VB.NET et la programmation objet

SOMMAIRE

INTRODUCTION

Ce support traite des concepts de Programmation Orientée Objet en langage VB.NET. Il est constitué d'une liste d'exercices permettant de construire pas à pas une classe d'objet . Les corrigés types de chaque étape sont présentés dans des répertoires séparés, il s'agit en fait du même cas qui va évoluer, étape par étape, pour aboutir à un fonctionnement correct de la classe dans l'univers C#.
Chaque exercice est structuré de la façon suivante :

- Description des objectifs visés.
- Explications des techniques à utiliser (Ce qu'il faut savoir).
- Enoncé du problème à résoudre (Travail à réaliser).
- Renvois bibliographiques éventuels dans les ouvrages traitant de ces techniques (Lectures).

Tous ces exercices sont corrigés et commentés dans le document intitulé :

- Proposition de corrigé C#_BPC.
  Apprentissage d'un langage de programmation Orientée Objet

- Dans ce support, les programmes produits fonctionneront en mode Console

Pour pouvoir utiliser ce support, les bases de la programmation en langage C# doivent être acquises.

T.P. N°1 - CLASSE D'OBJET - ENCAPSULATION

1.1 OBJECTIFS

- Notion de classe d'objet
- Définition d'une nouvelle classe d'objet en C#.
- Encapsulation des propriétés et des méthodes de cet objet.
- Instanciation de l'objet.

1.2 CE QU'IL FAUT SAVOIR

1.2.1 Notion de classe

Créer un nouveau type de données, c'est **modéliser** de la manière la plus juste un objet, à partir des possibilités offertes par un langage de programmation.
Il faudra donc énumérer toutes les propriétés de cet objet et toutes les fonctions qui vont permettre de définir son comportement. Ces dernières peuvent être classées de la façon suivante :

- **Les fonctions d'entrée/sortie**. Comme les données de base du langage C#, ces fonctions devront permettre de lire et d'écrire les nouvelles données sur les périphériques (clavier, écran, fichier, etc.).
- **Les opérateurs de calcul**. S'il est possible de calculer sur ces données, il faudra créer les fonctions de calcul.
- **Les opérateurs relationnels**. Il faut au moins pouvoir tester si deux données sont égales. S'il existe une relation d'ordre pour le nouveau type de donnée, il faut pouvoir aussi tester si une donnée est *inférieure* à une autre.
- **Les fonctions de conversion**. Si des conversions vers les autres types de données sont nécessaires, il faudra implémenter également les fonctions correspondantes.
- **Intégrité de l'objet**. La manière dont est modélisé le nouveau type de données n'est probablement pas suffisant pour représenter l'objet de façon exacte. Par exemple, si l'on représente une fraction par un couple de deux entiers, il faut également vérifier que le dénominateur d'une fraction n'est pas nul et que la représentation d'une fraction est unique (simplification).

La déclaration d'un nouveau type de données et les fonctions qui permettent de gérer les objets associés constituent une ***classe***.de l'objet.
Les propriétés de l'objet seront implantées sous la forme de ***données membres*** de la classe.
Les différentes fonctions ou méthodes seront implémentées sous la forme de ***fonctions membres*** de la classe.
De façon courante, dans le *patois* de la programmation orientée objet, ***données membres*** et ***fonctions membres*** de la classe sont considérées respectivement comme synonymes de ***propriétés*** et ***méthodes*** de l'objet.

1.2.2 Encapsulation

L'encapsulation est un concept important de la Programmation Orientée Objet.
L'encapsulation permet de rassembler les ***propriétés*** composant un objet et les ***méthodes*** pour les manipuler dans une seule entité appelée ***classe*** de l'objet.
Une classe, en C# se déclare par le mot clé **`class`** suivi d'un identificateur de classe choisi par le programmeur de la façon suivante :

```
Public Class NomDeLaClasse

   ' Déclaration des propriétés et
   ' des méthodes de l'objet
End Class
```

Le fichier contenant le code source portera le nom **NomdeLaClasse.cs.**
Les identifiants de classe obéissent aux même règles que les identifiants de variables et de fonctions. Par convention, ils commencent par une Majuscule (C# considère les majuscules comme des caractères différents des minuscules).
Les propriétés et les méthodes de l'objet seront déclarées et implémentées dans le bloc fermé par End Class contrôlé par le mot clé **`Class`**. Cela constitue l'implémentation du concept d'encapsulation en VB .NET

1.2.3 Déclaration des propriétés d'un objet

Les propriétés d'un objet sont déclarées, comme des variables, à l'intérieur du bloc **`{ }`** contrôlé par le mot clé **`class`**.

```
Public Class NomDeLaClasse

   Public    NomDeLaPropriete As TypeDeLaProprieté


   ' Déclaration des méthodes de l'objet
End Class
```

- Les propriétés peuvent être déclarées à tout moment à l'intérieur du corps de la classe.
- Chaque déclaration de propriété est construite sur le modèle suivant :

```
Public NomDeLaPropriete As TypeDeLaProprieté
```

- Une propriété peut être initialisée lors de sa déclaration :

```
Public NomDeLaPropriete As TypeDeLaProprieté =
valeurInitiale;
```

- Les identifiants de propriété par convention commencent par une majuscule.

1.2.4 Implantation des méthodes

Les méthodes peuvent être implémentées sous forme de procédures ou de fonctions.

