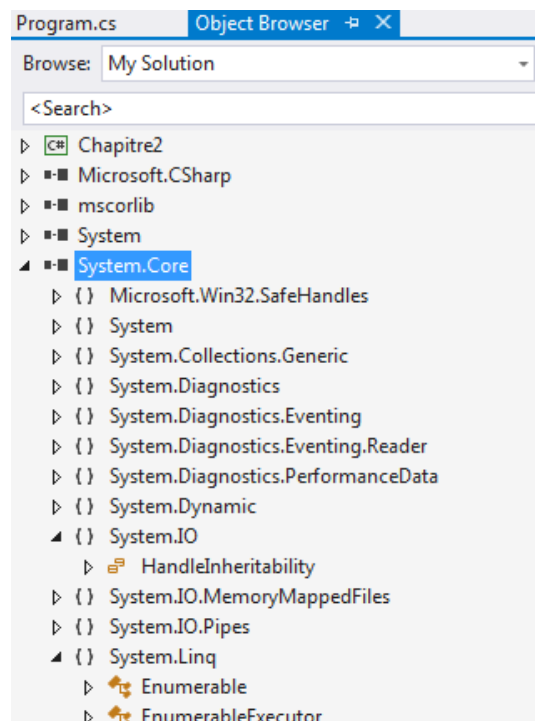


Vous devriez voir ceci :



Et ainsi pouvoir observer que la DLL `System.Core` définit différents types non seulement dans `System.Core`, mais aussi dans `System.IO`, `System.Threading`, `System.Linq`, `System.Collections`, etc.

C'est donc dire que **chaque type doit être défini dans un espace de noms particulier**, dont il devient un membre.

Un avantage considérable avec une telle approche est que tout langage .NET (C#, VB.NET, C++, F#, ou tout autre langage compatible .NET) utilisent exactement les mêmes espaces de noms et les mêmes types, ce qui rend le développement multi-langagiers vraiment plus attractif.

2.1.2 Principaux espaces de noms

Un des objectifs principaux lorsqu'on veut programmer en .NET devient donc d'avoir une idée assez bien définie des nombreux types contenus dans le FCL, et surtout dans quel espace de noms on peut les retrouver (et/ou les *assemblies* qui les contiennent).

L'espace de noms le plus fondamental est certainement **System** et tous ses sous-espaces de noms. Il contient la plupart des types centraux (lire « nécessaire ») à tout développement d'applications .NET. Entre autres, les types de données de base y sont définis.

Voici un tableau qui regroupe pour vous des espaces de noms contenus dans *System* parmi les plus importants du framework, avec une description de ce qu'ils contiennent :

Espace de noms .NET	Description
System	System contient des types utiles pour travailler avec les données, faire des calculs mathématiques (System.Math), générer des nombres aléatoires (System.Random), gérer des variables d'environnement (System.Environment), gérer le ramasseurs de miettes et un bon nombre d'exceptions.
System.Collections System.Collections.Generic	Ces espaces de noms définissent la plupart des types conteneurs importants, ainsi que les types de base et interfaces permettant de construire vos propres collections personnalisées, génériques ou non.
System.Data System.Data.Common System.Data.EntityClient System.Data.SqlClient	Ces espaces de noms sont utilisés pour interagir avec les bases de données relationnelles via ADO.NET.
System.IO System.IO.Compression System.IO.Ports	Ces espaces de noms définissent de nombreux types utilisés pour travailler avec l'écriture et la lecture de fichiers, la compression de données et la manipulation de ports.
System.Reflection	Ces espaces de noms définissent des types supportant la création dynamique de type et

Espace de noms .NET	Description
<code>System.Reflection.Emit</code>	la découverte de types à l'exécution. Nous y reviendrons lorsque nous discuterons du concept de réflexion en POO.
<code>System.Runtime.InteropServices</code>	Cet espace de noms fournit le nécessaire aux types .NET pour interagir avec du code non géré (c'est-à-dire provenant de composants développés avant l'ère .NET).
<code>System.Drawing</code> <code>System.Windows.Forms</code>	Ces espaces de noms définissent des types utilisés pour construire des applications de bureau (interfaces utilisateurs) en utilisant les <i>Windows Forms</i> . La tendance plus récente est plutôt d'utiliser WPF (Windows Présentation Foundation).
<code>System.Windows</code> <code>System.Windows.Controls</code> <code>System.Windows.Shapes</code>	<code>System.Windows</code> et tous ses sous-espaces a pour objectif de définir tous les types nécessaires au développement d'application WPF.
<code>System.Linq</code> <code>System.Xml.Linq</code> <code>System.Data.DataSetExtensions</code>	Ces espaces de noms définissent les types utilisées pour travailler avec LINQ, qui est un API visant à manipuler/interroger/comparer/filtrer/trier des collections de données de sources/natures diverses entre elles
<code>System.Web</code>	Cet espace de noms est lié à la construction d'application web ASP.NET.
<code>System.Threading</code> <code>System.Threading.Tasks</code>	Ces espaces de noms définissent de nombreux types nécessaires à la construction d'applications multi-threads (c'est-à-dire dans lesquels on désire distribuer la charge de travail/calcul à travers plusieurs unités de traitements).
<code>System.Security</code>	Cet espace de noms contient des types en lien avec les autorisations et la cryptographie, entre autres.
<code>System.Xml</code>	Cet espace de nom contient les types utilisés pour interagir/gérer des données/documents XML.

Note :

Un autre espace de noms, second en importance, nommé `Microsoft`, contient des types utilisés pour interagir avec des services uniques avec des systèmes d'exploitation Windows. Au courant de la session, nous n'utiliserons pas intensément des types issus de cet espace de noms, mais si nous le faisons, ils seront présentés en bonne et due forme.

Si nécessaire, vous pouvez toujours consulter la documentation `.NET Framework 4.5 SDK` (et les versions plus récentes) dans laquelle tous les détails pertinents peuvent être retrouvés.

2.1.3 Using / alias

À partir d'un nouveau projet – et en fonction du type de projet choisi – vous aurez donc un certain nombre d'assemblies référés par défaut, qui donneront accès à un certain nombre de types distribués à travers différents espaces et sous-espaces de noms. Vous avez donc accès à un sous-ensemble du FCL.

Si vous créez un nouveau projet de type application console et que vous ouvrez le fichier `Program.cs`, vous devriez voir le code suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

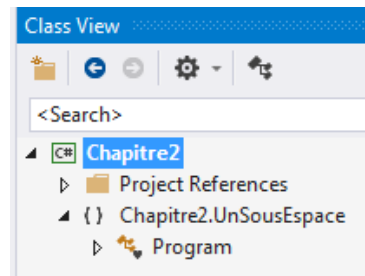
namespace Chapitre2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Deux remarques s'imposent :

- Observez que la classe `Program` est défini, par défaut, à l'intérieur de l'espace de nom `Chapitre2`, qui est le nom du projet/de la solution.
- Rien ne vous empêche de modifier cet espace de nom; par exemple :

```
namespace Chapitre2.UnSousEspace
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Dans ce cas, dans le visualisateur de classe, la nouvelle arborescence sera observable :



- Sachez qu'un type doit être unique pour un espace de nom particulier, mais que plusieurs types avec un identificateur identique peuvent coexister dans des espaces de noms différents; par exemple, **ceci est possible** :

```
namespace Chapitre2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

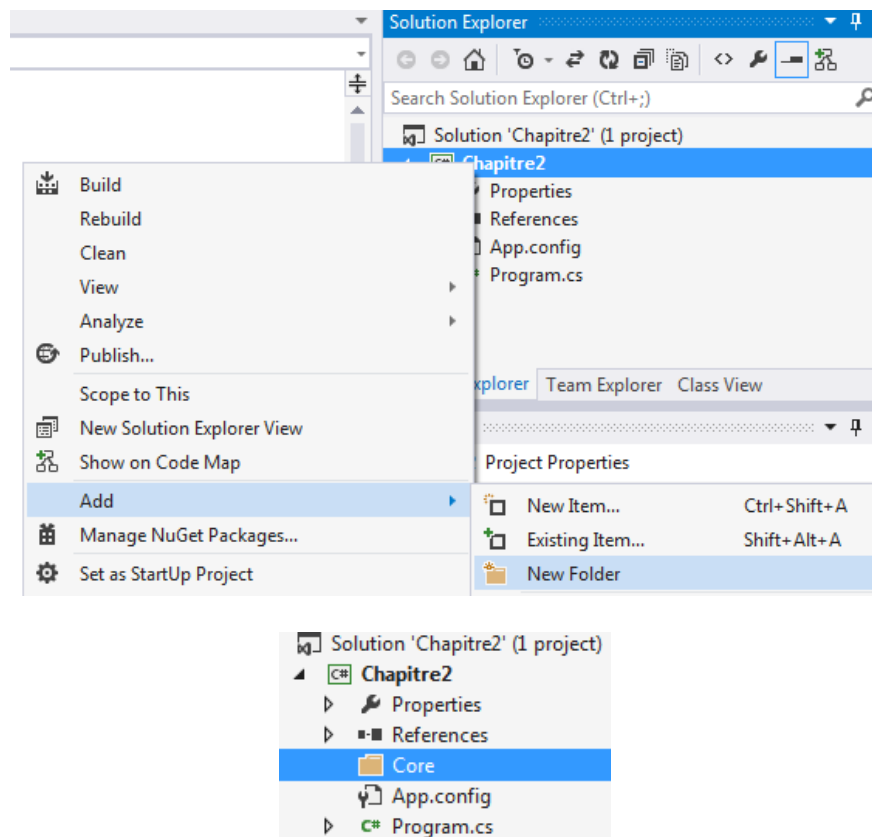
namespace Chapitre2.UnSousEspaceQuelconque
{
    class Program
    {
    }
}
```

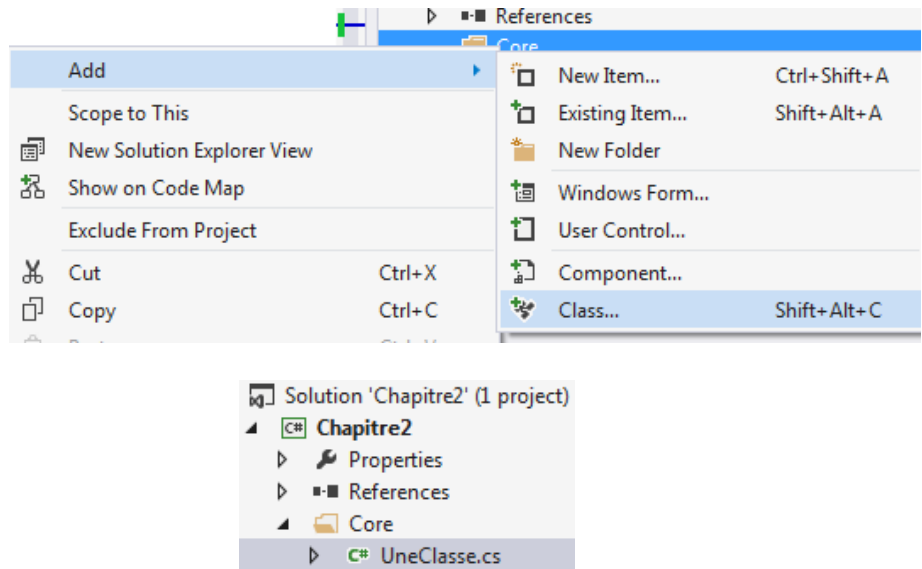
Mais **pas ceci** :

```
namespace Chapitre2
{
    class Program
    {
        static void Main(string[] args)
```

