Workshop: Introduction au C#

GConfs

19 novembre 2010

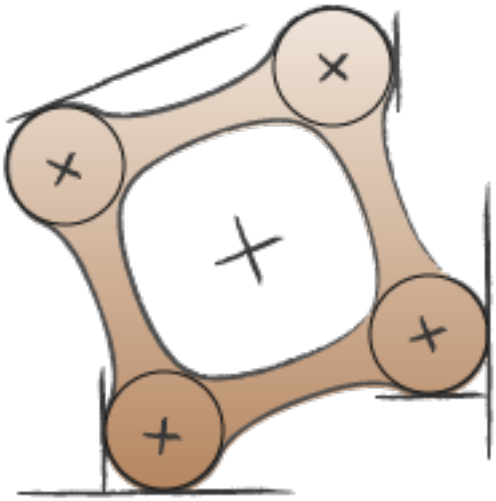

Table des matières

1 Introduction

1.1 Environnement de Travail

- Lancer Visual Studio 2008
- Fichier / Nouveau / Projet / C# / Application console
- Une portion de code minimale sera générée :

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Vous venez de mettre en place votre environnement de travail. Il est composé de plusieurs fichiers. Parmi ceux-ci, seul un nous intéresse pour le moment : Program.cs
De nombreux autres fichiers seront générés tels que des fichiers contenant des informations de débogage.
Un fichier est toutefois important : ControleApplication1.sln. Il s'agit du fichier que vous devrez ouvrir si vous fermez puis ré-ouvrez VS2008. En effet, il contient toutes les information concernant votre projet. (En réalité, il représente une "solution", et vous trouverez aussi un fichier ConsoleApplication1.csproj qui représente un "projet". De ce fait vous pouvez avoir plusieurs "projets" dans une solution, mais ce n'est pas important pour ce TP).

Comme vous pouvez le voir, le fichier Program.cs contient une fonction "Main". Il s'agit du point d'entrée de l'application : la première fonction appelée automatiquement lors du démarrage de votre programme.
C'est donc ici que tout commence et que vous ferez des appels à d'autres fonctions. Par la suite, nous allons écrire d'autres fonctions que vous devrez tester. Vous les écrirez dans ce fichier, mais gardez en tête qu'il est parfois préférable de travailler dans plusieurs fichiers séparés.

1.2 Hello World !

Comme tout bon tutoriel qui se respecte, la première chose à vous faire faire est d'afficher Hello World !. Pour cela il vous faudra utiliser la procédure Console.WriteLine( ). Tous les paramètres donnés seront convertis en chaîne de caractères (pour les types

prédéfinis) puis affichés sur la sortie standard. Dans un premier temps nous allons le placer dans le fichier Program.cs, et compléter la fonction Main(string[] args).

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world");
            Console.ReadLine();
        }
    }
}
```

- *Remarque* La fonction ReadLine() va attendre que l'utilisateur appuie sur Entrée. Cela permet que le programme ne s'arrête pas directement mais nous laisse le temps de lire notre Hello World !

1.3 Voir le résultat de votre travail !

Coder c'est bien, mais ça ne sert à rien si on ne peut pas exécuter le fruit de son travail ! Vous allez donc compiler votre code.
Visual Studio vous permet de compiler très intuitivement et facilement si vous ne souhaitez pas d'options particulières. D'ailleurs, vous avez surement deviné que pour compiler il faut utiliser la petite flèche verte dans la barre d'outils ! Eh bien ce n'est pas tout à fait exact, ce bouton fera deux choses pour vous : la compilation et l'exécution. Ainsi, si vous cliquez sur ce bouton, vous verrez au bout d'un moment votre console affichant "Hello World". Il est toutefois possible de compiler SANS exécuter le programme automatiquement. Pour cela, choisissez le menu "Générer – Générer la solution". Compiler sans exécuter est utile si vous souhaitez juste voir les messages d'erreurs, ou vérifier que tout compile.
Toutefois, utiliser les boutons devient rapidement pénible. Il existe donc deux raccourcis : F5 pour compiler+exécuter et F6 pour compiler uniquement.
Faites donc un F5 et regardez le résultat !

1.4 Première Fonction

Votre première fonction va écrire Hello World ! sur la sortie. Vous devrez la placer dans le fichier Program.cs en dehors de la fonction Main(), mais à intérieur de la classe (délimitée par des accolades { }) Program. Votre fonction ressemblera à la

fonction Main actuelle, mais sans arguments.
Toute fonction a un type de retour : int, float, string, etc... Toutefois, une fonction n'est pas toujours obligée de renvoyer quelque-chose ! Elle peut donc renvoyer 'rien', c'est à dire 'void'. Une telle fonction est usuellement appelée "procédure".
Lorsque vous déclarez une fonction, vous lui donnez donc : des attributs, un type, un nom, des paramètres, un contenu (le code exécuté lorsqu'elle est appelée). Pour le moment, nous ne ferons pas attention aux attributs et mettrons 'static' qui signifie que cette fonction peut être appelée sans avoir à créer un objet.
Voici donc, à quoi devrait ressembler votre procédure HelloWorld() :

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }

        static void HelloWorld()
        {
            Console.WriteLine("Hello world");
            Console.ReadLine();
      }
   }
}
```

