque d'endommager le code existant. Si vous partez du principe qu'une interface est une sorte de contrat, il apparaît clairement que les deux parties liées par ce contrat ont un rôle à jouer. La partie qui publie une interface accepte de ne jamais modifier cette dernière et l'implémenteur accepte de l'implémenter exactement comme elle a été conçue.

Les versions de Visual Basic antérieures à Visual Basic .NET étaient capables d'utiliser des interfaces mais pas de les créer directement. Visual Basic .NET vous permet de définir de vraies interfaces à l'aide de l'instruction **Interface** et de les implémenter avec une version améliorée du mot clé **Implements**.

# Définition des interfaces

Les définitions d'interfaces sont comprises dans les instructions **Interface** et **End Interface**. A la suite de l'instruction **Interface**, vous pouvez ajouter une instruction **Inherits** facultative qui répertorie une ou plusieurs interfaces héritées. Les instructions **Inherits** doivent précéder toutes les autres instructions dans la déclaration, à l'exception des commentaires. Les autres instructions figurant dans la définition de l'interface doivent être **Event**, **Sub**, **Function** et **Property**. Les interfaces ne peuvent pas contenir de code d'implémentation ou d'instructions associées au code d'implémentation, telles que **End Sub** ou **End Property**.

Les instructions présentes dans les interfaces sont publiques par défaut, mais elles peuvent également être explicitement déclarées comme **Public**, **Friend**, **Protected** ou **Private**.

> **Remarque** Les seuls modificateurs valides pour les instructions **Sub**, **Function** ou **Property** déclarées dans une définition d'interface sont les mots clés **Overloads** et **Default**. Aucun autre modificateur — **Public**, **Private**, **Friend**, **Protected**, **Shared**, **Static**, **Overrides**, **MustOverride** ou **Overridable** — n'est autorisé.

# Implements, mot clé

Le mot clé **Implements** permet d'indiquer qu'un membre d'une classe implémente une interface spécifique.

L'implémentation d'une instruction **Implements** suppose l'existence d'une liste des membres d'interface ayant la virgule comme séparateur. En général, un seul membre d'interface est spécifié, mais vous pouvez tout à fait spécifier plusieurs membres. La spécification d'un membre d'interface se présente sous la forme du nom de l'interface (qui doit être spécifié dans une instruction implements dans la classe), suivi d'un point et du nom de la fonction membre, de la propriété ou de l'événement à implémenter. Le nom d'un membre qui implémente un membre d'interface peut utiliser n'importe quel identificateur conforme et ne doit pas obligatoirement respecter strictement la convention **InterfaceName_MethodName** utilisée dans les versions antérieures de Visual Basic. Par exemple, le fragment de code ci-dessous indique comment déclarer une sous-routine appelée Sub1 qui implémente une méthode d'une interface :

```
Sub Sub1(ByVal i As Integer) Implements interfaceclass.interface2.Sub1
```

Les types de paramètres et types de retour du membre qui implémente doivent correspondre à la propriété d'interface ou à la déclaration de membre dans l'interface. La façon la plus répandue d'implémenter un élément d'interface consiste à utiliser un membre portant le même nom que l'interface, comme dans l'exemple précédent.

Pour déclarer l'implémentation d'une méthode d'interface, vous pouvez utiliser tous les attributs conformes dans les déclarations de méthodes d'instances, notamment **Overloads**, **Overrides**, **Overridable**, **Public**, **Private**, **Protected**, **Friend**, **Protected Friend**, **MustOverride**, **Default** et **Static**. L'attribut **Shared** n'est pas conforme parce qu'il définit une classe au lieu d'une méthode d'instance.

**Implements** permet également d'écrire une seule méthode implémentant plusieurs méthodes définies dans une interface, comme dans l'exemple suivant :

```
Protected Sub M1 Implements I1.M1, I1.M2, I2.M3, I2.M4
```

Vous pouvez utiliser un membre privé pour implémenter un membre de classe. Lorsqu'un membre privé implémente un membre d'interface, ce membre devient disponible via l'interface, même s'il n'est pas directement disponible avec les variables objets pour la classe.

# Exemples d'implémentation d'interfaces

Les classes qui implémentent une interface doivent implémenter toutes ses propriétés, méthodes et événements.

L'exemple à suivre définit deux interfaces. La seconde interface, `Interface2`, hérite de `Interface1` et définit une propriété et une méthode supplémentaires.

```
Interface Interface1
   Sub sub1(ByVal i As Integer)
End Interface
```

```
Interface Interface2
   Inherits Interface1 ' Demonstrates interface inheritance.
   Sub M1(ByVal y As Integer)
   ReadOnly Property Num() As Integer
End Interface
```

L'exemple suivant implémente Interface1, l'interface définie dans l'exemple précédent :

```
Public Class ImplementationClass1
  Implements Interface1
  Sub Sub1(ByVal i As Integer) Implements Interface1.Sub1
     ' Place code here to implement this method.
   End Sub
End Class
```

Ce dernier exemple implémente Interface2, y compris une méthode héritée de Interface1 :

```
Public Class ImplementationClass2
  Implements Interface2
  Dim INum As Integer = 0
  Sub sub1(ByVal i As Integer) Implements Interface2.Sub1
     ' Place code here that implements this method.
  End Sub
  Sub M1(ByVal x As Integer) Implements Interface2.M1
     ' Place code here to implement this method.
  End Sub

  ReadOnly Property Num() As Integer Implements _
     Interface2.Num
     Get
       Num = INum
     End Get
  End Property
End Class
```

# Quand utiliser des interfaces

Les interfaces représentent de puissants outils de programmation parce qu'elles permettent de séparer la définition des objets de leur implémentation ; vos

objets peuvent ainsi évoluer sans risque d'endommagement des applications existantes. Les interfaces et l'héritage de classes présentent tous deux des avantages et des inconvénients, ce qui peut vous amener à associer l'utilisation des deux dans vos projets. Cette rubrique ainsi que ~~Quand utiliser l'héritage~~ peuvent vous aider à identifier l'approche la mieux adaptée à votre situation.

## ~~Comment modifier le code sans risque~~

Les interfaces éliminent un problème important posé par l'héritage de classes : le risque d'endommagement du code lorsque vous modifiez le design de votre application après son implémentation. Même si l'héritage de classes permet à vos classes d'hériter l'implémentation d'une classe de base, elle vous contraint également à prendre toutes vos décisions de design au moment de la première publication de la classe. Si vos hypothèses de base s'avèrent être incorrectes, vous ne pouvez pas toujours modifier votre code sans risque dans les versions plus récentes.

Supposons, par exemple, que vous définissiez une méthode de classe de base qui requiert un argument **Integer**, puis que vous décidiez plus tard que cet argument doit être du type de données **Long**. Il est risqué de modifier la classe d'origine, car les applications conçues pour les classes dérivées de la classe d'origine risquent de ne pas être compilées correctement. Ce problème peut avoir des répercussions importantes, dans la mesure où une seule classe de base peut affecter des centaines de sous-classes.

Il existe une solution consistant à définir une nouvelle méthode qui surcharge la classe d'origine et prend en compte un argument de type **Long**. Cependant, cette solution peut très bien ne pas convenir, car une classe dérivée peut devoir se substituer à la méthode qui prend en compte le nombre entier et risque de ne pas fonctionner correctement si la méthode qui accepte un argument **Long** n'est pas également substituée. Les interfaces résolvent ce problème en vous donnant la possibilité de publier une interface mise à jour qui accepte le nouveau type de données.

