



Concepteur Développeur en Informatique

Développer des composants d'interface

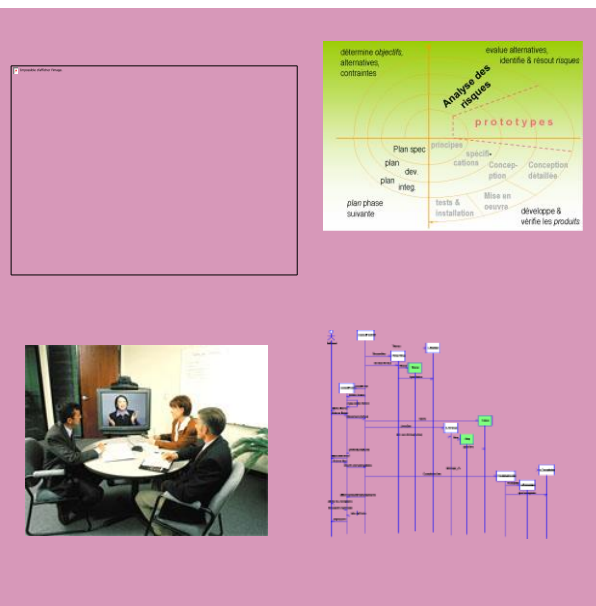
POO – Persistance

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S02

SOMMAIRE

1.	Introduction.....	3
2.	Manipuler des fichiers	4
2.1.	Les acc�s aux fichiers	4
2.2.	Les flux.....	5
2.3.	Vue d'ensemble des op�rations de traitement d'un fichier	6
2.3.1.	Ouverture du fichier avant la premi�re utilisation ;.....	6
3.	Manipuler un fichier Texte	7
3.1.	Ouverture du fichier	7
3.1.1.	Modes d'ouverture de fichier FileMode.....	7
3.1.2.	Op�rations autoris�es : FileAccess	8
3.1.1.	Autorisation de partage : FileShare	8
3.1.2.	Exemples d'ouverture d'un fichier.....	8
3.1.	Op�rations sur les donn�es	9
4.	Assurer la persistance des objets	13
4.1.	S�rialisation illustr�e	14
4.1.1.	S�rialisation binaire	14
4.1.2.	S�rialisation XML	15
4.1.3.	Limite de la s�rialisation XML.....	16
5.	Gestion des ressources	17
5.1.	Instruction Using	17
5.2.	Instruction Try / Catch	17

1. Introduction

Nous allons poursuivre dans ce document l'apprentissage des principaux concepts de la programmation orientée objet en nous intéressant à la persistance.

Assurer la persistance d'un objet est la capacité à conserver les valeurs prises par ce dernier au-delà du contexte où il a été créé.

L'objet est le plus souvent stocké dans une mémoire non volatile qui permettra de conserver son état au sein d'un fichier ou d'une base de données.

Nous allons voir ici comment stocker nos objets dans des fichiers et devrons donc découvrir comment manipuler des fichiers.

La persistance des objets s'appuie sur un mécanisme de conversion de l'état d'un objet en mémoire en un objet persistant ou transportable : la **sérialisation**.

Le complément de la sérialisation est la dé-sérialisation, qui convertit un flux de données en un objet.

Ces deux processus permettent donc de stocker et de transférer facilement les données.

Un flux (stream) issu de la sérialisation peut donc être ensuite transporté sur le réseau ou stocké dans un fichier. Les mécanismes qui permettent de sérialiser nos objets sont donc mis en œuvre pour :

- les communiquer à d'autres processus en dehors du contexte de l'application,
- assurer leur persistance et leur stockage durable dans des espaces mémoire non volatiles.

Nous allons donc traiter dans ce document des mécanismes de lecture et écriture dans des fichiers et des techniques de sérialisation.

2. Manipuler des fichiers

Il vous faut tout d'abord différencier la notion de flux et de fichier.

Un flux (stream) est un tuyau connecté entre deux entrepôts de données.
Ces entrepôts peuvent être situés en zone mémoire, sur le réseau ou sur disque.

Un fichier est constitué d'un ensemble de données structurées identifié par un nom et généralement stocké sur un disque sous formes d'octets (byte) au sein d'enregistrements.

Nous n'entrerons pas dans les détails de stockage physique des fichiers en fonction de la manière dont est gérée l'espace adressable du système de fichiers du système d'exploitation (FAT16, NTFS, F2FS, LTFS, ...).

L'utilisateur, comme le programmeur, n'aura jamais à se soucier de l'indexation des fichiers et de l'organisation de leur stockage physique.

Ce qu'ils voient est une vue logique, présentée sous la forme d'une arborescence de répertoires, leur permettant de s'affranchir de toute la complexité sous-jacente à cette organisation.