```
    {  
    }  
}  
  
namespace Chapitre2  
{  
    class Program  
    {  
    }  
}
```

- Vous pouvez aussi créer des dossiers dans votre projet; dans ce cas, l'espace de noms attribué par défaut sera ajusté en fonction du nom du dossier; par exemple,





Cela générera le code par défaut suivant :

```
namespace Chapitre2.Core
{
    class UneClasse
    {
    }
}
```

- On retrouve en début de fichier un certain nombre d'instructions `using (...)` :
 - Contrairement à ce qui pourrait être intuitif, l'espace de nom décrit suite à `using` n'agit pas comme « permission » de pouvoir utiliser des types définis dans l'espace de noms en question, car comme nous l'avons mentionné, cette vue sur les types référencés est acquise dès le référencement de l'assembly contenant au projet;
 - Plutôt, il s'agit de donner des chemins d'espace de noms implicites afin d'éviter, lorsque vient le temps de coder, d'avoir à déballer toute la hiérarchie de l'espace de noms pour utiliser un type donné. On peut donc parler en quelque sorte de « raccourci ». Par exemple :

Avec l'instruction `using System;`, on peut avoir la ligne de code suivante, pour lequel le chemin du type est résolu implicitement grâce à l'instruction `using` :

```
Int32 unType;
```

Sans cette instruction, on aurait une erreur de compilation à moins d'écrire le chemin explicitement comme ceci :

```
System.Int32 unType;
```

- L'instruction `using` peut également faire appel à la notion d'alias. Si on le souhaite, on peut associer un alias avec un chemin, qu'on utilisera à chaque fois pour faire référence à un espace de noms en particulier. Par exemple, on pourrait avoir le code suivant :

```
using m = System;
using colgen = System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapitre2.Core
{
    class UneClasse
    {
        m.Int32 unType;
        colgen.List<m.Int32> uneListe;
    }
}
```

`m` agit à titre d'alias pour l'espace de nom `System`, et `colgen` pour `System.Collections.Generic`. Les types déclarés à l'intérieur de `UneClasse` doivent alors utiliser ces alias pour faire appel à des types qui en font partis.

- L'intérêt des alias (dans ce contexte) est de résoudre des situations où un développeur devrait utiliser plusieurs types de mêmes noms mais provenant d'espaces de noms différents. L'utilisation d'alias deviendrait alors mandatoire, à moins de donner à chaque fois les chemins complets vers chacun des types afin de les distinguer.

2.2 ANATOMIE GÉNÉRALE D'UNE APPLICATION C# SIMPLE

2.2.1 Méthode *Main()*

C# demande que toute la logique d'un programme soit contenu à l'intérieur de définition de types.

Contrairement à certains langages, il n'est pas possible en C# de créer des fonctions globales ou d'utiliser des données globales. Plutôt, toutes les membres de données et toutes les méthodes doivent être contenues à l'intérieur de la définition du type.

Revoyons ensemble le code généré par défaut lorsqu'on crée une application console afin de décrire davantage les différentes instructions :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapitre2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

L'anatomie générale d'un fichier de code de projet aura le patron suivant :

```
<instructions usings>

namespace UnEspaceDeNom
{
    class UneClasse
    {
        <champs de la classe>

        UnMembre(comme une méthode)
        {
            <instructions>
        }

        UnAutreMembre
        {
            <instructions>
        }

        ...
    }

    class UneAutreClasse
    {
        ...
    }

    ...
}
```

```
namespace UnAutreEspaceDeNom
{
    ...
}
```

Typiquement, on définira un type différent par fichier de code.

Par défaut, la classe `Program` sera ajoutée au projet, contenant une méthode nommée `Main()`.

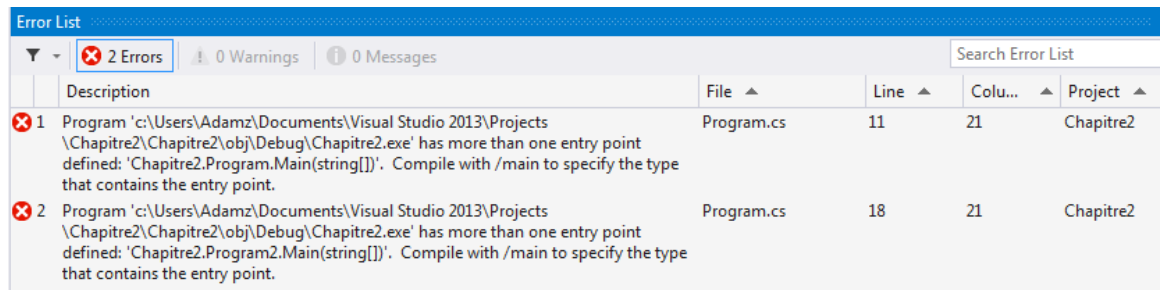
La suite de l'explication est **fondamentale** :

- Toute application C# exécutable (application de bureau WPF ou Windows Forms, de console, ou de service) doit contenir une classe définissant une méthode `Main()` (le « M » majuscule est important, car C# est sensible à la case), qui constitue le **point d'entrée dans l'application**;
- Formellement, la classe définissant la méthode `Main()` est appelée « **objet d'application** » ou « **objet de démarrage** »;
- Bien qu'il soit possible pour une application d'avoir plusieurs objets d'applications (par exemple pour exécuter différents tests unitaires), le compilateur doit être informé quel objet de démarrage vous désirez sélectionner comme point d'entrée pour l'exécution donnée. Par exemple, si vous écrivez ceci :

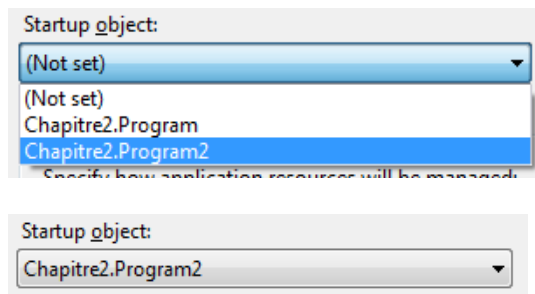
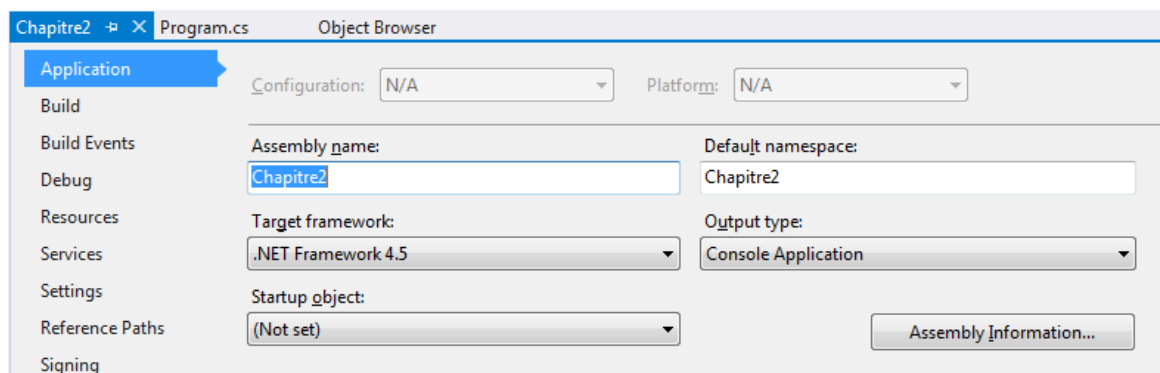
```
namespace Chapitre2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }

    class Program2
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Vous aurez droit à une erreur de compilation :



Ainsi, dans le cas où vous auriez plusieurs objets potentiels de démarrage dans votre projet, vous devez aller dans les propriétés de votre projet dans l'onglet par défaut, et sélectionner un objet de démarrage en particulier :



Suite à quoi le programme compilera normalement.

- Notez également que la signature d'une méthode `Main()` doit être décorée du mot clé **static**. Nous rediscuterons de ce terme plus en profondeur dans les prochains chapitres, mais pour l'instant il suffit de comprendre qu'un membre statique peut être invoqué sans avoir à créer d'objets;
- Par ailleurs, on peut remarquer que la méthode `Main()` possède un paramètre, `string[] args`, qui est un tableau de chaînes de caractères, qui contient un nombre quelconque (de 0 à n) d'arguments en ligne de commande. Nous y reviendrons à la section 2.2.2;

- Enfin, la méthode est définie de façon à ne retourner aucune valeur (`void`), et on n'a donc pas à indiquer de valeur de retour avec l'instruction `return`.

Bien que la méthode générée par défaut ait cette forme, il existe toutefois d'autres formes possibles de méthodes `Main()`. Voici différentes possibilités acceptées en C# :

```
class Program
{
    static void Main(string[] args)
    {
    }
}

class Program2
{
    static void Main()
    {
    }
}

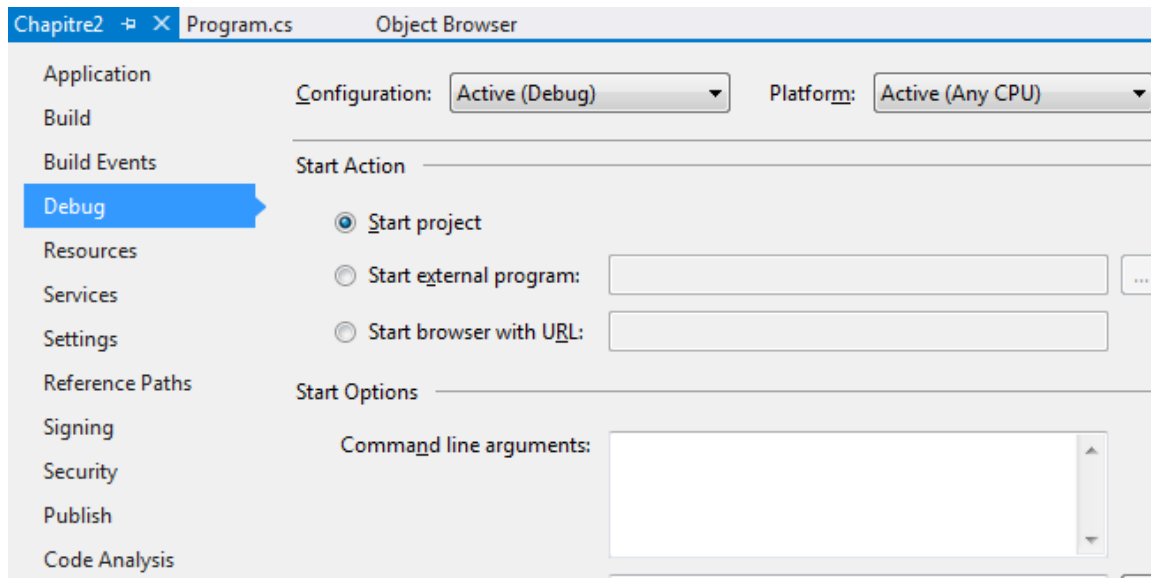
class Program3
{
    static int Main(string[] args)
    {
        return 0;
    }
}

class Program4
{
    static int Main()
    {
        return 0;
    }
}
```

Évidemment, si vous avez une valeur de retour, vous devrez vous assurer que tous les chemins d'exécution possible puissent retourner une valeur du type défini (comme pour toute méthode en général, d'ailleurs).

