Gestion des variables locales

Vous avez déjà vu au travers de cet ouvrage des utilisations de variables locales. Il s'agit de variables qui sont accessibles et utilisables dans le bloc où elles sont déclarées.

```
Dim X As Integer
For X = 1 to 10
  Dim Y As Integer
  Y = 0
  MessageBox.Show("X = " & X)
  MessageBox.Show("Y = " & Y)
  Next X
MessageBox.Show("X final = " & X)
MessageBox.Show("Y final = " & Y)
```

Dans ce bout de programme, la boucle devrait faire dix tours et afficher les valeurs de `X` entre 1 et 10, et dix fois `Y = 0`. Enfin une boîte de dialogue devrait afficher `X = 10` et `Y = 0`. Or, ce bout de programme provoque une erreur.

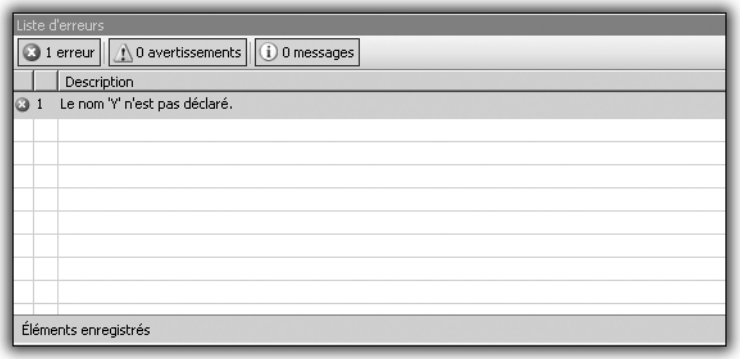

Figure 9.21 : *Accès à une variable hors de portée*

En effet, lors de l'affichage final de `Y`, la variable n'est plus accessible. Étant définie dans la boucle, `Y` n'est accessible que dans cette boucle. Sa portée, c'est-à-dire son accessibilité, est limitée à ladite boucle. On dit que `Y` est une variable locale de cette boucle.

Les variables locales sont gérées dans la pile. La pile est une zone mémoire dédiée à l'utilisation des variables locales. On peut la symboliser comme une vraie pile, composée de plusieurs petits blocs de mémoire que sont les variables :

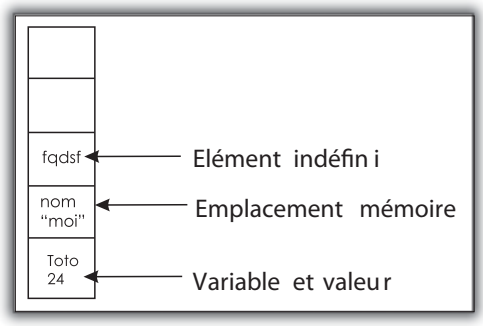

Figure 9.22 : *Représentation simplifiée de la pile mémoire*

Dans cette pile, ce qui va marquer l'accessibilité des variables est un petit marqueur :

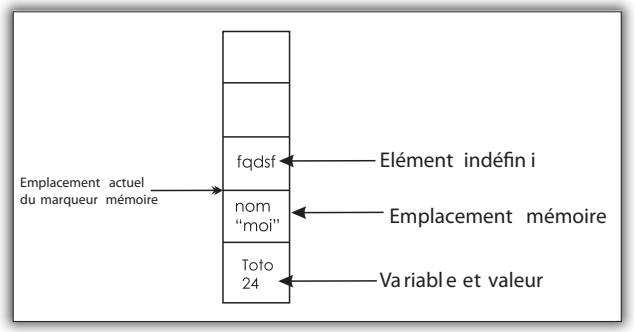

Figure 9.23 : *Pile mémoire avec le marqueur*

Quand vous déclarez une variable, un emplacement mémoire va être utilisé, et ce marqueur va être déplacé :

```
Dim X As Integer
```

Le programme pourra alors accéder aux parties de la mémoire qui sont au-dessus de ce marqueur. `X` vient d'être déclarée, elle est au-dessus du marqueur, on peut l'utiliser. Cependant, vous n'avez pas donné à `X` de valeur initiale. Entre sa déclaration et l'entrée dans la boucle, on ne sait pas ce que vaut `X`. Lorsque l'on déclare des variables, le marqueur se déplace vers le bas, donnant ainsi l'accès à de nouvelles variables.

Or, ce marqueur se déplace aussi vers le haut, bloquant ainsi l'accès à d'anciennes variables. Il se déplace essentiellement vers le haut lorsque l'on sort d'un bloc. En Visual Basic, on peut dire que la fin d'un bloc correspond à un endroit où il y a un `End`.

Examinons le programme précédent. On déclare `X` tout d'abord. Un accès lui est donné dans la pile et le marqueur descend donc.

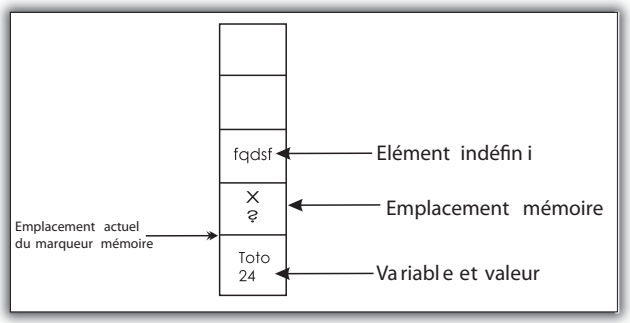

Figure 9.24 : *Pile mémoire, avec X accessible*

On entre dans un nouveau bloc, la boucle `For`. La position du marqueur est enregistrée pour que l'on puisse y revenir quand on sortira de ce bloc.

```
Dim Y As Integer
```