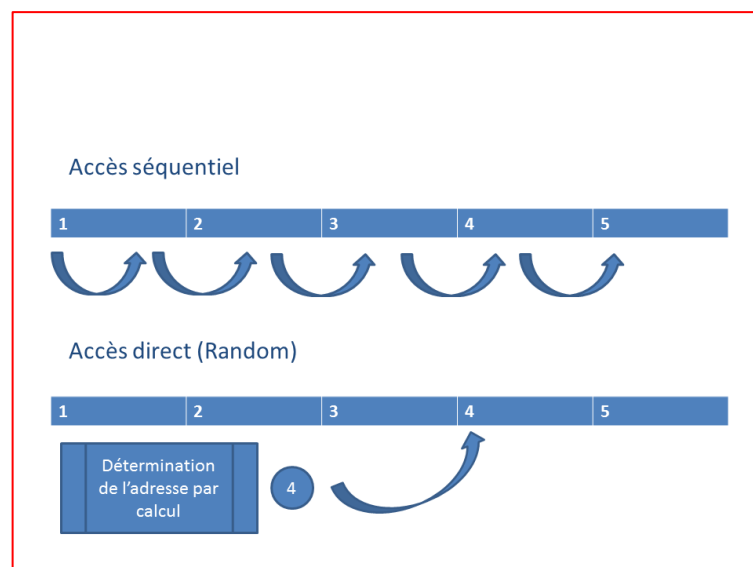
Du point de vue du programmeur, un fichier sera identifié par son nom et son chemin d'accès (path). Le développeur devra par contre tenir compte de la méthode d'accès au fichier.

2.1. Les accès aux fichiers

Nous distinguerons deux types d'accès, l'accès séquentiel et l'accès aléatoire (ou direct).

Il existe des méthodes d'accès dérivées, comme le séquentiel indexé, mais qui sont aujourd'hui, à l'époque de la généralisation des bases de données relationnelles, tombées en désuétude. *Pour la petite histoire, les bases de données s'appuient en interne sur des mécanismes de gestion séquentielle indexée, mais point n'est là notre propos.*

Pour comprendre très simplement la distinction entre accès séquentiel et accès direct ou aléatoire (Random Access in English qui vous rappelle peut être un acronyme célèbre RAM : Random Access Memory) ce petit schéma :



Dans le cas d'un acc s s quentiel, vous devez lire s quentiellement les enregistrements stock s dans le document depuis le premier jusqu'au dernier ou tout du moins jusqu'  l'enregistrement recherch .

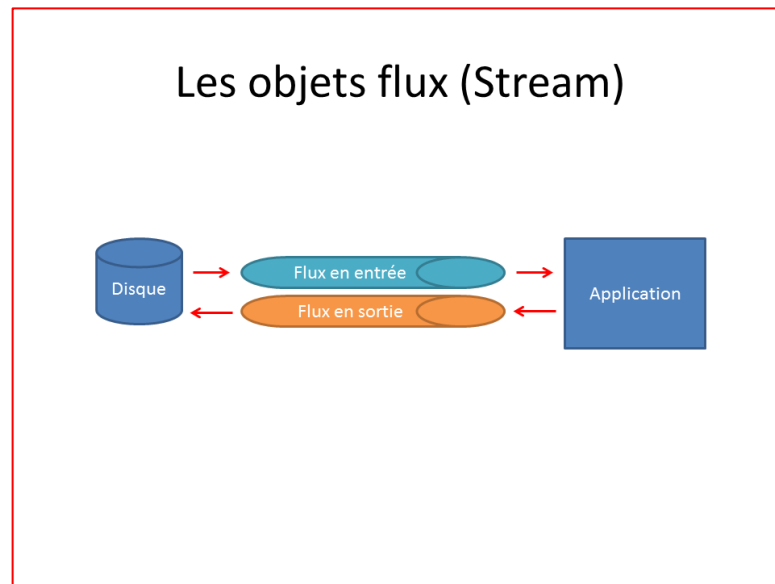
Dans le cas d'un acc s direct ou al atoire, l'adresse de l'enregistrement est d termin e par calcul et vous pouvez vous positionner directement sur l'offset qui repr sente la position du premier octet de l'enregistrement.

2.2. Les flux

Dans l'architecture .Net, un fichier sera toujours manipuler via un flux.

Un flux est unidirectionnel et permet d'extraire (lecture) ou de d poser des donn es ( criture).

Le sch ma ci-dessous pr sente les m canismes d' changes entre le disque et l'application via deux flux, un en lecture l'autre en  criture. Il s'agit donc ici d'une application qui serait amen    lire et modifier des donn es qui pourraient  tre stock s au sein d'un ou deux fichiers.



Dans l'architecture .Net, les flux d' changes sont toujours « bufferis s » (plac s dans une zone tampon). Les octets sont  crits ou lus par blocs d'octets. La taille des blocs d' changes peut  tre d finie par le programmeur via la m thode **SetLength** de l'objet Stream.

Les flux dans .Net sont d finis sous forme de types d riv s de la classe de base abstraite **Stream** qui sont fonction de la nature de l'entrep t de donn es.

Quelques exemples et non une liste exhaustive :

- Le type **FileStream** qui permet de manipuler des flux coupl s avec des fichiers sur disque.
- Le type **MemoryStream** qui permet de manipuler des flux en m moire
- Le type **NetworkStream** qui permet d' changer des donn es avec un socket Windows via le r seau.

La classe **FileStream** permet d'écrire et de lire des octets stockés au sein de fichiers. Elle dispose d'un flux de base qui permet de manipuler des flux d'octets bruts (des tableaux de Bytes).