## ~~Souplesse d'implémentation~~

Il existe plusieurs autres raisons qui peuvent vous amener à préférer les interfaces à l'héritage de classes :

- Les interfaces sont mieux adaptées aux applications exigeant la création de nombreux types d'objets pas nécessairement liés pour assurer certaines fonctionnalités.

- Les interfaces sont plus souples à utiliser que les classes de base, car il suffit de définir une seule implémentation pour pouvoir implémenter plusieurs interfaces.
- Les interfaces sont plus indiquées lorsque vous n'avez pas besoin d'hériter une implémentation d'une classe de base.
- Les interfaces sont utiles lorsque vous ne pouvez pas utiliser l'héritage de classes. Par exemple, les structures ne peuvent pas hériter des classes, mais elles peuvent implémenter des interfaces.

# Création et implémentation d'une interface

Conformément à ce qui est expliqué dans la rubrique Vue d'ensemble des interfaces, les interfaces décrivent les propriétés, méthodes et événements d'une classe sans en assurer l'implémentation.

**Pour créer une interface**

. Définissez votre interface en y ajoutant du code commençant par le mot clé **Interface** et le nom de l'interface et se terminant par l'instruction **End Interface**. Par exemple, le code suivant définit une interface appelée Encryption :

```
. Interface Encryption
  End Interface
```

. Ajoutez des instructions définissant les propriétés, méthodes et événements pris en charge par votre interface. Par exemple, le code suivant définit deux méthodes, une propriété et un événement :

```
. Interface Encryption
.   Function Encrypt(ByVal estring As String) As String
.   Function Decrypt(ByVal dstring As String) As String
.   Property EncryptionBits() As Integer
.   Event FinishedDecoding(ByVal RetVal As Integer)
  End Interface
```

**Pour implémenter une interface**

- Si l'interface que vous implémentez ne fait pas partie de votre projet, ajoutez une référence à l'assembly qui contient l'interface.
- Créez une nouvelle classe qui implémente votre interface et ajoutez le mot clé **Implements** dans la ligne à la suite du nom de la classe. Par exemple, pour implémenter l'interface Encryption à l'aide du cryptage **Xor**, vous pouvez nommer la classe d'implémentation XorEnc, comme dans le code suivant :

```
Class XorEnc
  Implements Encryption
End Class
```

- Ajoutez des procédures pour implémenter les propriétés, méthodes et événements de la classe conformément au code ci-dessous, qui complète l'exemple fourni à l'étape précédente :

```
Class XorEnc
  Implements Encryption
  Event FinishedDecoding(ByVal RetVal As Integer) _
    Implements Encryption.FinishedDecoding

  Function Encrypt(ByVal estring As String) _
    As String Implements Encryption.Encrypt
    ' Place code to perform Xor encryption here.
  End Function
  Function Decrypt(ByVal dstring As String) _
    As String Implements Encryption.Decrypt
    ' Place code to perform Xor decryption here.
  End Function

  Property EncryptionBits() As Integer _
    Implements Encryption.EncryptionBits
    Get
      'Place code to return the value of this property here.
    End Get
    Set
      'Place code to set this property here.
    End Set
  End Property
End Class
```

# Héritage

Visual Basic .NET prend en charge l'héritage, qui vous permet de définir les classes servant de base pour les classes dérivées. Les classes dérivées héritent des propriétés, des méthodes et des événements de la classe de base et peuvent les étendre. Elles peuvent également substituer des méthodes héritées par de nouvelles implémentations. Toutes les classes créées avec Visual Basic .NET peuvent être héritées par défaut.

L'héritage vous permet d'écrire et de déboguer une seule fois une classe et de réutiliser ensuite ce code plusieurs fois comme base des nouvelles classes. L'héritage vous permet également d'utiliser le polymorphisme par héritage, permettant de définir des classes pouvant être utilisées de manière interchangeable par le code client au moment de l'exécution, avec des méthodes et des propriétés différentes d'un point de vue fonctionnel mais portant le même nom.

**Dans cette section**

Eléments fondamentaux de l'héritage
: Présente les modificateurs d'héritage, la substitution de méthodes et de propriétés, **MyClass** et **MyBase**.

Utilisation de Inherits pour créer une classe dérivée
: Décrit étape par étape le processus de création des classes dérivées.

Quand utiliser l'héritage
: Explique dans quels cas il convient d'utiliser l'héritage au lieu des interfaces.

Héritage et le .NET Framework
: Explique comment le .NET Framework prend en charge l'héritage.

Procédure pas à pas : implémentation de l'héritage avec les objets COM
: Décrit l'utilisation d'objets COM existants en tant que base des nouveaux objets.

Héritage et la classe Object de base
: Propose une vue d'ensemble de la classe servant de base pour toutes les autres classes.

Fonctionnement des méthodes New et Finalize dans une hiérarchie de classe
: Décrit l'appel des constructeurs et des destructeurs dans une hiérarchie de classe.

Polymorphisme
: Présente le polymorphisme et son utilisation.

Design d'une hiérarchie d'héritage

Fournit des informations sur la conception et l'implémentation d'une hiérarchie de classe.

# Eléments fondamentaux de l'héritage

L'instruction **Inherits** permet de déclarer une nouvelle classe, appelée une classe dérivée, basée sur une classe existante, appelée une classe de base. Les classes dérivées héritent des propriétés, des méthodes, des événements, des champs et des constantes de la classe de base et peuvent les étendre. La section suivante décrit certaines règles de l'héritage et les modificateurs permettant de modifier la façon dont les classes peuvent hériter ou être héritées :

- Toutes les classes peuvent être héritées par défaut, sauf si elles sont marquées avec le mot clé **NotInheritable**. Les classes peuvent hériter d'autres classes dans votre projet ou de classes dans d'autres assemblys faisant référence à votre projet.
- Contrairement aux langages autorisant des héritages multiples, Visual Basic .NET ne permet qu'un seul héritage dans des classes ; les classes dérivées ne peuvent avoir qu'une seule classe de base. Bien que les héritages multiples ne soient pas autorisés dans les classes, les classes peuvent implémenter plusieurs interfaces, qui peuvent accomplir efficacement les mêmes buts.
- Pour empêcher l'exposition d'éléments restreints dans une classe de base, le type d'accès d'une classe dérivée doit être égal à sa classe de base ou être plus restrictif. Par exemple, une classe **Public** ne peut pas hériter d'une classe **Friend** ou **Private**, et une classe **Friend** ne peut pas hériter d'une classe **Private**.

## Modificateurs d'héritage

Visual Basic .NET contient les instructions et les modificateurs de niveau classe suivants pour prendre en charge l'héritage :

- Instruction **Inherits** : spécifie la classe de base.
- Modificateur **NotInheritable** : empêche les programmeurs d'utiliser la classe comme une classe de base.
- Modificateur **MustInherit** : indique que la classe doit être utilisée uniquement comme une classe de base. Les instances de la classe **MustInherit** ne peuvent pas être créées directement ; elles ne peuvent être créées que comme des instances de classe de base d'une classe dérivée (d'autres langages de programmation, tels que C++ et C#, utilisent le terme classe abstraite pour décrire une telle classe).