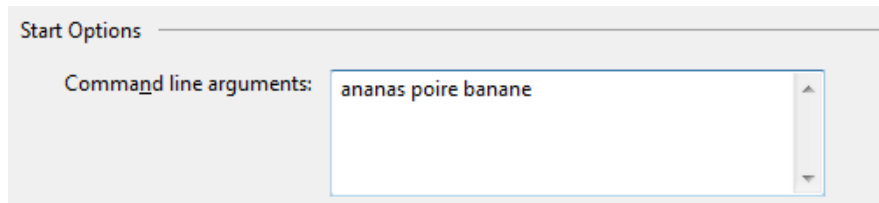
2.2.2 Traitement d'arguments en ligne de commande

Outre l'exécution de l'application via l'invite de commandes, vous pouvez spécifier des arguments de lignes de commandes via les propriétés du projet, dans l'onglet « *Debug* » :



Les arguments donnés dans cette boîte seront automatiquement passés à la méthode `Main()` lors du débogage ou de l'exécution de l'application à l'intérieur de Visual Studio. Vous en conviendrez, cela est beaucoup plus pratique que de devoir utiliser l'invite de commande.

Par exemple, si vous ajoutez ceci comme arguments :

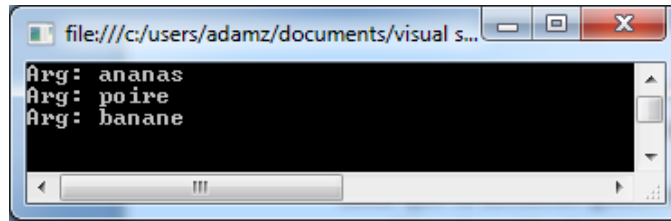


Et le code suivant :

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i < args.Length; i++)
            Console.WriteLine("Arg: {0}", args[i]);

        Console.ReadLine();
    }
}
```

Ne vous préoccupez pas trop du code pour l'instant, excepté du fait qu'on parcourt un à un les éléments du tableau de chaînes de caractères `args` et qu'on affiche à la console chacun de ces éléments. Vous obtiendrez :



Il existe d'autres alternatives pour parcourir un à un tous les arguments en utilisant d'autres structures de contrôles similaires (que nous verrons plus en profondeur un peu plus loin dans ce chapitre). Par exemple, on peut utiliser la boucle `foreach` comme ceci :

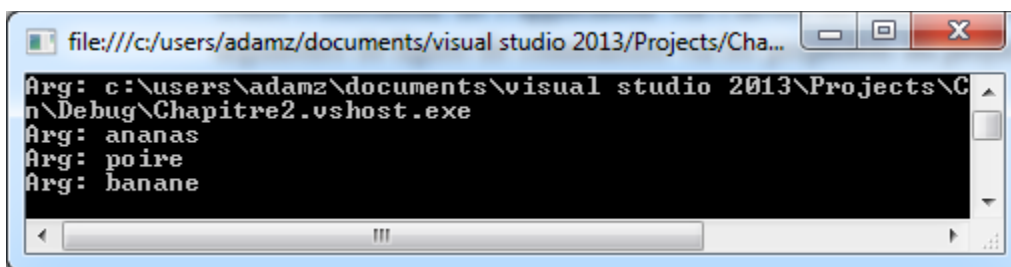
```
foreach (string arg in args)
    Console.WriteLine("Arg: {0}", arg);
```

Enfin, vous pouvez accéder aux arguments de ligne de commande en utilisant la méthode statique `GetCommandLineArgs` défini à l'intérieur du type `System.Environment`, dont la valeur de retour est un tableau de chaînes de caractères dans lequel le premier index identifie le nom de l'application et la suite la liste des arguments. Par exemple :

```
static void Main(string[] args)
{
    foreach (string arg in Environment.GetCommandLineArgs())
        Console.WriteLine("Arg: {0}", arg);

    Console.ReadLine();
}
```

Donnera ceci à l'exécution (remarquez le premier argument) :



En utilisant cette approche, vous n'avez pas besoin d'utiliser `string[] args` comme argument dans votre méthode. C'est d'ailleurs le principal intérêt. Par exemple, vous pourriez ainsi avoir ceci :

```
static int Main()
{
    foreach (string arg in Environment.GetCommandLineArgs())
        Console.WriteLine("Arg: {0}", arg);
}
```

```
        Console.ReadLine();

        return -1;
    }
}
```

2.3 UTILISATION DE SYSTEM.ENVIRONMENT

Mis à part la méthode `GetCommandLineArgs()`, la classe `Environment` propose un certain nombre de membres très intéressants pour vos applications .NET.

Par exemple, grâce à certaines méthodes et propriétés, vous pouvez avoir accès à un certain nombre de détails en ce qui concerne le système d'exploitation qui héberge l'application .NET en cours d'exécution. Les membres et leurs descriptions sont donnés dans le tableau suivant :

Méthode/Propriété	Description
<code>GetLogicalDrives()</code>	Retourne sous forme de tableau de chaînes de caractères l'ensemble des unités logiques.
<code>OSVersion</code>	Retourne la description complète du système d'exploitation hébergeant l'application en cours.
<code>ProcessorCount</code>	Retourne le nombre de cœurs de la machine.
<code>Is64BitOperatingSystem</code>	Retourne vrai ou faux selon si la machine hébergeante roule sur une version 64 bits du système d'exploitation ou non.
<code>SystemDirectory</code>	Retourne le chemin complet au répertoire du système.
<code>MachineName</code> <code>UserName</code>	Retourne le nom de la machine ou de l'utilisateur qui a démarré l'application.
<code>Version</code>	Retourne un objet de type <code>Version</code> qui représente la version de la plateforme .NET d'exécution de l'application.

Par exemple, vous pourriez avoir le code suivant :

```
class Program
{
```

```
static int Main()
{
    Console.WriteLine(Environment.MachineName + "(sous l'utilisateur " +
        Environment.UserName + ")");

    Console.WriteLine(Environment.OSVersion +
        (Environment.Is64BitOperatingSystem ? "(64 bits)" : "(32 bits)") + ",
        " + Environment.ProcessorCount + " coeurs");

    Console.WriteLine("Répertoire système: {0}",
        Environment.SystemDirectory);

    Console.WriteLine("Version de .NET: {0}", Environment.Version);

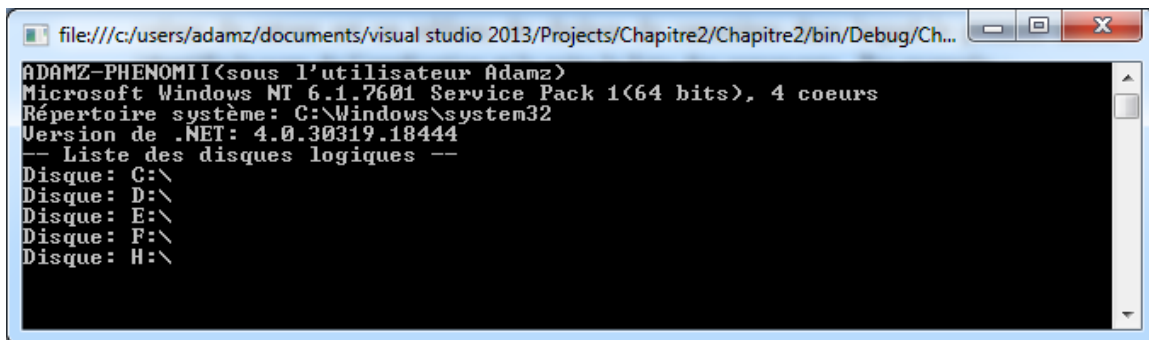
    Console.WriteLine("-- Liste des disques logiques --");

    foreach (string drive in Environment.GetLogicalDrives())
        Console.WriteLine("Disque: {0}", drive);

    Console.ReadLine();

    return -1;
}
```

Qui, à l'exécution, afficherait ceci :



The screenshot shows a Windows command prompt window with the following output:

```
file:///c:/users/adamz/documents/visual studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/Ch...
ADAMZ-PHENOMII(sous l'utilisateur Adamz)
Microsoft Windows NT 6.1.7601 Service Pack 1(64 bits), 4 coeurs
Répertoire système: C:\Windows\system32
Version de .NET: 4.0.30319.18444
-- Liste des disques logiques --
Disque: C:\
Disque: D:\
Disque: E:\
Disque: F:\
Disque: H:\
```

2.4 UTILISATION DE SYSTEM.CONSOLE

2.4.1 Entrées et sorties en console

La plupart des exemples et des laboratoires que nous ferons ensemble au courant de la session utiliseront la console. Il importe donc d'en décrire les principaux membres afin d'en faire une utilisation adéquate.

La raison pour laquelle nous nous attarderons peu à l'utilisation des Windows Forms ou même de WPF est que le focus du cours est davantage centré sur la syntaxe de C# que sur

les aspects interfaces utilisateurs, pour lesquels des cours comme INF1003 et INF1013 sont davantage le créneau.

La classe `Console` en C# remplit naturellement ce rôle en possédant des membres permettant de lire, d'écrire et de manipuler des flux d'erreurs pour des applications de type console.

Le tableau suivant présente les membres les plus importants (il y en a d'autres) avec lesquels vous pourriez être appelés à travailler au courant de la session :

Méthode/Propriété	Description
<code>Beep()</code>	Force l'émission d'un son d'une fréquence et longueur spécifiées en paramètres.
<code>BackgroundColor</code> <code>ForegroundColor</code>	Spécifie les couleurs d'avant et d'arrière-plan de la console/des sorties, en spécifiant une couleur membre de l'énumération <code>ConsoleColor</code> .
<code>BufferHeight</code> <code>BufferWidth</code>	Contrôle la hauteur et la largeur de la zone de buffer de la console.
<code>Title</code>	Spécifie le titre de la fenêtre de console.
<code>WindowHeight</code> <code>WindowWidth</code> <code>WindowTop</code> <code>WindowLeft</code>	Contrôle les dimensions (en nombre de lignes ou de caractères par ligne) et la position de la console en fonction du buffer établi.
<code>Clear()</code>	Efface le buffer et la zone d'affichage de la console.
<code>Write()</code> <code>WriteLine()</code>	Écrit en sortie sur la ligne courante ou sur une nouvelle ligne.
<code>Read()</code> <code>ReadLine()</code>	Lit et retourne le prochain caractère tapé ou une chaîne de caractères (jusqu'à ce que l'utilisateur appuie sur « <i>enter</i> »).

Voyons ensemble quelques exemples d'utilisation de `System.Console` pour faire la démonstration de ces membres.

Le code suivant :

```
class Program
{
    static int Main()
    {
        Console.Title = "Mon application console";
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.DarkRed;

        Console.WriteLine("### Mon application console ###");
        Console.Write("--"); //se rajoute comme nouvelle ligne puisque c'est
le 1er .Write après un .WriteLine
        Console.Write("-"); // rajoute un 3e -
        Console.WriteLine("##"); // ajoute à la fin de la dernière ligne; la
prochaine instruction (.Write ou .WriteLine) sera une nouvelle ligne.
        Console.WriteLine("---");