2.3. Vue d'ensemble des opérations de traitement d'un fichier

Le traitement d'un fichier comporte trois phases :

1. L'ouverture du fichier
2. Les opérations de lecture et d'écriture
3. La fermeture du fichier

2.3.1. Ouverture du fichier avant la première utilisation ;

Lors de l'ouverture, on spécifiera le **mode d'utilisation** du fichier :

- En **lecture**, les informations pourront être extraites sans être modifiées.
- En **écriture**, les informations seront insérées dans le fichier. Si des informations existent, elles seront remplacées.
- En **ajout**, on ne peut ni lire, ni modifier les informations existantes. Il est possible par contre d'ajouter de nouveaux enregistrements.

Un fichier peut être ouvert en lecture et écriture.

2. Traitement du fichier

Des opérations d'écriture et de lecture

3. Fermeture du fichier après la dernière utilisation

Force l'écriture des données présentes dans le tampon si le fichier est en écriture. Déverrouille le fichier et le rend disponible pour d'autres processus.

3. Manipuler un fichier Texte

Nous allons nous int resser plus particuli rement   la lecture de fichiers o  sont stock es des donn es textuelles.

Pour lire et  crire des donn es textuelles, nous associerons   notre objet `FileStream` des objets sp cialis s propos s par le Framework .Net plus efficient que les m thodes du flux de base. Il s'agit des types `StreamReader` et `StreamWriter`.

D couvrons par l'exemple ces m canismes et la documentation associ e.

3.1. Ouverture du fichier

La classe `FileStream` permet d'ouvrir ou de cr er un fichier en sp cifiant

- le mode d'ouverture au moyen de l' num ration `FileMode`
- le mode d'acc s (lecture uniquement,  criture uniquement ou lecture/ criture) au moyen de l' num ration `FileAccess`
- le mode de partage au moyen de l' num ration `FileShare`

3.1.1. Modes d'ouverture de fichier `FileMode`

Les valeurs de l' num ration `FileMode` v rifient si un fichier doit  tre remplac , cr e ou ouvert ou une combinaison de ces actions.

Utilisez **Open** pour ouvrir un fichier existant Pour ajouter des  l ments   un fichier, utilisez **Append**. Pour tronquer un fichier ou le cr er s'il n'existe pas, utilisez **Create**.

Principales valeurs de l'�num�ration <code>FileMode</code>	
Append	Ouvre le fichier s'il existe et acc�de � la fin du fichier, ou cr�e un nouveau fichier. <code>FileMode.Append</code> peut seulement �tre utilis� conjointement avec <code>FileAccess.Write</code> . Toute tentative de lecture �choue et l�ve un <code>ArgumentException</code> .
Create	Sp�cifie que le syst�me d'exploitation doit cr�er un fichier. Si le fichier existe, il est remplac�. Cela n�cessite <code>FileIOPermissionAccess.Write</code> et <code>FileIOPermissionAccess.Append</code> .
CreateNew	Sp�cifie que le syst�me d'exploitation doit cr�er un fichier. Cela n�cessite <code>FileIOPermissionAccess.Write</code> . Si le fichier existe, un <code>IOException</code> est lev�.
Open	Sp�cifie que le syst�me d'exploitation doit ouvrir un fichier existant. La possibilit� d'ouvrir le fichier d�pend de la valeur sp�cifi�e par <code>FileAccess</code> . <code>System.IO.FileNotFoundException</code> est lev� si ce fichier n'existe pas.
OpenOrCreate	Sp�cifie que le syst�me d'exploitation doit ouvrir un fichier s'il existe ; sinon, un nouveau fichier doit �tre cr�e. Si le fichier est ouvert avec <code>FileAccess.Read</code> , <code>FileIOPermissionAccess.Read</code> est requis. Si l'acc�s au fichier est <code>FileAccess.ReadWrite</code> et si le fichier existe, <code>FileIOPermissionAccess.Write</code> est requis. Si l'acc�s au fichier est <code>FileAccess.ReadWrite</code> et si le fichier n'existe pas, <code>FileIOPermissionAccess.Append</code> est requis en plus de <code>Read</code> et <code>Write</code> .
Truncate	Sp�cifie que le syst�me d'exploitation doit ouvrir un fichier existant. Une fois ouvert, le fichier doit �tre tronqu� de mani�re � ce que sa taille soit �gale � z�ro octet. Ceci n�cessite <code>FileIOPermissionAccess.Write</code> . Toute tentative de lecture d'un fichier ouvert avec Truncate entra�ne une exception.

3.1.2. Op rations autoris es : FileAccess

Les valeurs de l' num ration FileAccess permettent de d terminer quelles op rations de lecture et d' criture seront autoris es.

Principales valeurs de l'�num�ration FileAccess	
Read	Acc�s en lecture au fichier. Les donn�es peuvent �tre lues � partir de ce fichier.
ReadWrite	Acc�s en lecture et en �criture au fichier. Les donn�es peuvent �tre �crites dans le fichier et lues � partir de celui-ci.
Write	Acc�s en �criture au fichier. Les donn�es peuvent �tre �crites dans le fichier.