Méthode en tant que fonction

Les méthodes d'une classe sont implémentées, à l'intérieur du bloc entre Function et End Function.
Quand on considère une méthode par rapport à l'objet à laquelle elle est appliquée, il faut voir l'objet comme étant sollicité de l'extérieur par un ***message***. Ce ***message*** peut

comporter des paramètres. L'objet doit alors réagir à ce ***message*** en exécutant cette fonction ou méthode.

```
Public Class NomDeLaClasse
' Déclaration des propriétés de l'objet
   Public Vitesse As Integer = 30

   Public Function NomDeMethode(par1 As Type1, par2 As Type2)
      As TypeResultat

          ' Instructions de la méthode
          ' retour de la fonction
          NomDeMethode = expression

                ou

       Return expression

   End Function
End Class
```

- Les identifiants de méthodes commencent par une majuscule.

- Si aucun paramètre n'est désigné explicitement entre les parenthèses, le compilateur considère la méthode comme étant sans paramètre. Dans ce cas, les **parenthèses sont néanmoins obligatoires**.

- Pour renvoyer un résultat de la fonction, l'une des instructions doit être **une affectation au nom de la fonction d'une expression de même type ou un instruction Return (ce qui est plus standard)**

.

Méthode en tant que sous-programme

Les méthodes d'une classe sont implémentées, à l'intérieur du bloc entre Sub et End Sub. Le passage de paramètres se passe comme pour une fonction, par valeur ou par référence. Il n' ya pas de retour de valeur , ou alors il faut modifeir une valeur de paramètre passée par référence.

1.2.5 Instanciation

Pour qu'un objet ait une existence, il faut qu'il soit ***instancié***. Une même classe peut être instanciée **plusieurs fois**, chaque instance ayant des propriétés ayant des valeurs spécifiques.

En C#, il n'existe qu'une seule manière de créer une ***instance*** d'une classe. Cette création d'instance peut se faire en deux temps :

- Déclaration d'une variable du type de la classe de l'objet,
- Instanciation de cette variable par l'instruction **`New`**.

Par exemple, l'instanciation de la classe **`String`** dans la fonction **`Main`** d'une classe application se passerait de la façon suivante :

```
   Dim s As String
   s = new String("AFPA")
```

La déclaration d'une variable **`s`** de classe **`String`** n'est pas suffisante. En effet, **`s`** ne *contient* pas une donnée de type **`String`**. **`s`** est une variable qui contient une référence sur un objet. Par défaut, la valeur de cette référence est **`null`**, mot clé C# signifiant que la variable n'est pas référencée. La référence d'un objet doit être affectés à cette variable. Cette référence est calculée par l'instruction **`New`** au moment de l'instanciation.

Ces deux étapes peuvent être réduites à une seule instruction :

```
   Dim sb As String = New String("toto")
ou
   Dim sc As String = "titi"
```

1.2.6 Accès aux propriétés et aux méthodes

Bien qu'un accès direct aux propriétés d'un objet ne corresponde pas exactement au concept d'encapsulation de la programmation orientée objet, il est possible de le faire en VB.NET On verra, au chapitre suivant, comment protéger les données membres en en interdisant l'accès.

L'accès aux propriétés et aux méthodes d'une classe se fait par l'opérateur **.**.

Opérateur .

```
sc.Substring(2,1)
```

1.2.7 Conventions d'écriture

Quand on débute la Programmation Orientée Objet, l'un des aspects le plus rebutant est l'impression d'éparpillement du code des applications. En respectant quelques conventions dans l'écriture du code et en organisant celui-ci de façon rationnelle, il est possible de remédier facilement à cet inconvénient.

Ces conventions ne sont pas normalisées et peuvent différer en fonctions des ouvrages consultés.

Les classes

Tous les identificateurs de classe commencent par une Majuscule. Par exemple : **`Fraction`**.

Les propriétés

Tous les identificateurs public de propriétés commencent par une majuscule, les propriétés private ou protected par une minuscule
. Par exemple :
private **`denominateur`**, **`numerateur`**. (sans accent)

Méthode

Elle commence par une lettre en majuscule.
e**xemple** : AjouterVehicule() ; Demarrer() ; InsererFichier () ; le premier mot est un verbe.

Les noms de fichiersConvention d'écriture:

On codera chaque classe d'objet dans des fichiers différents dont l'extension sera **`.cs`**
Le nom de fichier sera identique au nom de la classe.
Exemple pour la classe **`Fraction`** : **`Fraction.cs`**

Les noms de fichiersConvention d'écriture:

Les noms de vairiables de travail commencent par une minuscule

1.3 TRAVAIL A REALISER

- Créer la classe **`Salarie`**. Cette classe aura 5 propriétés de type public:
  - matricule M**`atricule`** **`Integer`**
  - catégorie C**`ategorie`** **`Integer`**
  - service S**`ervice`** **`Integer`**
  - nom **`Nom`** **`String`**
  - salaire S**`alaire`** **`Double`**

- Créer une méthode en tant que fonction **`CalculerSalaire()`** pour afficher la mention "Le salaire de " suivie du nom du salarié, suivi de " est de ", suivi de la valeur du salaire.

  Implanter une classe application, avec une méthode **`Main`** dans laquelle la classe **`Salarie`** sera instanciée pour en tester les propriétés et les méthodes.

T.P. N°2 - ENCAPSULATION PROTECTION ET ACCES AUX DONNEES MEMBRES

2.1 OBJECTIFS

- Protection des propriétés (données membres).
- Fonctions de type ***Get*** et ***Set***, d'accès aux propriétés.

2.2 CE QU'IL FAUT SAVOIR

2.2.1 Protection des propriétés

En Programmation Orientée Objet, on évite d'accéder directement aux propriétés par l'opérateur **.** . En effet, cette possibilité ne correspond pas au concept d'encapsulation. Certains langages l'interdisent carrément. En C#, c'est le programmeur qui choisit si une donnée membre ou une fonction membre est accessible directement ou pas.
Par défaut, VB.NET toutes les propriétés et méthodes sont accessibles directement. Il faut donc préciser explicitement les conditions d'accès pour chaque propriété et chaque méthode. Pour cela, il existe trois mots-clés :

- **`public`** - Après ce mot clé, toutes les données ou fonctions membres sont accessibles.
- **`private`** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées et ne seront pas accessibles dans les classes dérivées.
- **`protected`** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées mais sont néanmoins accessibles dans les classes dérivées.

