1 Module A.7 : Gestion des entrées sorties

Lors du développement des précédents modules, les différents programmes réalisaient des actions sans aucune entrée et fournissaient des sorties par l'intermédiaire de la console. Dans le monde réel, les programmes informatiques nécessitent des entrées et des sorties plus élaborées.

1.1 Console

La manière la plus simple d'échanger des données avec l'extérieur est la console. Cette façon d'interagir avec l'environnement à l'avantage de la simplicité, mais se révèle limité pour des entrées et sorties complexes. L'exemple ci-dessous présente un exemple d'utilisation de la console.

Exemple :

```
1 Console.WriteLine("Veuillez saisir une phrase et valider.");
2 string saisie = Console.ReadLine();  // Lire une phrase dans la console.
3 Console.WriteLine("Vous avez saisi : " + saisie);
```

La classe **Console** fournit des fonctions pour lire et écrire des éléments dans la console.

Méthodes	**Explications**
int Read ()	Lecture du caractère suivant du flux d'entrée standard.
string ReadLine ()	Lecture de la ligne de caractères suivante du flux d'entrée standard.
ConsoleKeyInfo ReadKey ()	Obtention du caractère suivant ou de la touche de fonction sur laquelle l'utilisateur a appuyé.
void Write (**string** value)	Écriture de la valeur de chaîne spécifiée dans le flux de sortie standard.
void WriteLine (**string** value)	Écriture de la ligne de caractères suivante du flux de sortie standard.
void Clear ()	Effacement de la mémoire tampon et de la fenêtre affichant la console.

1.2 Gestion des fichiers

Les fichiers sont les périphériques de stockage les plus courantes. **.NET** fournit l'objet **File** pour les manipuler avec simplicité. Il permet d'agir sur les fichiers à proprement dit (création, suppression...) et sur leur contenu (écriture, lecture...). Cette section aborde de manière succincte ces deux aspects.

Pour commencer, **File** fournit plusieurs méthodes courts-circuits afin d'écrire ou lire en une seule fois dans un fichier qui existe.

Ces méthodes sont présentées ci-dessous :