3.1.1. Autorisation de partage : FileShare

Les valeurs de cette  num ration permettent de d finir quelles op rations simultan es peuvent  tre r alis es par les autres processus. A manipuler avec pr cautions. Par exemple, vous pouvez permettre   deux processus d'acc der en lecture   un m me fichier.

Principales valeurs de l'�num�ration FileShare	
None	Refuse le partage du fichier en cours. Toute demande d'ouverture du fichier (par ce processus ou un autre) �chouera jusqu'� la fermeture du fichier.
Read	Permet l'ouverture ult�rieure du fichier pour la lecture. Si cet indicateur n'est pas sp�cifi�, toute demande d'ouverture du fichier pour la lecture (par ce processus ou un autre) �chouera jusqu'� la fermeture du fichier.
ReadWrite	Permet l'ouverture ult�rieure du fichier pour la lecture ou l'�criture. Si cet indicateur n'est pas sp�cifi�, toute demande d'ouverture du fichier pour la lecture ou l'�criture (par ce processus ou un autre) �chouera jusqu'� la fermeture du fichier.
Write	Permet l'ouverture ult�rieure du fichier pour l'�criture. Si cet indicateur n'est pas sp�cifi�, toute demande d'ouverture du fichier pour l'�criture (par ce processus ou un autre) �chouera jusqu'� la fermeture du fichier.

3.1.2. Exemples d'ouverture d'un fichier

Privil giez de d finir le plus pr cis ment les options pour  viter de recourir aux valeurs par d faut.

Ainsi, pr f rez la forme 2   la forme 1

1. En sp cifiant son nom et son mode d'ouverture

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open);
```

2. En sp cifiant son nom, son mode d'ouverture, un mode d'acc s et un mode de partage

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open, FileAccess.Read, FileShare.Read);
```

Ouverture du fichier Exemple.txt existant (FileMode.Open), en lecture seule (FileAccess.Read), autorisant les autres processus   r aliser des lectures.

3.1. Op rations sur les donn es

Nous utiliserons ici les classes **StreamReader** et **StreamWriter**. Elles traitent par d faut des flux de caract res encod s en UTF 8 o  chaque caract re est encod  sur 1   4 octets.

La classe **StreamReader** permet de lire un fichier texte, ligne par ligne, caract re par caract re, par bloc ou en totalit .

Elle permet  galement de sp cifier le type d'encodage.

Pour lire le contenu du fichier texte Exemple.txt

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open);  
StreamReader sr = new StreamReader (fs);  

string strLine = sr.ReadLine();  
while (strLine != null)  
{
    // traitement de la ligne lue
    // ...
    strLine = sr.ReadLine();  
}

sr.Close();  
fs.Close();  
```

  Ouverture du fichier **Exemple.txt** existant

  Cr ation d'un objet StreamReader avec r f rence au FileStream

  Lecture de la premi re ligne du fichier

  Boucle de lecture de tous les enregistrements du fichier

Lire un fichier s quentiel de bout en bout suppose de programmer une **boucle**. ;

Comme on sait rarement   l'avance combien d'enregistrements comporte le fichier, il faut tester la fin de fichier. Si aucun enregistrement n'est lu, la m thode **ReadLine** renvoie **null**

  Lecture de l'enregistrement suivant

  Fermeture du lecteur en cours.

  Fermeture du fichier

Principales m thodes de StreamReader

Close	Ferme le lecteur en cours et le flux sous-jacent.
DiscardBufferedData	Permet � StreamReader d'ignorer ses donn�es en cours.
Peek	Substitu�. Retourne le prochain caract�re disponible, mais ne le consomme pas.
Read	Surcharg�. Substitu�. Lit le caract�re ou le jeu de caract�res suivant dans le flux d'entr�e.
Read(char [] buffer, int index, int count)	Lecture d'un jeu de caract�res suivant dans le flux d'entr�e.
ReadBlock	Lit un maximum de caract�res � partir du flux en cours et �crit les donn�es dans <i>buffer</i> , en commen�ant par <i>index</i> .
ReadLine	Substitu�. Lit une ligne de caract�res � partir du flux en cours et retourne les donn�es sous forme de cha�ne. Retourne null en fin de fichier.
ReadToEnd	Substitu�. Lit le flux entre la position actuelle et la fin du flux.

Il est pr f rable de pr ciser l'encodage lors de la cr ation du Flux de lecture

```
FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.Read);
StreamReader sr = new StreamReader(fs, System.Text.Encoding.UTF8);
    string s = sr.ReadLine();
    while (s != null)
    {
        // Traitement de la ligne
        s = sr.ReadLine();
    }
```

La classe **StreamWriter** permet d' crire dans un flux.

Exemple : lecture du fichier texte Exemple.txt et  criture des lignes lues dans un nouveau fichier Out.txt.