Ensuite, on déclare `Y`. Encore une fois, le marqueur descend et un emplacement mémoire est créé pour `Y` :

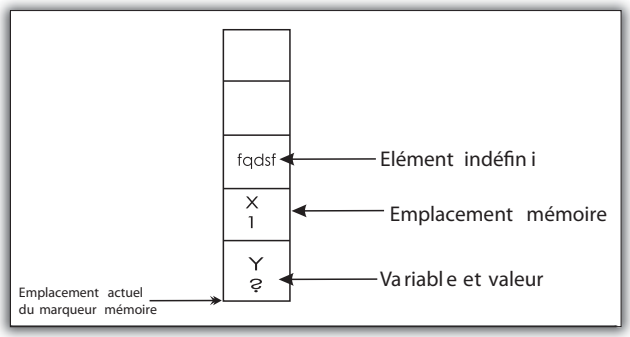

Figure 9.25 : *Pile mémoire, avec X et Y accessibles*

Les traitements sont faits avec cet environnement. L'environnement est l'ensemble des points accessibles à partir de là où l'on est. Ces points peuvent être des fonctions, des méthodes, des variables, etc.

Enfin, on quitte la boucle `For`. On quitte donc un bloc et le marqueur remonte à l'endroit enregistré lorsque l'on est entré dans cette boucle.

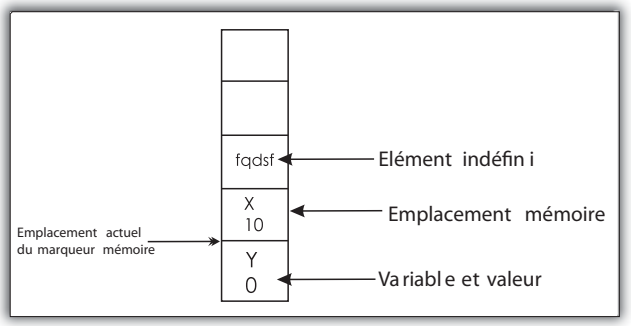

Figure 9.26 : *Pile mémoire, après retour dans l'ancien bloc*

L'emplacement mémoire correspondant à `Y` étant maintenant en dessous du marqueur, il n'est plus dans l'environnement courant et n'est donc plus accessible. Bien que `Y` ne soit pas accessible, son emplacement mémoire existe encore, avec sa valeur, tant que l'état de la pile n'a pas changé.

L'environnement change souvent : à l'entrée ou à la sortie d'une fonction, à l'entrée et à la sortie d'une structure de contrôle. La mémoire est en constants mouvements.

Désormais, vous devriez mieux comprendre les concepts de portée et d'accessibilité. C'est un sujet vaste et intéressant. Nous vous invitons à l'approfondir pour découvrir justement les subtilités que nous n'exposons pas ici.

Les constructeurs d'objets

Lorsque nous avons présenté les mécanismes de la programmation orientée objet, il a été question du mot-clé `New` dans le cadre de la création d'un objet. Ce mot-clé permet d'instancier un objet.

Qu'est-ce que l'instanciation ? C'est le mécanisme qui permet de créer un objet, une instance de classe. Son rôle réel est en fait de réserver un espace mémoire pour l'utilisation de l'objet et, le cas échéant, d'initialiser certains de ses attributs. Cependant, la gestion de la mémoire n'est pas la même pour les objets et pour les variables locales. En effet, un objet doit rester en mémoire jusqu'à ce que vous n'en ayez plus besoin. Il ne peut alors pas être stocké dans la pile. Les objets sont stockés dans le tas managé.

Le tas managé est un gros espace de mémoire réservé au programme. Celui–ci pourra y stocker tous les objets nécessaires à son exécution. Lorsqu'une classe est instanciée, le programme lui crée un espace mémoire dans le tas managé. Cela se fait grâce au constructeur.

```
Dim monChasseur As New AvionDeChasse()
```

Le constructeur est la partie qui suit `New`. C'est une méthode qui a pour nom celui de la classe et lui seul permet d'instancier cette classe. Il existe un constructeur par défaut. C'est pourquoi vous n'avez pas eu à l'indiquer lors de l'écriture de la classe `AvionDeChasse`. Dans ce cas, il réserve juste l'espace mémoire. Cependant, il est possible de redéfinir le constructeur, et ainsi de pouvoir faire un certain nombre d'opérations lors de l'instanciation. Vous pouvez modifier sa signature, en ajoutant des paramètres par exemple. Il se crée comme une méthode qui ne renvoie aucune valeur de retour et dont le nom est `New`. Appliquons cela à l'avion de chasse :

```
Public Class AvionDeChasse
' Nous ne gardons qu'une partie de la classe
' pour cette illustration
  Private vitesseMax As Integer
  Private nbreAilerons As Integer
Private reserveKerozene As Integer

  Public Sub afficheKero()
    MessageBox.Show("Niveau kéro : " & Me.reserveKerozene)
  End Sub

  Public Sub New(ByVal keroOriginal As Integer)
Me.reserveKerozene = keroOriginal
End Sub
End Class
```

On redéfinit le constructeur en lui ajoutant un paramètre qui permettra d'initialiser son niveau de kérosène à l'instanciation. Pour appeler ce constructeur, il suffira de tenir compte de la nouvelle signature, et donc d'ajouter ce paramètre :

```
Dim monChasseur As New AvionDeChasse(100)
monChasseur.afficheKero()
```

Un objet `AvionDeChasse` a été créé avec un niveau de kérosène valant 100. Cela est confirmé par la boîte de dialogue, qui affichera "*Niveau kéro : 100*".