Maintenant que nous avons codé la fonction il faut l'appeler, c'est dans le Main que vous allez le faire. Celui-ci va donc contenir :

```
static void Main(string[] args)
       {
            HelloWorld();
        }
```

Bravo, vous êtes maintenant prêts à coder des fonctions plus compliquées !

2 Factorielle

2.1 Définition de Factorielle

Pour une gestion plus simple des cas d'erreurs nous allons imposer une définition de factorielle spéciale qui vaut 1 pour des nombres négatifs.

$$n! = \begin{cases} 1 & \text{si } n <= 1 \\ n * (n-1)! & \text{si } n > 1 \end{cases}$$

Afin de pouvoir tester votre code, voici quelques exemples.

n	-42	-1	0	1	2	3	5	10
n!	1	1	1	1	2	6	120	3628800

- *Conseil* Lorsque vous aurez à choisir des exemples par vous même par la suite, veillez à bien prendre en compte tous les cas particuliers (et à les implémenter) ! Ici il ne faut pas oublier les nombres négatifs, ni 0, ni 1.

2.2 Version Recursive

Nous allons débuter par une version récursive de la fonction qui à déjà été vue en Caml. Il s'agit d'écrire le code C# qui va retranscrire à la lettre la formule mathématique.

```
static int fact(int n)
{
}
```

Syntaxe : Pour réaliser des alternatives vous devez utiliser la structure

```
if (Condition)
{
Action1
}
else
{
Action2;
}
```

Si la Condition est égale à vrai alors l'Action1 est effectuée, sinon c'est l'Action2 qui l'est.

2.3 Version Impérative - While

Nous allons transcrire cette fonction en impératif. Nous allons utiliser des boucles pour simuler les appels récursifs. Il existe plusieurs types de boucles mais la plus simple pour commencer reste la boucle While.

```
static int fact_while(int n)
{
}
```

Syntaxe : La boucle while s'utilise de cette façon :

```
while (Condition)
{
Action;
}
```

Tant que la Condition est vérifiée, alors on effectue l'Action. Avec la Condition étant une expression de type bool.
Par exemple le code suivant va afficher les nombres de i à 0.

```
while (i>=0)
{
        Console.WriteLine(i);
        i--; // ou bien i = i - 1;
}
```

2.4 Version Imperative - For

Maintenant que vous avez réussi à retranscrire grâce à la boucle While la fonction factorielle, nous allons utiliser la boucle For. Lorsque l'on connait le nombre d'itérations à l'avance elle est plus pratique à utiliser.

```
static int fact_for(int n)
{
}
```

Syntaxe : La boucle For s'utilise de cette façon :

```
for(int i=0; i < 10; i++)
{
        action;
}
```

