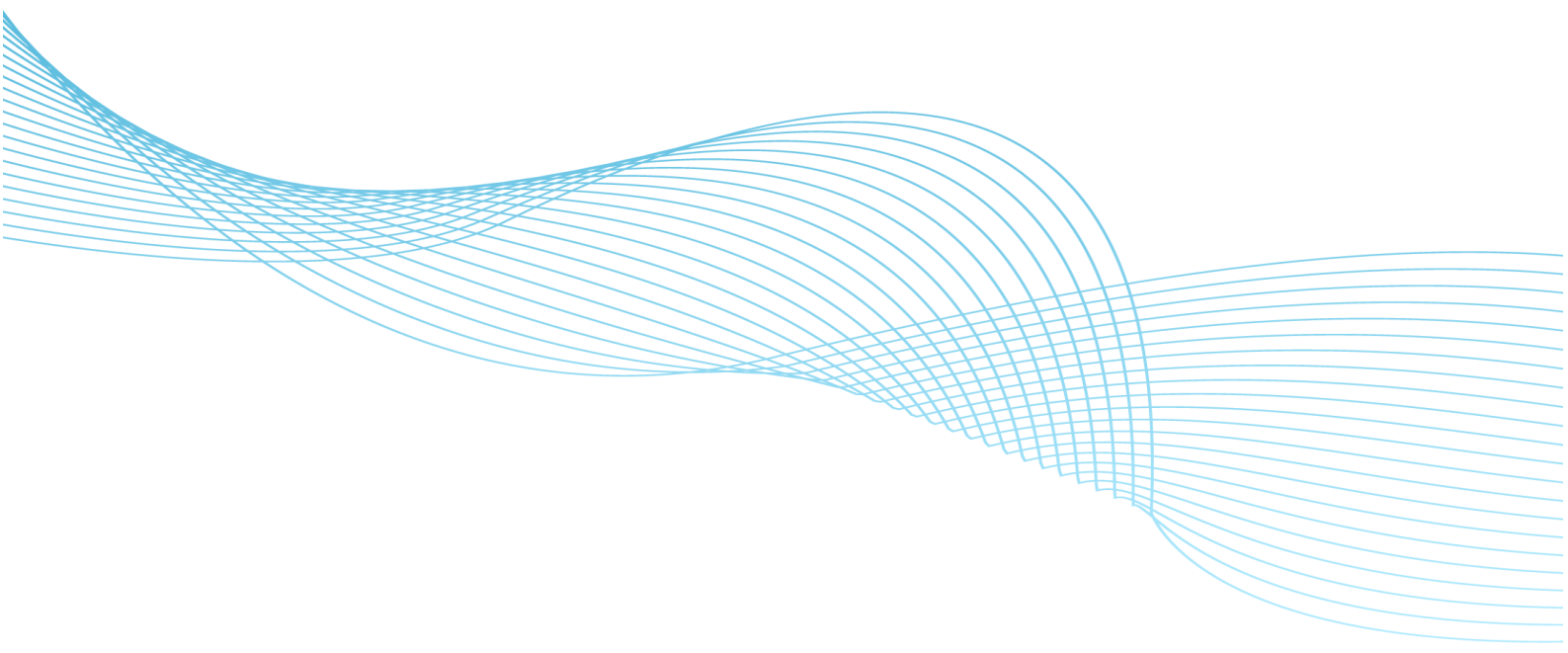
Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Notions fondamentales du langage C#

Version 1.0



Sommaire

1 Tout ce qu'il faut savoir pour bien commencer

Dans cette première partie, nous allons présenter les concepts de base qu'il faut connaitre pour apprendre à programmer et aussi introduire la plateforme .NET. Ce sont des concepts simples mais fondamentaux. Évidemment, les personnes ayant déjà des notions avancées de programmation ne verront ici rien de bien savant et pourront passer directement à la partie consacrée à l'apprentissage du langage C# (prononcez ci-sharpe).