## Substitution de propriétés et de méthodes dans des classes dérivées

Par défaut, une classe dérivée hérite des méthodes de sa classe de base. Si une propriété ou une méthode héritée doit se comporter différemment dans la classe dérivée, elle peut être substituée : vous pouvez définir une nouvelle implémentation de la méthode dans la classe dérivée. Les modificateurs suivants sont utilisés pour contrôler la façon dont les propriétés et les méthodes sont substituées :

- **Overridable** : permet la substitution d'une propriété ou d'une méthode d'une classe dans une classe dérivée.
- **Overrides** : substitue une propriété ou une méthode **Overridable** définie dans la classe de base.
- **NotOverridable** : empêche la substitution d'une propriété ou d'une méthode dans une classe qui hérite. Par défaut les méthodes **Public** sont **NotOverridable**.
- **MustOverride** : la propriété ou la méthode doit être substituée par une classe dérivée. Lorsque le mot clé **MustOverride** est utilisé, la définition de méthode se compose uniquement de l'instruction **Sub**, **Function** ou **Property**. Aucune autre instruction n'est autorisée et particulièrement l'instruction **End Sub** ou **End Function**. Les méthodes **MustOverride** doivent être déclarées dans les classes **MustInherit**.

Pour plus d'informations sur les méthodes de substitution, consultez Substitution de propriétés et de méthodes.

## MyBase, mot clé

Le mot clé **MyBase** permet d'appeler des méthodes dans une classe de base lors de la substitution de méthodes dans une classe dérivée. Supposons par exemple que vous créez une classe dérivée qui substitue une méthode héritée d'une classe de base. La méthode substituée peut appeler la méthode dans la classe de base et modifier la valeur de retour, comme illustré dans le fragment de code suivant :

```
Class DerivedClass
  Inherits BaseClass
  Public Overrides Function CalculateShipping(ByVal Dist As Double, _
                           ByVal Rate As Double) As Double
    ' Call the method in the base class and modify the return value.
    Return MyBase.CalculateShipping(Dist, Rate) * 2
  End Function
End Class
```

La liste suivante décrit les restrictions de l'utilisation de **MyBase** :

- **MyBase** fait référence à la classe de base immédiate et à ses membres hérités. Il ne peut pas être utilisé pour accéder aux membres **Private** de cette classe.
- **MyBase** est un mot clé, pas un objet réel. **MyBase** ne peut pas être assigné à une variable, passé à des procédures ou utilisé dans une comparaison **Is**.
- La méthode qualifiée par **MyBase** ne doit pas être définie dans la classe de base immédiate mais elle doit être définie dans une classe de base héritée indirectement. Pour qu'une référence qualifiée par **MyBase** soit correctement compilée, une classe de base doit contenir une méthode qui correspond au nom et aux types de paramètres qui apparaissent dans l'appel.
- Vous ne pouvez pas utiliser **MyBase** pour appeler des méthodes de la classe de base **MustOverride**.
- **MyBase** ne peut pas être utilisé pour se qualifier lui-même. Le code suivant est donc non conforme :

  ```
  MyBase.MyBase.BtnOK_Click()   ' Syntax error.
  ```

- **MyBase** ne peut pas être utilisé dans des modules.
- **MyBase** ne peut pas être utilisé pour accéder à des membres d'une classe de base marqués **Friend** si la classe de base se situe dans un assembly différent.

MyClass, mot clé

Le mot clé **MyClass** vous permet d'appeler une méthode **Overridable** implémentée dans votre classe et de garantir que l'implémentation de la méthode dans cette classe est appelée à la place d'une méthode substituée dans une classe dérivée.

- **MyClass** est un mot clé, pas un objet réel. **MyClass** ne peut pas être assigné à une variable, passé à des procédures ou utilisé dans une comparaison **Is**.
- **MyClass** fait référence à la classe conteneur et à ses membres hérités.
- **MyClass** peut être utilisé en tant que qualificateur pour les membres **Shared**.
- **MyClass** ne peut pas être utilisé dans des modules standard.
- **MyClass** peut être utilisé pour désigner une méthode définie dans une classe de base et ne possède aucune implémentation de la méthode fournie dans cette classe. La signification d'une telle référence est identique à **MyBase.Method**.

# Utilisation de Inherits pour créer une classe dérivée

Avec l'instruction **Inherits**, une classe hérite de tous les membres non privés de la classe spécifiée.

**Pour hériter d'une autre classe**

- Ajoutez une instruction **Inherits** portant le nom d'une classe que vous souhaitez utiliser comme classe de base en tant que première instruction dans votre classe dérivée. L'instruction **Inherits** doit être la première instruction sans commentaire après l'instruction de classe.

Exemple d'héritage

L'exemple suivant définit deux classes. La première classe est une classe de base comprenant deux méthodes. La seconde classe hérite des deux méthodes

de la classe de base, substitue la seconde méthode et définit un champ intitulé Field.

```
Class Class1
  Sub Method1()
    MessageBox.Show("This is a method in the base class.")
  End Sub
  Overridable Sub Method2()
    MessageBox.Show("This is another method in the base class.")
  End Sub
End Class

Class Class2
  Inherits Class1
  Public Field2 as Integer
  Overrides Sub Method2()
    Messagebox.Show("This is a method in a derived class.")
  End Sub
End Class

Protected Sub TestInheritance()
  Dim C1 As New class1()
  Dim C2 As New class2()
  C1.Method1() ' Calls a method in the base class.
  C1.Method2() ' Calls another method from the base class.
  C2.Method1() ' Calls an inherited method from the base class.
  C2.Method2() ' Calls a method from the derived class.
End Sub
```

Lorsque vous exécutez la procédure TestInheritance, les messages suivants apparaissent :

```
"This is a method in the base class."
"This is another method in the base class."
"This is a method in the base class."
"This is a method in a derived class."
```

# Quand utiliser l'héritage

L'héritage est un concept de programmation utile mais pouvant être utilisé de façon incorrecte. Les interfaces effectuent souvent mieux cette opération. Cette rubrique et la rubrique Quand utiliser des interfaces vous permettent de mieux déterminer les situations dans lesquelles il convient d'utiliser ces approches.

L'héritage est approprié dans les situations suivantes :

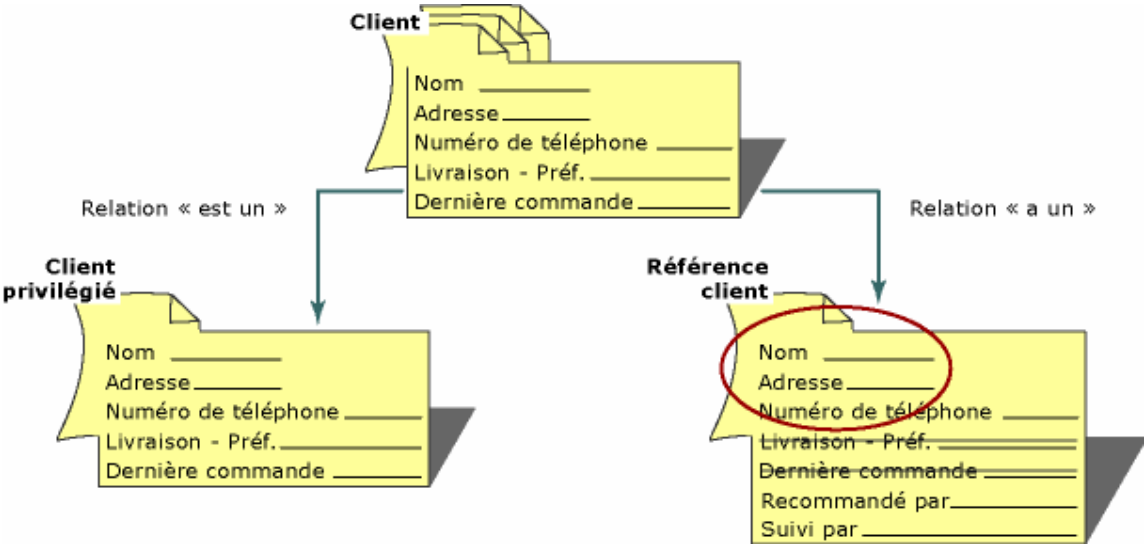
- Votre hiérarchie d'héritage est une relation de type être et non une relation de type avoir.
- Vous pouvez réutiliser du code des classes de base.
- Vous devez appliquer la même classe et les mêmes méthodes à divers types de données.
- La hiérarchie de classe est assez superficielle et il est peu probable que d'autres développeurs ajoutent des niveaux supplémentaires.
- Vous souhaitez apporter des modifications globales à des classes dérivées en modifiant une classe de base.

