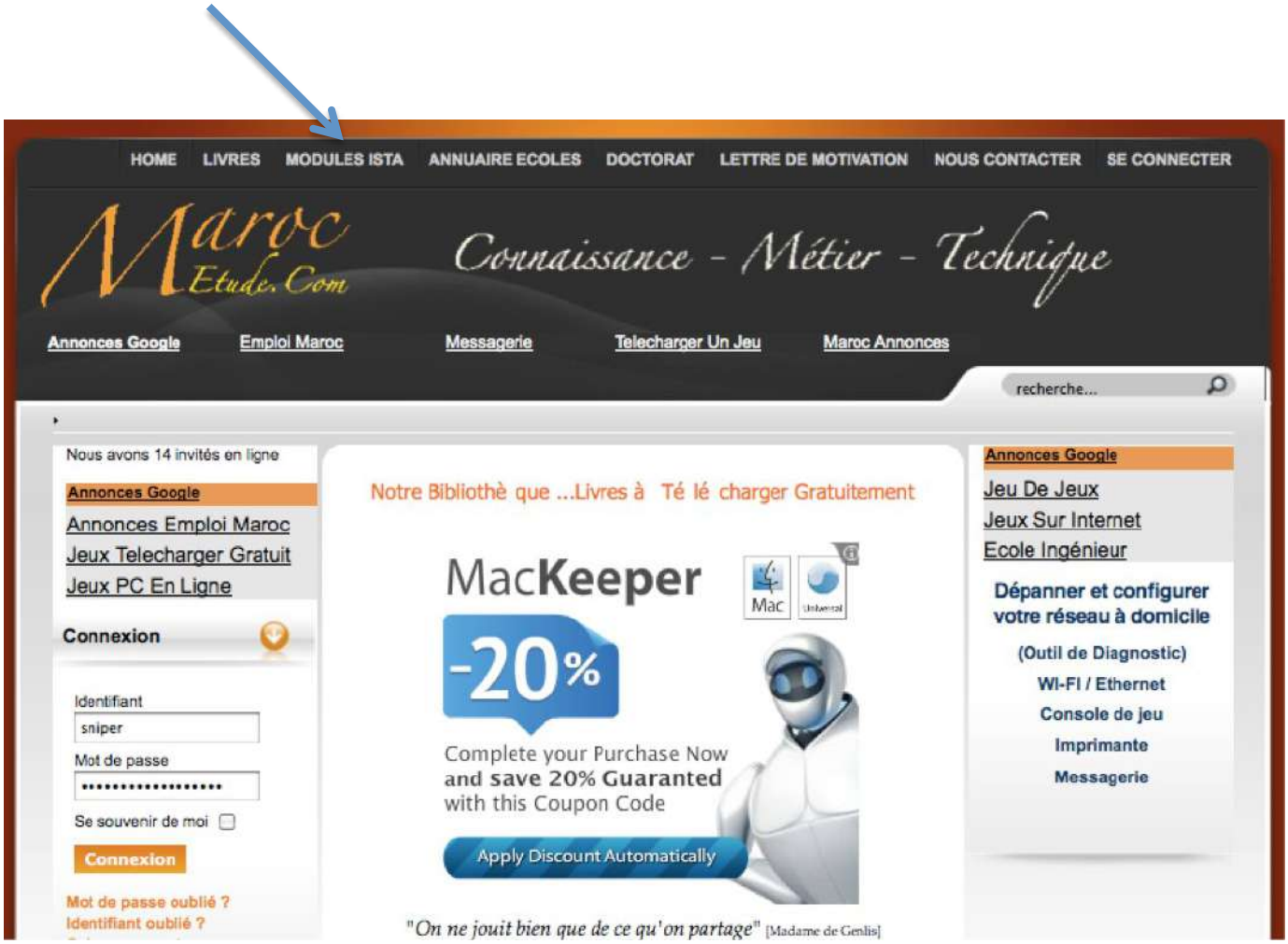
PORTAIL DE LA FORMATION PROFESSIONNELLE AU MAROC