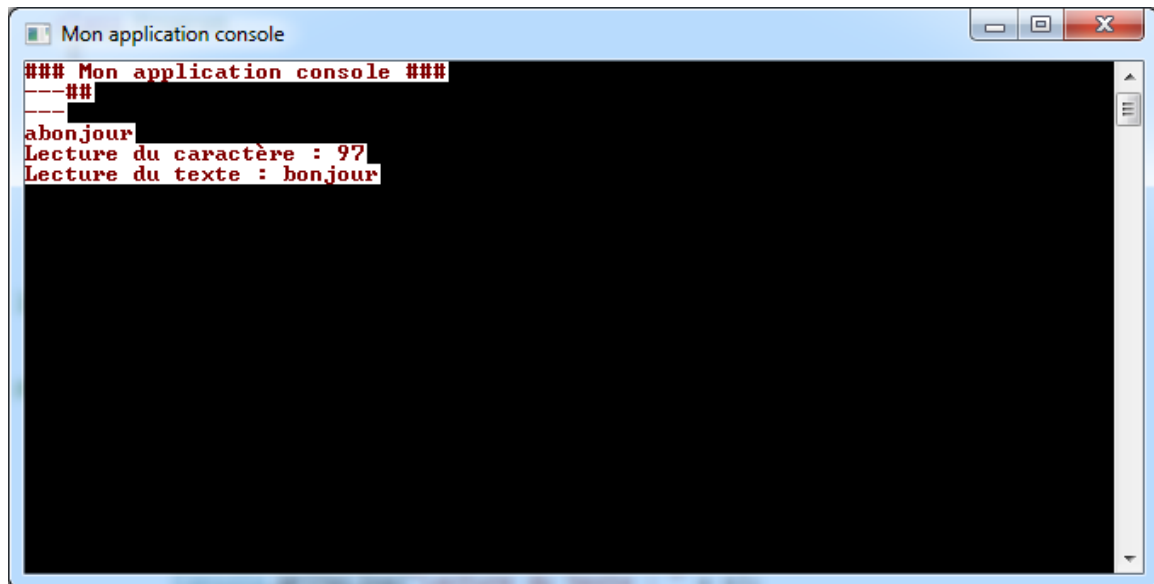
        string c = Console.Read().ToString();
        string s = Console.ReadLine();

        Console.WriteLine("Lecture du caractère : " + c);
        Console.WriteLine("Lecture du texte : " + s);

        Console.ReadLine();

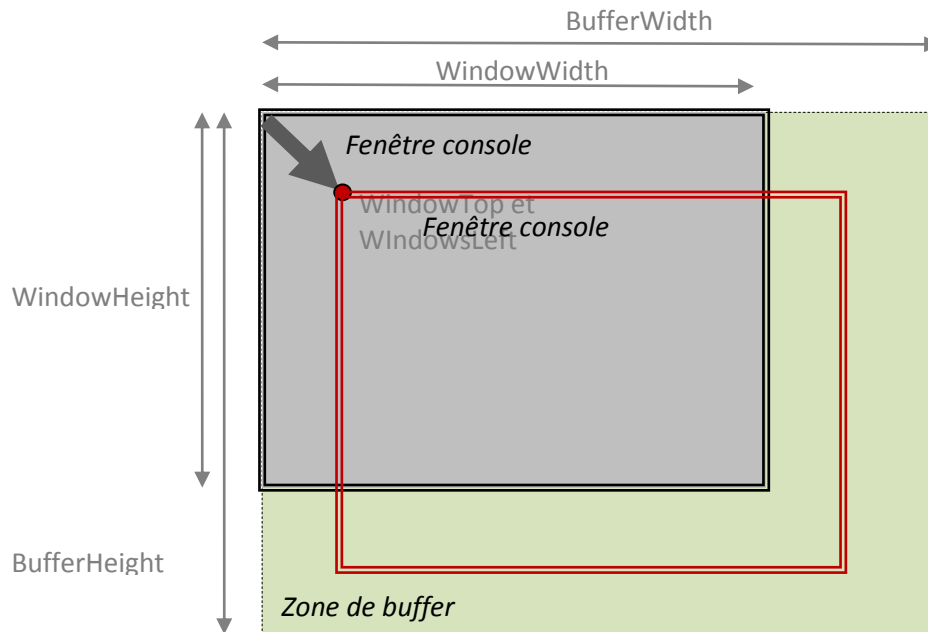
        return -1;
    }
}
```

Retournera ceci :



Remarquez que la lecture du caractère « a » retourne le code ASCII 97 (un entier). Il existe des façons de reconvertir en un véritable caractère.

Les propriétés de dimensionnement `xxxBuffer` et `xxxWindow` sont quelque peu difficiles à bien cerner. Sachant que la hauteur (`Height`) est exprimée en nombres de lignes et la largeur (`Width`) en nombres de colonnes (ou caractères), voici un schéma qui illustre comment cela fonctionne :



- `WindowWidth` et `WindowHeight` représentent respectivement la largeur et la hauteur de la fenêtre console;
- `BufferWidth` et `BufferHeight` représentent la largeur totale et le nombre de lignes pouvant être stockées dans le buffer :
 - `BufferWidth` et `BufferHeight` doit valoir au moins `WindowWidth` et `WindowHeight`;
 - Si les valeurs sont supérieures, on aura alors des barres de défilement horizontales et/ou verticales.
- `WindowTop` et `WindowLeft` représentent la position du buffer présentée à l'utilisateur (ce qui veut dire que si nécessaire les barres de défilement horizontale et verticale seront déplacées au bon endroit en fonction de ces valeurs) :
 - Il faut s'assurer que `WindowLeft + WindowWidth` est inférieur à `BufferWidth`.

Si on a par exemple le code suivant :

```
class Program
{
    static int Main()
    {
        Console.Title = "Mon application console";
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.DarkRed;

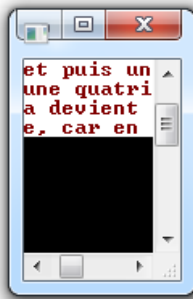
        Console.WindowHeight = 10;
        Console.WindowWidth = 10;
        Console.BufferHeight = 30;
        Console.BufferWidth = 30;
        Console.WindowTop = 3;
        Console.WindowLeft = 3;

        Console.WriteLine("Une ligne quelconque");
        Console.WriteLine("Et pourquoi pas une autre! Allez hop! La voilà!");
        Console.WriteLine("Oh et puis une troisième!");
        Console.WriteLine("Et une quatrième!");
        Console.WriteLine("Cela devient une mauvaise habitude, car en voici une cinquième!");

        Console.ReadLine();

        return -1;
    }
}
```

On observera la fenêtre console à l'exécution :

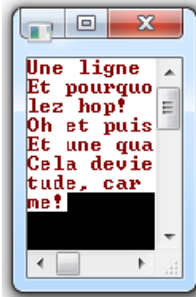


Remarquez les éléments suivant :

- La première colonne affichée est le 4^e caractère de chaque ligne du buffer;
- Certaines expressions demandent plus que les 30 caractères par ligne de buffer et exige donc une ligne supplémentaire, qui compte parmi la capacité du buffer.

Par ailleurs, si à force d'écrire à la console on dépasse la capacité du buffer, les premières lignes écrites seront effacées, à la manière d'une file (*FIFO*).

Sans les propriétés `WindowTop` et `WindowLeft` définies, l'affichage montrerait plutôt ceci :



Si on modifiait le code de façon à mettre un buffer inférieur à la dimension de la fenêtre, on aurait à l'exécution le message d'erreur suivant :

```
Console.WindowHeight = 10;  
Console.WindowWidth = 10;  
Console.BufferHeight = 8;  
Console.BufferWidth = 8;  
Console.WindowTop = 3;  
Console.WindowLeft = 3;  
  
Console.WriteLine("Une ligne");  
Console.WriteLine("Et pourquo lez hop!");  
Console.WriteLine("Oh et puis");  
Console.WriteLine("Et une qua");  
Console.WriteLine("Cela devie");  
Console.WriteLine("tude, car");  
Console.WriteLine("me!");
```

ArgumentOutOfRangeException was unhandled

An unhandled exception of type 'System.ArgumentOutOfRangeException' occurred in mscorlib.dll

Additional information: The console buffer size must not be less than the current size and position of the console window, nor greater than or equal to `Int16.MaxValue`.

Si on modifiait le code en faisant en sorte de déplacer la position du coin supérieur gauche de la fenêtre alors que la largeur de la fenêtre plus cette position dépasse la largeur ou la hauteur totale du buffer, on aurait également droit à une levée d'exception :

```
Console.WindowHeight = 10;  
Console.WindowWidth = 10;  
Console.BufferHeight = 30;  
Console.BufferWidth = 30;  
Console.WindowTop = 22;  
Console.WindowLeft = 22;  
  
Console.WriteLine("Une ligne");  
Console.WriteLine("Et pourquo lez hop!");  
Console.WriteLine("Oh et puis");  
Console.WriteLine("Et une qua");  
Console.WriteLine("Cela devie");  
Console.WriteLine("tude, car");  
Console.WriteLine("me!");
```

ArgumentOutOfRangeException was unhandled

An unhandled exception of type 'System.ArgumentOutOfRangeException' occurred in mscorlib.dll

Additional information: The window position must be set such that the current window size fits within the console's buffer, and the numbers must not be negative.

2.4.2 Formatage des sorties

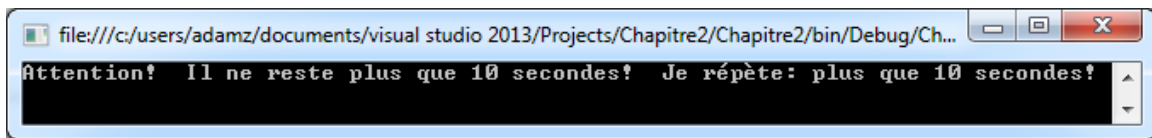
Vous avez sans doute remarqué l'utilisation à quelques reprises de jetons de style `{0}` à l'intérieur de littéraux en conjonction avec l'utilisation de la méthode `Console.WriteLine()`. C'est que .NET supporte un style de formatage de chaînes de caractères similaires à ce que vous retrouvez avec `printf()` en C/C++.

Un des avantages de cette façon de procéder par rapport à l'utilisation directe d'une concaténation avec l'opérateur + (nous revisiterons la concaténation bientôt) est que vous pouvez utiliser le même jeton plusieurs fois, puis faire référence une seule fois à la valeur/la variable qui doit s'y insérer.

Autrement dit, vous pourriez avoir le code suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("Attention! Il ne reste plus que {0} secondes! Je répète: plus que {0} secondes!", 10);
    Console.ReadLine();
}
```

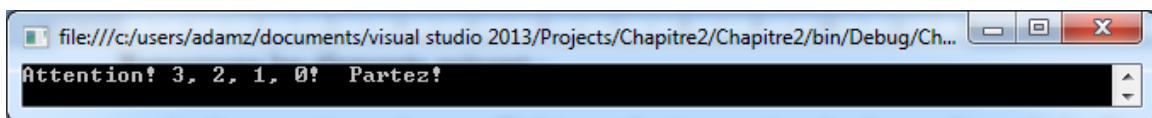
Donnera ceci :



Si vous désirez avoir plusieurs arguments, vous devez les séparer par des virgules supplémentaires et appeler les jetons selon cet ordre. Par exemple :

```
static void Main(string[] args)
{
    Console.WriteLine("Attention! {0}, {2}, {1}, {3}! Partez!", 3, 1, 2, 0);
    Console.ReadLine();
}
```

Aura comme résultat :



Par ailleurs, vous pouvez formater de façon plus élaborer les valeurs (ou les valeurs des variables) pour chaque jeton individuellement. Le tableau suivant liste les formatages les plus communs :

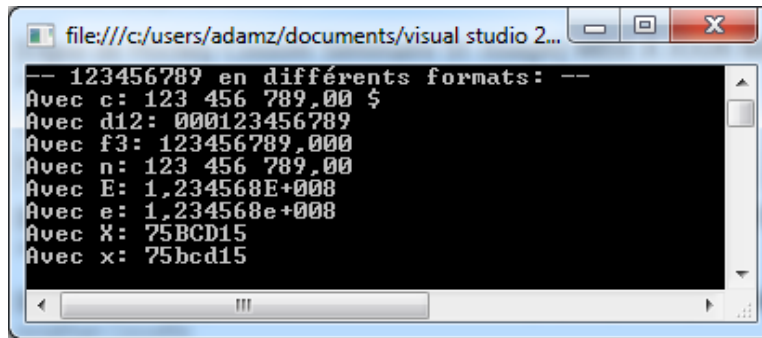
Caractère de formatage (casse non sensible)	Description
c	Utilisé pour formater en monnaie. Le signe de monnaie dépend de la propriété de culture locale associé à l'assembly. Par défaut, c'est le signe de \$.
d	Utilisé pour formater en nombre. Vous pouvez ajouter un nombre pour indiquer le nombre minimal de chiffres (p. ex. d3 pour au minimum 3 chiffres).
e	Utilisé pour formater en une notation exponentielle. La casse dans ce cas déterminera si à l'affichage on verra E ou e.
f	Utilisé pour formater des nombres à virgules de longueur fixée. Comme pour d, vous pouvez spécifier le nombre de chiffres minimaux à afficher.
n	Utilisé pour le formatage numérique de base.
x	Utilisé pour du formatage hexadécimal. En utilisant X plutôt que x, le format hexadécimal apparaîtra avec des lettres majuscules.