Récupération de la mémoire : le ramasse-miettes

Comme nous l'avons expliqué, le tas managé est un espace de mémoire réservé à l'application, dans lequel le programme va pouvoir stocker les objets qu'il utilise. Cependant, cet espace est limité, et vous n'avez pas donné de limite au nombre possible d'objets à instancier.

Pourtant il y en a une : celle de la capacité du tas managé. Si celui-ci est trop plein et que vous vouliez instancier des objets, une exception `OutOfMemoryException` est levée. On a un espace mémoire, on peut réserver des emplacements dans cet espace, mais il faut pouvoir libérer ceux dont on ne se sert pas.

Cette tache est effectuée par un mécanisme appelé "*ramasse-miettes*" (*garbage collector*), qui s'occupe de libérer l'espace mémoire occupé par des objets dont vous ne vous servez plus. Le programme sait quels sont les objets que vous allez pouvoir utiliser ou pas. Si vous pouvez les utiliser, on dit qu'ils sont référencés. Dans le cas contraire, on dit qu'ils sont déréférencés.

Prenons ce morceau de programme relatif aux avions :

```
Public Sub essaiDereferencement()
  Dim monChasseur2 As New AvionDeChasse()
End Sub

Dim monChasseur1 As New AvionDeChasse()
essaiDereferencement()
```

On déclare une fonction qui instancie un chasseur, puis on instancie un autre chasseur et l'on fait appel à cette fonction.

À la fin du programme, en admettant qu'il continue après, on pourra toujours utiliser `monChasseur1`. `monChasseur1` est donc référencé.

Par contre `monChasseur2` est instancié dans la fonction `essaiDereferencement`. Or, en sortant de cette fonction, on ne peut plus y accéder. En effet, il a été déclaré dans une fonction, et bien qu'il existe en mémoire dans le tas managé, sa portée ne peut aller au-delà de la fonction `essaiDereferencement`. À la fin de ce bout de programme, vous ne pouvez plus atteindre `monChasseur2`, il est déréfencé.

Le ramasse-miettes vérifie les objets qu'il a en charge et détermine ceux qui sont référencés et ceux qui ne le sont plus. Étant donné que ceux qui sont déréférencés ne sont plus accessibles, garder leur espace mémoire est inutile. Celui-ci est alors libéré pour permettre la création de nouveaux objets.

Les destructeurs d'objets

Le ramasse-miettes est un mécanisme qui permet de gérer automatiquement la mémoire. Il permet d'éviter les fuites de mémoire et enlève au programmeur une grosse part de réflexion. En effet, dans les langages de programmation où il faut gérer la mémoire manuellement, il n'est pas rare que l'espace dédié à certains objets ne soit pas libéré, provoquant ainsi des fuites de mémoire. Le programmeur doit savoir où sont utilisés ses objets, où ils ne le sont plus, et à ce moment-là libérer la mémoire. Grâce au ramasse-miettes, cela peut être évité.

Mais, il arrive que l'on ait besoin de libérer manuellement la mémoire, ou encore que l'on veuille effectuer une opération particulière juste avant que celle-ci ne soit libérée. Si l'on reprend l'analogie avec l'avion, la libération de la mémoire serait la destruction de celui-ci, mais on aimerait pouvoir lancer le siège éjectable juste avant, et ainsi épargner la vie du pilote !

Imaginons que vous vouliez effectuer une opération, juste avant la destruction de l'objet. On entend par destruction de l'objet sa libération. Il faut savoir que tous les objets possèdent une méthode `Finalize`. En effet, tous les objets en Visual Basic .NET dérivent implicitement de la classe `Object` ; celle-ci définit une méthode `Finalize`, qui est `Overridable`. Cette méthode est appelée lorsque le ramasse-miettes libère un objet. En redéfinissant cette méthode (avec le mot-clé `Overrides`, en tant que membre protégé), vous pouvez effectuer les instructions que vous voulez lorsque l'objet est détruit :

```
Public Class AvionDeChasse
  Public Sub siegeEjectable()
  ' L'implémentation n'est pas nécessaire
  ' pour la compréhension
  End Sub

  Protected Overrides Sub Finalize()
    Me.siegeEjectable()
  End Sub
End Class
```

`Finalize` étant redéfinie, la méthode `siegeEjectable` sera appelée lorsque l'objet sera détruit. Attention cependant, il y a quelques règles à respecter. Il ne faut pas appeler explicitement `Finalize`, ce sera fait par le ramasse-miettes. Si l'on veut effectuer la libération de l'objet manuellement, il y a une manière valable d'opérer, décrite ci-après. De plus, il ne faut pas qu'une exception soit levée. Si c'est l'objet qui gère l'exception, elle ne pourra pas être récupérée, car celui-ci sera détruit ; cela peut provoquer l'arrêt du programme.

De plus, le ramasse-miettes libère la mémoire selon son bon vouloir. En redéfinissant la méthode, vous ne pouvez pas forcer son utilisation, qui se fera au moment où le ramasse-miettes l'aura décidé, par exemple quand un besoin de mémoire se fera sentir, ou alors si le nombre d'objets déréférencés est trop important.

Si vous voulez forcer la libération, il faut passer par l'implémentation d'un modèle de conception, le `DisposeFinalize`. Nous présenterons les modèles de conception plus loin dans ce chapitre, car nous verrons juste après comment gérer la sauvegarde et le chargement de vos objets de manière efficace. Nous vous invitons donc à laisser faire le ramasse-miettes dans un premier temps, car quoique l'on puisse en dire, celui-ci est efficace dans la gestion de la mémoire.