Télécharger tous les modules de toutes les filières de l'OFPPT sur le site dédié à la formation professionnelle au Maroc : **www.marocetude.com**

Pour cela visiter notre site www.marocetude.com et choisissez la rubrique :

MODULES ISTA



M14 : POO

VB.Net

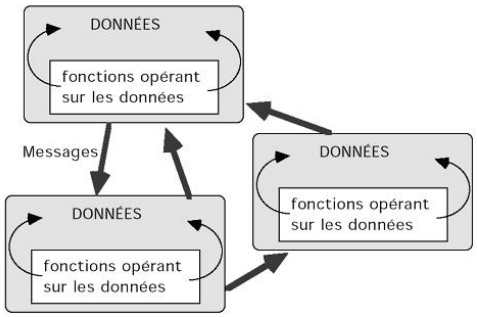
Formateur : Driouch(cfmoti.driouch@gmail.com)
Etablissement : OFPPT/CFMOTI
24/06/2011

Plan

- Notion d'Objet – Classe - Instanciation
- Encapsulation (Propriété et Méthodes d'accès)
- Constructeurs (Constructeur de copie) – Destructeur(Garbage Collector)
- Héritage (Surcharge Méthode – Redéfinition)
- Polymorphisme
- Les Exceptions
- Les Objets Courants(en .Net)
- Objet de collection
- Les Interfaces
- La persistance (Sérialisation binaire et XML)

Notion d'objet

- Un objet est une entité cohérente rassemblant des données et du code travaillant sur ces données.



Classe

- Une classe est une description abstraite d'un objet. Elle peut être considérée en quelque sorte comme un moule…

Véhicule
+Marque : string(idl) +Puissance fiscale : int +Vitesse maximale : int +Vitesse courante : int
+Creer un véhicule() +Détruire un véhicule() +Démarrer() +Accélérer(entrée Taux : int)

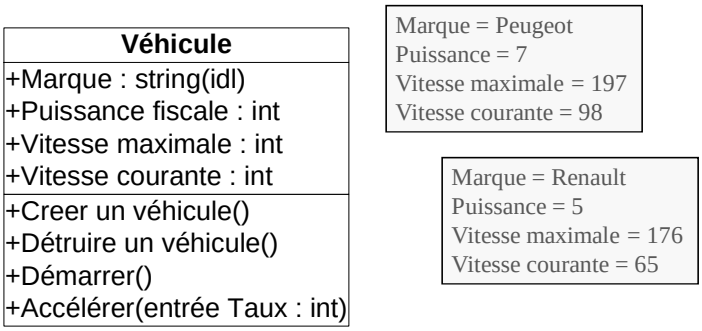
Nom de la classe

Description des attributs ou données membres

Description des méthodes = code associé

Instanciation

- Chaque objet correspond à une instance de la classe à laquelle il fait référence.



Instanciation

- La création d'un objet est constituée de deux phases :
  - Une phase du ressort de la classe : allouer de la mémoire et un contexte d'exécution minimaliste. Méthode de classe
  - Une phase du ressort de l'objet : initialiser ses attributs d'instance. Méthode d'instance
- Dans les langages tels que Java, C++, VB ou C#, ces deux phases ne sont pas différenciées.
- Appel à une méthode spéciale : le constructeur

Encapsulation

- Abstraction de données :
  La structure d'un objet n'est pas visible de l'extérieur. Seule son interface est accessible par le biais de messages invocables par le client.
- Abstraction procédurale :
  L'invocation d'un message est une opération atomique. Aucun élément d'information sur les traitements internes mis en œuvre n'est nécessaire.
- Donc pour chaque objet créer en peut limiter les accès ou les contrôler selon nos besoin pour cet objet.
  - Intérieur de l'objet protégé
  - Complexité dissimulé
  - Maintenance simplifié (centralisation)
  - Echanges avec l'externe sont codifiés

Propriétés

- 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.

Méthodes d'accès

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.