*La distinction entre **`private`** et **`protected`** n'est visible que dans le cas de la déclaration de nouvelles classes par héritage. Ce concept sera abordé ultérieurement dans ce cours.*

Afin d'implanter **correctement** le concept d'encapsulation, il convient de **verrouiller** l'accès aux propriétés et de les déclarer **`private`**, tout en maintenant l'accès aux méthodes en les déclarant **`public`**.

Exemple :

```
Public Class Client

   Private numeroClient As Integer       ' numéro client
   Private nomClient As String           ' nom du client
   Private caClient As Double            ' chiffre d'affaire client

   Public Function AugmenterCA(ByVal montant As Double)

      caClient = montant
   End Function
End Class
```

2.2.2 Fonctions d'accès aux propriétés

Si les propriétés sont verrouillées, on ne peut plus y avoir accès de l'extérieur de la classe. Il faut donc créer des méthodes dédiées à l'accès aux propriétés pour chacune d'elles. Ces méthodes doivent permettre un accès dans les deux sens :

- **pour connaître la valeur de la propriété**. Ces méthodes sont appelées méthodes de type "***Get***.". La réponse de l'objet, donc la valeur retournée par la méthode ***Get,*** doit être cette valeur.

  Par exemple, pour la propriété **`numeroClient`**, déclarée **`Integer`**, la fonction ***Get*** serait déclarée de la façon suivante :

```
Public Function GetNumeroClient() As Integer
      GetNumeroClient = numeroClient
  End Function
```

  Cette fonction pourra être utilisée dans la fonction **`Main`**, par exemple :

```
Dim cli As New Client
Dim numero As Integer
Numero = cli.GetNumeroClient()
```

- **pour modifier la valeur d'une propriété**. Ces méthodes sont appelées méthodes ***Set***. Cette méthode ne retourne aucune réponse. Par contre, un paramètre de même nature que la propriété doit lui être indiqué.

  Par exemple, pour la propriété **numeroClient**, déclarée **`int`**, la fonction ***Set*** sera déclarée de la façon suivante :

```
Public Function SetNumeroClient(ByVal numeroClient As Integer)
      Me.numeroClient = numeroClient
End Function
```

  Me désigne l'instance en cours qui exécute la méthode.

  Cette fonction pourra être utilisée dans la fonction **`main`**, par exemple :

```
Dim cli As New Client
cli.SetNumeroClient(1)
```

  L'intérêt de passer par des fonctions ***Set*** est de pouvoir y localiser des contrôles de validité des paramètres passés pour assurer la cohérence de l'objet, en y déclenchant des exceptions par exemple. La sécurisation des classes sera abordée ultérieurement dans ce cours.

Remarque :

Les modifiers Set peuvent être implémentés par des sous programmes Sub.

2.2.3 Accès aux attributs membres par les Property de classe

Il est possible de définir les accès aux attributs d'une classe par des propriétés Property à la place des méthodes Get et Set. Cette possibilité existe aussi en VB 6.

Exemple de la classe Client :

```
Public Property Nom() As String
      Get
         Return nomClient
      End Get
      Set(ByVal Value As String)
         nomClient = Value
      End Set
   End Property
```

Utilisation des "***properties***"

```
Dim cli As New Client
cli.SetNumeroClient(1)
' property
cli.Nom = "Toto"
Console.WriteLine("nom client " & cli.Nom())
```

2.3 TRAVAIL A REALISER

A partir du travail réalisé au T.P. N° 1, modifier la classe **`Salarie`** pour :

- protéger les propriétés et en interdire l'accès de l'extérieur de l'objet (l'accès aux fonctions membres doit toujours être possible), créer les méthodes d'accès aux propriétés Get et Set.
- Modifier la fonction **`Main`** de la classe Application pour tester.

Bonus : vous pouvez tester les propriétés de classe Property …

T.P. N°3 - CONSTRUCTION ET DESTRUCTION

3.1 OBJECTIFS

- Constructeurs et destructeur des objets
- Propriétés et méthodes de classe

3.2 CE QU'IL FAUT SAVOIR

3.2.1 Constructeurs

Quand une instance d'une classe d'objet est créée au moment de l'instanciation d'une variable avec **new**, une fonction particulière est exécutée. Cette fonction s'appelle le ***constructeur***. Elle permet, entre autre, d'initialiser chaque instance pour que ses propriétés aient un contenu cohérent.

Un constructeur est déclaré comme les autres fonctions membres à deux différences près :

- Le nom de l'identificateur du constructeur est New .
- Un constructeur ne renvoie pas de résultat. On utilisera donc des méthodfes de type Sub pour les constructeurs

Exemple de définition dans la classe Client

```
' constructeur par défaut vide
   Public Sub New()
      ' vide
   End Sub

   ' constructeur d'initialisation le CA est augmenté par la méthode AugmenterCA
   Public Sub New(ByVal numero As Integer, ByVal nom As String)
      Me.numeroClient = numero
      Me.nomClient = nom
   End Sub
```

Innstanciation d'un objet Client :

```
Dim cli As Client = New Client(12, "Titi")
Console.WriteLine("numéro client " & cli.GetNumeroClient())
```

3.2.2 Surcharge des constructeurs

Il peut y avoir plusieurs constructeurs pour une même classe, chacun d'eux correspondant à une initialisation particulière. Tous les constructeurs ont le même nom mais se distinguent par le nombre et le type des paramètres passés (cette propriété s'appelle ***surcharge*** en programmation objet). Quand on crée une nouvelle classe, il est indispensable de prévoir tous les constructeurs nécessaires. Deux sont particulier :

Constructeur d'initialisation

Ce constructeur permet de procéder à une instanciation en initialisant toutes les propriétés, la valeur de celles-ci étant passée dans les paramètres.

