Fichiers, dossiers, disques (System.IO)

(J-C Armici, 2006)

D'autres documents et exemples concernant la programmation, sur les sites:
www.unvrai.com, www.unfaux.com, jca.developpez.com et www.developpez.com

Ce document basé sur le Framework .NET 2.0 et Visual Studio 2005.

Introduction

Le namespace **System.IO** contient les classes de base prenant en charge les opérations d'entrée/sortie pour les fichiers et la mémoire.

Le tableau suivant présente les principaux membres du namespace **System.IO** :

Classes	Description
BinaryReader **BinaryWriter**	Permettent de sauvegarder et de lire des informations en tant que données binaires.
BufferedStream	Offre un espace de stockage temporaire pour un stream de bytes.
Directory **DirectoryInfo**	Permettent de manipuler la structure des dossiers.
DriveInfo	Fournit les informations concernant les unités de disques.
File **FileInfo**	Permettent de manipuler des ensembles de fichiers.
FileStream	Permet de gérer des fichiers à accès direct dont les données sont représentées par un stream de bytes.
FileSystemWatcher	Permet de surveiller les modifications concernant les fichiers.
MemoryStream	Permet un accès direct à des données en mémoire.
Path	Permet de gérer une chaîne de caractères contenant des informations de fichiers ou dossiers.
StreamWriter **StreamReader**	Permettent de sauvegarder et de lire des informations de type texte dans et depuis un fichier.
StringWriter **StringReader**	Analogues à StreamWriter/StreamReader mais les informations sont stockées dans un string buffer plutôt que dans un fichier.

Des énumérations et un ensemble de classes abstraites sont également disponibles.

Directory, DirectoryInfo, File, FileInfo

Les types **Directory** et **File** permettent des opérations de création, suppression, copie et déplacement en utilisant des membres statiques. Les types **DirectoryInfo** et **FileInfo** exposent des fonctionnalités analogues, mais par l'intermédiaire de méthodes d'instance (les objets correspondant doivent être créés avec **new**).

Les deux premiers (**Directory** et **File**) types généralement retournent des valeurs de type string. Les deux derniers (**DirectoryInfo** et **FileInfo**) types retournent des objets fortement typés et sont à utilisés de préférence pour des opérations récursives, telles que le parcours récursif d'une structure de répertoires.

Les types **DirectoryInfo** et **FileInfo** exposent des fonctionnalités héritées de la classe abstraite **FileSystemInfo**, dont les membres permettent de retrouver les caractéristiques d'un dossier ou d'un fichier. Le tableau suivant présente les propriétés de **FileSystemInfo** :

Propriété	Description
Attributes	Permet de connaître ou de spécifier les attributs associés au fichier courant, représentés par l'énumération **FileAttributes**.
CreationTime	Permet de connaître ou de spécifier la date/heure de création du fichier ou dossier
Exists	Détermine si un fichier ou un dossier existe.
Extension	Permet de retrouver l'extension d'un fichier.
FullName	Permet de retrouver de chemin complet d'un fichier ou dossier.
LastAccessTime	Permet de connaître ou de spécifier la date/heure du dernier accès à un fichier ou dossier.
LastWriteTime	Permet de connaître ou de spécifier la date/heure de la dernière écriture d'un fichier ou d'un dossier.
Name	Retourne le nom du fichier ou du dossier

Le type **FileSystemInfo** dispose également d'une méthode **Delete()** qui permet de supprimer un fichier ou un dossier.

Utilisation du type DirectoryInfo

En plus des fonctionnalités fournies par la classe de base **FileSystemInfo**, **DirectoryInfo** offre les fonctionnalités suivantes :

Membres	Description
Create() **CreateSubdirectory()**	Crée un dossier (ou une suite de sous-dossiers) depuis un emplacement donné.
Delete()	Supprime un dossier ainsi que son contenu.
GetDirectories()	Retourne un tableau de string contenant tous les sous-dossiers du dossier courant.
GetFiles()	Retourne un tableau du type **FileInfo** contenant les fichiers d'un dossier donné.
MoveTo()	Déplace un dossier et son contenu vers un nouvel emplacement.
Parent	Retourne le dossier parent de l'emplacement spécifié.
Root	Retrouve la racine du chemin spécifié.

Exemples d'utilisation de DirectoryInfo

Travail avec le dossier courant de l'application:

```
DirectoryInfo dossier1 = new DirectoryInfo (".");
```

Travail avec un dossier spécifié:

```
DirectoryInfo dossier2 = new DirectoryInfo (@"c:\Documents and Settings");
```

Création d'un dossier "test" dans le dossier "c:\windows\temp":

```
DirectoryInfo dossier3 = new DirectoryInfo (@"c:\windows\temp\test");
dossier3.Create();
```