Ces différents aspects sont décrits, dans l'ordre, ci-dessous.

## Héritage et relations de type « être »

Il existe deux façons de présenter les relations de classe dans la programmation orientée objet : les relations de type « être » et « avoir ». Dans une relation de type « être », la classe dérivée est un type de la classe de base. Par exemple, une classe intitulée « PremierCustomer » représente une relation « est un » dont la classe de base est « Customer », car un premier client est un client. Toutefois, une classe intitulée « CustomerReferral » représente une relation de type « avoir » avec la classe « Customer » car une référence de client appartient à un client mais n'est pas un type de client.

Les objets d'une hiérarchie d'héritage doivent avoir une relation de type « être » avec leur classe de base car ils héritent des champs, des propriétés, des méthodes et des événements définis dans la classe de base. Les classes qui représentent une relation de type « avoir » avec d'autres classes ne conviennent pas aux hiérarchies d'héritage car elles peuvent hériter de propriétés et de méthodes inadaptées. Par exemple, si la classe « CustomerReferral » était dérivée de la classe « Customer », elle pourrait hériter de propriétés absurdes, telles que « ShippingPrefs » et « LastOrderPlaced ». Une telle relation de type « avoir » doit être représentée en utilisant des classes ou des interfaces non apparentées. L'illustration suivante montre des exemples des relations « est un » et « a un ».




## Classes de base et réutilisation de code

L'utilisation de l'héritage présente l'avantage de pouvoir réutiliser du code. Les classes bien conçues peuvent être déboguées une seule fois et réutilisées comme base des nouvelles classes.

Un exemple courant d'une réutilisation efficace du code concerne les bibliothèques qui gèrent des structures de données. Supposons par exemple que vous ayez une grande application d'entreprise qui gère plusieurs types de listes en mémoire. L'une de ces listes est une copie en mémoire de votre base de données des clients, lue depuis une base de données située au début de la session pour augmenter la vitesse. La structure de données peut se présenter comme suit :

```
Class CustomerInfo
  Public PreviousCustomer As CustomerInfo
  Public NextCustomer As CustomerInfo
  Public ID As Integer
  Public FullName As String

  Function InsertCustomer As CustomerInfo
    ' Add code to add a CustomerInfo item to the list.
  End Function

   Function DeleteCustomer As CustomerInfo
    ' Add code to remove a CustomerInfo item from the list.
   End Function

  Function GetNextCustomer As CustomerInfo
    ' Add code to get the next CustomerInfo item from the list.
```

```
  End Function

  Function GetPrevCustomer As CustomerInfo
      'Add code to get the previous CustomerInfo item from the list.
  End Function
End Class
```

Votre application peut également posséder une liste similaire des produits ajoutés par un utilisateur à son panier d'achats, comme l'illustre le fragment de code suivant :

```
Class ShoppingCartItem
  Public PreviousItem As ShoppingCartItem
  Public NextItem As ShoppingCartItem
  Public ProductCode As Integer
  Function GetNextItem As ShoppingCartItem
    ' Add code to get the next ShoppingCartItem from the list.
  End Function
End Class
```

En voici un modèle : deux listes se comportent de la même façon (insertions, suppressions et extractions) mais fonctionnent sur divers types de données. La gestion de deux bases de code pour effectuer essentiellement les mêmes fonctions peut entraîner des problèmes. La solution la plus efficace consiste à isoler la gestion de la liste dans sa propre classe et d'hériter ensuite de cette classe pour divers types de données :

```
Class ListItem
  Public PreviousItem As ListItem
  Public NextItem As ListItem
  Function GetNextItem() As ListItem
    ' Add code to get the next item in the list.
  End Function
  Function InsertNextItem As ListItem
    ' Add code to add a item to the list.
  End Function

   Function DeleteNextItem As ListItem
    ' Add code to remove a item from the list.
   End Function

  Function GetPrevItem As ListItem
      'Add code to get the previous item from the list.
  End Function

End Class
```

La classe ListItem ne doit être déboguée qu'une seule fois. Vous pouvez ensuite construire des classes qui l'utilisent sans devoir vous préoccuper de la gestion de la liste. Exemple :

```
Class CustomerInfo
  Inherits ListItem
  Public ID As Integer
  Public FullName As String
End Class
Class ShoppingCartItem
  Inherits ListItem
  Public ProductCode As Integer
End Class
```

Bien que la réutilisation du code d'héritage soit un outil puissant, elle présente également certains risques. Même les systèmes les mieux conçus peuvent varier d'une façon dont les concepteurs n'avaient pas prévu. Les modifications d'une hiérarchie de classe existante peuvent parfois entraîner des conséquences inattendues. Pour plus d'informations, consultez la section « Problème de classe de base fragile » sous la rubrique ~~Modifications du design de la classe après le déploiement~~.

## ~~Classes dérivées interchangeables~~

Les classes dérivées d'une hiérarchie de classe peuvent parfois être utilisées de façon interchangeable avec leur classe de base. Ce processus est appelé le polymorphisme d'héritage. Cette approche associe les meilleures fonctionnalités du polymorphisme d'interface avec l'option permettant de réutiliser ou de substituer du code d'une classe de base.

Un package de dessin représente un exemple où cette méthode pourrait s'avérer utile. Par exemple, examinons le fragment de code suivant qui n'utilise pas l'héritage :

```
Sub Draw(ByVal Shape As DrawingShape, ByVal X As Integer, _

    ByVal Y As Integer, ByVal Size As Integer)

  Select Case Shape.Type

    Case shpCircle

    ' Circle drawing code here.

    Case shpLine
```

```
    ' Line drawing code here.

  End Select

End Sub
```

Cette approche présente plusieurs problèmes. Si quelqu'un décide d'ajouter ultérieurement une option d'ellipse, le code source devra être modifié ; dans ce cas, il est possible que les utilisateurs cibles ne puissent plus accéder à votre code source. Un problème plus subtil est le suivant : pour dessiner une ellipse, vous avez besoin d'un autre paramètre (les ellipses ont un diamètre majeur et mineur) qui n'est pas adapté aux lignes. Si quelqu'un veut ensuite ajouter un polyline (plusieurs lignes reliées), un autre paramètre doit être ajouté qui ne correspondrait pas aux autres cas.