```
FileStream fsI = new FileStream("Exemple.txt", FileMode.Open);
FileStream fsO = new FileStream("Out.txt", FileMode.CreateNew);❶

StreamReader sr = new StreamReader (fsI);
StreamWriter sw = new StreamWriter(fsO);❷

string strLine = sr.ReadLine();
while (strLine != null)
{
    // traitement de la ligne lue
    sw.WriteLine(strLine);❸
    strLine = sr.ReadLine();
}

sr.Close();
sw.Close();❹

fsI.Close();
fsO.Close();
```

- ❶ Cr ation et ouverture du fichier Out.txt
- ❷ D claration d'un objet StreamWriter sw   partir de l'objet FileStream de sortie.
- ❸  criture de la ligne lue dans le flux de sortie ;
- ❹ Fermeture du lecteur.

Principales m�thodes de StreamWriter	
Close	Ferme le lecteur en cours et le flux sous-jacent.
Flush	Force l'�criture sur le disque.
Write	Surcharg�. �crit dans le flux.
WriteLine	Surcharg�. �crit des donn�es de la mani�re sp�cifi�e par les param�tres surcharg�s, suivies d'un terminateur de ligne.

Un exemple au complet.

Ici, j'ai créé une fonction de conversion d'un fichier source vers un fichier cible dont je précise les chemins d'accès complets et les encodages.

```
/// <summary>
/// Conversion d'un fichier texte depuis un encodage vers un autre encodage
/// </summary>
/// <param name="pathSource">Chemin complet du fichier source</param>
/// <param name="pathCible">Chemin complet du fichier cible</param>
/// <param name="encodingSource">Type de l'encodage du fichier source</param>
/// <param name="encodingCible">Type de l'encodage du fichier cible</param>
|
static void ConvertirFichier(string pathSource, string pathCible,
    Encoding encodingSource, Encoding encodingCible)
{
    // Ouverture du fichier Source en lecture

    FileStream fSSource = new FileStream(pathSource, FileMode.Open, FileAccess.Read,
        FileShare.Read);

    StreamReader sRSource = new StreamReader(fSSource, encodingSource);

    // Ouverture du fichier cible en écriture.
    // Il est créé s'il n'existe pas
    // Accès exclusif

    FileStream fSCible = new FileStream(pathCible, FileMode.OpenOrCreate, FileAccess.Write,
        FileShare.None);
    StreamWriter sRCible = new StreamWriter(fSCible, encodingCible);

    string enregistrement = sRSource.ReadLine();
    while (enregistrement != null)
    {
        sRCible.WriteLine(enregistrement);
        enregistrement = sRSource.ReadLine();
    }
    sRCible.Close();
    sRSource.Close();
    fSCible.Close();
    fSSource.Close();
}
```

Le fichier source est encodé à l'origine au format ANSI, ou chaque caractère est représenté par un octet. La représentation du caractère sera donc dépendante de la page de codes de ma machine pour restituer correctement les caractères définis dans la deuxième partie de la table ASCII.

Premier appel de la méthode en demandant une conversion en Unicode (UTF16). Les caractères seront donc stockés sur 2 octets.

```
static void Main(string[] args)
{
    ConvertirFichier(@"D:\_CDI\Mod01\06 C# Objet Partie 2\Exemples\Fichier\TexteANSI.txt",
        @"D:\_CDI\Mod01\06 C# Objet Partie 2\Exemples\Fichier\TexteUTF16.txt",
        ASCIIEncoding.Default,
        ASCIIEncoding.Unicode);
}
```

La classe **ASCIIEncoding** nous permet de pr ciser l'encodage.

ASCIIEncoding.Default correspond   l'encodage **ANSI**.

ASCIIEncoding.Unicode correspond   l'encodage **UTF16**

Le document d'origine affich  avec NotePad ++

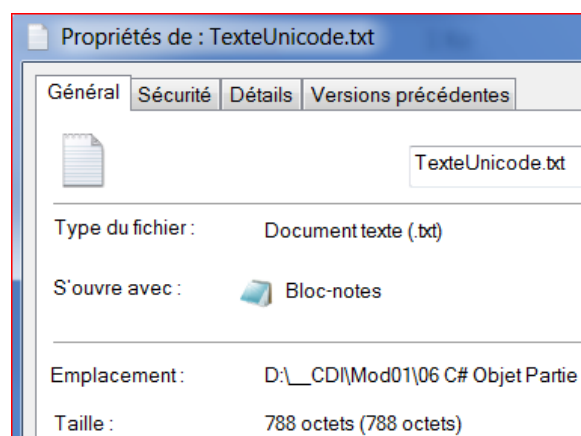
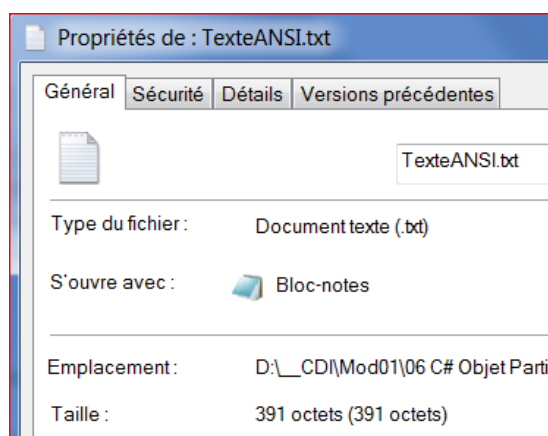
```
1 Ceci est un document cod  en ANSI
2 Les caract res accentu s seront correctement pr sent s en tenant compte de la page de code
3 de ma machine : la pr sentation d'un caract re de m me poids (valeur num rique) peut donc
4 diff r  en fonction de la culture. Ce fichier sera transform  en UTF8 et Unicode (UTF16)
5 UTF8 peut  tre encod  sur 1   4 octets. Unicode (UTF16) est encod  sur 2 octets.
```

Le document cod  en Unicode