Afin d'implanter **correctement** le concept d'encapsulation, il convient de **verrouiller** l'accès aux propriétés et de les déclarer **private**

L' Accès aux attributs membres peut se faire par des méthodes simple ou par les Property de classe

```
Public Property nomClient() as type
  Get
    Return iNomClient
  End Get
  Set(ByVal Value as type)
    iNomClient = Value
  End Set
End Property
```


Constructeur

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 autres, d'initialiser chaque instance pour que ses propriétés aient un contenu cohérent.

```
Public Sub New(ByVal numero As Integer, ByVal nom As String)
    Me.IDClient = 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, "Sharraf")
```

Constructeur de recopie

Le constructeur de recopie permet de recopier les propriétés d'un objet existant vers une nouvelle instance de même type.

```
Public Sub New(ByVal unClient As Client)
   Me.IDClient = unClient.IDClient
   Me.NomClient = unClient.NomClient
   Me.CaClient = unClient.CaClient
End Sub

Dim oClient1 as new Client()
Dim oClient As Client = New Client(oClient1)
```


Exemple

```
Public Class Vehicule
   Private Marque As String
   Private Puissance As Integer
   Private VitesseMax As Integer
   Private VitesseCour As Integer
   ' Constructeur par defaut
   Public Sub New()
      Marque = "Marque inconu"
      Puissance = 0
      VitesseMax = 0
      VitesseCour = 0
   End Sub
   'Constructeur d'initialisation
   Public Sub New(ByVal M As String, ByVal P As Integer, ByVal VM As Integer)
      Marque = M
      Puissance = P
      VitesseMax = VM
      VitesseCour = 0
   End Sub
   'Constructeur de Copie
   Public Sub New(ByVal Vh As Vehicule)
      Marque = Vh.PMarque
      Puissance = Vh.PPuissanceF
      VitesseMax = Vh.PVitesseM
      VitesseCour = Vh.PVitesseC
   End Sub
```

Exemple

```
Public Property PMarque() As String
    Get
        Return Marque
    End Get
    Set(ByVal value As String)
        Marque = value
    End Set
End Property
Public Property PPuissanceF() As Integer
    Get
        Return Me.Puissance
    End Get
    Set(ByVal value As Integer)
        Me.Puissance = value
    End Set
End Property
Public Property PVitesseM() As Integer
    Get
        Return Me.VitesseMax
    End Get
    Set(ByVal value As Integer)
        Me.VitesseMax = value
    End Set
End Property
```


Exemple

```
Public ReadOnly Property PVitesseC() As Integer
    Get
        Return Me.VitesseCour
    End Get
End Property
Public Sub accelerer(ByVal taux As Integer)
    VitesseCour += taux
    If VitesseCour > VitesseMax Then
        VitesseCour = VitesseMax
    End If
End Sub
 Public Overrides Function ToString() As String 'ToString redéfinie a partire
 de la classe de base Objet
    Return Me.Id & ": " & Me.PMarque & "( -P:" & Me.PPuissanceF & " -VM:"
 & Me.PVitesseM & " -VC:" & Me.PVitesseC & ")"
End Function
End Class
```

Programme principale

```
Sub main()
  Dim Rn As new vehicule()
  Console.Writeline(Rn.ToString())
  Dim Pg As new vehicule(« Peugeot »,7,187)
  Console.Writeline(Pg.ToString())

  Pg.Accelerer(60)
  Console.Writeline(Pg.ToString())

  Pg.Accelerer(80)
  Pg.Accelerer(60)
  Console.Writeline(Pg.ToString())
  Console.ReadLine()
End Sub
```


Destructeur

En VB.NET on ne déclenche pas explicitement la destruction d'un objet. Les instances seront détruites par le système lorsqu'elles ne sont plus référencées et qu'il sera nécessaire de récupérer des ressources mémoire. Le programme qui se charge de cette tâche s'appelle le ***Garbage Collector*** ou, en français, le ***ramasse-miettes***.