L'héritage permet de résoudre la plupart de ces problèmes. Les classes de base bien conçues laissent l'implémentation de méthodes spécifiques aux classes dérivées, de sorte qu'il est possible d'accommoder tout type de forme. D'autres développeurs peuvent implémenter des méthodes dans des classes dérivées en utilisant la documentation de la classe de base. D'autres éléments de classe (tels que les coordonnées x et y) peuvent être construits dans la classe de base car tous les descendants les utilisent. Par exemple, `Draw` peut être une méthode **MustOverride** :

```
MustInherit Class Shape
  Public X As Integer
  Public Y As Integer
  MustOverride Sub Draw()
End Class
```

Vous pouvez ensuite ajouter des paramètres à cette classe pour diverses formes. Par exemple, il est possible qu'une classe `Line` ne nécessite qu'un champ `Length` :

```
Class Line
  Inherits Shape
  Public Length As Integer
  Overrides Sub Draw()
    ' Place code here to implement Draw for this shape.
  End Sub
End Class
```

Cette approche est utile car les autres développeurs, qui ne peuvent pas accéder à votre code source, peuvent étendre leur classe de base avec de nouvelles

classes dérivées si nécessaire. Par exemple, une classe intitulée Rectangle peut être dérivée de la classe Line :

```
Class Rectangle
  Inherits Line
  Public Width As Integer
  Overrides Sub Draw()
    ' Place code here to implement Draw for the Rectangle shape.
  End Sub
End Class
```

Cet exemple illustre la façon dont vous pouvez passer des classes générales à des classes très spécifiques en ajoutant des détails d'implémentation à chaque niveau.

A ce niveau, il est recommandé de réévaluer si la classe dérivée représente réellement une relation de type « être » ou si une relation de type « avoir » a été utilisée par inadvertance. Si la nouvelle classe rectangle se compose uniquement de lignes, l'héritage ne représente pas la méthode la mieux adaptée. Toutefois, si le nouveau rectangle est une ligne comprenant une propriété de largeur, la relation de type « être » est conservée.

### Hiérarchies de classes superficielles

L'héritage est recommandé pour les hiérarchies de classes assez superficielles. Les hiérarchies de classe très profondes et complexes peuvent être difficiles à développer. La décision d'utiliser une hiérarchie de classe implique une analyse des avantages liés à l'utilisation d'une hiérarchie de classe par rapport à la complexité. En règle générale, vous devez limiter les hiérarchies à six niveaux maximum. Toutefois, la profondeur maximale d'une hiérarchie de classe particulière dépend d'un nombre de facteurs, y compris de la quantité de complexité à chaque niveau.

### Modifications globales des classes dérivées par l'intermédiaire de la classe de base

L'une des fonctionnalités les plus puissantes de l'héritage est la possibilité d'apporter des modifications dans une classe de base pour les répercuter dans les classes dérivées. Si vous procédez avec prudence, il suffit de mettre à jour l'implémentation d'une seule méthode pour que des dizaines, voire des centaines de classes dérivées puissent utiliser le nouveau code. Toutefois, cette opération peut être dangereuse car de telles modifications peuvent engendrer des problèmes avec les classes héritées conçues par de tierces personnes. Vous devez être prudent pour garantir que la nouvelle classe de base est compatible

avec les classes qui utilisent la classe d'origine. Vous devez particulièrement éviter de modifier le nom ou le type des membres de la classe de base.

Supposons par exemple que vous conceviez une classe de base avec un champ de type **Integer** pour stocker des informations concernant le code postal et que d'autres développeurs aient créé des classes dérivées qui utilisent le champ de code postal hérité. Supposons également que votre champ de code postal puisse stocker 5 chiffres mais que le bureau de poste ait modifié les codes postaux en leur ajoutant un tiret et 4 chiffres supplémentaires. Dans le pire scénario, vous pouvez modifier le champ de la classe de base pour stocker une chaîne de 10 caractères, mais les autres développeurs doivent modifier et recompiler les classes dérivées afin d'utiliser la nouvelle taille et le nouveau type de données.

Le moyen le plus sûr pour modifier une classe de base est de lui ajouter de nouveaux membres. Par exemple, vous pouvez ajouter un nouveau champ pour stocker les 4 chiffres supplémentaires du code postal de l'exemple précédent. Les applications clientes peuvent ainsi être mises à jour pour utiliser le nouveau champ sans endommager les applications existantes. La possibilité d'étendre les classes de base dans une hiérarchie d'héritage est un avantage important qui n'existe pas avec les interfaces.

# Héritage et la classe Object de base

Quasiment toutes vos actions dans Visual Basic .NET reposent sur l'utilisation d'objets, y compris des entités qui ne sont pas forcément perçues comme des objets, par exemple les chaînes et les entiers. Tous les objets de Visual Basic .NET sont dérivés de la classe de base **System.Object** et héritent de ses méthodes. La classe **System.Object** propose une classe de base commune pour tous les objets du .NET Framework, ce qui garantit une interopérabilité des objets développés à l'aide de Visual Studio .NET. Les nouvelles classes héritent implicitement de la classe **System.Object** ; il n'est pas nécessaire de nommer explicitement cette classe avec une instruction Inherits.

Parmi les méthodes dont héritent les objets de **System.Object**, l'une des plus utiles est la méthode **Object.GetType**, qui permet de retourner le type exact de l'objet en cours.

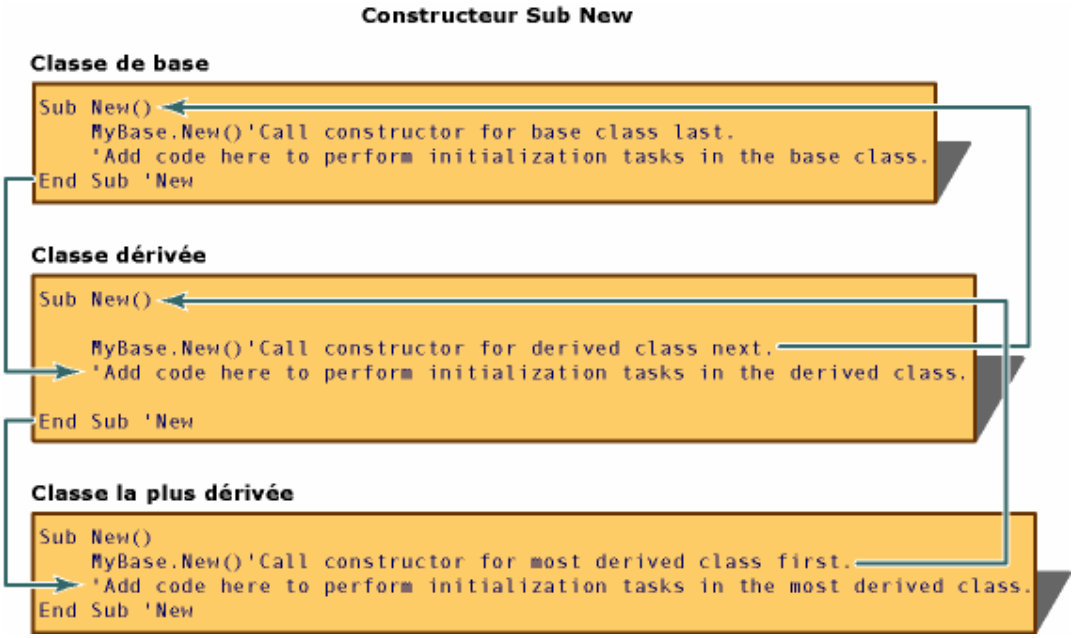
# Fonctionnement des méthodes New et Finalize dans une hiérarchie de classe