9.3. Enregistrer automatiquement vos objets : la sérialisation

Gérer les données peut devenir très fastidieux quand votre programme commence à devenir un peu important. Imaginez si vous deviez opter pour un enregistrement et un chargement à partir d'un fichier textuel... Pour chacun des attributs que vous avez à gérer, il vous faut créer une méthode de sauvegarde ou de chargement ou alors adapter celle qui est existe déjà. Cela limite un peu la croissance de vos structures ou alors au prix de quels efforts...

On a également vu qu'il est possible de les enregistrer dans des bases de données. Là encore, le travail peut se révéler important : installer le serveur, le configurer, s'assurer de son bon fonctionnement... Ceci n'est pas très efficace pour des applications indépendantes.

En revanche, nous avons également vu les notions d'objet et de classe, pour lesquelles le framework .Net définit et implémente un mécanisme

de sauvegarde et de chargement automatique. Vous pourrez en user et en abuser, à tel point que cela deviendra vite le moyen privilégié pour la gestion de la sauvegarde de vos objets.

Dans un premier temps, nous allons voir plus en détail ce qu'est la sérialisation et ses différents types. Nous préparerons également une petite application de test. Puis nous étudierons de manière plus détaillée deux types de sérialisations : binaires et XML.

Qu'est-ce que la sérialisation ?

Fort de vos connaissances concernant les enregistrements sur les bases de données et les fichiers, vous savez gérer des données, pas de souci...

Reportez-vous à ce sujet au chapitre ***Enregistrer des données****.*

Mais on admettra volontiers que, si on pouvait s'affranchir de tout ce travail pour pouvoir utiliser son temps à des parties de programmes plus intéressantes et plus constructives, ce serait quand même beaucoup plus sympa. La sérialisation nous permet cela. Pourquoi ? Car en entrée elle prend un objet, et elle vous ressort un fichier. La désérialisation, son opération opposée, prend ce fichier et vous renvoie votre objet. Qu'avez-vous fait pour cela ? Rien, si ce n'est avoir écrit les méthodes de sérialisation et de désérialisation. Et elles ne bougeront plus même si votre classe bouge avec des milliers d'attributs et de propriétés en plus.

Il existe plusieurs types de sérialisations :

- **Binaire**. Enregistre vos objets dans un format machine non lisible par l'homme. C'est la première que nous allons voir.
- **XML**. Enregistre vos objets au format XML, standard, et lisible par l'homme. Nous l'étudierons également.
- **SOAP**. On peut grossièrement dire qu'il s'agit d'une version améliorée de la version XML. C'est en fait un peu plus que cela, mais nous ne l'étudierons pas dans cet ouvrage.

Afin de vous présenter cette technique, nous allons créer une petite application qui se fondera sur une classe que nous allons sérialiser et désérialiser, la classe `Livre` :

```
Public Class Livre
    Private mTitre As String
    Private mAuteur As String
    Private mEditeur As String
    Private mAnneeDeParution As Integer
    Private mNbreDePages As Integer
    Private mIsbn As Long

    Property Titre() As String
        Get
            Return mTitre
        End Get
        Set(ByVal Value As String)
            mTitre = Value
        End Set
    End Property

    Property Auteur() As String
        Get
            Return mAuteur
        End Get
        Set(ByVal Value As String)
            mAuteur = Value
        End Set
    End Property

    Property Editeur() As String
        Get
            Return mEditeur
        End Get
        Set(ByVal Value As String)
            mEditeur = Value
        End Set
    End Property

    Property AnneeDeParution() As Integer
        Get
            Return mAnneeDeParution
        End Get
        Set(ByVal Value As Integer)
            mAnneeDeParution = Value
        End Set
    End Property

    Property NombreDePage() As Integer
        Get
            Return mNbreDePages
        End Get
        Set(ByVal Value As Integer)
            mNbreDePages = Value
        End Set
```

```
    End Property

    Property ISBN() As Long
        Get
            Return mIsbn
        End Get
        Set(ByVal Value As Long)
            mIsbn = Value
        End Set
    End Property
End Class
```

Cette classe définit six attributs membres (titre, auteur, année de parution, ISBN, nombre de pages, éditeur) ainsi que les propriétés (en lecture / écriture) associées.

Entrons maintenant dans le vif du sujet. Nous allons créer une légère application pour manier ces mécanismes.