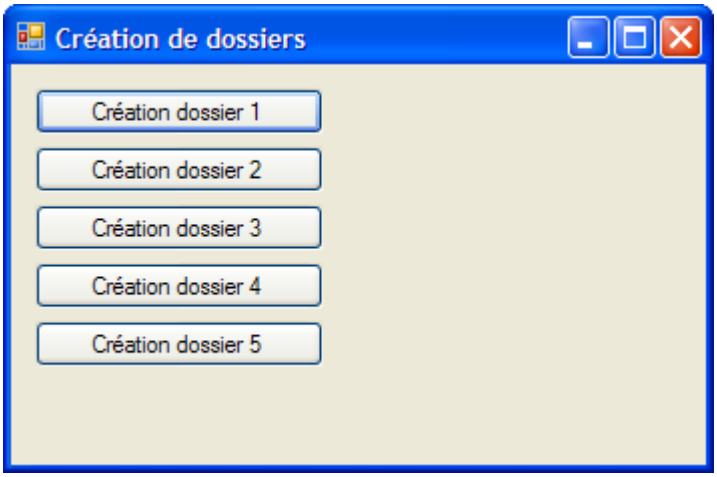
Enumération FileAttributes

L'énumération **FileAttributes** fournit la liste des attributs de fichiers ou de dossiers

```
public enum FilAttributes{
  ReadOnly,           // en lecture seule
  Hidden,             // fichier caché
  System,             // fichier système
  Directory,          // dossier
  Archive,            // pour marquage des fichiers
  Device,             // réservé pour usage futur
  Normal,             // si pas d'autres attributs
  Temporary,          // fichier temporaire
  SparseFile,         // fichier éparpillé
  ReparsePoint,       // contient des données utilisateur associées
  Compressed,         // fichier compressé
  Offline,            // fichier et données pas accessibles pour le moment
  NotContentIndexed,  // pas indexé par le service d'indexation des fichiers
  Encrypted           // fichier ou dossier crypté
}
```

Création de sous-dossiers

Le programme suivant illustre diverses manières de créer des dossiers et sous-dossiers.

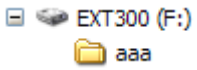


Remarque: *entre chaque clic sur un bouton le ou les dossiers créés précédemment sont supprimés.*

Le code du 1er bouton est:

```
DirectoryInfo dossier1 = new DirectoryInfo(@"f:\aaa");
dossier1.Create();
```

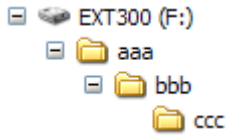
Résultat:



Le code du 2ème bouton est:

```
DirectoryInfo dossier2 = new DirectoryInfo(@"f:\aaa\bbb\ccc");
Dossier2.Create();
```

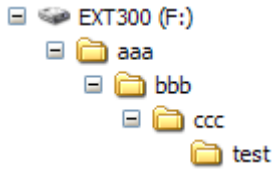
Résultat:



Le code du 3ème bouton est:

```
DirectoryInfo dossier3 = new DirectoryInfo(@"f:\aaa\bbb\ccc");
dossier3.CreateSubdirectory("test");
```

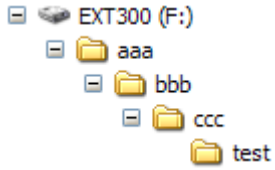
Résultat:



Le code du 4ème bouton est:

```
DirectoryInfo dossier4 = new DirectoryInfo(@"f:\aaa");
dossier4.CreateSubdirectory(@"bbb\ccc\test");
```

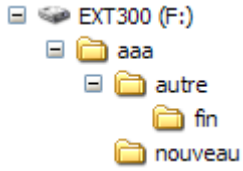
Résultat:

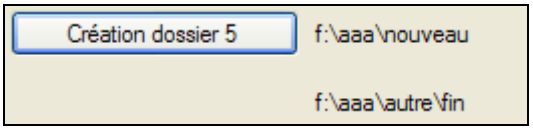


Le code du 5ème bouton est:

```
DirectoryInfo dossier5 = new DirectoryInfo(@"f:\aaa");
DirectoryInfo d = dossier5.CreateSubdirectory("nouveau");
label1.Text = d.FullName;
d = dossier5.CreateSubdirectory(@"autre\fin");
label2.Text = d.FullName;
```

Résultat:



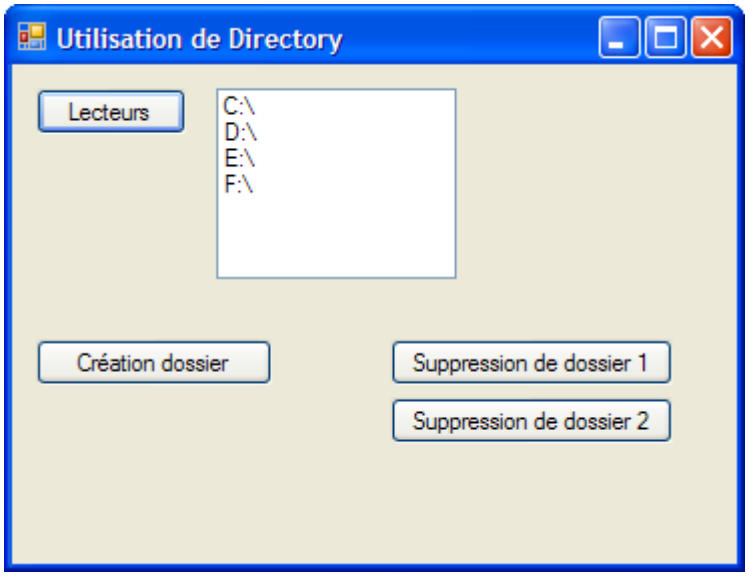


Ce dernier fragment illustre le fait que **CreateSubdirectory()** retourne un type **DirectoryInfo** lorsque la création est exécutée avec succès.