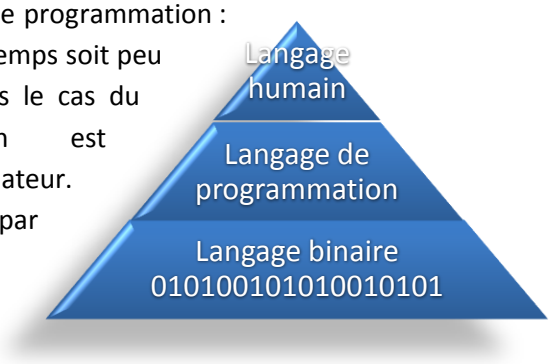
1.1 Qu'est ce qu'un langage de programmation ?

1.1.1 Introduction

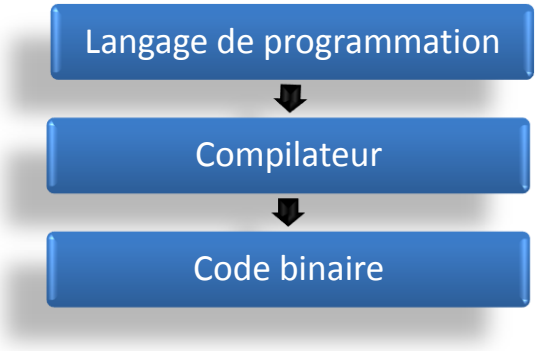
Nous, humains, communiquons avec un langage complexe. En effet, notre langage est composé de phrases, construites avec des mots qui sont eux-mêmes constitués de différentes lettres. Mais malheureusement pour nous, les ordinateurs, eux, ne comprennent que des 1 et des 0. Le problème est donc de savoir comment nous allons faire pour pouvoir communiquer avec eux, car notre but final est tout de même de programmer l'ordinateur afin d'obtenir ce que nous voulons de lui. Tout cela en gardant à l'esprit que programmer en binaire n'est pas une alternative envisageable.

1.1.2 Nécessité d'un langage intermédiaire

C'est donc pour cette raison que l'on a inventé le langage de programmation : afin de pouvoir communiquer avec la machine dans un langage un temps soit peu compréhensible par l'humain, ce qui n'est de toute évidence pas le cas du langage binaire. Le langage de programmation est donc un langage à mi-chemin entre notre langage et celui de l'ordinateur. Grossièrement, C'est un langage qui est compréhensible à la fois par l'homme et par le processeur de la machine.



Une fois notre code écrit en langage de programmation, un logiciel appelé compilateur va se charger de le traduire en langage binaire comme indiqué sur le schéma ci-dessous.



Le compilateur est une sorte de « traducteur » langage de programmation → binaire.

1.2 Introduction à la programmation orientée objet

1.2.1 Les différences avec un langage procédural

Avant d'introduire quelque différence que ce soit, nous allons d'abord définir ce qu'est un langage procédural. Un langage procédural est, comme son nom l'indique, un langage composé de procédures. Une procédure est ce qui va permettre au programme de traiter une suite d'informations, c'est-à-dire qu'on lui envoie simplement une instruction qu'il se doit d'exécuter. Dans un programme, une procédure n'a pas de place prédéfinie, donc une même procédure peut être en début, en milieu, en fin de programme, ou les trois à la fois. Le C#, lui, n'est pas un langage procédural, mais un langage de programmation orienté objet, souvent abrégé en « POO ». Nous allons voir ce qu'est cette notion.

Tout d'abord de quoi parle-t-on lorsque l'on parle d'objet ?

Un objet est en fait une structure de données, c'est-a-dire une structure qui sert à contenir des données pour faire de leur traitement un acte plus simple.

A quoi cela sert-il ?

On a vu précédemment ce qu'est un objet. Cet objet contient à l'intérieur de lui du code. Ce code va contenir tout ce qui est fonctions et variables voulues. On va donc pouvoir se servir d'un objet, ce qui est relativement simple, pour utiliser un code complexe. La principale différence entre ces deux méthodes de programmation est que lorsque l'on utilise un langage procédural, le programmeur doit tout coder, alors que lorsque l'on utilise un langage orienté objet, le programmeur peut utiliser des objets afin de rendre son code moins lourd et plus facile à programmer. Il y a un gain de temps et de simplicité dans cette approche.