1 Tout d'abord, on crée un nouveau projet.

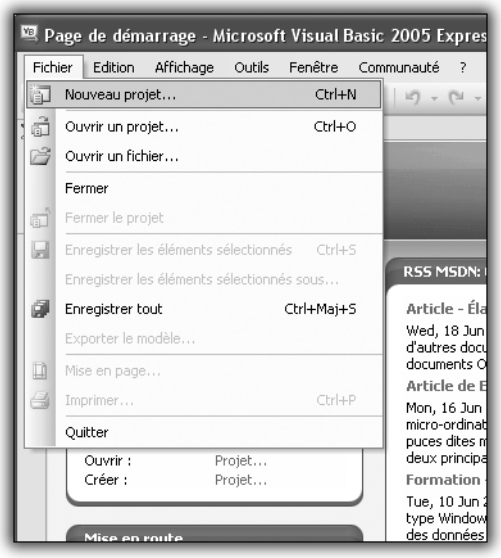

Figure 9.27 : *Création du nouveau projet*

2 C'est un projet `WindowsForms`, et `ProjetSerialisation` semble être un nom adéquat (voir Figure 9.28).

3 Il faut ajouter maintenant au projet la classe `Livre`, qui sera le point central du projet (voir Figure 9.29, 9.30).

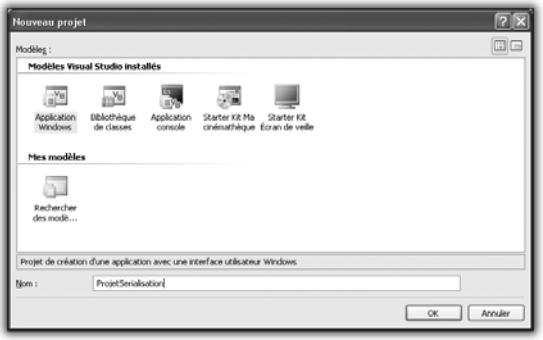

Figure 9.28 : *Choix du nom et du type de projet*

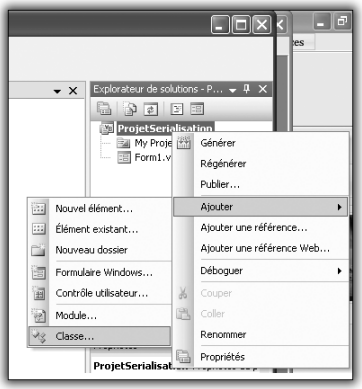

Figure 9.29 : *Ajout d'une nouvelle classe*

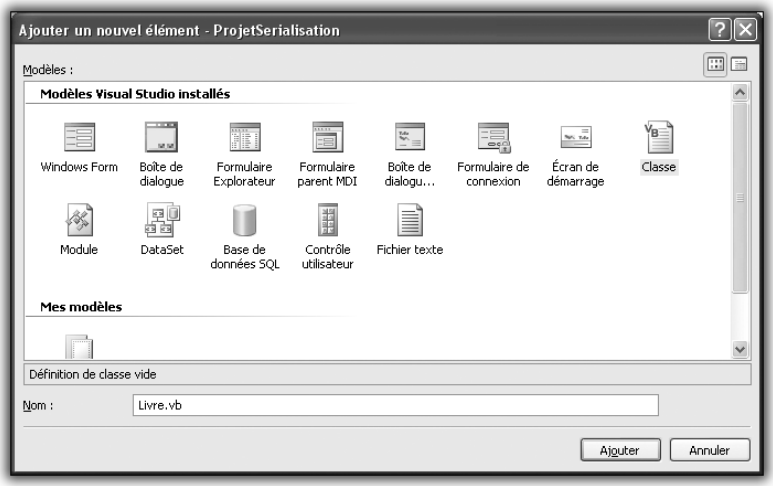

Figure 9.30 : *On la nomme Livre*

4 Créez maintenant une interface graphique, à base de `Label`, de `TextBox` et de `Bouton`, afin qu'elle ressemble à quelque chose dans ce style :

Figure 9.31 :
Interface du projet

Reportez-vous à ce sujet au chapitre ***Dialoguer avec un ordinateur.***

Maintenant que la base du programme est faite, nous allons pouvoir attaquer en profondeur et voir en détail comment manier sérialisation et désérialisation.

Les différentes méthodes de sérialisation

Nous allons étudier dans cette partie seulement deux des sérialisations existantes :

- la sérialisation binaire ;
- la sérialisation XML.

La sérialisation binaire

La sérialisation binaire est un type de sérialisation qui transforme votre objet en fichier binaire, c'est-à-dire directement en code machine. Globalement, si vous essayez de lire, vous n'y arriverez pas. En tout cas pas en entier.

Figure 9.32 :
Fichier binaire ouvert avec un éditeur de texte

Comme vous pouvez le constater, on ne comprend pas grand-chose, mais il est possible de discerner certaines chaînes de caractères. En effet, que ce soit dans un programme ou dans un fichier, le code machine est le même... Mais ce n'est pas vraiment l'objet de ce chapitre.

Tout d'abord, il faut savoir que, pour sérialiser et désérialiser, il faut un sérialiseur, qui fait aussi la désérialisation. Logique ! Le sérialiseur binaire est fourni par le framework .Net, dans le namespace `Binary`. Il vous faudra donc inclure ce namespace dans votre code.

```
Imports System.Runtime.Serialization.Formatters.Binary
```

Enregistrement : sérialisation

Une méthode de sérialisation écrit un fichier à partir d'un objet. Elle prendra donc cet objet en paramètre, ici un `Livre`, et ne renverra rien. Mais nous allons quand même lui faire renvoyer un booléen pour nous dire si l'opération s'est bien passée. Elle aura donc une signature comme ceci :

```
    Private Function BinarySerialize(ByVal monLivre As
✂ Livre) As Boolean
    End Function
```

Comme nous allons écrire dans un fichier, il faut créer celui-ci, par exemple `Livre.bin`. Il nous faut donc également ajouter le namespace `IO`.

Reportez-vous à ce sujet au chapitre ***Enregistrer des données, les fichiers.***

Le sérialiseur binaire est un `BinaryFormatter` qui possède un constructeur par défaut. Nous pouvons donc enrichir notre méthode de sérialisation avec ces nouveaux éléments :

```
    Private Function BinarySerialize(ByVal monLivre As
✂ Livre) As Boolean
        If monLivre Is Nothing Then Return False
        Dim stream As FileStream = New FileStream("Livre
      ✂ .bin", FileMode.Create)
        Return True
    End Function
```

Validité de l'objet

Attention à ne pas oublier la vérification de la validité de l'objet (`If monLivre Is Nothing Then Return False`), sans quoi vous risquez de provoquer une exception de type `NullReferenceException`.