Pour comprendre son fonctionnement, on peut la réécrire avec une boucle While. Vous remarquerez la concision de la boucle for.

```
int i = 0;
while (i < 10)
{
        action;
        i++; // ou bien i = i + 1;
}
```

2.5 Version Imperative - Passage par référence

Jusqu'à présent, nous avons utilisé des fonctions qui renvoient la valeur de la factorielle. Il est possible d'utiliser une procédure qui ne renvoie rien mais qui va modifier une des variables passées en paramètre.

```
static void fact_for(int n, ref int res)
{
}
```

Utilisation : Lorsque vous voudrez tester cette fonction, il va falloir déclarer une variable de type int et la donner à la procédure avec le mot clef ref. A la suite de l'appel, cette variable contiendra le résultat de l'opération.

```
int res = 1;
fact_ref(5 , ref res);
Console.WriteLine(res);
```

3 Point 2D

3.1 Type - Structure

Nous allons apprendre à créer un nouveau type : un vecteur mathématique. Il comporte deux coordonnées représentées par deux champs dans notre structure.

```
struct Point2D
    {
        public int x;
        public int y;
    }
```

Vous pouvez placer cette définition à deux endroits : avant la déclaration de la classe Program, ou dans un fichier séparé. Le plus propre est encore de le faire dans un fichier séparé.
Si toutefois, vous souhaitez le mettre dans le même fichier que votre Main, vous devrez le placer dans la partie "namespace", mais avant la déclaration "class Program". Vous obtiendrez ceci :

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

namespace ConsoleApplication1
{

    struct Point2D
    {
        public int x;
        public int y;
    }

    class Program
    {
        static void Main(string[] args)
        { /* du code ici */   }

        // d'autres fonctions ici
    }

}
```

Utilisation : Son utilisation est très simple. Il vous suffit de déclarer une variable du type Point2D. Ensuite pour accéder à chacun des champs il vous suffit de mettre un . devant son identifiant. Remarquez l'attribut "public" devant chaque définition de variable de la structure Point2D. Cela signifie que vous donnez à tout votre code le

droit de lire et modifier les champs (i-e x et y) de la structure. En général, c'est une pratique à éviter car cela brise certains principes liés à la Programmation Orientée Objet. Toutefois, ici notre structure est très simple, donc pour plus de simplicité, nous conserverons l'attribut public.

```
Point2D pt;
pt.x = 0;
pt.x = pt.x + 1; // ou bien pt.x++
Console.WriteLine(pt.x);
// Affiche 1
```

3.2 Afficher un Point2D

La première chose à faire avant de coder quoi que ce soit est de s'assurer que l'on peut afficher ce que l'on est en train de faire. Nous allons donc réaliser une fonction qui prend un Point2D en paramètre, par exemple $\binom{x=12}{y=5}$, et qui affiche dans la console :

```
x=12 ; y=-5
```

```
static void print_pt2d(Point2D p)
{
}
```

Attention : Lorsque l'on vous demande d'écrire quelque chose dans la sortie vous devez respecter le format à la lettre ! Pas d'espace autour du '=', les deux sur une même ligne ... Lorsque vous serez corrigés par une moulinette, la moindre faute de syntaxe vous vaudra 0.

3.3 Addition

Passons aux choses sérieuses. Vous devez coder l'addition de deux vecteurs, inutile de vous donner la formule ...

```
static Point2D add_pt2d(Point2D pa, Point2D pb)
{
}
```

3.4 Opposé

En prévision de la fonction de soustraction, nous allons coder une fonction qui donne l'opposé d'un Point2D, c'est à dire l'opposé de tous les champs. Par exemple :

$$-\binom{x=2}{y=-5} = \binom{x=-2}{y=5}$$

```
static Point2D opp_pt2d(Point2D pa)
{
}
```

3.5 Opposé par référence

Il faut savoir que lorsque l'on retourne un type structuré par une fonction, tous les champs sont recopiés dans une nouvelle variable. Même si dans cet exemple ce n'est pas important, lorsque ces opérations sont effectuées des millions de fois par seconde, toute optimisation, même aussi mineure que celle-ci, est bonne à prendre. La méthode pour palier ce problème est le passage par référence. En effet, la variable est directement modifiée et non copiée. Par contre il n'est plus possible d'utiliser l'ancienne variable.