Avec le code suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("-- 123456789 en différents formats: --");
    Console.WriteLine("Avec c: {0:c}", 123456789);
    Console.WriteLine("Avec d12: {0:d12}", 123456789);
    Console.WriteLine("Avec f3: {0:f3}", 123456789);
    Console.WriteLine("Avec n: {0:n}", 123456789);
    Console.WriteLine("Avec E: {0:E}", 123456789);
    Console.WriteLine("Avec e: {0:e}", 123456789);
    Console.WriteLine("Avec X: {0:X}", 123456789);
    Console.WriteLine("Avec x: {0:x}", 123456789);

    Console.ReadLine();
}
```

On obtiendra ceci :



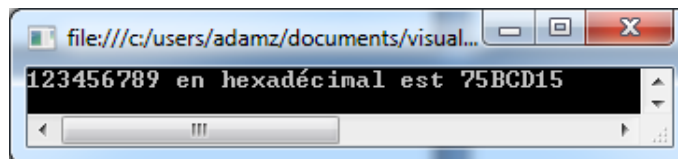
Notez enfin que vous pouvez utiliser ces formatages en général en utilisant la méthode `string.Format()`.

Par exemple, ce code :

```
static void Main(string[] args)
{
    string s = string.Format("123456789 en hexadécimal est {0:X}",
        123456789);
    Console.WriteLine(s);

    Console.ReadLine();
}
```

Donnera en sortie ceci :



2.5 TYPES DE DONNÉES

2.5.1 Types de données primitifs

C# définit un certain nombre de mots clés pour les types de données fondamentaux, qui sont utilisés pour représenter des variables locales, des variables de membres de classes, des valeurs de retour et des paramètres de méthodes.

Contrairement à certains langages de programmation, ces mots clés sont plus que de simples mots reconnus par le compilateur. Plutôt, ils agissent en tant que notations raccourcies à la place de types définis dans l'espace de noms `System`.

Le tableau qui suit donne la liste des types de données de base de `System` en C#.

Raccourci C#	Type System	Domaine de valeurs	Description
bool	System.Boolean	true ou false	Bit à 1 (vrai) ou 0 (faux).
sbyte	System.SByte	-128 à 127	Entier signé sur 8 bits.
byte	System.Byte	0 à 255	Entier non signé sur 8 bits.
short	System.Int16	-32768 à 32767	Entier signé sur 16 bits.
ushort	System.UInt16	0 à 65535	Entier non signé sur 16 bits.
int	System.Int32	-2.1G à 2.1G	Entier signé sur 32 bits.
uint	System.UInt32	0 à 4.3G	Entier non signé sur 32 bits.
long	System.Int64	-9.2E+18 à 9.2E+18	Entier signé sur 64 bits.
ulong	System.UInt64	0 à 18.4E+18	Entier non signé sur 64 bits.
char	System.Char	U+0000 à U+FFFF	Caractère Unicode sur 16 bits.
float	System.Single	-3.4E38 à 3.4E38	Nombre à virgule signé sur 32 bits.
double	System.Double	-5E-324 à 1.7E308	Nombre à virgule signé sur 64 bits.
decimal	System.Decimal	-7.9E-28 à 7.9E28	Nombre signé sur 128 bits
string	System.String	Selon la mémoire système	Ensemble (chaîne) de caractères Unicode.
Object	System.Object	Peut stocker tout type de données	Classe de base de tous les types .NET.

Quelques remarques :

- Par défaut, tout nombre à virgule est traité comme un `double` :
 - Pour forcer un nombre décimal à être un `float`, il faut utiliser le suffixe `f` ou `F`;
 - Pour forcer un nombre décimal à être de type `decimal`, il faut utiliser le suffixe `m` ou `M`;
 - Par exemple, `1.0f`, ou `5.23m`.
- Par défaut, tout nombre entier est traité comme un `int` :
 - Pour spécifier qu'une donnée doit être un long, il faut utiliser `l` ou `L` suite à ce nombre;
 - Par exemple, `4L`.

2.5.2 Déclaration et initialisation de variables

Lorsque vous déclarez une variable localement (c'est-à-dire à l'intérieur d'un membre), vous devez spécifier le type de donnée suivi du nom de la variable :

```
static void Main(string[] args)
{
    string s;
    char c;
    int i;
}
```

L'initialisation de la valeur de la variable peut se faire en même temps que la déclaration ou ultérieurement :

```
static void Main(string[] args)
{
    string s = "bonjour";
    char c = 'a';
    int i;
    i = 34;
}
```

Il est également possible de déclarer plusieurs variables d'un même type sur une même ligne de code :

```
static void Main(string[] args)
{
    string s = "bonjour", s2 = "allo";
    int i, j = 34, k, l = 33;
    bool estVisible = true, float f = 3.4f; // erreur de compilation car 2
    types différents sur la même ligne
}
```

Enfin, vous pouvez utiliser la notation raccourcie pour faire référence aux types, ou utiliser le type `System` correspondant: cela ne fait aucune différence:

```
static void Main(string[] args)
{
    String s = "bonjour", s2 = "allo";
    Int32 i, j = 34, k, l = 33;
    Boolean estVisible = true;
}
```

Par ailleurs, tous les types de données `System` supportent ce qu'on appelle un constructeur par défaut (nous reviendrons plus en détails sur ce concept). Cela permet l'utilisation du mot clé `new` lors de la création de la variable de façon à attribuer automatiquement une valeur par défaut à celle-ci.

Les valeurs par défaut des types de base sont les suivantes :

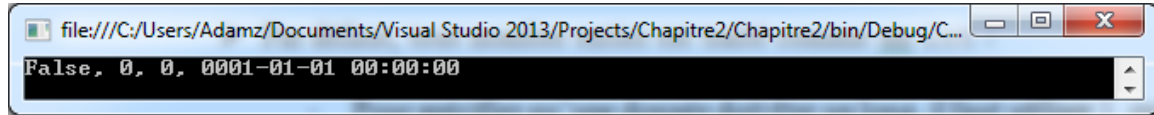
- `bool` : `false`;
- Données numériques : `0` pour les entiers, `0.0` pour les nombres à virgules;
- `char` : caractère vide;
- `DateTime` : `1/1/0001 12 :00 :00 AM`;
- Références objets (dont `string`) : `null`.

Autrement dit, le code suivant :

```
static void Main(string[] args)
{
    bool b = new bool();
    Int32 i = new Int32(), j = new Int32();
    double d = new Double();
    DateTime dt = new DateTime();

    Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
    Console.ReadLine();
}
```

Produira ceci en sortie :



2.5.3 Manipulation des types de données

En considérant que la classe `System.Object` est la classe mère d'entre toutes, il faut de ce fait comprendre que même les types de données de base dérivent de cette classe `Object`. Ainsi, il est possible de faire appel à des membres définis pour ces types afin de manipuler les données. Nous verrons dans cette section les principaux membres pour chacun d'entre eux.

2.5.3.1 TYPES DE DONNÉES NUMÉRIQUES

Voici les méthodes et propriétés inhérentes aux types numériques en C# :

Méthode/Propriété	Description
<i>Tout type numérique</i>	
MinValue MaxValue	Retourne la valeur minimale ou maximale du domaine des valeurs possibles pour le type numérique en question.
<i>Nombres à virgules seulement</i>	
Epsilon	Retourne le « pas » entre deux valeurs possibles pour le type de données.
PositiveInfinity NegativeInfinity	Retourne la valeur +Infini ou -Infini pour des calculs scientifiques.
NaN	Retourne une valeur non numérique.
IsInfinity() IsPositiveInfinity() IsNegativeInfinity() IsNaN()	Teste (vrai ou faux) si la valeur numérique vaut la propriété ou non.

Le code qui suit :

```
static void Main(string[] args)
```

```
{
    Console.WriteLine("-- Membres pour les types de données numériques --");
    Console.WriteLine("Valeur maximale d'un int: {0}", int.MaxValue);
    Console.WriteLine("Valeur minimale d'un int: {0}", Int32.MinValue);
    Console.WriteLine("Valeur maximale d'un float: {0}", float.MaxValue);
    Console.WriteLine("Valeur minimale d'un double: {0}",
        Double.MinValue);
    Console.WriteLine("Epsilon d'un float: {0}", float.Epsilon);
    Console.WriteLine("Epsilon d'un double: {0}", double.Epsilon);
    Console.WriteLine("Valeur d'infinité positive d'un double: {0}",
        double.PositiveInfinity);
    Console.WriteLine("Valeur d'infinité négative d'un double: {0}",
        double.NegativeInfinity);
    Console.WriteLine("Valeur non numérique attribuée à un double: {0}",
        double.NaN);
    Console.WriteLine("Test de valeur infini positive pour un float: {0}",
        double.IsInfinity(double.PositiveInfinity));

    Console.ReadLine();
}
```

Aura donc comme résultat à l’affichage ceci :

```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
-- Membres pour les types de données numériques --
Valeur maximale d'un int: 2147483647
Valeur minimale d'un int: -2147483648
Valeur maximale d'un float: 3.402823E+38
Valeur minimale d'un double: -1.79769313486232E+308
Epsilon d'un float: 1.401298E-45
Epsilon d'un double: 4.94065645841247E-324
Valeur d'infinité positive d'un double: +Infini
Valeur d'infinité négative d'un double: -Infini
Valeur non numérique attribuée à un double: Non Numérique
Test de valeur infini positive pour un float: True
-
```

2.5.3.2 BOOLEAN

Étant donné la simplicité du type de donnée, il n’y a que deux propriétés d’intérêts :

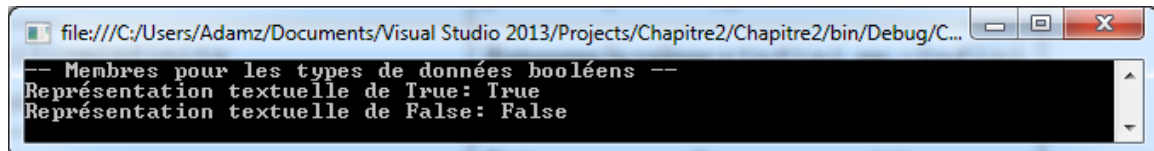
Méthode/Propriété	Description
TrueString FalseString	Retourne sous forme de chaîne de caractères « True » ou « False ».

Ce code :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Membres pour les types de données booléens --");
```

```
Console.WriteLine("Représentation textuelle de True: {0}",  
bool.TrueString);  
Console.WriteLine("Représentation textuelle de False: {0}",  
bool.FalseString);  
  
Console.ReadLine();  
  
}
```