Pour la classe **`Client`**, ce constructeur est déclaré comme ceci :

```
' constructeur d'initialisation le CA est augmenté par la méthode AugmenterCA
   Public Sub New(ByVal numero As Integer, ByVal nom As String)
      Me.numeroClient = numero
      Me.nomClient = nom
   End Sub
```

Cela va permettre d'instancier la classe **`Client`** de la façon suivante :

```
Dim cli As Client = New Client(12, "Titi")
Console.WriteLine("numéro client " & cli.GetNumeroClient())
```

Constructeur par défaut

```
' constructeur par défaut vide
   Public Sub New()
      ' vide
   End Sub
```

Un constructeur par défaut existe déjà pour chaque classe si aucun autre constructeur n'est déclaré. A partir du moment où le constructeur d'initialisation de la classe **`Client`** existe, il devient impossible de déclarer une instance comme on l'a fait dans le T.P. précédent :

```
Dim cli As Client = New Client
```

Pour qu'une telle déclaration, sans paramètre d'initialisation, soit encore possible, il faut créer un ***constructeur par défaut***. En fait ce n'est réellement indispensable que si une instanciation de l'objet, avec des valeurs par défaut pour ses propriétés, a un sens.

Pour la classe **`Client`**, on peut s'interroger sur le sens des valeurs par défaut des propriétés.

Constructeur de recopie

Le constructeur de recopie permet de recopier les propriétés d'un objet existant dans vers la nouvelle instance

```
Dim client1 As Client = New Client(1, "AAA")                    // 1
Dim client2 As Client = New Client(client1)                     // 2
```

En 1, client1 est un objet créer via le constructeur d'initialisation. La propriété nomClien" est initialisé par "AAA"

En 2, salarie2 est un objet différent de salarie1 mais les propriétés des deux objets ont les mêmes valeurs. Pour cela, le développeur doit écrire un constructeur de recopie.

```
' constructeur de recopie
  Public Sub New(ByVal unClient As Client)
    Me.numeroClient = unClient.numeroClient
    Me.nomClient = unClient.nomClient
    Me.caClient = unClient.caClient
  End Sub
```

3.2.3 Propriétés de classe

Jusqu'à présent, les propriétés déclarées étaient des ***propriétés d'instance***. C'est à dire que les propriétés de chaque objet, instancié à partir de la même classe, peuvent avoir des valeurs différentes (numero, nom, etc).

Supposons donc maintenant, que dans la classe **`Client`**, il soit nécessaire de disposer d'un compteur d'instance, dont la valeur serait le nombre d'instances en cours à un instant donné.

En VB.NET il est possible de créer des ***propriétés de classe***. Leur valeur est la même pour toutes les instances d'une même classe. Pour déclarer une telle propriété, on utilise le mot-clé **`Shared`**. Par exemple, dans la classe **`Client`**, le compteur d'instance pourrait être déclaré :

```
Private Shared compteur As Integer = 0
```

La propriété de classe **`compteur`**, initialisée à 0 lors de sa déclaration, sera incrémentée de 1 dans tous les constructeurs développés pour la classe **`Client`**. Sa valeur sera le nombre d'instances valides à un instant donné.

3.2.4 Méthodes de classe

Comme pour les autres propriétés déclarées **`private`**, il est nécessaire de créer les méthodes d'accès associées. Pour ce compteur, seule la méthode ***Get*** est nécessaire.

Cependant, comme les propriétés de classe sont partagées par toutes les instances de la classe, le fait d'envoyer un message ***Get*** pour obtenir leur valeur n'a pas de sens.

En reformulant, on pourrait dire que le message ***Get*** à envoyer pour obtenir le nombre d'instances de classe **`Client`** ne doit pas être envoyée à une instance donnée de cette classe, mais plutôt à la classe elle-même. De telles méthodes sont appelées ***méthodes de classe***.

Une méthode de classe est déclarée par le mot-clé **`Shared`**. Pour la méthode ***Get*** d'accès au compteur d'instance on déclarerait :

```
Public Shared Function GetCompteur() As Integer
    Return compteur
End Function
```

L'appel à une méthode **`Shared`** est sensiblement différent aux appels standard. En effet, ce n'est pas à une instance particulière que le message correspondant doit être envoyé, mais à la classe.

Dans la fonction **`Main`**, par exemple, si l'on veut affecter à une variable le nombre d'instances de la classe Salarie, cela s'écrirait :

```
Dim nInstances As Integer
nInstances = Client.GetCompteur
```

3.2.5 Destructeur

En VB.NET on ne peut pas déclencher explicitement la destruction d'un objet. Les instances sont automatiquement détruites lorsqu'elles ne sont plus référencées. Le programme qui se charge de cette tâche s'appelle le ***Garbage Collector*** ou, en français, le ***ramasse-miettes***. Le ***Garbage Collector*** est un système capable de surveiller les objets créés par une application, de déterminer quand ces objets ne sont plus utiles, d'informer sur ces objets et de détruire les objets pour récupérer leurs ressources.

Or si l'on veut que le compteur d'instances soit à jour, il est nécessaire de connaître le moment ou les instances sont détruites pour décrémenter la variable **`compteur`**.