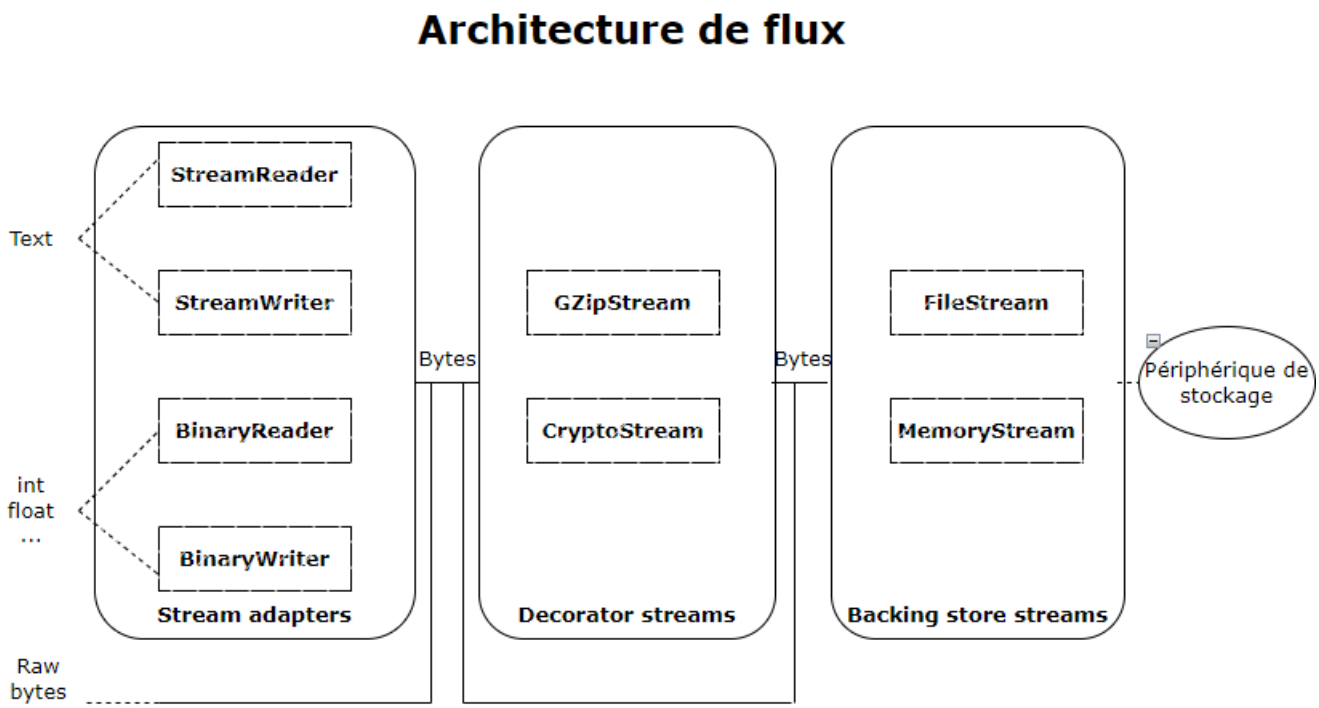
Méthodes	**Explications**
string[] ReadAllLines (**string** path)	Ouverture du fichier, lecture de toutes les lignes puis fermeture du fichier.
string ReadAllText (**string** path)	Ouverture du fichier, lecture de tout le texte dans une chaîne puis fermeture du fichier.
byte[] ReadAllBytes (**string** path)	Ouverture du fichier, lecture de tout le texte dans un tableau d'octets puis fermeture du fichier.
void WriteAllLines (**string** path, **string[]** contents)	Ouverture du fichier, écriture de toutes les lignes puis fermeture du fichier.
void WriteAllText (**string** path, **string** contents)	Ouverture du fichier, écriture de tout le texte depuis une chaîne puis fermeture du fichier.
void WriteAllBytes (**string** path, **byte[]** contents)	Ouverture du fichier, écriture de tout le texte depuis un tableau d'octets puis fermeture du fichier.

Le tableau suivant illustre plusieurs méthodes d'action sur des fichiers.

Méthodes	**Explications**
FileStream Create (**string** path)	Création ou remplacement du fichier spécifié. Renvoie d'un flux sur ce fichier (voir section suivante).
bool Exists (**string** path)	Est-ce que le fichier spécifié existe ?
void Delete (**string** path)	Suppression du fichier spécifié.
void Copy (**string** sourceName, **string** destName)	Copie du fichier source vers l'emplacement de destination.
void Move (**string** sourceName, **string** destName)	Déplacement du fichier source vers l'emplacement de destination.

1.3 Architecture de flux

Dans la plupart des contextes, un programme interagit avec son environnement en récupérant des informations en entrée et écrivant les résultats de sa tâche en sortie. **.NET** définit *une architecture de flux* afin de réaliser les différentes opérations I/O. Cette section introduit ces principaux concepts et en présente plusieurs utilisations. Il est conseillé de s'appuyer sur la documentation Microsoft pour des usages plus avancés.



L'architecture de flux repose sur la notion fondamentale de **flux**. Un **flux** présente une vue sur une séquence d'octets sur laquelle il peut agir (utilisation de tableaux de **byte** en entrée et sortie).
L'architecture de flux introduit trois concepts : les **magasins de stockage**, les **décorateurs** et les **adaptateurs** de flux.

Un **magasin de stockage** est un flux point final qui communique avec le périphérique de stockage. Celui-ci est soit une source avec des octets qui vont être lus de manière séquentielle ou une destination avec des octets qui vont être écrits de manière séquentielle. Le **magasin de stockage** sélectionné renvoie au type de périphérique de stockage (par exemple : **FileStream** pour un fichier). En général, ce type de flux n'est pas exposé par le développeur. D'une manière générale, dans les applications, les flux de ce type ne sont pas exposés par le développeur.