- Les destructeurs ne peuvent pas être définis dans des struct. Ils sont utilisés uniquement avec les classes.
- Une classe peut posséder un seul destructeur.
- Les destructeurs ne peuvent pas être hérités ou surchargés.
- Les destructeurs ne peuvent pas être appelés. Ils sont appelés automatiquement.
- Un destructeur n'accepte pas de modificateurs ni de paramètres.

Finalize(). Cette méthode s'appelle ***destructeur***.

```
Protected Overrides Sub finalize()
  'libération des ressources
End Sub
```

Dans le programme principale on peut forcé le Garbage Collector par la procédure: GC.Collect()

Propriétés et Méthodes partagé

```
Public class test
  Public Shared Attribut As Type
  Public Shared Sub NomMeth()
    …
  End Function
End Class
```

L'utilisation de l'attribut ou la méthode se fait via la classe directement.
Test.NomMeth Ou test.Attribut
Exp :

```
Private Shared iCompteur As Integer

Public Sub new()
 iCompteur += 1
End Sub

Protected Overrides Sub finalize()
   iCompteur -= 1
End Sub
```


Les opérateurs d'accessibilité

Mot clé	Définition
Public	Accessible partout
Private	Accès dans la classe uniquement
Friend	Accès Classe - Espace de nom Assemblage
Protected	Accès classe et classes dérivées

Héritage

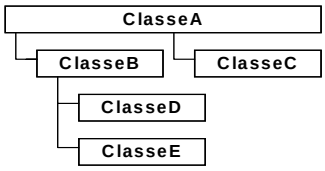
Le concept d'héritage est l'un des trois principaux fondements de la Programmation Orientée Objet, le premier étant l'encapsulation vu précédemment et le dernier étant le polymorphisme qui sera abordé plus loin dans ce document

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 :

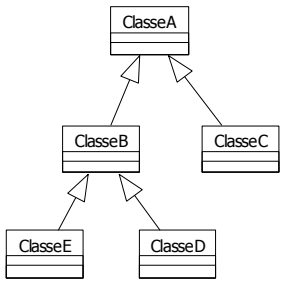
- **Récupérer** le comportement standard d'une classe d'objet (classe parente) à partir des propriétés et des méthodes définies dans celles-ci.
- **Ajouter** des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée.
- **Modifier** le comportement standard d'une classe d'objet (classe parente) en surchargeant certaines méthodes de la classe parente dans la classe dérivée.

Représentation



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 :



Exemple

```
Public Class Client
    Protected NomClient As String
    Protected CAClient As Single
    Public Sub New()
    End Sub
    Public Sub New(ByVal nom As String, ByVal ca As Single)
        Me.NomClient = nom
        Me.CAClient = ca
    End Sub
    Public Property Nom() As String
        Get
            Return Me.NomClient
        End Get
        Set(ByVal value As String)
            Me.NomClient = value
        End Set
    End Property
    Public WriteOnly Property CA() As Single
        Set(ByVal value As Single)
            Me.CAClient = value
        End Set
    End Property
    Public Overridable Function finance() As Single
        Return Me.CAClient
    End Function
    Public Overrides Function ToString() As String
        Return "Id: " & Me.Id & "  Nom: " & Me.Nom & "  CA : " & Me.finance()
    End Function
End Class
```


Exemple

```
Public Class Grossiste
    Inherits client
    Private TxRemiseClient As Single
    Public Sub New()
        MyBase.New()
    End Sub
    Public Sub New(ByVal nom As String, ByVal ca As Single, ByVal rm As Single)
        MyBase.New(nom, ca)
        Me.TxRemise = rm
    End Sub
    Public Property TxRemise() As Single
        Get
            Return Me.TxRemiseClient
        End Get
        Set(ByVal value As Single)
            Me.TxRemiseClient = value
        End Set
    End Property
    Public Function calrm() As Single
        Return Me.CAClient * Me.TxRemise
    End Function
    Public Overrides Function finance() As Single
        Return Me.CAClient * (1 - Me.TxRemise)
    End Function
End Class
```

Surcharge des méthodes

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

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

*Le mot clé **Overrides** indique une surcharge avec la même signature ou bien une redéfinition dans l'héritage.*