```
1 Ceci est un document cod  en ANSI
2 Les caract res accentu s seront correctement pr sent s en tenant compte de la page de code
3 de ma machine : la pr sentation d'un caract re de m me poids (valeur num rique) peut donc
4 diff r  en fonction de la culture. Ce fichier sera transform  en UTF8 et Unicode (UTF16)
5 UTF8 peut  tre encod  sur 1   4 octets. Unicode (UTF16) est encod  sur 2 octets.
```

Les deux fichiers sont correctement encod s.

Quelle diff rence alors ? **La taille**



Les enregistrements dans le fichier en Unicode (UTF16) occupent 788 octets alors qu'au format ANSI ils n'occupent que 391 octets.

Je peux poursuivre les tests avec des fichiers cibles cod s en ASCII et en UTF8. Pour rappel en UTF8, un caract re occupera de 1   4 octets.

En ASCII, j'ai perdu les caract res accentu s. Le jeu de caract res ASCII est cod  sur 7 bits...

```
1 Ceci est un document cod  en ANSI
2 Les caract res accentu s seront correctement pr 
3 de ma machine : la pr sentation d'un caract re d
4 diff r  en fonction de la culture. Ce fichier s
5 UTF8 peut  tre encod  sur 1   4 octets. Unicode
```

4. Assurer la persistance des objets

Nous allons aborder ici cet objectif par le biais de la sérialisation des objets.

La sérialisation correspond au processus de conversion de l'état d'un objet en mémoire en un objet persistant ou transportable.

Le .NET Framework comprend deux technologies de base de sérialisation, la sérialisation binaire et la sérialisation XML. Elles peuvent être ensuite déclinées pour fournir des mécanismes normés, par exemple dans le cadre des services Web et du standard SOAP (Services Oriented Architecture Protocol). Nous verrons aussi par la suite un autre format pour la sérialisation très usité aujourd'hui JSON (JavaScript On Notation).

La sérialisation binaire, qui préserve le respect des types, permet de conserver l'état d'un objet entre plusieurs appels d'une application. Très facile à utiliser, très performante, elle reste limitée à l'échange entre deux applications sur l'architecture .Net.

Vous pouvez partager un objet entre plusieurs applications en le sérialisant dans le Presse-papiers.

Vous pouvez sérialiser un objet vers un flux, un disque, la mémoire, le réseau, et ainsi de suite.

L'accès distant utilise la sérialisation pour passer des objets « par valeur » d'un ordinateur ou d'un domaine d'application à un autre.

La sérialisation XML sérialise uniquement des propriétés et des champs publics mais ne conserve pas les types. XML étant une norme ouverte, elle constitue une option intéressante pour partager des données via le Web. Ces mécanismes de sérialisation XML sont mis en œuvre dans les services Web au travers du protocole SOAP : SOAP constitue une offre intéressante pour l'interopérabilité des applications est également une norme ouverte et représente par conséquent une option avantageuse.

Nous utiliserons ici ces principes pour vous permettre de comprendre les mécanismes sous-jacents mis en œuvre dans les services Web par exemple.

Nous verrons par la suite des apprentissages que nos objets métiers sont rendus persistants par les mécanismes de stockage dans des bases de données relationnelles.

En résumé, la sérialisation binaire a pour inconvénients :

- le manque d'universalité de la présentation des données sérialisées. Nous devons restreindre son utilisation à des objectifs d'échanges entre application d'une même architecture logicielle.

Et pour avantages :

- La conservation des types d'origine de nos objets
- Un volume de données beaucoup plus restreint et donc de meilleures performances.

La sérialisation XML a pour inconvénients :

- Un volume de données beaucoup plus important du fait du caractère verbeux d'XML.
- La perte d'informations sur les types même s'il existe des solutions pour pallier ce phénomène pas le biais des documents de description de type (DTD ou XSD)

Et pour avantages :

- D'une forme universellement reconnue qui nous permettra de communiquer nos objets à n'importe quel système.
- Une présentation compréhensible en l'état (ce qui n'est pas le cas du binaire)

4.1. Sérialisation illustrée

Dans cet exemple, nous allons sérialiser nos objets métiers de type **Stagiaire** qui auront été préalablement stockés en mémoire dans un objet de type **Stagiaires**, collection générique fortement typée.

Nous devons tout d'abord indiquer que notre classe est une collection et qu'elle est sérialisable.

Par convention, les collections sont toujours désignées avec la forme plurielle du type.