1.2.2 Les avantages de la programmation orientée objet

Le but de la POO est avant tout de rendre la programmation complexe plus facile. C'est-à-dire, dans un sens, la rendre plus familière à l'Homme. Cependant, que connaît et par quoi est entouré particulièrement l'Homme : des objets. C'est pourquoi la POO nous aide, concrétise certaines notions qui seraient difficiles à coder en langage procédural. Prenons par exemple la modélisation d'une voiture. Dans un langage procédural, le programmeur va devoir utiliser des outils abstraits. Alors qu'en POO, le programmeur pourra faire un objet "voiture" et lui donner ses caractéristiques, un objet "roue" par exemple, ainsi de suite, ce qui rend la programmation plus simple et surtout plus instinctive.

Ceci est un avantage non négligeable dans le monde de la programmation, qui permet donc un gain de temps, de compréhension mais aussi d'efficacité.

1.3 Présentation du Framework .Net

Ici nous allons voir ce qu'est la plateforme .NET. Nous commenceront par un bref historique puis nous rentreront dans le vif du sujet en nous intéressant à son mode de fonctionnement.

1.3.1 Introduction

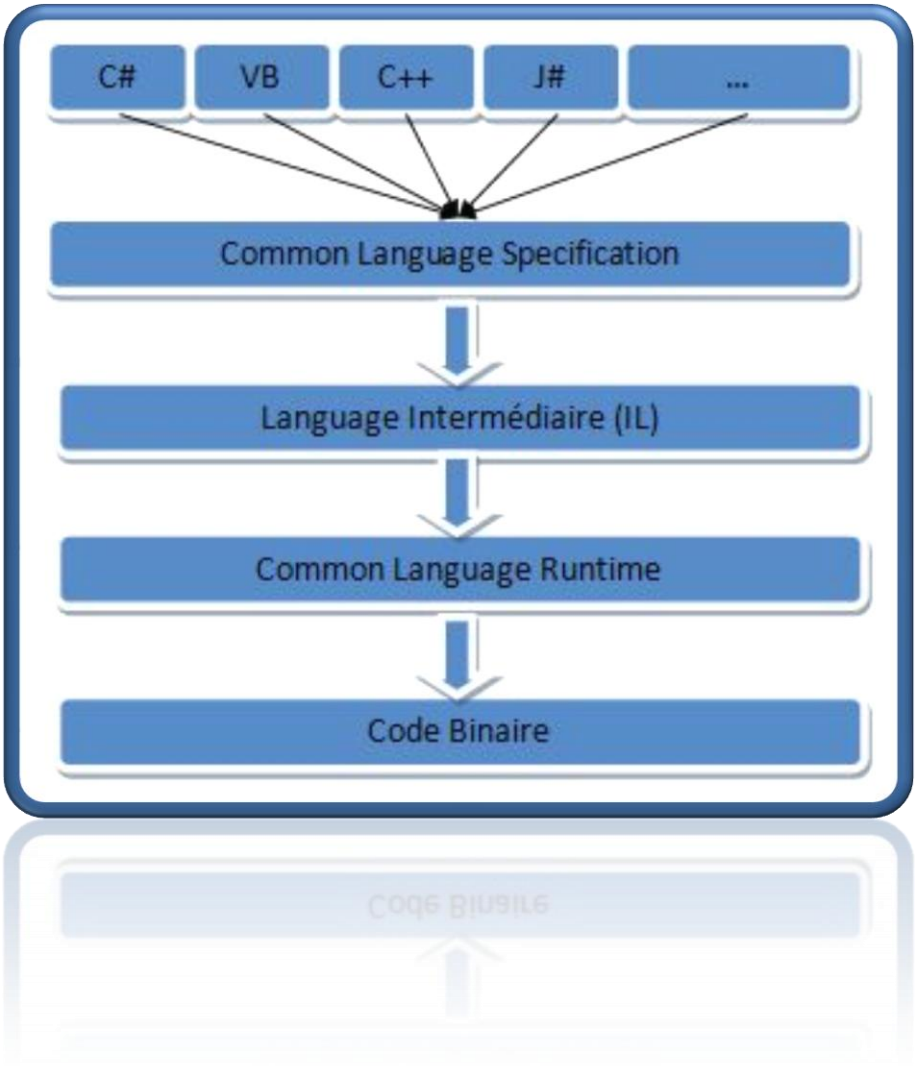
Nous sommes aujourd'hui à la version 3.5 du Framework .NET (et bientôt 4.0). La première version exploitable de celui-ci est sortie en 2002 et celle-ci n'a pas cessé d'évoluer de même que son environnement de programmation dédié : Visual Studio. En effet la version 1 .1 est sortie en 2003 puis la 2.0 en 2005 la 3.0 en 2006 et la version actuelle 3.5 sortie en 2007. Chaque mise à jour apportant d'importantes nouveautés et améliorations.