C'est pour cela que le ***ramasse –miettes*** envoie un message à chaque instance avant qu'elle soit détruite, et ceci quelle que soit la classe. Il suffit d'écrire la méthode de réponse à ce message, c'est la méthode **`Finalize()`**. Cette méthode s'appelle ***destructeur***. C'est , bien sûr, une méthode d'instance de la classe que l'on est en train de développer. Dans la classe **`Client`** par exemple, elle se déclare OBLIGATOIREMENT de la façon suivante :

```
Protected Overrides Sub Finalize()
    compteur = compteur - 1
End Sub
```

Le problème est que l'on ne sait pas quand le ramasse miettes intervient …

3.3 TRAVAIL A REALISER

- A partir du travail réalisé au T.P. N° 2, implémenter les constructeurs et le destructeur de la classe **`Salarie`**.
- Afin de mettre en évidence les rôles respectifs des constructeurs et du destructeur, implémenter ceux-ci pour qu'ils affichent un message à chaque fois qu'ils sont exécutés.
- Implémenter un compteur d'instances pour la classe **`Salarie`**.
- Ajouter une méthode de classe permettant de mettre le compteur à zéro ou à une valeur prédéfinie.
- Modifier le jeu d'essai de l'application Main pour tester ces fonctions.

T.P. N°4 - L'HERITAGE

4.1 OBJECTIFS

4.2

- Généralités sur l'héritage
- Dérivation de la classe racine **`Object`**

4.3 CE QU'IL FAUT SAVOIR

4.4

4.2.1 L'héritage

Le concept d'héritage est l'un des trois principaux fondements de la Programmation Orientée Objet, le premier étant l'encapsulation vu précédemment dans les T.P. 1 à 3 et le dernier étant le polymorphisme qui sera abordé plus loin dans ce document.
L'héritage consiste en la création d'une nouvelle classe dite ***classe dérivée*** à partir d'une classe existante dite ***classe de base*** ou ***classe parente***.

L'héritage permet de :

- **récupérer** le comportement standard d'une classe d'objet (classe parente) à partir de propriétés et des méthodes définies dans celles-ci,

- **ajouter** des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée,

- **modifier** le comportement standard d'une classe d'objet (classe parente) en surchargeant certaines méthodes de la classe parente dans la classe dérivée.

-

Exemple de la classe Client. Nous rajoutons une méthode avec le mot clé Overridable

```
' méthode que l'on peut redéfinir
   Public Overridable Function Finance() As String
      ' méthode crée pour exemple de redéfinition
      ' renvoie le CA pour un Client
      ' renvoie le CA + le taux de remise pour un Grossiste
      Return "Le ca est : " + Me.caClient

   End Function
```

Nous créons une classe Grossiste suivante :

```
Public Class Grossiste
   Inherits Client
   Private txRemise As Double
   ' le taux de remise appliqué au CA du client permet de calculer la remise
   Public Function GetTauxRemise() As Double
      Return txremise
```

```
    End Function
    ' calcul de la remise
    Public Function CalculRemise() As Double
        Return txremise * Me.GetCAClient
    End Function
End Class
```

Cette classe hérite de la classe Client par Inherits Client.

Elle possède une propriété supplémentaire txRemise et l'accesseur associé (il faut un Set en plus évidemment) . Elle possède une méthode spécifique CalculRemise.

Nous allons redéfinir la méthode Finance pour rajouter au chiffre d'affaire le taux de remise sous forme de chaîne.

```
Public Overrides Function Finance() As String
        Return MyBase.Finance + " et taux : " + txRemise
End Function
```

Le mot clé Overrides idique que l'on redéfini la méthode finance.

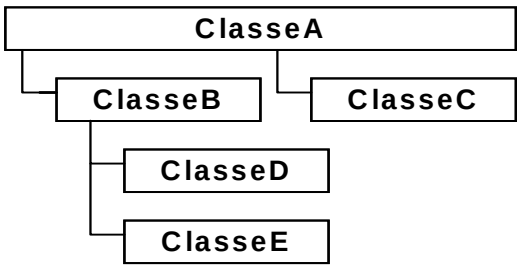
4.2.2 Protection des propriétés et des méthodes

En plus des mots-clés **`public`** et **`private`** décrits dans les chapitres précédents, il est possible d'utiliser un niveau de protection intermédiaire de propriétés et des méthodes par le mot-clé **`protected`**.

- **`public`** - Après ce mot clé, toutes les données ou fonctions membres sont accessibles.
- **`private`** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées et ne seront pas accessibles dans les classes dérivées.
- **`protected`** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées mais sont néanmoins accessibles dans les classes dérivées.

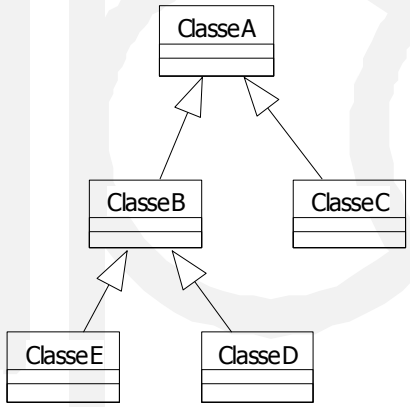
4.2.3 Mode de représentation

Le concept d'héritage peut être utilisé pratiquement à l'infini. On peut créer des classes dérivées à partir de n'importe quelle autre classe, y compris celles qui sont déjà des classes dérivées.
Supposons que **`ClasseB`** et **`ClasseC`** soient des classes dérivées de **`ClasseA`** et que **`ClasseD`** et **`ClasseE`** soient des classes dérivées de **ClasseB**. Les instances de la classe **`ClasseE`** auront des données et des fonctions membres communes avec les instances de la classe **`ClasseB`**, voire de la classe **`ClasseA`**. Si les dérivations sont effectuées sur plusieurs niveaux, une représentation graphique de l'organisation de ces classes devient indispensable. Voici la représentation graphique de l'organisation de ces classes :



Le diagramme ci-dessus constitue la représentation graphique de la ***hiérarchie de classes*** construite à partir de **`ClasseA`**.