La dernière étape de cette méthode est la sérialisation effective. Elle est réalisée par le sérialiseur et prendra comme paramètre le fichier dans lequel écrire, ainsi que l'objet à sérialiser :

```
serializer.Serialize(stream, monLivre)
```

Il ne faudra pas oublier de libérer les ressources utilisées pour le fichier, par l'appel à la méthode `Close`. Au final, la méthode de sérialisation ressemblera à ceci :

```
 Private Function BinarySerialize(ByVal monLivre As
✂ Livre) As Boolean
     If monLivre Is Nothing Then Return False
     Dim stream As FileStream = _
               New FileStream("Livre.bin", FileMode
             ✂ .Create)
     Dim serializer As New BinaryFormatter
     serializer.Serialize(stream, monLivre)
     stream.Close()
     Return True
 End Function
```

Liez maintenant cette méthode au bouton de sauvegarde de votre interface. Et en avant pour le test !

Démarrez le projet et remplissez les différents champs avec les valeurs de votre choix.

Figure 9.33 :
Description d'un livre

Cliquez sur sauvegarder, et là, surprise !

Figure 9.34 :
Erreur de sérialisation

Erreur de sérialisation ! Votre programme ne peut plus continuer à s'exécuter, car l'objet que l'on veut sérialiser est l'instance d'une classe qui n'est pas marquée comme sérialisable.

Qu'à cela ne tienne ! Marquons-la sérialisable en y ajoutant un attribut.

```
<Serializable()> _
Public Class Livre
    Private mTitre As String
    Private mAuteur As String
    Private mEditeur As String
    Private mAnneeDeParution As Integer
    Private mNbreDePages As Integer
    Private mIsbn As Long

    ReadOnly Property Titre() As String
        Get
            Return mTitre
        End Get
        'Set(ByVal Value As String)
        '    mTitre = Value
        'End Set
    End Property

    ReadOnly Property Auteur() As String
        Get
            Return mAuteur
        End Get
        'Set(ByVal Value As String)
        '    mAuteur = Value
        'End Set
    End Property
```

```
    ReadOnly Property Editeur() As String
        Get
            Return mEditeur
        End Get
        'Set(ByVal Value As String)
        '    mEditeur = Value
        'End Set
    End Property

    Property AnneeDeParution() As Integer
        Get
            Return mAnneeDeParution
        End Get
        Set(ByVal Value As Integer)
            mAnneeDeParution = Value
        End Set
    End Property

    Property NombreDePage() As Integer
        Get
            Return mNbreDePages
        End Get
        Set(ByVal Value As Integer)
            mNbreDePages = Value
        End Set
    End Property

    Property ISBN() As Long
        Get
            Return mIsbn
        End Get
        Set(ByVal Value As Long)
            mIsbn = Value
        End Set
    End Property
End Class
```

Vous pouvez maintenant réessayer. Et vous obtiendrez en sortie un fichier *Livre.bin* qui contiendra ceci :

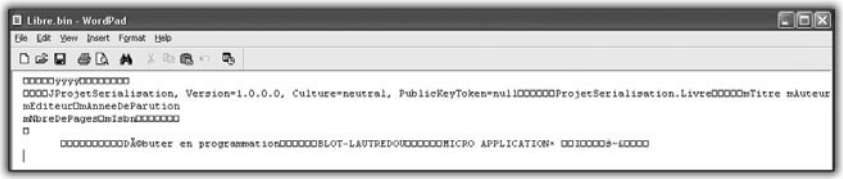

Figure 9.35 : *Fichier de sortie binaire*

Nous remarquons que certaines chaînes de caractères sont lisibles. Certaines ont été générées pour les besoins du framework .Net (nom du projet, version...) et d'autres sont les valeurs des attributs de votre objet. On peut y voir le titre, l'auteur, l'éditeur, mais remarquez qu'on ne voit pas les chiffres. Les types numériques en code machine ne sont pas lisibles, contrairement aux chaînes de caractères. Ils correspondent à certains des petits carrés que vous voyez.

Chargement : désérialisation

L'opération contraire à la sérialisation, qui permet la création de votre objet préalablement enregistré *via* la sérialisation, est la désérialisation. En toute logique, vous pensez qu'elle se fait avec un désérialiseur. Et vous avez raison de penser cela, sauf qu'elle se fait également en utilisant le sérialiseur.

En revanche, une méthode de désérialisation a une signature totalement différente. En effet, nous avons vu que la désérialisation consistait à utiliser un fichier pour créer un objet, ici de type `Livre`. Et, comme nous savons quel fichier aller chercher, cette méthode ne nécessite pas de paramètres. Voici donc la base de notre méthode de désérialisation :

```
Imports System.Runtime.Serialization.Formatters.Binary

    Private Function BinaryDeserialize() As Livre
        Dim monLivre As Livre = Nothing

        Dim deserializer As New BinaryFormatter

        Return result
    End Function
```

De la même manière que pour enregistrer, il va falloir ici gérer un fichier, sauf qu'on va l'ouvrir et non le créer.

Fichier manquant

Attention à bien vérifier l'existence du fichier, sans quoi une exception de type `IOException` peut arrêter votre programme !

```
    Private Function BinaryDeserialize() As Livre
        Dim fichier As String = "Livre.bin"
        Dim monLivre As Livre = Nothing

        If Not File.Exists(fichier) Then Return Nothing
        Dim deserializer As New BinaryFormatter
```

```
        Return result
    End Function
```

Il faut maintenant désérialiser et c'est encore plus facile que la sérialisation. Le seul paramètre nécessaire est le fichier à lire. La méthode qui fait cela est `Deserialize`. Là encore, il ne faudra pas oublier de libérer les ressources du fichier, sinon vous ne pourrez plus l'ouvrir avec un autre programme tant que notre programme de test sera actif. La méthode finale ressemble à ceci :

```
Private Function BinaryDeserialize() As Livre
    Dim fichier As String = "Livre.bin"
    Dim monLivre As Livre = Nothing

    If Not File.Exists(fichier) Then Return Nothing

    Dim deserializer As BinaryFormatter
    Dim stream As FileStream = New FileStream(fichier,
   ✂ FileMode.Open)
    monLivre = deserializer.Deserialize(stream)
    stream.Close()

    Return monLivre
End Function
```

Maintenant, il vous suffit de charger sur votre interface graphique les informations récupérées à partir de l'instance de `Livre` chargée pour vérifier le chargement :

Figure 9.36 :
Objet chargé

On retrouve bien notre objet préalablement enregistré. Quelque part, c'est rassurant, n'est-ce pas ?