Nativement, la plate-forme .NET fut crée pour Windows, mais celle-ci est désormais portable sur des systèmes Linux ou Unix, grâce à un projet intitulé MONO.

Le Framework .NET regroupe un ensemble d'outils pour le programmeur. En effet les technologies .Net vont nous permettre de programmer autant des applications console et WindowsForm que des services Web, des applications Windows mobile et bien d'autres.
Un des avantages du Framework .Net, c'est qu'il est multi-langages. C'est-à-dire que l'on va pouvoir programmer en VB.NET, en C#, en C++, en J# et aussi en d'autres langages compatibles. Nous allons voir comment l'architecture du Framework va nous permettre cela.

1.3.2 Le Common Language Specification et le Common Language Runtime.

Nous allons voir ici le fonctionnement de la plateforme .NET grâce à un schéma simple.



Pour commencer on code dans un langage compatible .NET. La Common Language Specification (CLS) s'occupe, elle, de la compatibilité de chaque langage avec le Framework de sorte qu'en codant avec n'importe quel langage compatible, on aboutisse au même code intermédiaire. Chaque langage est donc compilé par son compilateur associé pour aboutir à un code intermédiaire : le MicroSoft Intermediate Language (MSIL).

Cette caractéristique est typique du Framework.NET, et elle n'est pas là par hasard. Bien évidemment, cette compatibilité multi-langages vise à faciliter la migration d'un langage de développement spécifique vers la plateforme .Net (exemple du Visual Basic et le VB.Net). Le processus de compilation est assez semblable au fonctionnement du langage JAVA. Notre code source est « Pré »compilé dans un langage intermédiaire et celui-ci sera interprété lors de l'exécution par une machine virtuelle qui le traduira en code natif.
Cette « machine virtuelle » dans le Framework .Net, c'est la Common Language Runtime (CLR).
La CLR a deux grandes fonctions : d'abord, elle s'occupe de l'interprétation du code à la volée. C'est-à-dire qu'elle va compiler le code MSIL en fonctions des besoins ce qui optimise les performances. Cela s'appelle la compilation « Just in Time » ou « JIT ». Grâce à cette méthode on va uniquement compiler du code « utile ». Et ensuite la Common Language Runtime va gérer la mémoire.

1.4 Qu'est ce que le C#

1.4.1 Un langage simple et performant pour .NET

est le langage par excellence de .Net, apparu en 2001. Ce langage est comme dit précédemment un langage de programmation orientée objet, étant un carrefour entre différent langage comme le Java, le C++ ou encore le Visual Basic, tout en restant un langage à part. C'est un langage à typage fort, c'est à dire que le type des variables est fixé à la compilation et que celle-ci peut vérifier les erreurs de typage (à la différence du langage PHP par exemple). Le C Sharp est très polyvalent, Il permet de coder de simples applications consoles jusqu'à de gros programmes avec une multitude de fenêtres, en passant par les jeux.

Souvent, le langage C# a été comparé au java, pour qui, tout comme le C#, tout est objet. Cependant, le langage de .NET, de part ses amélioration au fur et à mesure de son existence, a su se forger son propre profil. En effet depuis sa parution, on est passé de la version 2.0 à 2.5, de 2.5 à 3.0 et enfin de la version 3.0 à 3.5, ayant dans chaque évolution bien entendu son nombre de nouveautés. De nouveaux mots clés sont ajoutés, de nouvelles sortes de classes et certaines syntaxes simplifiées… C'est donc au fil des années que ce langage devint de plus en plus efficace et plus facile d'accès que beaucoup d'autres langages orientés objet, ce qui fait aujourd'hui sa popularité.