Un **décorateur** est un flux qui transforme les données fournies en entrée afin de réaliser une opération particulière. Le type de **décorateur** dépend de la transformation souhaitée (par exemple : **GZipStream** pour la compression/décompression ou **CryptoStream** pour le chiffrement/déchiffrement). Un **décorateur** est connecté à un autre **décorateur** ou directement à un **magasin de stockage**.

Les **magasins de stockage** et les **décorateurs** sont des flux et n'agissent que sur des séquences d'octets. Généralement, les programmes travaillent sur des types de données haut niveau comme des chaînes de caractères. Les **adaptateurs** remplissent cette tâche et se connectent à la sortie d'une file d'attente de flux. Le choix d'adaptateur dépend des actions à réaliser et du type haut niveau des données de travail (par exemple : **StreamReader** / **StreamWriter** pour lire/écrire du texte).

1.3.1 Fondamentaux sur Stream

Stream est le type de base pour les flux. Il déclare les méthodes et propriétés permettant de réaliser la lecture, l'écriture et la recherche d'éléments, ainsi que les tâches administratives comme la fermeture, libération des ressources et configurations des délais d'attente du flux.

Le tableau suivant fournit une série de méthodes et propriétés associées à l'ensemble des types de flux hérités de **Stream**.

Méthodes / Propriétés	**Explications**
int Read (byte[] buffer, int offset, int count)	Lecture d'un bloc de données comme un tableau de bytes.
bool CanRead	Le flux prend-il en charge la lecture ?
void Write (byte[] buffer, int offset, int count)	Écriture d'un bloc de données à partir du tableau de bytes.
bool CanWrite	Le flux prend-il en charge l'écriture ?
long Seek (long offset, **SeekOrigin** origin)	Modification de la position au sein du flux actuel. **SeekOrigin** définit le point de référence.
bool CanSeek	Le flux prend-il en charge la recherche ?
long Position / Length	Position actuelle / longueur du flux
int ReadTimeout / WriteTimeout ()	Délai d'attente en lecture / écriture.
bool CanTimeout	Le flux d'attente peut-il dépasser le délai d'attente ?
void Dispose ()	Libération de l'ensemble des ressources utilisées par le flux.
void Close ()	Fermeture du flux.

1.3.2 Magasins de stockage : FileStream

Afin d'introduire le fonctionnement et l'utilisation des **magasins de stockage**, cette section présente le type **FileStream** qui est un flux associé aux fichiers. Pour construire une instance de type **FileStream**, il existe deux manière différentes : utiliser **File** ou l'instancier directement.

Le type **File** fournit une série de méthodes utiles pour traiter des fichiers. L'exemple suivant montre trois utilisations différentes de ces méthodes pour créer des instances de **FileStream** :

```
1 FileStream fs1 = File.OpenRead ("readme.bin");         // Lecture seulement.
2 FileStream fs2 = File.OpenWrite ("writeme.txt")        // Ecriture seulement.
3 FileStream fs3 = File.Create (@"c:\dossier\file.txt") // Lecture/Ecriture.
```

L'exemple suivant illustre une instanciation directe à l'aide d'un nom de fichier et un mode de création (ici récupération d'un fichier préexistant).

La documentation Microsoft présente les différentes variantes de création directe.

```
// Lecture/Ecriture.
FileStream fs = new FileStream ("readwrite.tmp", FileMode.Open);
```

Ci-dessous, nous présentons une utilisation du flux terminal (type **FileStream**) qui permet de directement écrire et lire dans un fichier des bytes.