*Le mot clé **Overloads** est utilisé pour préciser une surcharge avec une signature différente : le nombre ou le type des paramètres de la méthode diffèrent.*

Exemple

```
' 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

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

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

   Public Overrides Function Finance() As Single
      Return MyBase.ca * (1-Me.txRemise)
   End Function
End Class
```

Réutilisation des méthodes

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

' 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
```

Méthode Finalize: le destructeur vu précédemment.

Méthode ToString:

```
  Pour la classe fraction
  Public Overrides Function ToString() As String
            return numerateur + "/" + denominateur
  End Function
```

Méthode Equals:

```
Public Overloads Function Equals(ByVal fr As Fraction) As Boolean
    Dim c1, 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
```


Complément héritage(Modificateur)

- MustInherit : Ce mot clé indique qu'une classe ne peut être instancier, et qu'elle ne peut donc être utilisée que comme classe de base, une classe abstret.
- NotInheritable : A l'inverse, ce mot-clé indique que cette classe ne peut pas être héritée, et peut etres instancier, c'est-à-dire servir de classe de base.
- Overridable : ce mot-clé indique que cette méthode peut être redéfinie dans une classe dérivé.
- MustOverride : Ce mot clé indique qu'une méthode doit être redéfinie pour chaque classe dérivé.
- NotOverridable : A l'inverse, ce mot-clé indique que cette méthode ne peut être redéfinie dans une classe dérivé.
- Shadows : Spécifie qu'un élément de classe déclaré redéclare et masque un élément de classe de base de même nom.

Polymorphisme

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

```
Dim x As Client
Dim cl As New Client("Ali", 1000)
Dim clg As New Grossiste("Ahmed", 7000, 0.2)
x = cl
Console.WriteLine(x.ToString)
x = clg
Console.WriteLine(x.ToString)
```


Les Interfaces (Implements)

Une interface est une collection de prototypes représentant les membres (propriétés, procédures et événements) que l'interface encapsule. Les interfaces contiennent uniquement les **déclarations** des membres, les classes et les structures implémentent ces membres

Exp:

```
Interface TestInterface
    Property Prop1() As Integer
    Sub Method1(ByVal X As Integer)
End Interface
Class ImplementationClass
    Implements TestInterface
    Private pval As Integer
    Public Sub Method1(ByVal X As Integer) Implements TestInterface.Method1
        MsgBox("The X parameter for Method1 is " & X)
    End Sub
    Public Property Prop1() As Integer Implements TestInterface.Prop1
        Get
            Return pval
        End Get
        Set(ByVal value As Integer)
            pval = value
        End Set
    End Property
End Class
```

Les Exceptions(Gestion Erreur)

Il y a plusieurs types d'erreurs :

- **Les erreurs de syntaxe :** Elle surviennent en mode conception quand on tape le code.
  Exp: A+1=B `Erreur d'affectation
  2 For... et un seul Next
- **Les erreurs de logique :** quand la conception du programme (logiciel) qui est incorrect, des données justes nous donnes des résultats faut. Donc il faut revoir la conception.
- **Les erreurs d'exécution :** Elle surviennent en mode Run ou lors de l'utilisation de l'exécutable, une instruction ne peut pas être effectuée. Le logiciel s'arrête brutalement, c'est très gênant!! Pour l'utilisateur c'est un 'BUG`
  division par zéro
  dim a(3) as string ; ➔ a(5)=« A »
  Soit une erreur de l'utilisateur, Il faut toujours vérifier ce que fait l'utilisateur et prévoir toutes les possibilités.
  Exp: On lui demande de taper un chiffre, il tape une lettre ou rien puis valide

Pour éviter ces derniers il faut capté l'erreur avec Try … Catch … finally

Syntaxe :