2 Les bases du C#

2.1 Environnement de développement

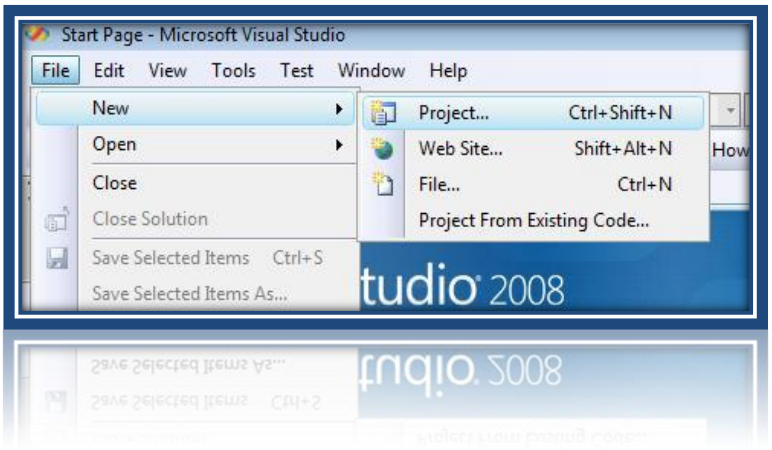
Microsoft Visual studio 2008 est l'environnement de développement dédié à la plateforme .NET, son utilisation est donc ici plus que vivement conseillée. Toutefois les codes de C# présentés ici pourront également être écrits au bloc-notes puis compilés par un compilateur C# en lignes de commande.

2.1.1 Installation de Visual studio 2008

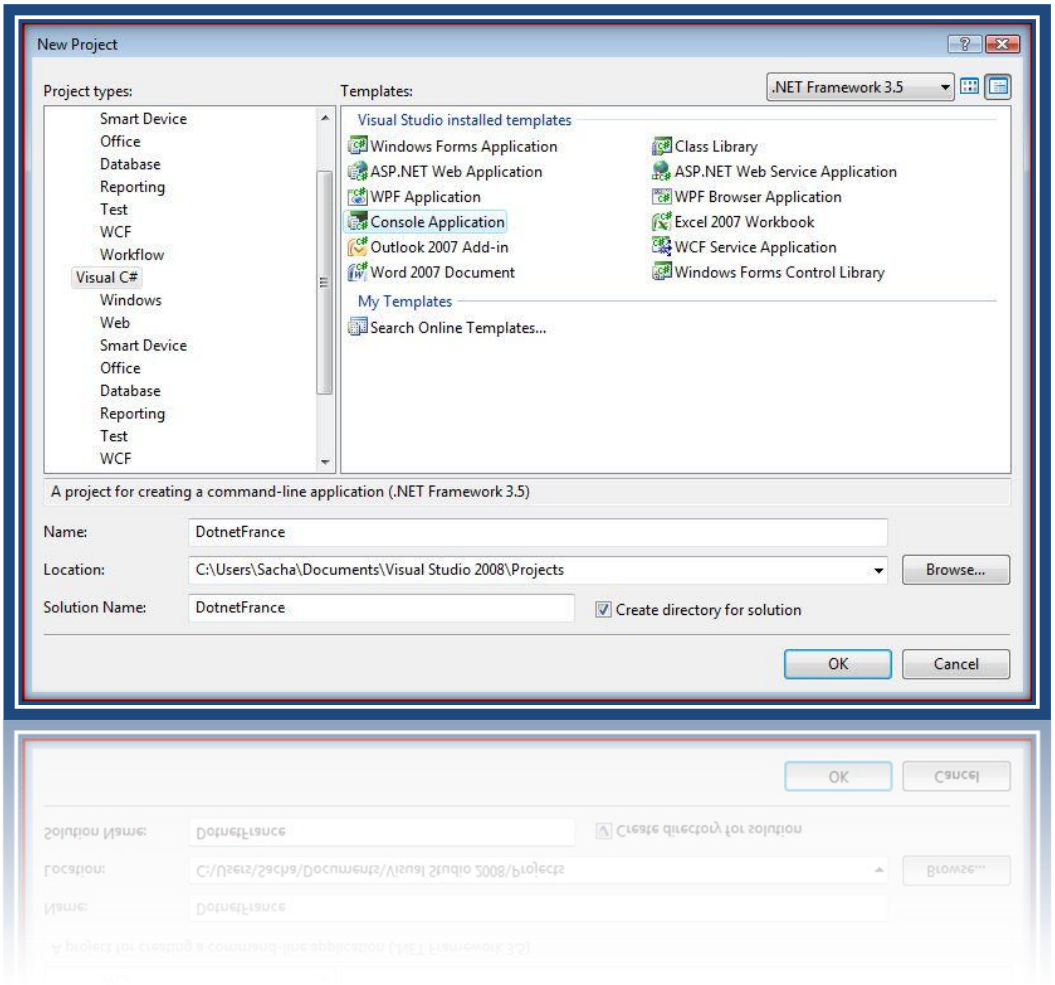
L'installation du logiciel Visual studio 2008 est toute simple, sélectionnez le premier lien sur la première fenêtre d'installation et suivez les différentes étapes.