Exemple :

```
// Création du fichier appelé text.txt dans le répertoire courant.
using (Stream s = new FileStream ("text.txt", FileMode.Create))
{
  Console.WriteLine (s.CanRead);    // True
  Console.WriteLine (s.CanWrite);   // True
  Console.WriteLine (s.CanSeek);    // True

  s.WriteByte (101);                          // 101
  s.WriteByte (102);                          // 101 102
  byte[] block = { 1, 2, 3, 4, 5 };
  s.Write (block, 0, block.Length); // 101 102 1 2 3 4 5

  Console.WriteLine (s.Length);     // 7
  Console.WriteLine (s.Position);   // 7
  s.Position = 0;                   // On retourne au début.

  Console.WriteLine (s.ReadByte()); // 101, Position : 1
  Console.WriteLine (s.ReadByte()); // 102, Position : 2

  Console.WriteLine (s.Read (block, 0, block.Length)); // 5, Position 7
  // 5 étant le nombre de bytes chargés.
  Console.WriteLine (s.Read (block, 0, block.Length)); // 0
  // 0 car fin du fichier.
}
```

1.3.3 Adaptateurs : StreamReader/StreamWriter

Afin d'introduire le fonctionnement et l'utilisation des **adaptateurs**, cette section présente les types **StreamReader** et **StreamWriter** qui travaillent en lecture et écriture à partir de chaînes de caractères. Pour construire une instance d'un de ces types, on peut partir d'un flux (**magasin de données** ou **décorateur**) ou utiliser l'une des méthodes de **File**.

L'exemple suivant présente plusieurs instanciations :

```
// Instanciation avec FileStream
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
{
  // bloc utilisation de writer
}
// Utilisation de File
```

```
8  using (TextWriter   wrt = File.AppendText("text.txt")) {}
9  using (TextReader reader = File.OpenText ("text.txt")) {}
```

Les tableaux suivants présentent les principales méthodes utilisables avec une instance de **StreamReader** ou **StreamWriter** respectivement.

StreamReader

Méthodes	Explications
int Read ()	Lecture du prochain caractère dans la chaîne d'entrée.
int Read (char[] buffer, int index, int count)	Lecture du bloc de données comme un tableau de caractères.
string ReadLine ()	Lecture d'une ligne de caractères à partir de l'entrée et retour des données sous forme d'une chaîne
void Close ()	Fermeture du flux.
void Dispose ()	Libération de l'ensemble des ressources utilisées par le flux.

StreamWriter

Méthodes	Explications
void Write (**string** format, params object[] arg)	Écriture de données en argument dans la sortie.
string WriteLine (**string** value)	Écriture une chaîne suivie d'une marque de fin de ligne dans le flux de texte en sortie.
void Close ()	Fermeture du flux.
void Dispose ()	Libération de l'ensemble des ressources utilisées par le flux.

Le code suivant présente l'écriture de deux lignes dans le fichier *text.txt* et la lecture de ces dernières.

Exemples :

```
1  using (FileStream fs = File.Create ("test.txt"))
2  using (StreamWriter writer = new StreamWriter (fs))
3  {
4    writer.WriteLine ("Line1");
5    writer.WriteLine ("Line2");
6  }
7
8  using (FileStream fs = File.OpenRead ("test.txt"))
9  using (StreamReader reader = new StreamReader (fs))
10 {
11   Console.WriteLine (reader.ReadLine()); // Line1
12   Console.WriteLine (reader.ReadLine()); // Line2
13 }
```

1.3.4 Décorateurs : DeflateStream

Il existe de nombreux décorateurs, chacun correspondant à un usage particulier. Ce module présente les décorateurs associés à la compression et la décompression selon l'algorithme derrière le format *ZIP* à savoir **DeflateStream**. En connectant la file de flux, toutes les données écrites sont compressées et celles lues sont décompressées.

L'exemple suivant présente l'écriture et la lecture dans un fichier fichier *compressed.bin* au format compressé à l'aide de **DeflateStream**.

Exemple :

```
1  // Création et écriture dans le fichier
2  using (Stream s = File.Create ("compressed.bin"))
3  using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
4  {
5    for (byte i = 0; i < 100; i++)
6      ds.WriteByte (i);
7  }
8
9  // Ouverture et lecture dans le fichier
10 using (Stream s = File.OpenRead ("compressed.bin"))
11 using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
12 {
13   for (byte i = 0; i < 100; i++)
14     Console.WriteLine (ds.ReadByte());
15 }
```