**Voir aussi**

~~Durée de vie d'un objet : création et destruction des objets~~

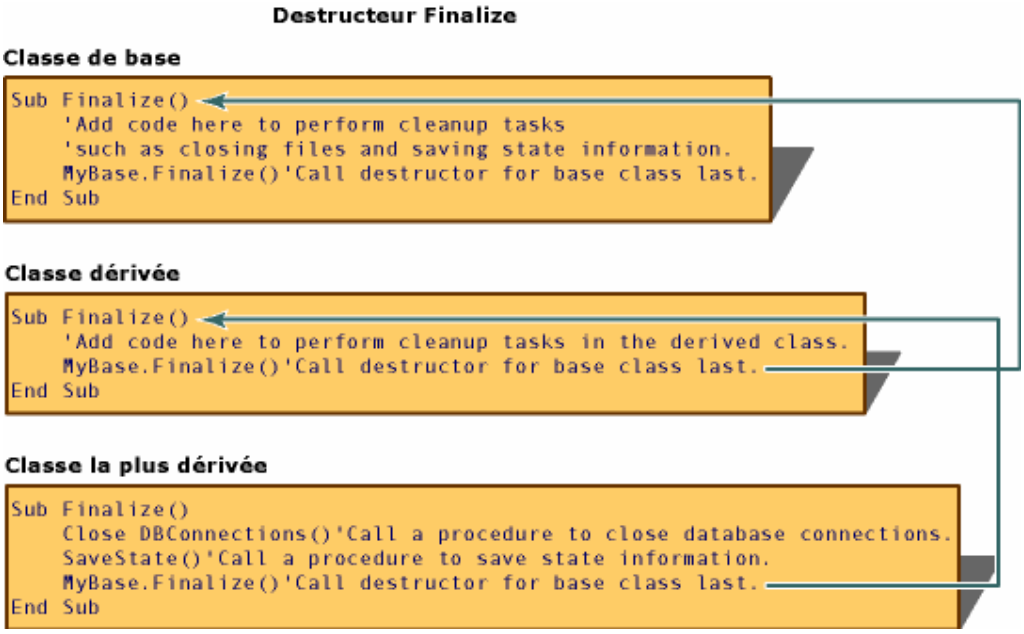
Quand une instance d'une classe est créée, le Common Language Runtime (CLR) tente d'exécuter une procédure appelée **New**, si elle existe dans cet objet. **New** est un type de procédure (appelé un constructeur) utilisé pour initialiser de nouveaux objets avant qu'un autre code d'un objet ne s'exécute. Un constructeur **New** peut être utilisé pour ouvrir des fichiers, se connecter à des bases de données, initialiser des variables et réaliser d'autres tâches qui doivent être effectuées avant qu'un objet ne soit utilisé.

Lorsqu'une instance d'une classe dérivée est créée, le constructeur **Sub New** de la classe de base s'exécute premièrement, suivi des constructeurs des classes dérivées. En effet, la première ligne de code d'un constructeur **Sub New** utilise la syntaxe MyBase.New()pour appeler le constructeur de la classe situé immédiatement au-dessus de celle-ci dans la hiérarchie de classe. Le constructeur **Sub New** est ensuite appelé pour chaque classe dans la hiérarchie de classe jusqu'à ce que le constructeur de la classe de base soit atteint. A ce stade, le code du constructeur de la classe de base s'exécute, suivi du code de chacun des constructeurs de toutes les classes dérivées ; le code des classes plus dérivées s'exécute en dernier lieu.




Lorsqu'un objet n'est plus nécessaire, le CLR appelle la méthode **Finalize** pour cet objet avant de libérer sa mémoire. La méthode **Finalize** est appelée un destructeur car elle effectue des tâches de nettoyage, telles qu'un enregistrement d'informations d'état, une fermeture de fichiers et des connexions à des bases de données, ainsi que d'autres tâches qui doivent être effectuées avant de libérer l'objet.

Lorsque la méthode **Sub Finalize** d'une classe dérivée est appelée, elle commence par effectuer les tâches de nettoyage nécessaires et utilise ensuite la syntaxe MyBase.Finalize() pour appeler explicitement la méthode **Sub Finalize** de sa classe de base. Les méthodes **Sub Finalize** s'exécutent d'abord depuis les classes les plus dérivées et exécutent le code de la classe de base en dernier lieu.




# Le polymorphisme dans Visual Basic

A l'origine, les interfaces permettaient généralement d'accomplir le polymorphisme dans Visual Basic et elles peuvent toujours être utilisées dans ce but. A l'heure actuelle, Visual Basic .NET permet toutefois d'utiliser l'héritage pour assurer également le polymorphisme. Tout comme d'autres éléments de la programmation orientée objet, l'option à utiliser dépend de vos besoins spécifiques. En règle générale, utilisez l'héritage pour créer des fonctionnalités de base pouvant être étendues par les classes dérivées et utilisez les interfaces lorsque des fonctionnalités similaires doivent être fournies par plusieurs implémentations possédant peu de caractéristiques communes.

# Polymorphisme d'héritage

La majorité des systèmes de programmation orientée objet assurent le polymorphisme via l'héritage. Le polymorphisme d'héritage implique une définition de méthodes dans une classe de base et leur substitution par de nouvelles implémentations dans des classes dérivées.

Vous pouvez par exemple définir une classe BaseTax qui propose des fonctionnalités de base pour calculer le taux de T.V.A. dans un état. Les classes dérivées de BaseTax, telles que CountyTax ou CityTax, peuvent implémenter des méthodes telles que CalculateTax, si nécessaire.

Le polymorphisme est issu de la possibilité d'appeler la méthode CalculateTax d'un objet appartenant à une classe dérivée de BaseTax, sans connaître la classe à laquelle appartient cet objet.

Dans l'exemple suivant, la procédure TestPoly illustre le polymorphisme d'héritage :

```
Const StateRate As Double = 0.053 ' %5.3 State tax
Const CityRate As Double = 0.028  ' %2.8 City tax
Public Class BaseTax
  Overridable Function CalculateTax(ByVal Amount As Double) As Double
    Return Amount * StateRate ' Calculate state tax.
  End Function
End Class

Public Class CityTax
  ' This method calls a method in the base class
  ' and modifies the returned value.
  Inherits BaseTax
  Private BaseAmount As Double
  Overrides Function CalculateTax(ByVal Amount As Double) As Double
    ' Some cities apply a tax to the total cost of purchases,
    ' including other taxes.
    BaseAmount = MyBase.CalculateTax(Amount)
    Return CityRate * (BaseAmount + Amount) + BaseAmount
  End Function
End Class

Sub TestPoly()
   Dim Item1 As New BaseTax()
   Dim Item2 As New CityTax()
   ShowTax(Item1, 22.74) ' $22.74 normal purchase.
   ShowTax(Item2, 22.74) ' $22.74 city purchase.
End Sub

Sub ShowTax(ByVal Item As BaseTax, ByVal SaleAmount As Double)
  ' Item is declared as BaseTax, but you can
  ' pass an item of type CityTax instead.
  Dim TaxAmount As Double
  TaxAmount = Item.CalculateTax(SaleAmount)
    MsgBox("The tax is: " & Format(TaxAmount, "C"))
End Sub
```

Dans cet exemple, la procédure ShowTax accepte un paramètre intitulé Item de type BaseTax, mais vous pouvez également passer les classes dérivées de la classe de forme, telle que CityTax. Cette conception présente l'avantage que vous pouvez ajouter de nouvelles classes dérivées de la classe BaseTax sans modifier le code client dans la procédure ShowTax.

# Design d'une hiérarchie d'héritage

Il est plus facile d'implémenter des hiérarchies d'héritage que de les créer. Il est donc conseillé d'identifier clairement vos besoins avant de commencer le codage. Pour corriger des erreurs de design dans une hiérarchie de classe après l'implémentation, vous devrez peut-être modifier le code qui pourrait désactiver des applications existantes. Cette section décrit les considérations de design d'une hiérarchie d'héritage et tente d'éviter de telles erreurs.

**Dans cette section**

*Considérations sur le design de hiérarchies de classes pour l'extensibilité*
: Présente le design de hiérarchies de classes de sorte qu'elles puissent être mises à jour ou étendues par d'autres développeurs.