Utilisation du type Directory

Il convient de rappeler que les membres de **Directory** sont analogues à ceux de **DirectoryInfo**. Toutefois pour **DirectoryInfo** ils retournent des données de type **FileInfo** ou **DirectoryInfo**, alors que pour **Directory**, ils retournent des données de type string.

Voici un exemple comportant deux parties. La première affiche les lecteurs logiques de l'ordinateur. La seconde illustre la suppression de dossiers à l'aide des fonctionnalités de **Directory**.



Le code derrière le bouton "Lecteurs" est:

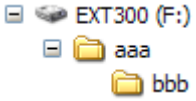
```
string[] lecteurs = Directory.GetLogicalDrives();
lbDrives.Items.AddRange(lecteurs);
```

On remarque qu'il n'y a pas de "new" devant Directory.

Voici maintenant le code déclenché par le bouton "Création de dossier":

```
DirectoryInfo dossier1 = new DirectoryInfo(@"f:\aaa\bbb");
dossier1.Create();
```

La création de dossier est effectuée comme nous l'avons vu précédemment et son effet est:

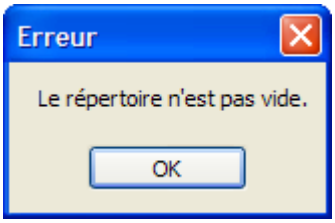


Le bouton "Suppression de dossier 1" déclenche le code suivant:

```
try
{
  Directory.Delete(@"f:\aaa\bbb");
}
catch (IOException ex)
{
  MessageBox.Show(ex.Message, "Erreur");
}
```

L'exécution de ce code ne provoque, en principe, aucune erreur. Le dossier "bbb" est supprimé.

Il n'en va pas de même si l'on essaie de supprimer le dossier "aaa" (qui n'est pas vide). Dans ce cas, voici ce qui est affiché:



D'où l'importance de trapper les erreurs éventuelles à l'aide de la structure **try…catch**.

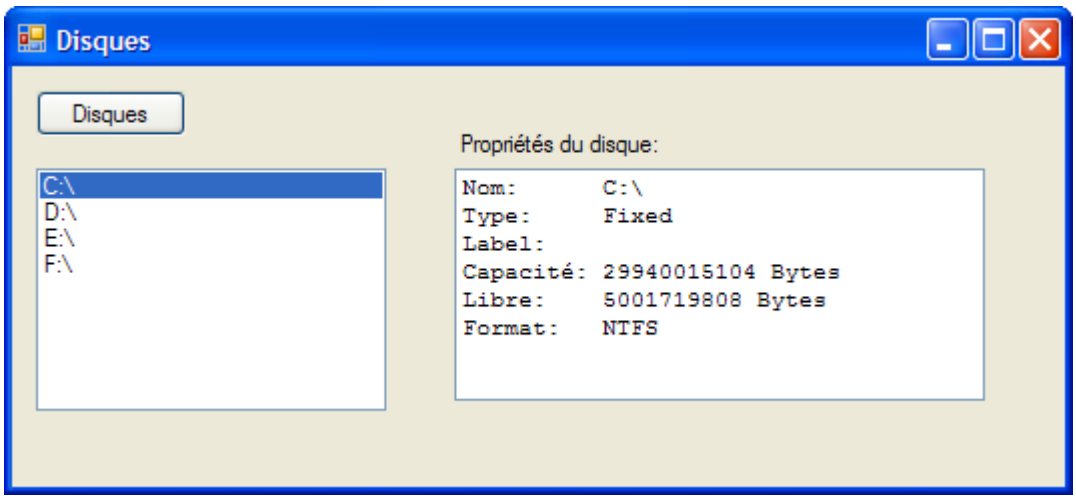
Le bouton "Suppression de dossier 2" déclenche le code suivant:

```
try
{
  Directory.Delete(@"f:\aaa",true);
}
catch (IOException ex)
{
  MessageBox.Show(ex.Message, "Erreur");
}
```

Nous voyons qu'il est possible de passer un second paramètre à **Delete()** autorisant une suppression récursive des dossiers. Bien entendu le contenu (fichiers et dossiers) du dossier "aaa" ainsi que celui de tous les dossiers sous-jacents seront supprimés

Utilisation de la classe DriveInfo

La classe **DriveInfo** permet d'obtenir divers renseignement concernant les unités logiques. En voici une illustration dans l'exemple suivant:



Le bouton "Disques" remplit la liste de gauche avec la dénomination des disques logiques. Son code est le suivant:

```
private void btnDisques_Click(object sender, EventArgs e)
{
  DriveInfo[] disques = DriveInfo.GetDrives();
  lbDisques.Items.AddRange(disques);
}
```