Mène donc vers cette sortie :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...  
-- Membres pour les types de données booléens --  
Représentation textuelle de True: True  
Représentation textuelle de False: False
```

2.5.3.3 CHAR

Une valeur de type `char` est représenté par un caractère Unicode. Les méthodes statiques de `System.Char` permettent de déterminer si un caractère donné est de nature alphabétique ou numérique, un caractère de ponctuation, et plus encore :

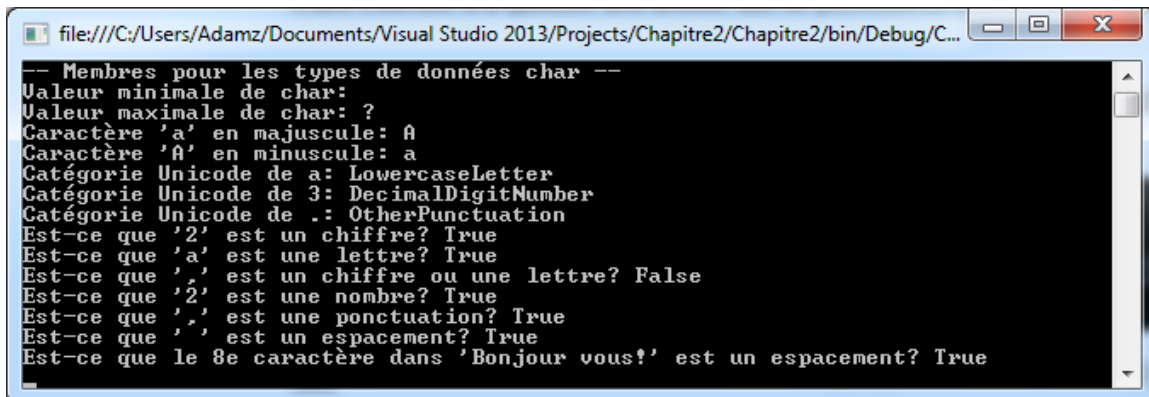
Méthode/Propriété	Description
<code>MinValue</code> <code>MaxValue</code>	Retourne la valeur minimale ou maximale du domaine des valeurs Unicode possibles pour un caractère.
<code>ToUpper()</code> <code>ToLower()</code>	Retourne une version majuscule ou minuscule dans le cas d'un caractère alphabétique.
<code>GetUnicodeCategory()</code>	Retourne la catégorie Unicode d'un caractère donné, qui est une valeur de l'énumération <code>System.Globalization.UnicodeCategory</code> .
<code>IsDigit()</code> <code>IsNumber()</code> <code>IsLetter()</code> <code>IsLetterOrDigit()</code> <code>IsPunctuation()</code> <code>IsWhiteSpace()</code> ...	Retourne <code>true</code> ou <code>false</code> selon si le caractère appartient oui ou non au type de caractère indiqué en question.

Considérons le code suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Membres pour les types de données char --");
    Console.WriteLine("Valeur minimale de char: {0}", char.MinValue);
    Console.WriteLine("Valeur maximale de char: {0}", char.MaxValue);
    Console.WriteLine("Caractère 'a' en majuscule: {0}",
        char.ToUpper('a'));
    Console.WriteLine("Caractère 'A' en minuscule: {0}",
        char.ToLower('A'));
    Console.WriteLine("Catégorie Unicode de a: {0}",
        char.GetUnicodeCategory('a'));
    Console.WriteLine("Catégorie Unicode de 3: {0}",
        char.GetUnicodeCategory('3'));
    Console.WriteLine("Catégorie Unicode de .: {0}",
        char.GetUnicodeCategory('.')');
    Console.WriteLine("Est-ce que '2' est un chiffre? {0}",
        char.IsDigit('2'));
    Console.WriteLine("Est-ce que 'a' est une lettre? {0}",
        char.IsLetter('a'));
    Console.WriteLine("Est-ce que ',' est un chiffre ou une lettre? {0}",
        char.IsLetterOrDigit(','));
    Console.WriteLine("Est-ce que '2' est une nombre? {0}",
        char.IsNumber('2'));
    Console.WriteLine("Est-ce que ',' est une ponctuation? {0}",
        char.IsPunctuation(','));
    Console.WriteLine("Est-ce que ' ' est un espacement? {0}",
        char.IsWhiteSpace(' '));
    Console.WriteLine("Est-ce que le 8e caractère dans 'Bonjour vous!' est
        un espacement? {0}", char.IsWhiteSpace("Bonjour vous!", 7));

    Console.ReadLine();
}
```

En sortie, le résultat sera tel que :



```
-- Membres pour les types de données char --
Valeur minimale de char:
Valeur maximale de char: ?
Caractère 'a' en majuscule: A
Caractère 'A' en minuscule: a
Catégorie Unicode de a: LowercaseLetter
Catégorie Unicode de 3: DecimalDigitNumber
Catégorie Unicode de .: OtherPunctuation
Est-ce que '2' est un chiffre? True
Est-ce que 'a' est une lettre? True
Est-ce que ',' est un chiffre ou une lettre? False
Est-ce que '2' est une nombre? True
Est-ce que ',' est une ponctuation? True
Est-ce que ' ' est un espacement? True
Est-ce que le 8e caractère dans 'Bonjour vous!' est un espacement? True
```

Il existe quelques autres méthodes supplémentaires que vous pouvez toujours consulter dans la documentation Microsoft.

2.5.3.4 PARSING

Les types de données .NET viennent avec l'abileté de générer une variable de leur type à partir de leur équivalent textuel.

Cette technique de conversion très importante est ce que nous appelons du « *parsing* ».

Elle est particulièrement utile, par exemple, pour convertir du texte entré par un utilisateur en console ou via des contrôles utilisateurs dans une interface utilisateur en une valeur numérique.

Par exemple, si nous avons le code suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Parsing de données --");

    bool b = bool.Parse("true");
    Console.WriteLine("Valeur de b: {0}", b);

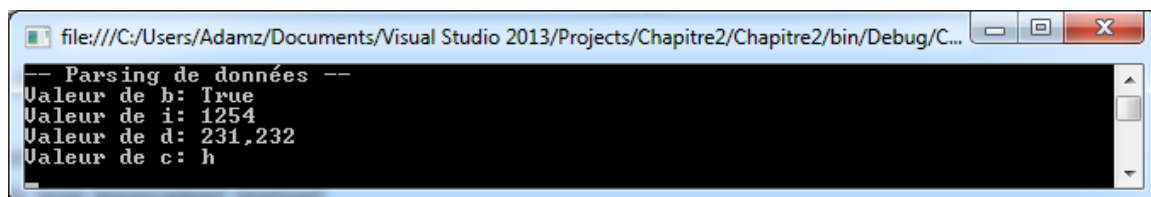
    int i = Int32.Parse("1254");
    Console.WriteLine("Valeur de i: {0}", i);

    double d = double.Parse("231,232");
    Console.WriteLine("Valeur de d: {0}", d);

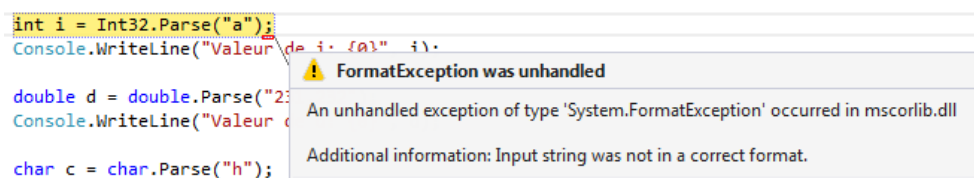
    char c = char.Parse("h");
    Console.WriteLine("Valeur de c: {0}", c);

    Console.ReadLine();
}
```

La console montrera ceci :



À noter que si à l'exécution la conversion échoue une exception de type `FormatException` sera levée. Par exemple :



2.5.3.5 DATETIME

L'espace de noms `System` définit quelques types de données sans notation raccourci comme la structure `DateTime`.

Le type `DateTime` contient des données qui représentent une date spécifique et une valeur de temps, chacune des parties peuvent être formatée de différentes façons grâce aux membres fournis.

Méthode/Propriété	Description
<code>Date</code> <code>TimeOfDay</code>	Retourne la partie correspondant soit à la date, soit à l'heure du jour.
<code>Millisecond</code> <code>Second</code> <code>Minute</code> <code>Hour</code> <code>Day</code> <code>Month</code> <code>Year</code>	Retourne une des composantes de la date et de l'heure.
<code>DayOfWeek</code> <code>Day</code> <code>DayOfYear</code>	Retourne soit le jour de la semaine, du mois, ou de l'année.
<code>IsDaylightSavingTime()</code>	Retourne vrai si la date et l'heure fait partie de l'heure avancée d'été.
<code>AddMilliseconds()</code> <code>AddSeconds()</code> <code>AddMinutes()</code> <code>AddHours()</code> <code>AddDays()</code> <code>AddMonths()</code> <code>AddYears()</code>	Ajoute du temps à la date et heure.
<code>Date.Now</code>	Cette méthode statique donne dynamiquement la date et heure actuelle.

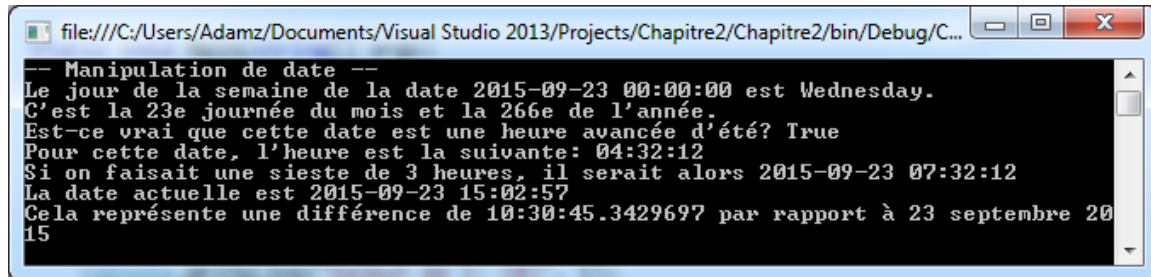
À titre de démonstration, si vous écrivez le code suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Manipulation de date --");

    DateTime dt = new DateTime(2015, 9, 23, 4, 32, 12); // création d'une
    date AAAA, MM, JJ, HH, MM, SS
    Console.WriteLine("Le jour de la semaine de la date {0} est {1}.",
        dt.Date, dt.DayOfWeek);
    Console.WriteLine("C'est la {0}e journée du mois et la {1}e de
        l'année.", dt.Day, dt.DayOfYear);
    Console.WriteLine("Est-ce vrai que cette date est une heure avancée
        d'été? {0}", dt.IsDaylightSavingTime());
    Console.WriteLine("Pour cette date, l'heure est la suivante: {0}",
        dt.TimeOfDay);
    Console.WriteLine("Si on faisait une sieste de 3 heures, il serait
        alors {0}", dt.AddHours(3));
    Console.WriteLine("La date actuelle est {0}", DateTime.Now);
    Console.WriteLine("Cela représente une différence de {0} par rapport à
        {1}", DateTime.Now - dt, dt.ToLongDateString());

    Console.ReadLine();
}
```

Vous générerez en console l'affichage suivant :



Nous ne l'explorerons pas dans le cadre de ce cours, mais sachez qu'il existe un autre type de données de manipulation du temps nommé `TimeSpan` qui peut présenter un certain intérêt.

2.5.4 Manipulation des Strings

2.5.4.1 MANIPULATIONS GÉNÉRALES

Il existe une panoplie de membres fournis avec le type `String` pour manipuler les chaînes de caractères. Le tableau qui suit présente les principales d'entre elles. Des exemples d'utilisation suivront.