2.1.2 Création d'un projet C#

Nous allons créer ici notre premier projet C# avec notre première application console. Lancer Visual Studio 2008 et dans le menu supérieur allez dans : File/New/Project comme indiqué sur la capture d'écran ci-dessous.



Puis, dans la fenêtre qui s'ouvre, on va choisir Visual C# dans le menu déroulant et « Console Application » et donner un nom à notre projet.



Si tout s'est bien passé, vous devriez avoir un code qui ressemble à ça :

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

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

A présent, direction l'étape suivante pour comprendre un peu tout çà.

2.2 Les bases de la programmation C#

2.2.1 Les pré-requis

2.2.1.1 La structure de base

Nous allons tout d'abord voir la structure de notre code de base. Le mot clé « `using`» est une directive qui indique avec quel espace de noms nous allons travailler, c'est peut être un peu flou pour le moment mais vous allez rapidement voir son utilité dans un exemple. Le mot clé « `namespace`» défini lui un espace de nom facultatif. Ceci est essentiel à comprendre car c'est afin de mieux organiser le code. Nous avons ensuite la classe « Program » qui est le cœur de notre programme. `static void Main` indique notre méthode principale, c'est la que l'on va placer notre code source, c'est le code placé dans cette partie qui sera exécuté en premier.

Voyons alors l'architecture générique :

```
//C#
using <notre espace de noms>

namespace <espace de noms facultatif>
{
    class Program
    {
        static void Main(string[] args)
        {
            //Notre code source principal sera placé ici.
        }
    }
}
```

Nous allons maintenant exécuter notre premier programme.

2.2.1.2 Afficher un message dans la console

La commande « `System.Console.WriteLine(" ")` » permet d'afficher un message dans la Console.

Exécutez ce code dans Visual Studio (F5 pour exécuter).

```
//C#
class Program
    {
        static void Main(string[] args)
        {
         System.Console.WriteLine("Bienvenue sur Dotnet-France!!!");
        }
    }
```

Il ne se passe rien ? La console apparait et disparaît instantanément ? Ne vous inquiétez pas c'est tout à fait normal car le code s'exécute, le message s'affiche bien mais le programme se ferme car aucune instruction ne lui demande de se mettre en pause. Nous allons donc rajouter une instruction qui attend une entrée utilisateur. Cette instruction va mettre notre programme en « pause » attendant une entrée, ce qui va nous permettre de voir le message s'afficher.

2.2.1.3 Attendre une entrée utilisateur

La commande qui permet de récupérer une entrée au clavier est : « `System.Console.ReadLine(); ` »

Ajouter-la en fin de code comme ceci puis exécutez :

```
//C#
class Program
    {
        static void Main(string[] args)
        {
         System.Console.WriteLine("Bienvenue sur Dotnet-France!!!");
         System.Console.ReadLine();
        }
    }
```

Ici, la chaîne de caractère récupérée par « `System.Console.ReadLine(); ` » n'est stockée dans aucune variable. Cette fonction aura donc le rôle de mettre le programme en pause.

Voilà ! Nous avons à présent tout le temps de contempler notre message.

```
Bienvenue sur Dotnet-France !!!
```

2.2.1.4 La directive `using`

Nous allons maintenant voir l'utilité de la directive « `using`». Le mot clé « using » va nous permettre de ne pas réécrire à chaque fois le chemin entier des commandes. Dans l'exemple, on utilisait « `System.` » devant nos commandes. A présent nous allons insérer « `using System;` » au début de notre programme comme ceci :

```
//C#
using System;

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Bienvenue sur Dotnet-France!!!");
            Console.ReadKey();
        }
    }
```

Cela nous permet d'écrie « `Console.WriteLine` » au lieu de « `System.Console.WriteLine` » ce qui est quand même bien plus pratique si on utilise souvent cette commande.