Lorsque l'utilisateur choisit un lecteur dans la liste, des informations concernant ces lecteurs sont affichées dans la liste de droite.

```
private void lbDisques_SelectedIndexChanged(object sender, EventArgs e)
{
 // récupération de la lettre de l'unité
  DriveInfo di = new DriveInfo(lbDisques.Text);

  // affichage des informations des unités de disques
  lbInfos.Items.Clear();
  lbInfos.Items.Add("Nom:      " + di.Name);
  lbInfos.Items.Add("Type:     " + di.DriveType);
  if (di.IsReady)
  {
    lbInfos.Items.Add("Label:    " + di.VolumeLabel);
    lbInfos.Items.Add(string.Format("Capacité: {0} Bytes",di.TotalSize));
    lbInfos.Items.Add(string.Format("Libre:    {0} Bytes",di.TotalFreeSpace));
    lbInfos.Items.Add("Format:   " + di.DriveFormat);
  }
}
```

Utilisation de la classe FileInfo

La classe **FileInfo** permet d'obtenir divers renseignement concernant les fichiers.

En plus des fonctionnalités héritées de **FileSystemInfo**, la classe **FileInfo** contient les membres figurant dans le tableau ci-dessous:

Membre	Description
AppendText()	Crée un objet de type **StreamWriter** permettant d'ajouter du texte à un fichier.
CopyTo()	Copie un fichier existant vers un nouveau fichier.
Create()	Crée un nouveau fichier et retourne un objet **FileStream** permettant de gérer le nouveau fichier créé.
CreateText()	Crée un objet de type **StreamWriter** qui écrit un nouveau fichier texte.
Delete()	Supprime le fichier auquel une instance de **FileInfo** est attachée.
Directory	Retourne une instance du dossier parent.
DirectoryName	Retourne le chemin complet du dossier parent.
Length	Retourne la taille du fichier ou du dossier.
MoveTo()	Déplace le fichier spécifié vers un nouvel emplacement (avec possibilité de choisir un nouveau nom).
Name	Retourne le nom du fichier.
Open()	Ouvre un fichier en spécifiant divers privilèges de lecture/écriture et

	de sécurité.
OpenRead()	Crée un **FileStream** en lecture seule.
OpenText()	Crée un objet de type **StreamReader** permettant de lire un fichier existant.
OpenWrite()	Crée un **FileStream** en écriture seule.

La plupart des membres de la classe **FileInfo** retourne un objet permettant de lire ou d'écrire des données en utilisant le fichier associé, et ceci sous différents formats.

Méthode Create()

Une manière de créer un descripteur de fichier (handle) est d'utiliser la méthode **Create()**.

```
FileInfo f = new FileInfo (@"c:\tmp\donnees.dat");
FileStream fs = f.Create();

// ici utilisation de l'objet FileStream fs

fs.Close();
```

Il est à noter que l'objet **FileStream** retourné par **Create()** donne tous les droits de lecture/écriture.

Méthode Open()

Cette méthode permet d'ouvrir des fichiers existant ou d'en créer de nouveaux, avec plus de contrôle sur les droits d'accès que la méthode **Create()**. La méthode **Open()** retourne également un objet de type **FileStream**.

```
FileInfo f = new FileInfo (@"c:\tmp\donnees.dat");
FileStream fs = f.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite,
                       FileShare.None);

// ici utilisation de l'objet FileStream fs

fs.Close();
```

La méthode **Open()** utilisée ici dispose de trios paramètres:

- La manière dont sera utilisé le fichier: CreateNew (crée un nouveau fichier qui ne doit pas déjà exister), Create (si le nouveau fichier existe, il est écrasé), Open, OpenOrCreate (ouvre un fichier s'il existe, le crée s'il n'existe pas), Truncate, Append.
- Le mode d'accès au fichier: Read, Write, ReadWrite.
- Le mode de partage du fichier: None, Read, Write, ReadWrite.

Méthodes OpenRead() et OpenWrite()

Ces méthodes sont très proches de la méthode **Open()**. Elles fournissent directement un objet de type **FileStream** en mode readonly ou writeonly.

Méthode OpenText()

Contrairement aux méthodes précédentes, **OpenText()** retourne un objet de type **StreamReader** plutôt que **FileSTream**.

```
FileInfo f = new FileInfo (@"c:\tmp\donnees.dat");
StreamReader sr = f.OpenText();

// ici utilisation de l'objet StreamReader sr

sr.Close();
```

Méthodes CreateText() et AppendText()

Ces méthodes retournent un objet de type StreamWriter. La première en vue d'écrire des informations de type texte dans un nouveau fichier. La seconde pour ajouter des informations de type texte à un fichier existant.

Utilisation du type File

Comme FileInfo, File dispose également de méthodes AppendText(), Create(), CreateText(), Open(), OpenRead(), OpenWrite() et OpenText(). En voici une simple illustration:

```
FileStream fs = File.OpenRead(@"c:\tmp\donnees.dat");
…
fs.Close();

StreamReader sr = File.OpenText(@"c:\tmp\donnees.dat");
…
sr.Close();
```

Dans sa version 2.0 .NET enrichit le type **File** avec les méthodes ci-dessous:

Méthodes	Description
ReadAllBytes()	Ouvre le fichier spécifié, retourne son contenu sous forme binaire dans un tableau de bytes, puis ferme le fichier.
ReadAllLines()	Ouvre le fichier spécifié, retourne son contenu sous forme texte dans un tableau de strings, puis ferme le fichier.
ReadAlleText()	Ouvre le fichier spécifié, retourne son contenu sous forme texte dans un string, puis ferme le fichier.
WriteAllBytes()	Ouvre le fichier spécifié, y écrit le contenu d'un tableau de bytes, puis ferme le fichier.
WriteAllLines()	Ouvre le fichier spécifié, y écrit le contenu d'un tableau de strings, puis ferme le fichier.
WriteAllText()	Ouvre le fichier spécifié, y écrit les données, puis ferme le fichier.