Méthode/Propriété	Description
Length	Retourne la taille (en nombre de caractères) de la chaîne de caractère.
String.Compare() CompareTo()	Compare deux chaînes de caractères entre elles.
Contains()	Indique si la chaîne contient ou non le caractère ou la sous-chaîne donnée en paramètre.
Equals()	Indique si deux chaînes contiennent exactement la même expression ou non.
String.Format()	Formate la chaîne telle que vu précédemment.
IndexOf() LastIndexOf()	Retourne l'index de la première ou la dernière occurrence d'un caractère ou d'une sous-chaîne à partir d'une position également passée en paramètre.
Insert()	Insère une chaîne à l'intérieur de la chaîne à la position indiquée en paramètre.
PadLeft() PadRight()	Remplit une chaîne en ajoutant un caractère choisi avant ou après.
Remove() Replace()	Supprime ou remplace toutes les instances d'un caractère ou d'une sous-chaîne.
Split()	Sépare une chaîne en un tableau de chaîne en fonction d'un caractère donné en paramètre.
StartsWith() EndsWith()	Vérifie si oui ou non une expression débute ou se termine par un caractère ou une sous-chaîne donnée en paramètre.
Substring()	Extrait une sous-chaîne à partir d'un index et d'une certaine longueur.
Trim() TrimStart() TrimEnd()	Enlève des caractères (par défaut l'espace) au début et/ou à la fin de la chaîne.

Méthode/Propriété	Description
ToUpper() ToLower()	Modifie la chaîne pour rendre tous les caractères alphabétiques en majuscules ou en minuscules.

Pour tester ces membres, vous pourriez utiliser le jeu de tests suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Manipulation de String --");

    string prenom = "adam", nom = "joly";

    Console.WriteLine("'adam' possède {0} caractères.", prenom.Length);
    Console.WriteLine("'adam' comparé à 'joly': {0}",
        String.Compare(prenom, nom));
    Console.WriteLine("'adam' comparé à 'adam': {0}",
        String.Compare(prenom, prenom));
    Console.WriteLine("'joly' comparé à 'adam': {0}",
        nom.CompareTo(prenom));
    Console.WriteLine("Est-ce vrai que 'joly' contient un 'l'? {0}",
        nom.Contains('l'));
    Console.WriteLine("Est-ce vrai que 'joly' contient 'oli'? {0}",
        nom.Contains("oli"));
    Console.WriteLine("Est-ce vrai que la variable prenom vaut 'adam'?
        {0}", prenom.Equals("adam"));
    Console.WriteLine("Récupère une sous-chaîne de tout ce qui se trouve
        entre le 1er et le dernier 'a' dans le prénom: {0}",
        prenom.Substring(prenom.IndexOf('a'), prenom.LastIndexOf('a') -
            prenom.IndexOf('a') + 1));
    Console.WriteLine("PadLeft sur 'adam' avec un total de 10 caractère,
        le reste rempli avec '*': {0}", prenom.PadLeft(10, '*'));
    Console.WriteLine("Remplace les 'a' par des 'o', puis enlève le
        dernier caractère du prénom: {0}", prenom.Replace('a', 'o').Remove(3,
            1));
    Console.WriteLine("'adam' en lettres majuscules {0}",
        prenom.ToUpper());

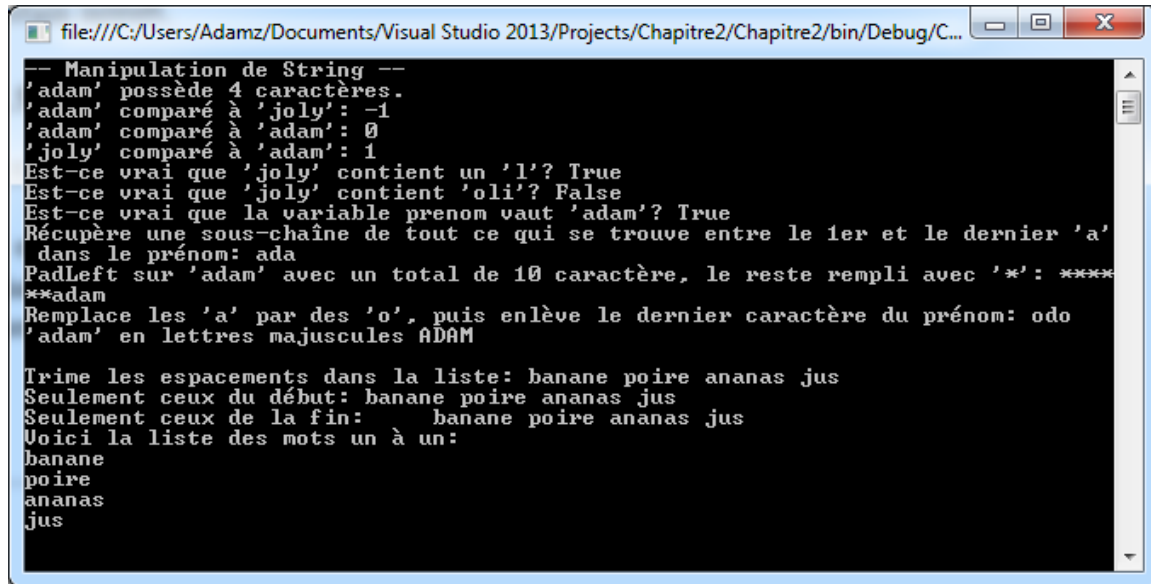
    string liste = "    banane poire ananas jus    ";

    Console.WriteLine();
    Console.WriteLine("Trime les espacements dans la liste: {0}",
        liste.Trim());
    Console.WriteLine("Seulement ceux du début: {0}", liste.TrimStart());
    Console.WriteLine("Seulement ceux de la fin: {0}", liste.TrimEnd());

    Console.WriteLine("Voici la liste des mots un à un:");
    foreach(string s in liste.Trim().Split(' '))
        Console.WriteLine(s);

    Console.ReadLine();
}
```

Qui mènerait alors vers cet affichage :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
-- Manipulation de String --
'adam' possède 4 caractères.
'adam' comparé à 'joly': -1
'adam' comparé à 'adam': 0
'joly' comparé à 'adam': 1
Est-ce vrai que 'joly' contient un 'l'? True
Est-ce vrai que 'joly' contient 'oli'? False
Est-ce vrai que la variable prenom vaut 'adam'? True
Récupère une sous-chaîne de tout ce qui se trouve entre le 1er et le dernier 'a'
dans le prénom: ada
Padleft sur 'adam' avec un total de 10 caractères, le reste rempli avec '*': ****
**adam
Remplace les 'a' par des 'o', puis enlève le dernier caractère du prénom: odo
'adam' en lettres majuscules ADAM

Trime les espacements dans la liste: banane poire ananas jus
Seulement ceux du début: banane poire ananas jus
Seulement ceux de la fin:      banane poire ananas jus
Voici la liste des mots un à un:
banane
poire
ananas
jus
```

2.5.4.2 CONCATÉNATION

Les chaînes de caractères sont concaténées en C# via l'opérateur +, qui à la compilation est reconnu et remplacé par un appel à la méthode statique `String.Concat()`.

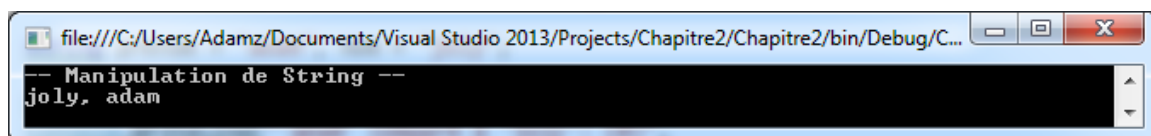
En d'autre termes, ceci :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Manipulation de String --");

    string prenom = "adam", nom = "joly";
    Console.WriteLine(nom + ", " + prenom);

    Console.ReadLine();
}
```

Donnera ceci :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
-- Manipulation de String --
joly, adam
```

2.5.4.3 CARACTÈRES D'ÉCHAPEMENT

Comme dans d'autres langages basés sur C, les littéraux de chaînes de caractères peuvent contenir des caractères d'échappement permettant de spécifier certains comportements/caractères spéciaux à générer en sortie.

Chaque caractère d'échappement débute avec une barre oblique arrière (\) suivi d'un caractère spécial.

Le tableau suivant donne les principaux parmi ceux-ci :

Caractère d'échappement	Description
\'	Insère une apostrophe.
\"	Insère une double apostrophe.
\\	Insère une barre oblique arrière, ce qui est particulièrement pratique pour décrire des chemins d'accès, par exemple.
\a	Déclenche un « beep » sonore.
\n	Insère une nouvelle ligne (sur Windows).
\r	Insère un retour chariot.
\t	Insère une tabulation horizontale.

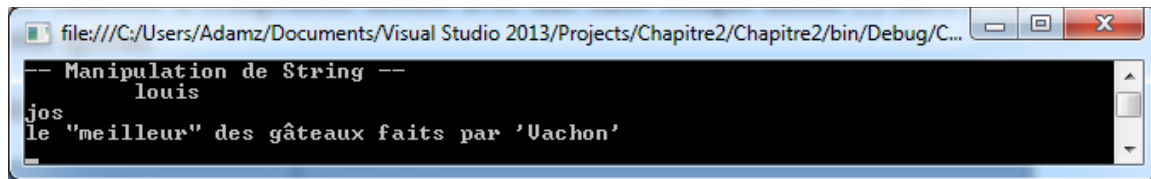
À titre d'illustration, ce code :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Manipulation de String --");

    string prenom = "jos", nom = "louis";
    Console.WriteLine("\t" + nom + "\n" + prenom + "\nle \"meilleur\" des
gâteaux faits par \"Vachon\"");

    Console.ReadLine();
}
```

Produit ceci :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
-- Manipulation de String --
louis
jos
le 'meilleur' des gâteaux faits par 'Uachon'
```

Notez aussi que vous pouvez utiliser le symbole @ pour créer une chaîne verbatim.

Une telle chaîne permet de désactiver le traitement des caractères d'échappement et d'écrire en sortie la chaîne telle qu'elle est décrite. C'est particulièrement utilisé pour décrire des chemins d'accès, ou pour des citations (avec le double "").

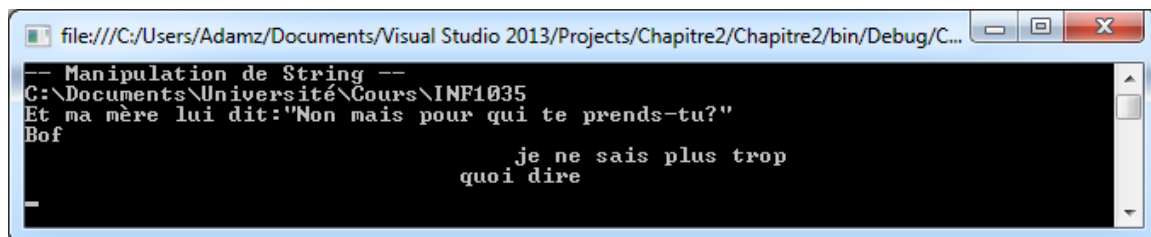
Par exemple :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Manipulation de String --");

    Console.WriteLine(@"C:\Documents\Université\Cours\INF1035");
    Console.WriteLine(@"Et ma mère lui dit:""Non mais pour qui te prends-
tu?""");
    Console.WriteLine(@"Bof
                        je ne sais plus trop
                        quoi dire");

    Console.ReadLine();
}
```

Donnera en console :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
-- Manipulation de String --
C:\Documents\Université\Cours\INF1035
Et ma mère lui dit:"Non mais pour qui te prends-tu?"
Bof
                        je ne sais plus trop
                        quoi dire
```

2.5.4.4 CONCEPT D'ÉGALITÉ DES CHAINES DE CARACTÈRES

Les Strings sont considérés comme des types références (nous reviendrons sur ce concept plus loin), c'est-à-dire un objet en mémoire.