Comme vous le voyez, il est très facile de charger et d'enregistrer des instances de classe grâce à la sérialisation. N'hésitez surtout pas à

utiliser ces méthodes. Elles vous feront gagner un temps précieux sur la gestion de vos données, temps que vous pourrez utiliser pour des choses de plus haut niveau.

Voyons maintenant la sérialisation XML, qui, à l'utilisation, sera quasiment identique à notre sérialisation binaire, mais qui aura un fichier de sortie totalement différent.

La sérialisation XML

Je ne reviendrai pas ici sur ce qu'est le XML mais, pour ceux qui auraient pris le livre en cours, il faut savoir que c'est un langage à balises permettant de décrire des éléments suivant une hiérarchie donnée.

Reportez-vous à ce sujet au chapitre ***Enregistrer des données, Sauvegarde et fichier : le XML.***

Mais, comme rien ne vaut un exemple, voici un extrait de fichier XML :

```
<?xml version="1.0" encoding="UTF-8"?>
  <!-- '''Commentaire''' -->
  <élément-document xmlns="http://exemple.org/" xml:lang=";fr">
    <élément>Texte</élément>
    <élément>élément répété</élément>
    <élément>
      <élément>Hiérarchie récursive</élément>
    </élément>
    <élément>Texte avec<élément>élément</élément>mêlé</élément>
    <élément/><!-- élément vide -->
    <élément attribut="valeur"></élément>
  </élément-document>
```

Figure 9.37 : *Exemple de fichier XML*

Vous remarquez que, contrairement au format binaire, le format XML est totalement lisible. *A fortiori*, il permet de savoir ce que l'on fait. Nous verrons que cela pourra devenir un moyen de modifier nos données à la volée.

De la même manière que pour la sérialisation et la désérialisation binaire, il faut un sérialiseur. Dans ce cas-là, il est dans le namespace `Xml.Serialization`. Il faut donc ajouter ce namespace au projet.

```
Imports System.Xml.Serialization
```

Enregistrement : sérialisation

Le principe de sérialisation, qu'il soit binaire, XML ou même SOAP (que nous ne traiterons pas ici) reste toujours le même : on crée un fichier à partir d'un objet. C'est pourquoi la signature de notre méthode de sérialisation ne va pas changer. Elle prendra toujours en paramètre un objet `Livre` et renverra toujours un booléen qui nous dira si l'opération s'est bien passée.

```
 Private Function XMLSerialize(ByVal monLivre As Livre)
✂ As Boolean
 End Function
```

Jusque-là tout est identique, si ce n'est le nom de la méthode. (Ceci dit, on aurait pu garder le même nom, ce qui aurait été très maladroit, car le nom ne correspondrait plus à l'action.)

Une fois encore, nous allons écrire dans un fichier, exactement de la même manière que pour la sérialisation binaire, mais on va lui donner un autre nom. `.xml` étant le standard pour définir un fichier XML, nous l'appelons *Livre.xml*.

Besoin de ressources

Nous n'oublierons pas de libérer les ressources du fichier ni de vérifier la validité de l'objet que nous voulons sérialiser.

```
 Private Function XMLSerialize(ByVal monLivre As Livre)
✂ As Boolean
     If monLivre Is Nothing Then Return False
     Dim stream As FileStream = _
       New FileStream("Livre.xml", FileMode.Create)
     stream.Close()
     Return True
 End Function
```

Il nous faut maintenant instancier un sérialiseur. Cependant, dans le cas du sérialiseur XML, il faut lui passer comme paramètre le type de l'objet que nous voulons sérialiser, tout simplement en utilisant `GetType`. L'utilisation se fait ensuite de manière totalement identique à un sérialiseur binaire, avec comme paramètre le lien vers le fichier et l'objet à sérialiser :

```
 Private Function XMLSerialize(ByVal monLivre As Livre)
✂ As Boolean
     If monLivre Is Nothing Then Return False
     Dim stream As FileStream =
```

```
                New FileStream("Livre.xml", FileMode
               ✂ .Create)
        Dim serializer As XmlSerializer = _
          New XmlSerializer(GetType(Livre))
        serializer.Serialize(stream, monLivre)
        stream.Close()
        Return True
    End Function
```

Nous obtiendrons donc un fichier de sortie, au format XML, contenant les informations de notre objet, qui, lui, est totalement lisible.

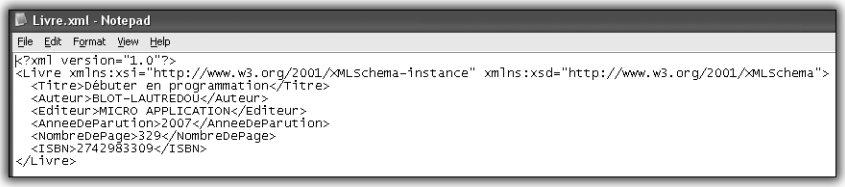

Figure 9.38 : *Fichier XML de notre objet sérialisé*

Chargement : désérialisation

Cette fois encore, la désérialisation sera avec un objet sérialiseur. Cela a au moins cet avantage qu'on l'instancie de la même manière.