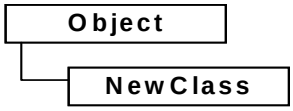
Dans le cadre de la conception orientée objet, la méthode UML (United Modeling Language) propose une autre représentation graphique d'une telle hiérarchie :



4.2.5 Insertion d'une classe dans une hiérarchie

Bien que l'on ait opéré comme cela depuis le début de ce cours, la création d'une nouvelle classe indépendante et isolée n'a pratiquement aucun intérêt en Programmation Orientée Objet. Afin de rendre homogène le comportement des instances d'une nouvelle classe il est important de l'insérer dans une bibliothèque de classe existante. Cette bibliothèque est fournie sous la forme d'une ***hiérarchie de classes*** construite à partir d'une ***classe racine***.

En VB.NET, il est impossible de créer une classe isolée. En effet, lorsqu'on crée une nouvelle classe sans mentionner de classe de base, c'est la classe **`Object`** , la classe racine de toutes les classes C#, qui est utilisée.



`NewClass` est une nouvelle classe insérée dans la hiérarchie de classes construite à partir de **`Object`**.

Le Framework .NET propose une hiérarchie de classes normalisées prêtes à l'emploi, ou à dériver.

4.2.6 *Insertion d'une nouvelle classe à partir de Object*

La dérivation d'une classe par rapport à une classe prédéfinie doit obéir à un certain nombre de règles définies dans la documentation de la ***classe de base*** à dériver. L'une de ces contraintes est la surcharge OBLIGATOIRE de certaines méthodes de la classe de base avec la même signature. Pour dériver la classe **`Object`**, cas le plus courant, cela se résume à ne réécrire que quelques méthodes :

Méthode Finalize

Il s'agit du destructeur dont on déjà décrit le rôle dans le chapitre précédent.

Méthode ToString

Cette méthode doit créer une chaîne de caractères (instance de la classe **`String`**) qui représente les propriétés des instances de la classe. Par exemple, pour la classe **`Fraction`**, la chaîne de caractères représentant une fraction pourrait être **`"nnnn/dddd"`** où **`nnnn`** et **`dddd`** correspondraient respectivement aux chiffres composant le numérateur et le dénominateur de la fraction.

```
Public Overrides Function ToString() As String

        return numerateur + "/" + denominateur;
End Function
```

La méthode **`ToString`** est appelée lorsqu'une conversion implicite d'un objet en chaîne de caractères est nécessaire comme c'est le cas pour la fonction **`WriteLine de l'objet Console`**, par exemple :

```
Console.WriteLine("fraction = " + fr);
```

Méthode Equals

Cette méthode doit répondre VRAI si deux instances sont rigoureusement égales. Deux conditions sont à vérifier :

- Il faut que les deux objets soient de la même classe. Le paramètre de la méthode **`Equals`** étant de type **`Object`**, notre instance peut donc être comparée à une instance d'une classe quelconque dérivée de **`Object`**.

- Il faut qu'une règle d'égalité soit appliquée : par exemple deux objets client sont égaux si leurs numéros sont égaux .; pour la classe **`Fraction`**, on peut dire que deux fractions sont égales si le produit des *extrêmes* est égal au produit des mo*y*ens. Ce qui peut s'écrire :

```
Public Overloads Function Equals(ByVal fr As Fraction) As Boolean
```

```
        Dim c1 As Long
        Dim c2 As Long
        c1 = Me.numerateur * (fr.denominateur)
        c2 = Me.denominateur * (fr.numerateur)
        If c1 = c2 Then
            Return True
        Else
            Return False
        End If
    End Function
```

Le mot clé Overloads est utilisé ici car la liste d'arguments est diufférentye de la liste d'origine . En effet, le paramètre passé est de la classe Fraction et non de la classe Object.

4.2.7 Constructeur et héritage

Chaque constructeur d'une classe dérivée doit obligatoirement appeler le constructeur équivalent de la classe de base.
Si **`BaseClass`** est une classe de base appartenant à la hiérarchie de classe construite à partir de **`Object`**, si **`NewClass`** est une classe dérivée de **`BaseClass`**, cela se fait de la façon suivante en utilisant le mot-clé **`MyBase`** :

Exemples sur la classe Grossiste dont la classe de base est Client :

Constructeur par défaut

```
' constructeur par défaut
    Public Sub New()
        MyBase.New()    ' appel constructeur par défaut de la classe de base
    End Sub
```

Constructeur d'initialisation

```
' constructeur d'initialisation le CA est augmenté par la méthode AugmenterCA
    Public Sub New(ByVal numero As Integer, ByVal nom As String, ByVal
txRemise As Double)
        MyBase.New(numero, nom)
        Me.txRemise = txRemise

    End Sub
```

L'appel de **Mybase(valeurs)** initialise la partie de l'objet créé avec Client en utilisant le constructeur d'initialisation de celle-ci . Puis on complète l'objet en initialisant les attributs spécifiques à la classe Grossiste.