```
/// <summary>
/// Collection Stagiaires
/// Cette classe propose des méthodes de sérialisation et désérialisation
/// </summary>
[Serializable()]
public class Stagiaires : List<Stagiaire>
{
```

La classe Stagiaires, collection d'objets de type Stagiaire, hérite de la classe de collection générique List<T>.

L'attribut **[Serializable()]** indique que cette collection pourra être sérialisée. Cet attribut est obligatoire.

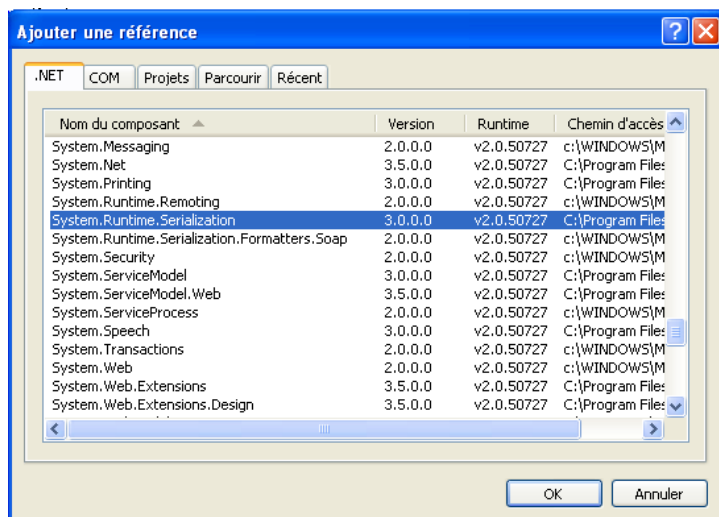
La classe Stagiaire doit être elle aussi marquée comme sérialisable.

Il est possible de ne pas sérialiser toutes les propriétés en recourant à l'attribut [not Serializable].

4.1.1. Sérialisation binaire

Nous allons ici utiliser une classe **BinaryFormatter** qui se trouve dans la bibliothèque de fonctions **Runtime.Serialization**.

Cette bibliothèque n'est pas référencée par défaut. La première chose à faire est donc de lier cette dll à notre projet.



Nous allons ensuite utiliser un objet qui formatera le flux en tenant compte du type à sérialiser.

Il s'agit d'un objet créé à partir d'une classe spécialisée de « Formatter », ici la classe `BinaryFormatter`.

Il existe de nombreux formateurs, aussi est-il utile de préciser l'espace de noms où se trouve référencé le `BinaryFormatter` :

```
using System.Runtime.Serialization.Formatters.Binary;
```

Ensuite, nous avons besoin d'un flux et d'un fichier, mais il s'agit ici d'une approche déjà connue.

```
FileStream fs = new FileStream(NomFichier, FileMode.Create)
{
    // Sérialisation à l'aide de BinaryFormatter
    BinaryFormatter bf = new BinaryFormatter();
```

Appel de la méthode pour sérialiser notre objet et le stocker dans le fichier.

```
bf.Serialize(fs, this);
```

this représente ici l'objet collection de la classe `Stagiaires` sur lequel nous travaillons.

Pour la dé-sérialisation, l'approche est similaire si ce n'est que le flux doit être ouvert en lecture et que la méthode du « formateur » utilisée est `Deserialize`.

```
// Désérialisation à l'aide de BinaryFormatter
BinaryFormatter bf = new BinaryFormatter();
// Désérialisation avec conversion de type car deserialize renvoie
// un objet et ajout à la liste

this.AddRange(bf.Deserialize(fs) as Stagiaires);
```

4.1.2. Sérialisation XML

Le principe retenu est le même que celui utilisé précédemment. Nous allons utiliser une classe de sérialisation de l'espace de noms `System.Xml.Serialization`. Il faudra donc l'introduire via une directive `using`.

Cette fois, nous utiliserons, comme lors de la manipulation de fichier texte, un flux spécialisé dans la manipulation de données XML. Nous précisons l'encodage utilisé.

This représente ici un objet collection à sérialiser dans le fichier cible.

```
FileStream fileStream = new FileStream(string.Format(CultureInfo.InvariantCulture, @"{0}\Salaries.xml", | pathXmlDocument)
XmlTextWriter xmlTW = new XmlTextWriter(fileStream, Encoding.UTF8);
XmlSerializer xmlS = new XmlSerializer(this.GetType());
xmlS.Serialize(xmlTW, this);
fileStream.Close();
```

Pour la dé-sérialisation, nous utiliserons un lecteur XML.

```
FileStream fileStream;  
XmlTextReader xmlTR;  
XmlSerializer xmlS;  
  
fileStream = new FileStream(string.Format(CultureInfo.InvariantCulture, @"{0}\Salaries.xml", pathXmlDocument),  
    FileMode.Open, FileAccess.Read, FileShare.Read);  
xmlTR = new XmlTextReader(fileStream);  
xmlS = new XmlSerializer(this.GetType());  
base.AddRange(xmlS.Deserialize(xmlTR) as Salaries);  
fileStream.Close();
```

Nous avons conservé ici la sérialisation de base d'un objet en XML dans le but principal de rendre persistantes ses propriétés.

Si notre sérialisation avait pour objet l'échange de données avec un système externe nous aurions pu éventuellement modifier la construction du flux en précisant par exemple que certaines propriétés devaient figurer comme attributs d'un élément plutôt que comme éléments XML.

Précisez que vous souhaitez inclure dans la sérialisation XML des salariés, classe Salaries, le type Commercial. C'est obligatoire en XML.