Ces nouvelles méthodes simplifient grandement les opérations de lecture et écriture dans les fichiers. Un des avantages contribuant à cette simplification est que les fichiers sont automatiquement fermés.

Voici un exemple montrant cette simplification de programmation. Ce programme utilise des données qui se trouvent dans un tableau de strings:

```
string[] donnees = { "ligne 1", "ligne 2", "ligne 3", "ligne 4",
                     "dernière ligne" };
```

Au moment du lancement (Form_Load) les données sont simplement affichées dans le listBox de gauche (on remarquera l'utilisation de la méthode **AddRange()** permettant d'éviter une boucle):

```
listBox1.Items.AddRange(donnees);
```

Le bouton "Ecriture dans le fichier" provoque simplement l'exécution de:

```
File.WriteAllLines(txtNom.Text, donnees);
```

txtNom.Text est le nom du fichier (ici "test.txt") et **donnees** est le tableau de strings contenant les données.

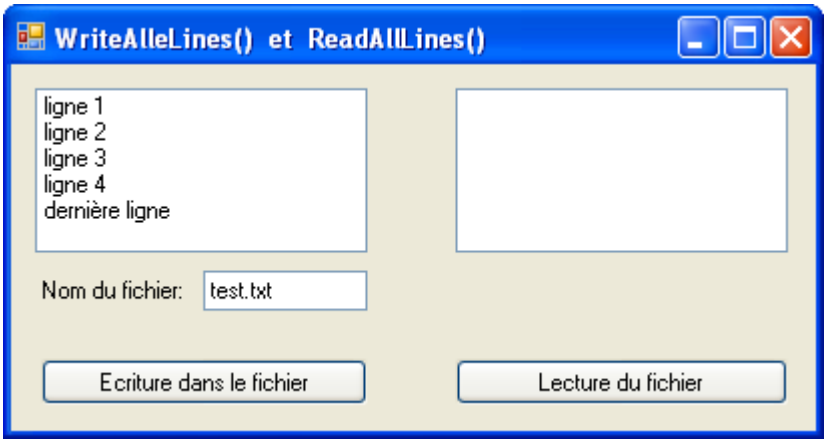
Le bouton "Lecture du fichier" lit les données du fichier et les affiche dans le listBox de droite. Voici le code correspondant:

```
foreach (string ligne in File.ReadAllLines(txtNom.Text))
  listBox2.Items.Add(ligne);
```

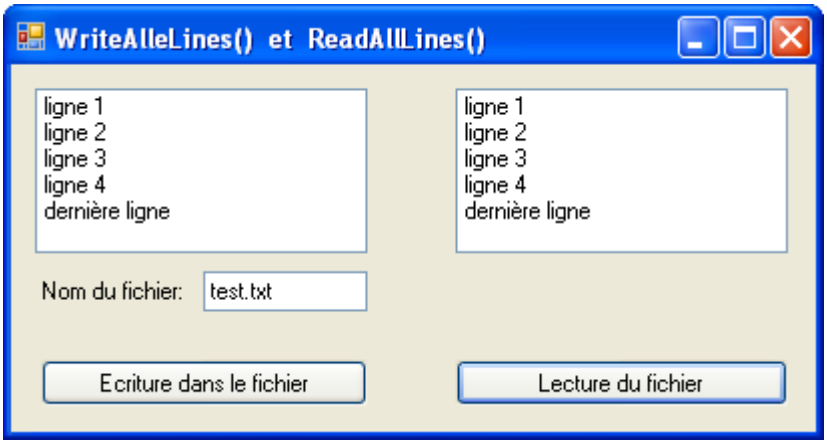
Mais il est encore plus simple, et pas plus hermétique, d'écrire, de manière équivalente:

```
listBox2.Items.AddRange(File.ReadAllLines(txtNom.Text));
```

Voici comment se présente le programme au moment du lancement:



Puis, après avoir cliqué sur les deux boutons, respectivement écriture et lecture:



Il est important de garder à l'esprit l'existence de ces méthodes compactes, pouvant rendre de précieux services dans la plupart des cas.

L'inconvénient de ne pas passer par un objet de type **FileInfo** est de ne pas disposer d'informations telles que la date de création ou les attributs du fichier.

La classe abstraite Stream

A ce stade nous avons vu comment obtenir des objets de type **FileStream**, **StreamReader** et **StreamWriter**, mais pas encore comment lire et écrire des données en utilisant ces objets. Ces opérations sont effectuées à l'aide de Streams (flux). Un Stream permet de gérer des flux de données indépendamment du support (fichier, mémoire, connexion réseau, etc).