4.2.8 Appel aux méthodes de la classe de base

Lors de la surcharge d'une méthode de la classe de base dans une classe dérivée, il peut être utile de reprendre les fonctionnalités standard pour y ajouter les nouvelles fonctionnalités. Pour ne pas avoir à réécrire le code de la classe de base (dont on ne dispose pas forcément), il est plus simple de faire appel à la méthode de la classe de base. Pour cela on utilise le mot-clé **MyBase** avec une syntaxe différente à celle utilisée pour le constructeur :

```
' méthode bidon pour tester l'appel à partir de la sous-classe Grossiste
   Public Overridable Function Bidon() As String
      Return "AAAAA"
   End Function
```

```
' appel méthode de la classe de base Client
   Public Overrides Function Bidon() As String
      Return MyBase.Bidon + "BBB"

   End Function
```

4.3 TRAVAIL A REALISER

Pour ce travail, à réaliser à partir de ce qui a été produit au T.P. N° 3, on procédera en deux temps :

- Ajouter à la classe **`Salarie`** les méthodes **`Equals`** et **`ToString`**. La règle d'égalité pour la classe **`Salarie`** peut s'énoncer de la façon suivante : deux salariés sont égaux s'ils ont le même numéro de matricule et le même nom. **`ToString`** doit renvoyer toutes les propriétés séparées par des virgules.
- Créer une classe **`Commercial`** en dérivant la classe **`Salarie`**. Cette classe aura 2 propriétés supplémentaires pour calculer la commission :
  - chiffre d'affaire **`chiffreAffaire`** **`Double`**
  - commission en % **`commission`** **`Integer`**
    - Créer les deux constructeurs standards de la classe **`Commercial`**. Ne pas oublier d'appeler les constructeurs équivalents de la classe de base.
    - Créer les méthodes d'accès aux propriétés supplémentaires.
    - Surcharger la méthode **`CalculerSalaire`** pour calculer le salaire réel (fixe + commission).
    - Surcharger les autres méthodes de la classe de base pour lesquelles on jugera nécessaire de faire ainsi.

Tester les classes Salarie et Commercial

T.P. N°5 – LES COLLECTIONS

5.1 OBJECTIFS

- Tableaux statiques
- ArrayList
- Dictionnaire Hashtable
- Dictionnaire trié SortedList

5.2 CE QU'IL FAUT SAVOIR

5.2.1 Les tableaux statiques

Les tableaux contiennent des éléments, chacun d'eux étant repéré par son indice. En VB.NET il existe une manière très simple de créer des tableaux "classiques" sans faire référence aux classes ***collections*** :

```
Dim aTab(20) As Integer
Dim i As Integer
For i = 0 To 19
   aTab(i) = i + 1900
Next

Dim aStr(5) As String
aStr(0) = "André"
aStr(1) = "Mohamed"
aStr(2) = "Marc"
aStr(3) = "Abdelali"
aStr(4) = "Paul"

Dim aSal(5) As Salarie
aSal(0) = New Salarie(16, 1, 10, "CAUJOL", 10900.0)
aSal(1) = New Salarie(5, 1, 10, "DUMOULIN", 15600.0)
aSal(2) = New Salarie(29, 3, 20, "AMBERT", 5800.0)
aSal(3) = New Salarie(20, 2, 20, "CITEAUX", 8000.0)
aSal(4) = New Salarie(34, 2, 30, "CHARTIER", 7800.0)
```

Le problème de ce type de tableaux réside en deux points :

- Tous les éléments du tableau doivent être de **même type**.
- Le nombre d'éléments que peut contenir un tableau est limité au moment de la déclaration. Cela sous-entend que le programmeur connaît, au moment de l'écriture du programme, le nombre maximum d'éléments que doit contenir le tableau.

Itération dans le tableau

```
For i = 0 To 4
        Console.WriteLine("nom : " + aStr(i))
Next
```

5.2.2 Classe ArraysList

C'est un tableau dynamique auquel on peut rajouter ou insérer des éléments, en supprimer.
Il faut utiliser le namespace System.Collections :

```
Imports  System.Collections
Dim al as ArrayList = new ArrayList()
al.Add(1)
al.Add(45)
al.Add(87)
al.Remove(1)
etc etc
```

Quelques méthodes de la classe ArrayList

Méthodes ou propriétés	But
Capacity	Détermine le nombre d'éléments que le tableau peut contenir
Count	Donne le nombre d'éléments actuelleùent dans le tableau
Add(object)	Ajoute un élément au tableau
Remove(object)	Enlève un élément du tableau
RemoveAt(int)	Enlève un élément à l'indice fourni
Insert(int, object)	Insère un élément à l'indice fourni
Clear()	Vide le tableau
Contains(object)	Renvoie un booléen vrai si l'objet fourni est présent dans le tableau
al[index]	Fournit l'objet situé à la valeur de index

Explorer le tableau (al contient

```
Dim al As ArrayList = New ArrayList
Dim objsal As new Salarie()
al.Add(sal)
'etc etc
For Each objsal In al
        Console.WriteLine(objsal)
Next
```

Accès à un élément du tableau (ici le deuxième)

```
sal = al(1)
```

5.2.3 SortedList

SortedList est un dictionnaire qui garantit que **les clés sont rangées de façon ascendante**.

Exemple :

```
static void Main()
{
        SortedList sl = new SortedList();
        sl.Add(32, "Java");
        sl.Add(21, "C#");
        sl.Add(7, "VB.Net");
        sl.Add(49, "C++");
        Console.WriteLine("Les éléments triés sont...");
        Console.WriteLine("\t Clé \t\t Valeur");
        Console.WriteLine("\t === \t\t =====");
        for(int i=0; i<sl.Count; i++)
        {
        Console.WriteLine("\t {0} \t\t {1}", sl.GetKey(i), sl.GetByIndex(i));
        }
 }
```

Méthodes ou propriétés	But
Keys	Collection des clés de la liste triée
Values	Collection des valeurs (objets) de la liste
Count	Donne le nombre d'éléments actuelleùent dans la liste
Add(object key, object value)	Ajoute un élément à la liste (paire clé-valeur)
Remove(object key)	Enlève un élément e la liste dont la valeur correspond à la clé
RemoveAt(int)	Enlève un élément à l'indice fourni
Clear()	Vide la liste
ContainsKey(object key)	Renvoie un booléen vrai si la clé fournie est présent dans la liste
ContainsValue(object value)	Renvoie un booléen vrai si la valeur fournie est présent dans la liste
GetKey(int index)	Renvoie la clé correspondant l'indice spécifié
GetByIndex(int index)	Renvoie la valeur correspondant à l'indice spécifié
IndexOfKey(object key)	Renvoie l'indice correspondant à une clé
IndexOfValue(object value)	Renvoie l'indice correspondant à une valeur