```
[Serializable()]  
[XmlInclude(typeof(Commercial))]  
public class Salaries : List<Salarie>  
{
```

4.1.3. Limite de la sérialisation XML

Contrairement à la sérialisation binaire, la sérialisation XML permet de sérialiser et dé-sérialiser uniquement les propriétés et champs **publics** de vos objets.

Elle ne sérialise pas les propriétés en lecture seule qu'elle ne pourrait d'ailleurs pas dé-sérialiser faute d'accesseur en écriture.

5. Gestion des ressources

Nous avons vu que le Garbage Collector prenait en charge la libération des ressources managées par le système. Les fichiers et les flux ouverts ne sont pas des ressources managées (il en est de même des fenêtres Windows que nous découvrirons par la suite). Vous devez donc mettre en place un mécanisme explicite de libération de ces ressources.

Deux approches sont possibles :

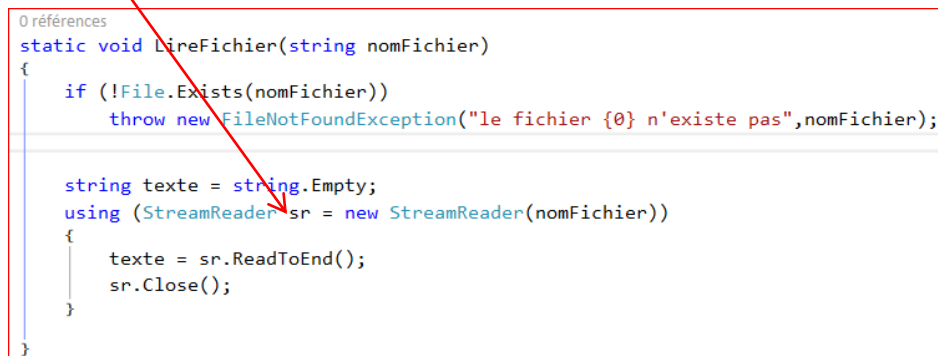
- L'utilisation de l'instruction using
- L'invocation de la méthode dispose dans un bloc try/catch

5.1. Instruction Using

L'approche privilégiée en C# et VB lorsqu'il est possible de l'utiliser pour la libération des ressources de types flux associés à des fichiers. Il s'agit d'un raccourci syntaxique. Le compilateur générera de fait un appel à la méthode Dispose dans un bloc try / catch similaire à l'option suivante.

Dans l'exemple suivant, le contenu d'un fichier est chargé en mémoire et la ressource est ensuite libérée.

L'objet géré doit être instancié lors de l'instruction using : ici un flux en lecture de type.

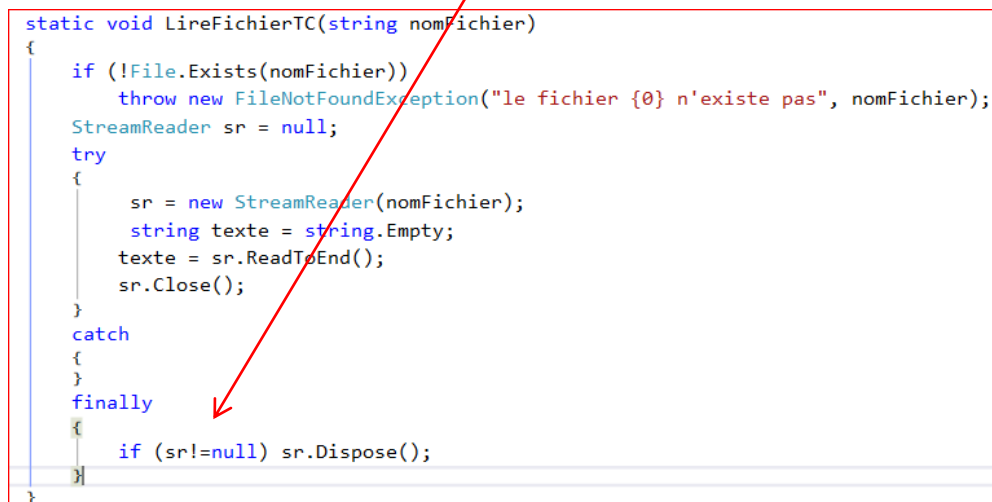


```
0 références
static void LireFichier(string nomFichier)
{
    if (!File.Exists(nomFichier))
        throw new FileNotFoundException("le fichier {0} n'existe pas", nomFichier);

    string texte = string.Empty;
    using (StreamReader sr = new StreamReader(nomFichier))
    {
        texte = sr.ReadToEnd();
        sr.Close();
    }
}
```

5.2. Instruction Try / Catch

Dans cet exemple, la méthode Dispose du flux est invoquée au niveau du bloc finally qui est toujours visité qu'il y ait ou non survenance d'une exception.



```
static void LireFichierTC(string nomFichier)
{
    if (!File.Exists(nomFichier))
        throw new FileNotFoundException("le fichier {0} n'existe pas", nomFichier);
    StreamReader sr = null;
    try
    {
        sr = new StreamReader(nomFichier);
        string texte = string.Empty;
        texte = sr.ReadToEnd();
        sr.Close();
    }
    catch
    {
    }
    finally
    {
        if (sr != null) sr.Dispose();
    }
}
```