De la classe abstraite **Stream** dérivent trois classes:

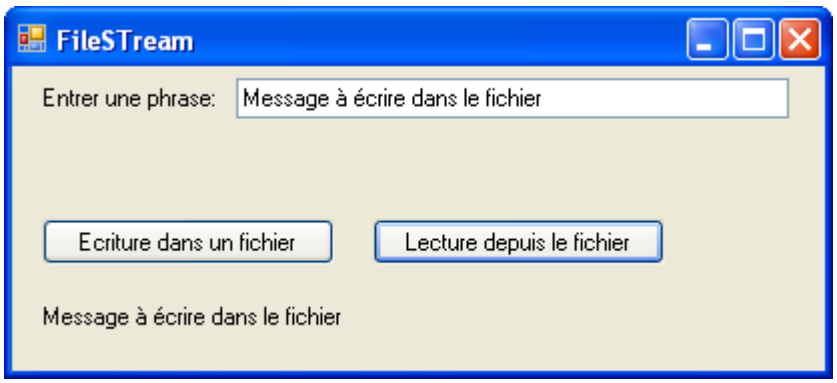
- FileStream
- MemoryStream
- BufferedStream

Le tableau suivant présente les membres de la classe **Stream**:

Membres	Description
CanSeek **CanRead** **CanWrite**	Détermine si un stream supporte les opérations de positionnement, lecture et/ou cériture.
Close()	Ferme le stream courant et libère les ressources éventuellement associées.
Flush()	Pour les streams qui supportent la bufferisation, cette méthode écrit le contenu du buffer, puis efface son contenu.
Length	Fournit la taille d'un stream, en bytes.
Position	Fournit la position dans le stream courant.
Read() **ReadByte()**	Lit une séquence de bytes (ou un seul byte) depuis le stream courant avance le pointeur de position du nombre de bytes lus.
Seek()	Spécifie la position dans le stream courant.
SetLength()	Spécifie la taille du stream courant.
Write() **WriteByte()**	Ecrit une séquence de bytes (ou un seul byte) dans le stream courant avance le pointeur de position du nombre de bytes écrits.

Utilisation de FileStream

La classe **FileStream** fournit une implémentation de la classe **Stream** permettant la gestion des flux pour les fichiers. Elle permet uniquement de lire et d'écrire des bytes ou tableaux de bytes dans un fichier, raison pour laquelle généralement on utilisera les classes de streaming de plus haut niveau. Afin d'illustrer l'utilisation de **FileStream**, voici un exemple qui écrit une phrase dans un fichier, qui la relit et l'affiche.



Le code est le suivant:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace FileStream2
{
  public partial class Form1 : Form
  {
    const string fnom = "data.txt";
    public Form1()
    {
      InitializeComponent();
    }

    private void btnEcrire_Click(object sender, EventArgs e)
    {
      // Création d'un objet FileStream
      FileStream fs = File.Open(fnom, FileMode.Create);

      // Transformation d'un string en tableau de bytes
      byte[] txtByte = Encoding.Default.GetBytes(txtPhrase.Text);

      // Ecriture dans le fichier
      fs.Write(txtByte, 0, txtByte.Length);
      fs.Close();
    }

    private void btnLire_Click(object sender, EventArgs e)
    {
      // Création d'un objet FileStream
```

```
            FileStream fs = File.Open(fnom, FileMode.Open);

            byte[] txtByte = new byte[txtPhrase.Text.Length];
            for (int i = 0; i < txtByte.Length; i++)
            {
              txtByte[i] = (byte)fs.ReadByte();
            }

            // Affichage de la phrase lue
            lRes.Text = Encoding.Default.GetString(txtByte);

            fs.Close();
          }
        }
      }
```

Comme on peut le voir, avec **FileStream** on travaille à bas niveau, directement sur des bytes. Toutefois .NET fournit d'autres manières plus simples (avec les readers et les writers) permettant de gérer les flux.

Utilisation de StreamWriter et StreamReader

Ces deux classes sont utiles lorsqu'il s'agit de gérer des informations textuelles (des chaînes de caractères). **StreamReader** dérive de la classe abstraite **TextReader** disposant essentiellement de méthodes pour lire un caractère et se positionner dans un stream. Il en va de même pour le couple **StreamWriter** et **TextWriter**.

Voici les principaux membres de **TextWriter**:

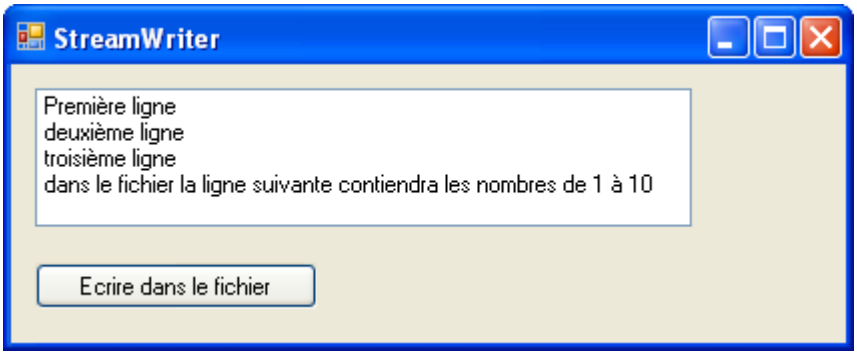
Membres	Description
Close()	Ferme le Writer et libère les ressources associées. Le contenu du buffer est automatiquement écrit sur dans stream
Flush()	Les données du buffer sont écrites et le buffer est vidé. Le Writer n'est pas fermé.
NewLine	Indique la séquence de fin de ligne (par défaut \r\n)
Write()	Ecrit une ligne de texte dans un stream, sans y adjoindre un saut de ligne
WriteLine()	Ecrit une ligne de texte dans un stream et effectue un saut de ligne.

Et les principaux membres de **TextReader**:

Membres	Description
Peek()	Retourne le caractère suivant sans modifier la position du Reader. Une valeur -1 indique la fin du fichier.
Read()	Lit des données depuis un Stream.
ReadLine()	Lit une ligne de caractères depuis le Stream courant et retourne le string correspondant. Une valeur **null** indique la fin du fichier.
ReadToEnd()	Lit tous les caractères depuis la position courant jusqu'à la fin du stream et retourne un string contenant tout les caractères lus.

Exemple d'écriture dans un fichier texte

Le programme suivant écrit dans un fichier donnees.txt le contenu d'un listbox ainsi qu'une liste de nombres:



Le contenu du fichier créé **donnees.txt** est le suivant:

```
Première ligne
deuxième ligne
troisième ligne
dans le fichier la ligne suivante contiendra les nombres de 1 à 10
1 2 3 4 5 6 7 8 9 10
```

Voici le code du programme:

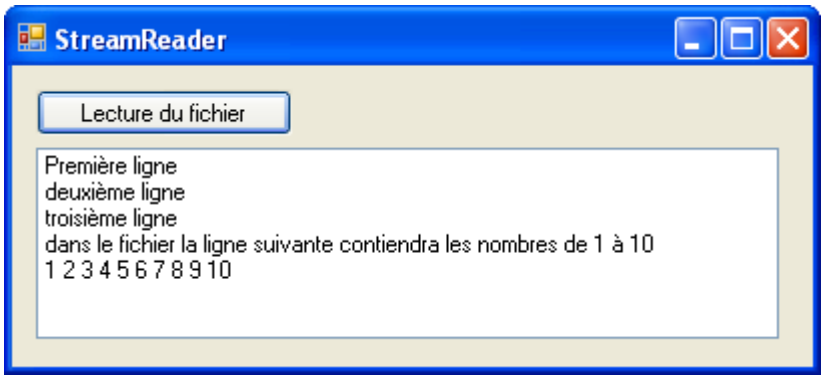
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace StreamWriter2
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }

    private void btnEcrire_Click(object sender, EventArgs e)
    {
      StreamWriter sw = File.CreateText("donnees.txt");
      foreach (string st in lbText.Items)
      {
        sw.WriteLine(st);
      }
      for (int i = 1; i < 11; i++)
        sw.Write(i + " ");
      sw.Close();
    }
  }
}
```

Exemple de lecture depuis un fichier texte

Dans cet exemple, le contenu du fichier "donnees.txt" créé à l'aide du programme précédent est lu et affiché.



Le code du programme est le suivant:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace StreamReader2
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
      StreamReader sr = File.OpenText("donnees.txt");
      string ligne = null;

      while ((ligne = sr.ReadLine()) != null)
      {
        lbText.Items.Add(ligne);
      }
      sr.Close();
    }
  }
}
```

Remarque:

Au lieu d'écrire:

```
StreamWriter sw = File.CreateText("donnees.txt");
```

ou

```
StreamReader sr = File.OpenText("donnees.txt");
```

Il est possible d'écrire directement:

```
StreamWriter sw = new StreamWriter ("donnees.txt");
```

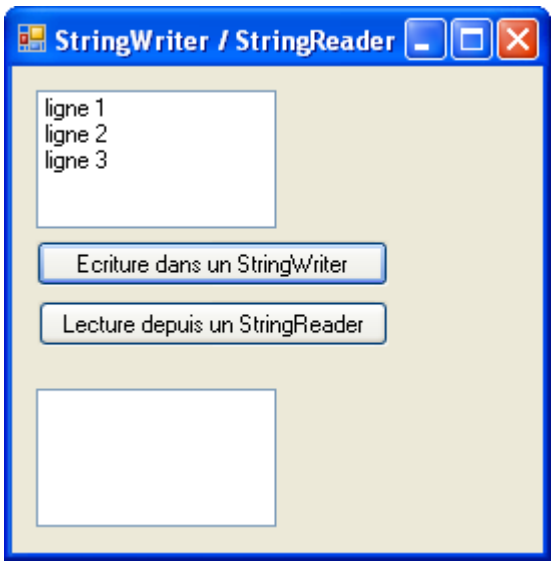
ou

```
StreamReader sr = new StreamReader ("donnees.txt");
```

Utilisation de StringWriter et StringReader

Ces objets permettent de traiter du texte en tant que Stream en mémoire plutôt que dans un fichier.

Voici un exemple d'utilisation de **StringWriter** et **StringReader**.