```
Dim X as integer = 0
try
 dim y as double= 100/x
catch e as ArithmeticException
  Console.WriteLine("ArithmeticException Handler: {0}", e.ToString()); }
catch (Exception e) { Console.WriteLine("Generic Exception Handler: {0}", e.ToString()); }
finally { Console.WriteLine("Executing finally block."); }
```


Les Exceptions

- Cette classe est la classe de base pour toutes les exceptions. Lorsqu'une erreur se produit, le système ou l'application en cours d'exécution la signale en levant une exception qui contient des informations sur l'erreur. Une fois levée, une exception est gérée par l'application ou par le gestionnaire d'exceptions par défaut.
- Exp:
  - ArithmeticException
  - DivideByZeroException
  - NotFiniteNumberException
  - OverflowException

Exceptions (Exemple)

```
Public Class Personne
    Private _Nom As String
    Private _Age As Integer
    Public Property Nom() As String
        Get
            Return _Nom
        End Get
        Set(ByVal value As String)
            _Nom = value
        End Set
    End Property
    Public Property Age() As Integer
        Get
            Return _Age
        End Get
        Set(ByVal value As Integer)
            If value < 0 Then
                Throw New AgeException("Erreur : Age Négatif")
            Else
                _Age = value
            End If
        End Set
    End Property
    Public Sub New(ByVal No As String, ByVal Ag As Integer)
        Me.Nom = No
        Me.Age = Ag
    End Sub
    Public Overrides Function ToString() As String
        Return Me.Nom & " (Age :" & Me.Age.ToString & ")"
    End Function
End Class
```

```
Public Class AgeException
    Inherits Exception
    Public Sub New()
        MyBase.New()
    End Sub
    Public Sub New(ByVal msg As String)
        MyBase.New(msg)
    End Sub
    Public Sub New(ByVal msg As String, ByVal Inner As
      Exception)
        MyBase.New(msg, Inner)
    End Sub
End Class
```

```
    Dim p As Personne
    Try
        p = New Personne("Ali", 0)
        'p.Age = -5
        p.Age = "A"
    Catch ex As AgeException
        Console.WriteLine(ex.Message)
    Catch ex As Exception
        Console.WriteLine("Error : " & ex.Message)
    End Try
    Console.ReadKey()
```


Objets String

Dim str as string
Avec ça on a un objet str de type string et chaque objet a des méthodes et des attributs

Length : Taille d'une chaîne en nombre de caractère.
Chars(i) : retourne le caractère à la position i.
ToUpper : Mettre en majuscules une chaîne de caractère.
ToLower : transforme par contre la chaîne en minuscule.
Trim : Permet de supprimer des caractères en début et fin de chaîne.
Insert (n,str1) : Insère une chaîne dans une autre.
Remove(N,L) : Supprime la sous-chaine à partir de la position N et de longueur L.
Replace (str1,str2) : Remplace partout dans une chaîne de départ, une chaîne par une autre.
IndexOf & LastIndexOf : Indique le numéro du caractère, la position (la première occurrence) ou une chaîne à chercher est trouvée dans une autre.
Substring(n,l) : Extrait une partie d'une chaîne
…
Exp :
Dim str as string =« Bonjour »
Console.write(str.length) ➔ 7
Console.write(str.replace(« jour », « soir »)) ➔ Bonsoir

L'Objet Regex

La classe Regex permet l'utilisation d'expression régulières. Celles-ci permettent de tester le format d'une chaîne de caractères. Ainsi on peut vérifier qu'une chaîne représentant une date est bien au format jj/mm/aa. On utilise pour cela un modèle et on compare la chaîne à ce modèle. Ainsi dans cet exemple, j m et a doivent être des chiffres. Le modèle d'un format de date valide est alors "\d\d/\d\d/\d\d" où le symbole \d désigne un chiffre.