De plus, nous gérerons également un fichier à ouvrir, que l'on prendra bien soin de refermer pour éviter de laisser un lien qui empêchera son utilisation par un autre programme. On n'oubliera pas non plus de vérifier son existence pour éviter des erreurs `IOException`. Voici la méthode de désérialisation XML :

```
Private Function XMLDeserialize() As Livre
    Dim fichier As String = "Livre.xml"
    Dim monLivre As Livre = Nothing

    If Not File.Exists(fichier) Then Return Nothing

    Dim deserializer As XmlSerializer = _
    New XmlSerializer(GetType(Livre))
    Dim stream As FileStream = New FileStream(fichier,
   ✂ FileMode.Open)
    Dim result As Object = Nothing
    monLivre = deserializer.Deserialize(stream)
    stream.Close()

    Return monLivre
End Function
```

Comme la méthode de désérialisation est quasiment identique en binaire ou en XML, je ne vais pas détailler une nouvelle fois les différents points de la méthode. Cependant, la sérialisation XML a certaines particularités.

Si vous chargez directement le fichier préalablement enregistré, vous obtiendrez le même objet que celui d'origine. Jusque-là, normal, nous pouvons vérifier dans votre interface de chargement :

Figure 9.39 :
Objet chargé

Maintenant, étant donné que le fichier XML est lisible et modifiable, essayez de changer certaines de ses valeurs :

Figure 9.40 :
Modification du fichier XML à la main

Et, là, si vous tentez de recharger le même fichier, voici ce que vous obtiendrez dans votre interface :

Figure 9.41 :
Interface obtenue à partir du fichier XML modifié

Comme vous le voyez, les éléments ont changé, prenant maintenant les valeurs contenues dans le fichier XML. Quelque part, c'est rassurant, car cela nous prouve que la désérialisation fonctionne. D'un autre côté, cela ouvre énormément de possibilités pour piloter des objets (et donc une application) à partir de fichiers. Par exemple, créons un objet `MailSender`, dont le but est d'envoyer des courriers électroniques. Ajoutons-lui des attributs "*Nombre de message*", "*destinataire*", "*corps*"... En le pilotant par sérialisation et désérialisation, il n'y a même plus besoin de modifier notre programme pour faire changer son comportement. On le définit dans le fichier XML de sérialisation et il sera à même de trouver les bonnes valeurs automatiquement. Je vous laisse le soin de créer un tel projet, cela pourrait être un très bon exercice de mise en pratique, pas trop difficile, concret et intéressant.

Modifions quelque peu notre code. Pour des raisons de sécurité, nous souhaiterions que certaines des propriétés de notre classe `Livre` soient en lecture seule (par exemple le titre, l'auteur et l'éditeur... Eh oui, on ne peut pas nous changer) :

```
ReadOnly Property Titre() As String
    Get
        Return mTitre
    End Get
End Property

ReadOnly Property Auteur() As String
    Get
        Return mAuteur
    End Get
End Property

ReadOnly Property Editeur() As String
    Get
        Return mEditeur
    End Get
End Property
```

Maintenant, retentez un chargement et vous obtiendrez ceci : (voir Figure 9.42)

Les champs dont les propriétés étaient en lecture seule n'apparaissent plus... En effet, c'est une particularité de la sérialisation XML. Elle se fonde sur les propriétés et utilise les `Get` et les `Set` pour sérialiser ou désérialiser le cas échéant. Alors que la sérialisation binaire sérialisera et désérialisera les attributs directement, quelle que soit leur accessibilité.

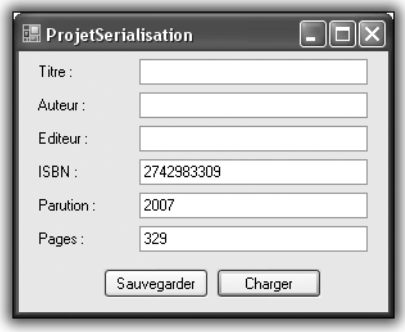

Figure 9.42 :
Résultat avec des propriétés en lecture seule

Quoi qu'il en soit, qu'elle soit binaire ou XML, n'hésitez pas à user et à abuser de la sérialisation. Choisissez celle qui vous convient selon le besoin de votre application. Si vous devez modifier des données à la volée, préférez la sérialisation XML car elle est lisible par l'homme. En revanche, étant donné que c'est du texte, elle sera moins performante et prendra plus de place en mémoire. D'un autre côté, la sérialisation binaire est plus performante, et elle ne nécessite pas de définir des propriétés. Elle est plus adaptée pour une utilisation directe qui ne demande pas de modification entre la sauvegarde et le chargement. Vous verrez, une fois acquise, la sérialisation est un concept dont on ne peut plus du tout se passer tellement elle facilite la gestion de la sauvegarde et du chargement des données.

9.4. Les modèles de conception

Un modèle de conception permet de résoudre un problème récurrent en tirant parti des mécanismes de la programmation orientée objet. Certains peuvent être simples, comme le `Singleton` que nous détaillerons et implémenterons complètement, mais ils sont le plus souvent assez subtils. Par contre, il garantisse une solution au problème, et ce de manière toujours propre. C'est pourquoi, il est fortement recommandé de les utiliser lorsque vous rencontrez un problème pour lequel il existe un modèle de conception.

Dans cette partie, vous verrez en détail le modèle de conception `Singleton`, qui permet de garantir l'unicité d'une instance de classe dans tout le programme et vous l'implémenterez. Puis, vous étudierez quelques-uns des modèles de conception les plus courants, ce qui vous donnera un point de départ si vous voulez approfondir le sujet.