```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace StringWriter2
{
  public partial class Form1 : Form
  {
    StringWriter sw;
    StringReader sr;
```

```
        public Form1()
        {
          InitializeComponent();
        }

        private void btnEcrire_Click(object sender, EventArgs e)
        {
          sw = new StringWriter();
          foreach (string ligne in lbDepart.Items)
          {
            sw.WriteLine(ligne);
          }
          sw.Close();
        }

        private void btnLire_Click(object sender, EventArgs e)
        {
          sr = new StringReader(sw.ToString());
          string ligne = null;

          while ((ligne = sr.ReadLine()) != null)
          {
            lbArrivee.Items.Add(ligne);
          }
          sr.Close();
        }
      }
    }
```

Comme on peut le voir l'utilisation de **StringWriter / StringReader** est analogue à celle de **StreamWriter / StreamReader**. Aucun fichier n'est utilisé, tout se passe en mémoire.

Utilisation de BinaryWriter et BinaryReader

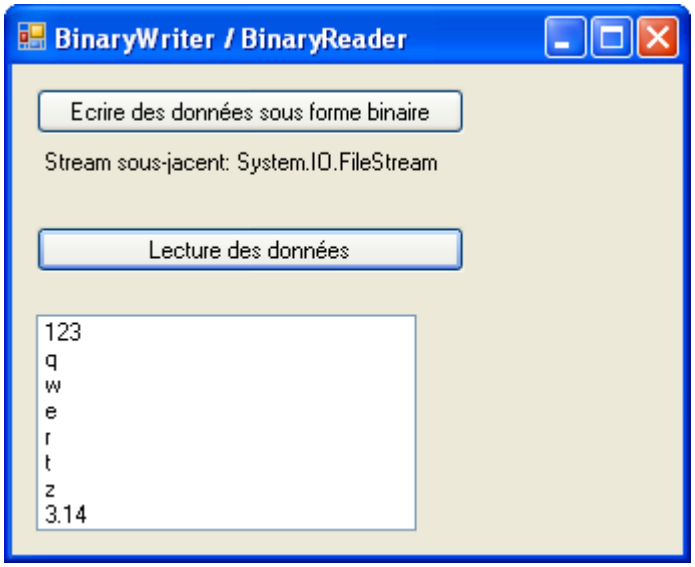
Ces deux classes dérivent directement de **System.Object**. Elles permettent de lire et d'écrire directement des données sous forme binaire dans le stream sous-jacent. Voici les principaux membres de **BinaryWriter**:

Membres	Description
BaseStream	Propriété en lecture seule donnant accès au stream utilisé par le **BinaryWriter**.
Close()	Ferme le **BinaryWriter**.
Flush()	Ecrit les données du buffer.
Seek()	Spécifie la position dans le stream courant.
Write()	Ecrit une donnée dans le stream courant.

Ainsi que ceux de **BinaryReader**:

Membres	Description
BaseStream	Propriété en lecture seule donnant accès au stream utilisé par le **BinaryReader**.
Close()	Ferme le **BinaryReader**.
PeekChar()	Retourne le prochain caractère disponible, sans modifier la position courante.
Read()	Lit un certain nombre de bytes ou de caractères et les stocke dans un tableau.
ReadXXX()	Lit la prochaine donnée depuis le stream courant (p.ex. **ReadBoolean()**, **ReadInt32()**, etc)

Le programme ci-dessous écrit des données sous forme binaire dans un fichier (un entier, un tableau de caractères et un double), puis les relit et les affiche pour vérification:



Le code est le suivant:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace BinaryWriter2
{
  public partial class Form1 : Form
  {
    int entier = 123;
```

```
    char[] tabChar = { 'q', 'w', 'e', 'r', 't', 'z' };
    double pi = 3.14;

    public Form1()
    {
      InitializeComponent();
    }

    private void btnEcrire_Click(object sender, EventArgs e)
    {
      // Ouverture d'un binarywriter pour un fichier
      FileInfo fichier = new FileInfo("binaire.dta");
      BinaryWriter bw = new BinaryWriter(fichier.OpenWrite());

      lStream.Text = "Stream sous-jacent: " + bw.BaseStream;

      // Ecriture de données
      bw.Write(entier);
      bw.Write(tabChar);
      bw.Write(pi);
      bw.Close();
    }

    private void btnLire_Click(object sender, EventArgs e)
    {
      // Ouverture d'un binaryreader
      FileInfo fichier = new FileInfo("binaire.dta");
      BinaryReader br = new BinaryReader(fichier.OpenRead());

      // Lecture d'un entier
      lbRes.Items.Add(br.ReadInt32());

      // Lecture du tableau de caractères
      for (int i=0; i<tabChar.Length; i++)
        lbRes.Items.Add(br.ReadChar());

      // Lecture d'un réel double
      lbRes.Items.Add(br.ReadDouble());
      br.Close();
    }
  }
}
```

Remarque:

Il est important de noter que le constructeur de **BinaryStream** accepte comme paramètre n'importe quel objet **Stream** (**FileStream**. **MemoryStream** etc). Il est donc possible d'écrire par exemple des données binaires en mémoire.