5.3 TRAVAIL A REALISER

- Tester quelques méthodes sur les différentes collections : ArrayList, SortedList

A partir de la classe **`Salarie`** implémentée dans les T.P. précédents, dans la fonction **`Main`** de `l'application`,

- Créer une instance d'une collection **SortedList** dans laquelle on va ranger des instances de la classe **`Salarie`** repérées par leur numéro de matricule.
- Créer au moins cinq instances de la classe **`Salarie`** et les insérer dans la collection.
- Faire l'itération de la collection pour afficher son contenu par **ordre croissant** des numéros de matricules.
- Chercher un salarié dans la collection en fournissant son matricule

T.P. N°6 - POLYMORPHISME

6.1 OBJECTIFS

6.2

- Polymorphisme
- Fonctions virtuelles
- Classes génériques

6.2 CE QU'IL FAUT SAVOIR

6.2.1 Polymorphisme

Considérons l'exemple suivant.

```
Dim c1 As Salarie = New Commercial
Salaire = sal.CalculerSalaire()
```

La variable **sal** est déclarée de type **Salarie**, mais elle est instanciée à partir de la classe **Commercial**. Une telle affectation est correcte et ne génère pas d'erreur de compilation. La question que l'on doit se poser maintenant est la suivante : lors de l'envoi du message **calculSalaire** à **sal**, quelle est la méthode qui va être exécutée ? La méthode **CalculerSalaire** de la classe **Salarie** qui calcul le salaire uniquement à partir du salaire de base, ou la méthode **CalculerSalaire** de la classe **Commercial** qui prend en compte les propriétés commission et chiffre d'affaire de la classe **Commercial** ?

Un simple test va permettre de mettre en évidence que c'est bien la méthode **CalculerSalaire** de la classe **Commercial** qui est exécutée. VB.NET *sait* et *se souvient* comment les objets ont été instancié pour pouvoir appeler la bonne méthode.

En Programmation Orientée Objet, c'est ce que l'on appelle le concept de ***Polymorphisme***. Pour toutes les instances de salarié, quelles soient instanciées à partir de la classe de base **Salarie** ou d'une classe dérivée comme **Commercial**, il faut pouvoir calculer le salaire. C'est le ***comportement polymorphique***. Par contre le calcul ne se fait pas de façon identique pour les salariés commerciaux et les non commerciaux.

6.2.2 Méthodes virtuelles

L'ambiguïté est levée du fait que la résolution du lien (entre le programme appelant et la méthode) ne se fait pas au moment de la compilation, mais pendant l'exécution en fonction de la classe qui a été utilisée pour l'instanciation. On parle de ***méthode virtuelle***. **`calculSalaire`** est une méthode virtuelle qui défini un comportement polymorphique sur la hiérarchie de classe construite sur **`Salarie`**.
Pour associer un comportement polymorphique à une méthode, il suffit de "surcharger" la méthode la classe de base, avec exactement la même signature, c'est-à-dire le même nom, le même type de résultat, le même nombre de paramètres, de mêmes types, dans le même ordre.
La classe racine **`Object`** est elle-même un polymorphisme. Les méthodes **`Finalize`**, **`ToString`** et **`Equals`**, surchargées dans les T.P. précédents pour les classes **`Salarie`** et **`Commercial`**, sont des fonctions virtuelles qui définissent un comportement polymorphique pour toutes les classes dérivées de **`Object`**.

6.2.3 Classes génériques

Dans certains cas, lors de l'élaboration d'une hiérarchie de classe ayant un comportement polymorphique, il peut être intéressant de mettre en facteur le comportement commun à plusieurs classes, c'est-à-dire créer une ***classe générique***. Il est probable que le fait de créer une instance de cette classe ne correspond pas à une réalité au niveau conceptuel. C'est le cas de la classe racine **`Object`** : Il est absurde de créer une instance de la classe **`Object`** qui, par ailleurs, définit le comportement commun à toutes les classes C#. C'est le cas également de la classe **`Dictionary`** qui définit le comportement commun à toutes les classes dictionnaire, mais qui ne peut être instanciée. Une telle classe est déclarée **`abstract`** :

```
Public abstract Class Dictionary

   ' Définitions des membres de la classe

End Class
```

Essayer d'instancier une classe **`abstract`** avec l'instruction **`New`** va générer une erreur de compilation.

6.3 TRAVAIL A REALISER

- Modifier la fonction **`Main`** développée dans le T.P. précédent en insérant quelques instances de la classe **`Commercial`** dans la liste triée.

- Vérifier le fonctionnement du *Polymorphisme* lié à la fonction **`CalculerSalaire`**.

CONCLUSION

Ici se termine la deuxième partie de ce cours. On peut en déduire une méthodologie de construction d'objets dont la démarche peut être résumée dans les étapes ci-dessous :

- Enumérer les données membres d'un objet.
- Protéger les données membres et créer les fonctions d'accès à ces propriétés.
- Créer, même s'ils ne sont pas indispensables, au moins les deux constructeurs (défaut, initialisation) et le destructeur si nécessaire.
- Enumérer les opérations à opérer sur l'objet (Entrées/sorties, calcul, comparaison, conversion, etc.).
- Utiliser l'héritage
- Utiliser le polymorphisme

Dans un support suivant nous allons apprendre à gérer les exceptions, l'agrégation et la composition et utiliser les interfaces .