```
static void opp_pt2d(ref Point2D pa)
{
}
```

3.6 Soustraction avec Opposé

Maintenant que vous avez les deux fonctions Addition et Opposé, il va faloir les combiner afin de coder la fonction Soustraction.

```
static Point2D sub_pt2d(Point2D pa, Point2D pb)
{
}
```

3.7 Distance

Il est possible à partir de vecteurs d'obtenir d'autres types que des vecteurs. Par exemple on peut calculer la distance entre deux vecteurs qui est un réel.

$$dist(\binom{x1}{y1}, \binom{x2}{y2}) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

```
static float dist_pt2d(Point2D pa, Point2D pb)
{
}
```

Remarque : Il existe plusieurs représentations des nombres réels en C#. Les plus couramment utilisés sont les float. Les double permettent de stocker de plus grandes valeurs.
Remarque : Vous devrez utiliser la fonction racine carrée suivante : Math.sqrt()

4 Bonus

Bravo, vous avez terminé le TP guidé, voici quelques exercices bonus pour parfaire vos notions de C#. Ils sont très similaires à ce que vous pourrez avoir pendant les partiels.

4.1 Histogramme

Le but de cet exercice est de vous faire manipuler les fichiers. Vous trouverez toute la documentation nécessaire sur le site de msdn, ou sur la documentation accessible en appuyant sur F1. Vous devez écrire une fonction qui prend en entrée deux noms de fichiers. Elle lira le premier fichier et écrira dans le second un histogramme du nombre d'occurence de chaque chiffe.

```
static void AfficherHistogramme(string path_in, string path_out)
{
}
```

4.1.1 Histogramme Largeur

Voici un exemple pour l'histogramme en largeur.

```
Fichier Entrée :
6 5 6 6 3 2 1 6 2 2 2 1 2 5 6 7 1 2 5 6 6 5 6 6 3 2 1 6 2

Fichier Sortie :
1 : * * * *
2 : * * * * * * * * *
3 : * *
4 :
5 : * * * *
6 : * * * * * * * * * *
7 : *
```

4.1.2 Histogramme Hauteur

Plus dur, le même histogramme mais en hauteur.

```
Fichier Sortie :
1 2 3 4 5 6 7
- - - - - - -
* * * * * *
* * * * *
* * * *
* * * *
* *
* *
```

```
* *
* *
*
*
```

4.2 Recherche de chaîne

Le but de cet exercice est de vous faire manipuler les chaînes de caractères. Vous devez écrire une fonction qui prend en entrée trois noms de fichiers. Elle lira le premier fichier et écrira dans le dernier toutes les occurences des mots du premier dans le second.

```
static void RechercheChaine(string in1, string in2, string out)
{
}
```

4.2.1 Recherche ligne par ligne

Dans une premiere partie, vous allez comparer les mots lignes par lignes.

```
Fichier Entrée 1 :

fooo
vjeux
banban
epita
renard


Fichier Entrée 2 :

epita
banban
belette
fooo
42
marmotte


Fichier Sortie :

fooo
banban
epita
```

4.2.2 Recherche complète

Il serait plus intéressant de donner des mots et de les rechercher dans un corpus de texte.

```
Fichier Entrée 1 :

famille
belette
carotte
cactus
confondu
renard


Fichier Entrée 2 :

La belette (Mustela nivalis) est le plus petit mammifère de la famille
des mustélidés et constitue également le plus petit mammifère
carnassier d'Europe avec une taille d'environ 20 cm pour une centaine
de grammes seulement. Vivant essentiellement dans les milieux
désertiques, la belette peut facilement être confondu avec une hermine.


Fichier Sortie :

famille
belette
confondu
```

5 Conclusion

Félicitation ! Maintenant que vous êtes habiles avec l'imperatif en C#, il est temps de vous plonger dans le monde merveilleux de la Programmation Orientée Objets (POO) avec le Workshop numéro 2 !