Comme nous le verrons, en temps normal, le test d'égalité pour des objets revient à vérifier si ces objets réfèrent au même objet en mémoire, mais pour les chaînes de caractères, l'opérateur d'égalité est redéfini de façon à agir de même sorte que lorsqu'on appelle la méthode `Equals()`.

La conséquence est que pour les lignes de codes qui suivent :

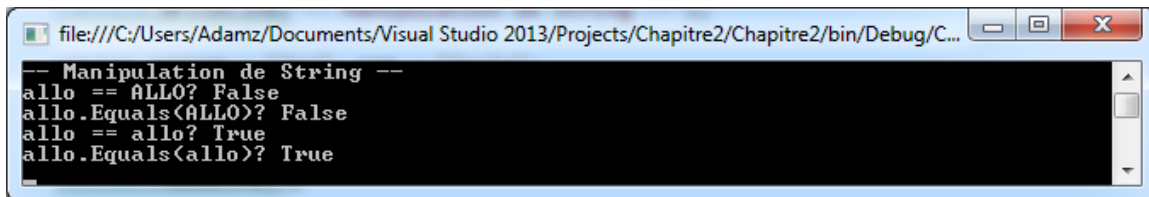
```
static void Main(string[] args)
{
    Console.WriteLine("-- Manipulation de String --");

    string s1 = "allo", s2 = "ALLO";

    Console.WriteLine("allo == ALLO? {0}", s1 == s2);
    Console.WriteLine("allo.Equals(ALLO)? {0}", s1.Equals(s2));
    Console.WriteLine("allo == allo? {0}", s1 == s2.ToLower());
    Console.WriteLine("allo.Equals(allo)? {0}", s1.Equals(s2.ToLower()));

    Console.ReadLine();
}
```

Le résultat en console est identique :



```
file:///C:/Users/Adamz/Documents/Visual Studio 2013/Projects/Chapitre2/Chapitre2/bin/Debug/C...
-- Manipulation de String --
allo == ALLO? False
allo.Equals(ALLO)? False
allo == allo? True
allo.Equals(allo)? True
```

2.5.5 Conversion de données

Un des éléments importants à maîtriser est certainement la conversion de données implicite et explicite :

- **La conversion de données implicite** se produit lorsque vous tentez de stocker la valeur d'une donnée d'un type x vers un type y sans l'indiquer spécifiquement dans votre code. À l'exécution il y aura tentative de conversion;
- **La conversion de données explicite** se produit lorsque vous indiquez dans votre code que vous désirez convertir une données d'un type x vers un autre type y.

Évidemment, certaines vérifications ont lieu dès la compilation afin d'assurer qu'une conversion est techniquement possible à l'exécution.

Un des problèmes émergent de la conversion de données est la possible perte d'information. Par exemple, la transformation d'un `Int32` en un `Int16` pourrait être problématique si la donnée de type `Int32` contient une valeur plus élevée que 32 767.

Pour cette raison, la conversion de données n'est pas toujours permise dans les deux sens pour éviter ces situations en cours d'exécution.

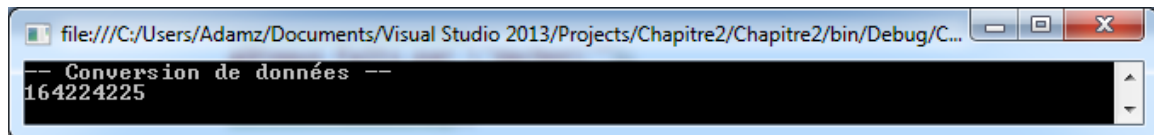
Pour démontrer cela, imaginons le code suivant, dans lequel on désire stocker le résultat d'une multiplication avec deux `shorts` dans un `int` :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Conversion de données --");

    short n = 12825;
    int i = n * n;

    Console.WriteLine(i);
    Console.ReadLine();
}
```

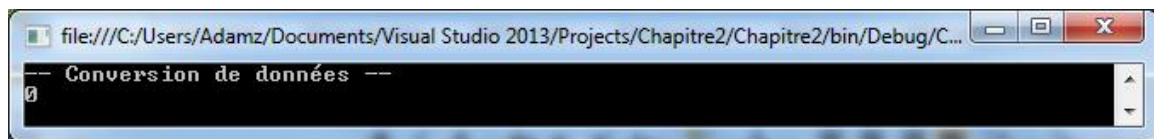
Une conversion implicite est effectuée et cela ne cause pas de problème :



Même si nous avions plutôt écrit :

```
int i = n^n;
```

Il n'y aurait eu aucun problème à la compilation. À l'exécution, toutefois, il y aurait dans ce cas un débordement de valeur et pour éviter un plantage la variable `i` prendrait 0 comme valeur :



Il faut donc être prudent lorsqu'on effectue des conversions, même lorsque permises.

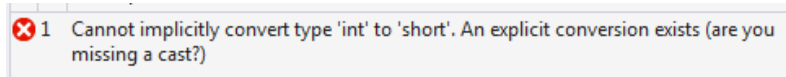
À l'inverse, si on tente de passer d'un `int` vers un `short`, le compilateur ne nous laissera pas faire :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Conversion de données --");

    int n = 12;
    short i = n * n;

    Console.WriteLine(i);
    Console.ReadLine();
}
```

Mènera vers ce message d'erreur :



Pourtant, on peut bien voir que le résultat, 144, est stockable dans une variable de type `short`. Cette conversion descendante est bloquée par le compilateur.

Pour informer le compilateur que vous souhaitez prendre le risque d'une perte de donnée à cause d'une conversion descendante, vous devez effectuer une conversion explicite en utilisant les opérateurs de conversion `()` avec au centre le type de données de conversion.

Dans l'exemple précédent, il faudrait alors avoir le code suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("-- Conversion de données --");

    int n = 12;
    short i = (short)(n * n);

    Console.WriteLine(i);
    Console.ReadLine();
}
```

Et dans ce cas le programme compilerait.

Règle générale, considérez qu'une conversion implicite est possible à partir d'un type entier vers un autre type entier de bits supérieur ou égal ou vers un type stockant un nombre décimal, mais pas dans le sens inverse.

Une liste plus exhaustive est disponible à ce lien : [https://msdn.microsoft.com/en-us/library/08h86h00\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/08h86h00(v=vs.110).aspx).

2.5.6 Variables locales typées implicitement (*var*)

Il est coutume lorsque vous déclarez des variables locales d'utiliser des types explicitement donnés, comme :

```
static void Main(string[] args)
{
    int monEntier = 0;
    bool monBool = true;
    string maChaine = "bonjour";
}
```

Depuis C# 2008, il est permis de déclarer localement des variables de type implicite, c'est-à-dire inféré selon la valeur d'initialisation de la dite variable, à la manière de la déclaration de variables en PHP, par exemple.

2.5.6.1 DÉCLARATION

Pour cela, vous devez utiliser le mot clé `var` à la place du type dans la déclaration. Par exemple :

```
static void Main(string[] args)
{
    var monEntier = 0;
    var monBool = true;
    var maChaine = "bonjour";
}
```

En effet, si vous laissez le curseur de la souris vis-à-vis un des « `var` », vous verrez le titre implicitement inféré par le compilateur :

```
class Program
{
    static void Main(string[] args)
    {
        var monEntier = 0;
        var monBool = true;
        var maChaine = "bonjour";
    }
}
```

`class System.String`
Represents text as a series of Unicode characters.

Le mot clé `var` peut être utilisé autant pour les variables de type primitif/valeur que des types complexes/références ou des collections.

Par exemple :

```
class Program
{
    static void Main(string[] args)
    {
        var nombres = new int[] { 5, 25, 2, 23 };
        var robot = new Robot();
        var robots = new List<Robot>();
    }
}
```

`class System.Collections.Generic.List<T>`
Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

T is AutresParticularitesCSharp.Robot

L'important est que la valeur ou la référence de la variable soit affectée (sinon, le compilateur n'a aucun moyen d'inférer le type).

2.5.6.2 UTILISATION DANS LES BOUCLES FOREACH

Vous pouvez utiliser également le typage implicite à l'intérieur d'une structure de contrôle `foreach` (que nous présenterons plus en détails à la prochaine section). Par exemple :

```
static void Main(string[] args)
{
    var nombres = new int[] { 5, 25, 2, 23 };

    foreach (var item in nombres)
        Console.WriteLine(item);
}
```

Ce qui est équivalent (et également permis) à :

```
static void Main(string[] args)
{
    var nombres = new int[] { 5, 25, 2, 23 };

    foreach (int item in nombres)
        Console.WriteLine(item);
}
```

2.5.6.3 CONTRAINTES D'UTILISATION

Il y a certaines restrictions dans l'utilisation du mot clé `var` :

1. Cela n'est syntaxiquement valide que pour des variables locales à l'intérieur d'une méthode ou d'une propriété; il est donc interdit d'utiliser `var` comme type de paramètres, de retour ou de variables d'instance, tel que :

```
static var uneMethode(var x) { }
```

2. La valeur initiale assignée doit être différente de `null`; ceci n'est pas acceptable :

```
var x = null;
```

Par contre, vous pouvez évidemment affecter une valeur nulle après que le type ait été implicitement déterminé, bien évidemment; ceci est valide :

```
var x = new Robot();
x = null;
```

3. Les données typées implicitement **sont des données fortement typées**; ainsi, bien que comme nous l'avons mentionné cette technique syntaxique ressemble à ce qu'on peut retrouver en PHP, il y a cependant une énorme différence : en C#,

le type n'est pas dynamiquement malléable en cours d'exécution, c'est-à-dire qu'une fois le type inféré, il ne peut être changé; le code suivant provoquerait donc une erreur à la compilation :

```
var x = new Robot();  
x = 44;
```

2.5.6.4 UTILITÉ

Rien ne vous empêche de les utiliser, mais étant donné que les types implicites n'offrent pas l'avantage du typage dynamique, il y a peu d'avantage à l'utiliser dans un contexte standard de programmation. En effet, cela peut amener certains flous à la lecture du code quant aux types des données.

Cependant, ils constituent d'importants alliés à l'API d'interrogation de collections de données LINQ, du fait que `var` est particulièrement utile pour traiter des données provenant d'ensemble de résultats dynamiquement retournés par les requêtes, dont le type est difficilement prévisible/définissable. Nous retrouverons du code du genre (ne vous en faites pas pour l'instant pour les détails de la syntaxe LINQ) :

```
static void Main(string[] args)  
{  
    int[] nombres = new int[] { 5, 25, 2, 23 };  
    var subset = from i in nombres where i < 24 select i;  
  
    foreach (var x in subset)  
        Console.WriteLine(x);  
  
    Console.ReadLine();  
}
```

Bien que LINQ dépasse la portée de ce cours, il demeure important de comprendre le fonctionnement des variables typées implicitement ne serait-ce qu'au cas où vous en rencontreriez dans du code C#.

2.6 STRUCTURES DE CONTRÔLE

Tous les langages de programmation fournissent un certain nombre de mécanismes de programmation permettant de répéter ou se diriger conditionnellement vers certains blocs de codes.

Vous les avez utilisées à maintes reprises lorsque vous avez appris Java lors des cours d'introduction à la programmation orientée objet et de structures de données.