*Considérations sur la sélection de niveaux d'accès pour les méthodes*
: Illustre l'utilisation correcte des niveaux d'accessibilité dans des classes.

*Modifications du design de la classe après le déploiement*
: Explique les problèmes auxquels sont confrontés les développeurs lorsqu'ils tentent d'apporter des modifications à une hiérarchie de classe.

# Considérations sur le design de hiérarchies de classes pour l'extensibilité

**Voir aussi**

MustInherit | MustOverride | ~~Considérations sur la sélection de niveaux d'accès pour les méthodes~~ ~~Modifications du design de la classe après le déploiement~~

Même les hiérarchies de classe bien conçues doivent évoluer au fil du temps. Les choix que vous effectuez lors de la conception d'une hiérarchie de classe peuvent simplifier les modifications ultérieures.

La liste suivante décrit quelques suggestions pour permettre d'étendre plus facilement vos hiérarchies de classe :

- Les hiérarchies de classe sont définies en partant des classes générales jusqu'aux classes spécifiques. Définissez des classes de chaque niveau d'une hiérarchie d'héritage aussi générales que possible. Les classes dérivées héritent des méthodes des classes de base et peuvent les réutiliser et les étendre.

  Supposons par exemple que vous conceviez une hiérarchie de classe qui décrit le matériel informatique. Lorsque vous commencez à décrire les périphériques de sortie, vous pouvez définir des classes appelées « Display », « Printer » et « File ». Vous pouvez ensuite définir les classes qui implémentent les méthodes définies dans les classes de base. Par exemple, la classe LCDDisplay peut être dérivée de Display et implémenter une méthode intitulée « EnterPowerSaveMode ».

- Définissez de grands types de données et de stockage pour éviter de devoir les modifier par la suite. Vous pouvez par exemple utiliser une variable de type **Long** même si vos données actuelles ne nécessitent qu'une variable standard de type **Integer**.
- Exposez uniquement les éléments nécessaires aux classes dérivées. Les méthodes et les champs **Private** réduisent les conflits de noms et empêchent d'autres utilisateurs d'utiliser les éléments que vous pourriez être amenés à modifier à un moment ou à un autre.
- Les membres qui sont uniquement nécessaires aux classes dérivées devraient être marqués en tant que protégés. Ceci réduit le nombre de

classes dépendantes de ces membres aux classes dérivées uniquement, ce qui facilite la mise à jour de ces membres lors du développement de la classe.

# Considérations sur la sélection de niveaux d'accès pour les méthodes

**Voir aussi**

Considérations sur le design de hiérarchies de classes pour l'extensibilité | Modifications du design de la classe après le déploiement

Si vous appliquez les niveaux d'accès optimaux aux membres d'une hiérarchie de classe, vous pouvez plus facilement gérer la hiérarchie car vous pouvez contrôler la façon dont ces membres sont utilisés.

En règle générale, vous devez déclarer des membres de classe avec des modificateurs d'accès qui fournissent l'accès minimum. La restriction d'accès aux membres de classe réduit les conflits de noms et empêche une utilisation inattendue des méthodes.

Les membres de classe interne doivent être déclarés comme **Private** ; vous ne pouvez accéder à ces membres que depuis la classe dans laquelle ils ont été définis.

Les méthodes qui sont uniquement utilisées à l'intérieur d'une classe ou par des descendants d'une classe doivent utiliser le modificateur d'accès **Protected**. Un membre **protégé** est accessible à partir de la classe dans laquelle il est déclaré et à partir de toute classe dérivée de la classe ayant déclaré ce membre.

Les données membres **Friend** sont accessibles depuis l'extérieur d'une classe mais uniquement par des modules faisant partie du projet lors de la définition de la classe.

Les données membres **Public** sont visibles à toute personne et sont souvent utilisés au bas d'une hiérarchie de classe.

# Objets

Lorsque vous créez une application dans Visual Basic .NET, vous travaillez constamment avec des objets. Vous pouvez utiliser des objets fournis pas Visual Basic (par exemple, des contrôles, des formulaires et des objets d'accès aux données) ou des objets issus d'autres applications dans votre application Visual Basic. Vous pouvez également créer vos propres objets et définir des propriétés ainsi que des méthodes supplémentaires pour ces objets. Les objets se comportent comme des blocs de construction préfabriqués : vous pouvez écrire un fragment de code une seule fois et le réutiliser continuellement.

Les objets sont traités en détail dans les rubriques suivantes :

**Dans cette section**

Qu'est-ce qu'un objet ?
: Présente une vue d'ensemble des objets et des classes.

Relation des objets entre eux
: Présente les hiérarchies d'objets, les collections et les conteneurs.

Eléments fondamentaux relatifs à l'utilisation d'objets
: Présente une vue d'ensemble de l'utilisation des objets.

Objets, propriétés, méthodes et événements
: Récapitule les propriétés, méthodes et événements exposés par objets.

# Qu'est-ce qu'un objet ?

Un objet est une combinaison de code et de données qui peut être traitée comme une entité unique. Un objet peut être une partie d'une application (telle qu'un contrôle ou un formulaire) ou une application dans son intégralité.

Les objets vous permettent de déclarer une seule fois des variables et des procédures, puis de les réutiliser quand bon vous semble. Pour ajouter un vérificateur d'orthographe à une application par exemple, vous pouvez définir toutes les variables et les fonctions de support pour assurer une vérification orthographique dans votre application. Toutefois, si vous créez votre vérificateur orthographique sous forme de classe, vous pouvez le réutiliser dans d'autres applications en ajoutant une référence à l'assembly compilé. Vous pouvez même utiliser une classe de vérificateur d'orthographe développée par quelqu'un d'autre pour vous éviter tout travail superflu.

Dans Visual Basic .NET, chaque objet est défini par une classe. Les classes décrivent les champs, les propriétés, les méthodes et les événements d'un objet. Les objets sont des instances de classes ; vous pouvez créer autant d'objets que nécessaire dès que vous avez créé une classe.

Pour comprendre la relation existant entre un objet et sa classe, pensez à celle existant entre un moule à gâteau et des gâteaux. Le moule à gâteau représente la classe. Il définit les caractéristiques de chaque gâteau (sa taille et sa forme, par exemple). La classe permet de créer des objets. Les objets peuvent être comparés aux gâteaux.

Les deux exemples suivants illustrent mieux la relation entre les classes et les objets dans Visual Basic .NET :

- Dans Visual Basic .NET, les contrôles de la boîte à outils représentent des classes. Lorsque vous déplacez un contrôle de la boîte à outils vers un formulaire par glisser-déplacer, vous créez un objet, c'est-à-dire une instance d'une classe.
- Le formulaire que vous utilisez au moment de la création est une classe. Au moment de l'exécution, Visual Basic .NET crée une instance de la classe du formulaire.

Les objets sont créés en tant que copies identiques de leur classe. Il est toutefois possible de modifier leurs propriétés dès qu'ils existent en tant qu'objets individuels. Par exemple, si vous ajoutez trois cases à cocher à un formulaire, chacun de ces objets est une instance de la classe **CheckBox**. Les objets **CheckBox** individuels partagent un ensemble commun de caractéristiques et de fonctionnalités (propriétés, champs, méthodes et événements), définies par la classe. Cependant, chacun possède un nom qui lui est propre, il peut être activé ou désactivé séparément et placé à un autre emplacement du formulaire.