```
' une expression régulière modèle
Dim modèle1 As String = "\d\d/\d\d/\d\d"
Dim regex1 As New Regex(modèle1)
' comparer un exemplaire au modèle
Dim exemplaire1 As String = " 123 "
If regex1.IsMatch(exemplaire1) Then
   affiche(("[" + exemplaire1 + "] correspond au modèle [" + modèle1 + "]"))
Else
   affiche(("[" + exemplaire1 + "] ne correspond pas au modèle [" + modèle1 +
"]"))
End If
```

Caractère	Description

Les collections

Une alternative aux tableaux est l'usage de Collection. Qui fait partie de l'espace de nom System.Collections

Une collection fonctionne plutôt comme un groupe d'éléments dans laquelle il est possible d'ajouter ou d'enlever un élément à n'importe quel endroit sans avoir à se préoccuper de sa taille ni ou se trouve l'élément.

Le nombre d'élément n'est pas défini au départ comme dans un tableau. Dans une collection il n'y a que les éléments que l'on a ajouté.

Une fois que vous avez créé une collection, vous pouvez effectuer l'une des actions suivantes :

- ajouter un élément par le biais de la méthode Add ;
- supprimer un élément avec la méthode Remove ;
- supprimer tous les éléments via la méthode Clear ;
- Connaître le nombre d'éléments contenus dans la collection avec la propriété Count.
- Vérifier la présence d'un élément spécifique avec la méthode Contains.
- Retourner un élément spécifique de la collection avec la propriété Item.
- parcourir la collection entière à l'aide de For Each...Next, instruction (Visual Basic).

Les éléments sont repérés grâce à un index ou avec une Clé unique

```
Dim Col As New Collection
Col.Add ("Toto") : Ajoute dans la collection
Col.Add("Lulu") :
Col.Add("Titi") :
Col.Remove(1) : supprime l'element « Lulu »
Col.Item(1) : contient "Titi" (le second Item de la collection)
NB: l'index du premier élément et 1
```


Array

La classe **Array** implémente un tableau. Nous utiliserons dans notre exemple les propriétés et méthodes suivantes :

ArrayList

Hashtable

La classe *Hashtable* permet d'implémenter un dictionnaire. On peut voir un dictionnaire comme un tableau à deux colonnes :

clé	valeur
clé1	valeur1
clé2	valeur2
..	...

Sortedlist

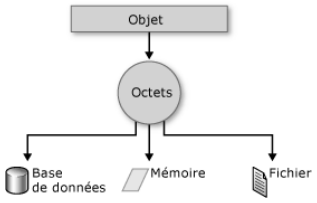
- La séquence d'index est basée sur la séquence de tri. Quand un élément est ajouté, il est inséré dans SortedList dans l'ordre de tri adéquat(une implémentation spécifique de IComparer), et l'indexation s'ajuste en conséquence. Quand un élément est supprimé, l'indexation s'ajuste aussi en conséquence. Par conséquent, l'index d'une paire clé/valeur spécifique peut changer en fonction de l'ajout ou de la suppression d'éléments dans l'objet SortedList.
- Les opérations sur un objet SortedList ont tendance à être plus lentes que les opérations sur un objet Hashtable en raison du tri. Cependant, un objet SortedList est plus flexible, car il permet l'accès aux valeurs par l'intermédiaire des clés associées ou des index.
- Il est possible d'accéder aux éléments de cette collection en utilisant un index d'entiers. Les index de cette collection sont des index de base zéro.

Exercice

- Écrire un programme pour la gestion des comptes bancaire(NumCompte, Solde, Client) et les Clients(Nom, Prénom)
- un client peut avoir plusieurs comptes)
  - Ajouter
  - Rechercher (NumCompte)
  - Supprimer (NumCompte)
  - Affichage (Tableau)

La persistance des objets

- Cette illustration affiche le processus global de la sérialisation :
- L'objet est sérialisé à un flux qui contient non seulement les données, mais également des informations sur le type d'objet, notamment sa version, sa culture et son nom d'assembly. À partir de ce flux, il peut être stocké dans une base de données, dans un fichier ou en mémoire.



Sérialisation binaire et XML

- **Sérialisation binaire :** La sérialisation binaire utilise le codage binaire afin de produire une sérialisation compacte destinée notamment au stockage ou au flux réseau socket. Il n'est pas convenable de faire passer les données dans un pare-feu mais les performances sont meilleures lors du stockage des données.

- **Sérialisation XML :** La sérialisation XML sérialise les champs et les propriétés publics d'un objet, ou les paramètres et valeurs de retour des méthodes, en un flux XML conforme à un document de langage XSD (XML Schema Definition) spécifique. La sérialisation XML favorise des classes fortement typées avec des propriétés et des champs publics convertis en XML. System.Xml.Serialization contient les classes nécessaires pour la sérialisation et la désérialisation XML

Sérialisation binaire (Exemple)

Pour lire ou écrire dans un fichier, on utilise les deux objets StreamReader et StreamWriter dans l'espace de nom System.IO.

Pour écriture:

```
Dim ligne As String = Nothing    ' une ligne de texte
Dim fluxInfos As StreamWriter = Nothing    ' le fichier texte
Try
    ' création du fichier texte
    fluxInfos = New StreamWriter("C:\infos.txt", True)
    Do
        Console.Write("ligne (rien pour arrêter) : ")
            ' lecture ligne tapée au clavier
        ligne = Console.ReadLine().Trim()
        fluxInfos.WriteLine(ligne)
    Loop Until ligne = ""
Catch e As Exception
    Console.WriteLine("L'erreur suivante s'est produite : " & e.Message)
Finally
    Try
        fluxInfos.Close()
    Catch
    End Try
End Try
```


Sérialisation binaire (Exemple)

Pour lecture:

```
Dim ligne As String = Nothing
Dim fluxInfos As StreamReader = Nothing
' lecture contenu du fichier
Try
    fluxInfos = New StreamReader("C:\infos.txt")
    Do
        ligne = fluxInfos.ReadLine()
        Console.WriteLine(ligne)
    Loop Until (ligne Is Nothing)
Catch e As Exception
    Console.WriteLine("L'erreur suivante s'est produite : " & e.Message)
Finally
    Try
        fluxInfos.Close()
    Catch
    End Try
End Try
Console.ReadLine()
```

Sérialisation XML Exemple (XML)

Class

```
Public Class Employee
    Public EmpName As String
    Public EmpID As String
    Public Sub New()
    End Sub
    Public Sub New(ByVal newName As String, ByVal newID As
    String)
        EmpName = newName
        EmpID = newID
    End Sub
End Class
```



```
Public Class Employees
    Implements ICollection
    Public CollectionName As String
    Private empArray As ArrayList = New ArrayList()
    Default Public Overloads ReadOnly Property Item(ByVal index As Integer) As Employee
        Get
            Return CType(empArray(index), Employee)
        End Get
    End Property
    Public Sub CopyTo(ByVal a As Array, ByVal index As Integer) Implements ICollection.CopyTo
        empArray.CopyTo(a, index)
    End Sub
    Public ReadOnly Property Count() As Integer Implements ICollection.Count
        Get
            Count = empArray.Count
        End Get
    End Property
    Public ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
        Get
            Return Me
        End Get
    End Property
    Public ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
        Get
            Return False
        End Get
    End Property
    Public Function GetEnumerator() As IEnumerator Implements IEnumerable.GetEnumerator
        Return empArray.GetEnumerator()
    End Function
    Public Function Add(ByVal newEmployee As Employee) As Integer
        empArray.Add(newEmployee)
        Return empArray.Count
    End Function
End Class
```

Exemple (Suite) : Main()

```
Dim Emps As Employees = New Employees()
' Note that only the collection is serialized -- not the
' CollectionName or any other public property of the class.
Emps.CollectionName = "Employees"
Dim John100 As Employee = New Employee("John0", "100xxx")
Emps.Add(John100)
Dim John101 As Employee = New Employee("John1", "101xxx")
Emps.Add(John101)
Dim x As XmlSerializer = New XmlSerializer(GetType(Employees))
Dim writer As TextWriter = New StreamWriter("coll.xml")
x.Serialize(writer, Emps)
writer.Close()
```

XML :

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfEmployee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Employee>
    <EmpName>John0</EmpName>
    <EmpID>100xxx</EmpID>
  </Employee>
  <Employee>
    <EmpName>John1</EmpName>
    <EmpID>101xxx</EmpID>
  </Employee>
</ArrayOfEmployee>
```