# Relation des objets entre eux

Les objets peuvent être reliés les uns aux autres de diverses manières. Une telle relation est hiérarchique. Les classes dérivées de classes plus fondamentales ont une relation hiérarchique. Les hiérarchies de classe sont utiles lors d'une description d'éléments qui représentent un sous-type d'une classe plus générale. Dans la mesure où les classes dérivées héritent des membres de la classe dont

elles sont issues, elles peuvent devenir plus complexes au fur et à mesure que vous avancez dans une hiérarchie de classe.

Les objets sont également liés par une relation entre un objet et un conteneur d'objet. Les objets conteneur encapsulent d'autres objets. Par exemple, un objet conteneur peut être une instance de classe comprenant une propriété qui stocke un objet.

Les collections représentent un type de relation contenant-contenu d'un objet. Les collections ressemblent à des tableaux d'objets pouvant être énumérés. Visual Basic .NET prend en charge une syntaxe spécifique sous la forme de blocs de syntaxe **For Each…Next** permettant d'itérer les éléments d'une collection. En outre, les collections vous permettent souvent d'utiliser une méthode appelée propriété **Item** pour extraire des éléments par leur index ou en les associant à une chaîne unique. Les collections sont plus simples à utiliser que les tableaux car elles vous permettent d'ajouter ou d'enlever des éléments sans utiliser d'index. Grâce à leur simplicité d'utilisation, elles sont souvent utilisées pour stocker des formulaires et des contrôles.

# Eléments fondamentaux relatifs à l'utilisation d'objets

La plupart des opérations effectuées dans Visual Basic .NET impliquent des objets. Cette rubrique présente diverses méthodes d'utilisation des éléments d'objets.

## Accès au membre

Pour accéder aux membres d'un objet, vous devez spécifier le nom de l'objet suivi d'un point et du nom du membre à utiliser. Par exemple, si vous possédez un objet intitulé Obj1 dont la propriété est Prop1, la syntaxe `Obj1.Prop1` vous permet d'accéder à cette propriété. Vous devez parfois utiliser plusieurs points car les objets peuvent contenir d'autres objets. Vous pouvez par exemple utiliser la syntaxe `Obj1.Prop1.X` si la propriété Prop1 est un objet qui contient un champ X.

## Champs et propriétés

Les propriétés et les champs sont des informations stockées dans un objet. Les valeurs des champs et des propriétés sont extraites et définies avec des

instructions d'affectation de la même façon que les variables. L'exemple suivant définit et extrait la valeur d'un champ et d'une propriété.

```
Dim X As Integer = 8
Dim Obj1 As New Class1   ' Declares an instance of a class.
Obj1.Prop1 = X   ' Property assignment. Assigns the number 8 to Prop1.
X = Obj1.Prop1 + 1   ' Retrieval. The variable X contains the number 9.
Obj1.Field1 = X   ' Field assignment. Assigns the number 9 to Field1.
X = Obj1.Prop1 + 1   ' Retrieval. The variable X contains the number 10.
```

## Méthodes

Les méthodes sont des procédures définies dans une classe. Elles sont utilisées comme des sous-routines ou des fonctions, mais elles sont accessibles via l'objet auquel elles appartiennent. Le fragment de code suivant appelle une fonction et une sous-routine déclarée dans un objet :

```
Dim Obj2 As New Class2   ' Declares an instance of a class
Dim Y As Integer
' Retrieves a value from a method declared as a function.
Y = Obj2.GetNumber
Obj2.StoreNumber(Y)   ' Calls a method declared as a subroutine.
```

## Evénements

Les événements sont des notifications qui entraînent l'exécution d'une procédure. Les objets peuvent traiter et générer des événements. Les événements sont générés par l'instruction **RaiseEvent** et gérés par le mot clé **Handles** ou l'instruction **AddHandler**. L'exemple suivant déclenche et gère un événement déclaré par l'intermédiaire du mot clé **WithEvents** :

```
' Declares a WithEvents variable in the declarations section.
  Dim WithEvents EClass As EventClass

  Class EventClass
    Public Event AnEvent()   ' Declares an event.

    Sub CauseEvent()
      RaiseEvent AnEvent()   'Raises an event.
    End Sub
  End Class

' Declares an event handler
  Sub EventHandler() Handles EClass.AnEvent
    MessageBox.Show("Received Event.")
  End Sub

' Test the class and the event handler.
```

```
Sub TestEvents()
  EClass = New EventClass()   ' Creates an instance of the class.
  EClass.CauseEvent()   ' Calls a method that raises an event.
End Sub
```

# Objets, propriétés, méthodes et événements

**Voir aussi**

~~Objets~~ | ~~Définition et récupération de propriétés~~ | ~~Exécution d'actions à l'aide de méthodes~~ | ~~Evénements et délégués~~

Les objets, les propriétés, les méthodes et les événements représentent les unités de base de la programmation orientée objet. Un objet est un élément d'une application, représentant une instance d'une classe. Les propriétés, les méthodes et les événements sont les blocs de construction des objets et constituent leurs membres.

## ~~Objets~~

Un objet représente une instance d'une classe telle que **Form**, **Control** ou **Component**. Dans le code Visual Basic, vous devez instancier un objet avant de pouvoir appliquer l'une des méthodes de l'objet ou modifier la valeur de l'une de ses propriétés. L'instanciation est le processus selon lequel une instance d'une classe est créée et assignée à une variable d'objet. C'est ce que montre l'exemple suivant :

```
Dim x As New MyClass()
```

Dans cet exemple, la variable x est assignée pour faire référence à une nouvelle instance de la classe MyClass.

## ~~Propriétés~~

Une propriété est un attribut d'un objet qui définit l'une des caractéristiques de l'objet (sa taille, sa couleur, son emplacement à l'écran, etc.) ou un aspect de son comportement (s'il est activé ou visible, par exemple). Pour modifier les caractéristiques d'un objet, vous devez modifier les valeurs de ses propriétés correspondantes.

Pour définir la valeur d'une propriété, vous devez faire suivre la référence à un objet d'un point, puis du nom de la propriété, du signe égal (=) et de la valeur de la nouvelle propriété. Par exemple, la procédure suivante modifie la légende du Windows Form Visual Basic en définissant la propriété **Text** :

```
Public Sub ChangeName(newTitle)
  myForm.Text = newTitle
End Sub
```

Vous ne pouvez pas définir certaines propriétés. La rubrique d'aide de chacune des propriétés indique si vous pouvez définir cette propriété (lecture/écriture), uniquement lire la propriété (lecture seule) ou uniquement écrire la propriété (écriture seule).

Vous pouvez extraire des informations sur un objet en retournant la valeur de l'une de ses propriétés. La procédure suivante affiche, dans un message, le titre qui apparaît au-dessus du formulaire actif.

```
Public Sub GetFormName()
  Dim formName As String
  formName = myForm.Text
  MsgBox(formName)
End Sub
```

## Méthodes

Une méthode est une action que peut effectuer un objet. Par exemple, **Add** est une méthode de l'objet **ComboBox**, car elle ajoute une nouvelle entrée à une zone de liste.

L'exemple suivant illustre la méthode **Start** d'un composant **Timer** :

```
' Instantiates a Timer object.
Dim myTimer As New System.Windows.Forms.Timer()
' Invokes the Start method of myTimer.
MyTimer.Start
```

## Evénements

Un événement est une action reconnue par un objet, telle qu'un clic de souris ou une pression sur une touche, pour laquelle vous pouvez écrire du code pour y répondre. Les événements peuvent se produire suite à une action d'un utilisateur ou au code de programme, ou être déclenchés par le système. Vous pouvez également développer vos événements personnalisés déclenchés par vos objets et gérés par d'autres objets.