Grâce à la ligne « `using System;` » notre programme "sait" maintenant où trouver l'instruction « Console.WriteLine » sans que nous ayons à lui indiquer le chemin absolu à chaque fois.

2.2.2 Variables

2.2.2.1 Introduction

Comme vous le savez certainement, les variables nous permettent de stocker des valeurs en mémoire. Nous allons voir comment déclarer une variable en C#. Pour ceux qui auraient des connaissances en langage C, C++, ou Java cette partie ne sera qu'un rappel car la syntaxe du C# est quasiment identique aux langages énoncés précédemment.

2.2.2.2 Les types de variable

La première notion à savoir est qu'il y a plusieurs sortes de variables, selon la place prise en mémoire par celles-ci. En effet la déclaration de la variable va réserver un espace en mémoire. Vous imaginer bien que l'on ne va pas demander à l'ordinateur de réserver une place mémoire pouvant accepter des nombres à 10 chiffres si on désire stocker une valeur binaire. Ce serait ne pas tenir compte du fait que le C# est un langage à typage fort ! C'est pour cela qu'avant le nom de notre variable, nous allons placer un mot clé indiquant le type de celle-ci. Le tableau ci-dessous fait la correspondance entre le type de variable, le mot clé et la place réservée en mémoire.

Mot clé	Le Type de notre variable	Place réservée en mémoire
bool	Valeur booléenne	8 bits
char	Caractère Ascii	16 bits
string	Chaîne de caractères	*
sbyte	De -128 à 127	8 bits
byte	De 0 à 255	8 bits
short	De -32768 à 32767	16 bits
ushort	De 0 à 65535	16 bits
int	De -2147483648 à 2147483647	32 bits
uint	De 0 à 4294967295	32 bits
long	De -9223372036854775808 à 9223372036854775807	64 bits
ulong	De 0 à 18446744073709551615	64 bits
float	De 1.5×10^{-45} à 3.4×10^{38}	32 bits
double	De 5.0×10^{-324} à 1.7×10^{308}	64 bits
decimal	De 1.0×10^{-28} à 7.9×10^{28}	128 bits

2.2.2.3 Les constantes

Les constantes fonctionnent comme des variables, à l'exception que l'on indique au compilateur que leur valeur restera inchangée tout au long de l'exécution du programme. Pour cela on viendra juste placer le mot clé « `const` » devant le type de la variable et on s'assurera d'initialiser la constante des la déclaration.

2.2.2.4 Déclaration de variable

Le code suivant va vous paraitre un peu conséquent, mais il est en fait assez simple. Il présente un exemple de déclaration et une initialisation pour chaque type de variable que nous avons vu.

```
//C#
using System;

    class Program
    {
        static void Main(string[] args)
        {
            //Ceci est un commentaire, il sera ignoré par le
            compilateur !
            bool exempleBooleen = true; //true ou false

            char exempleChar = 'w';      //caractère alphanumérique

            string exempleString = "une chaîne de caractères!";

            sbyte exempleSbyte = 8;
            byte exempleByte = 139;

            short exempleShort = -345;
            ushort exempleUshort = 35000;

            int exempleEntier = 40000;
            uint exempleUint = 79500;

            long exempleLong = 402340540607;
            ulong exempleUlong = 345678905768;

            float exempleFloat = 1.72f;
            //C'est un point et pas une virgule !(f à la fin)

            double exempleDouble = 8.7654398765235423552 5e12d;
            //(d à la fin)

            decimal exempleDecimal = 1.24567234257347385903428492 49e-4m;
            // nombres à très grande précision (m à la fin)

            //déclaration d'une constante
            const float g = 9.80665f;


      //Ici on fait afficher nos valeur à la console.

            Console.WriteLine(exempleBooleen);
            Console.WriteLine(exempleChar);
            Console.WriteLine(exempleString);
            Console.WriteLine(exempleSbyte);
```

```
//C#


            Console.WriteLine(exempleByte);
            Console.WriteLine(exempleShort);
            Console.WriteLine(exempleUshort);
            Console.WriteLine(exempleEntier);
            Console.WriteLine(exempleUint);
            Console.WriteLine(exempleLong);
            Console.WriteLine(exempleUlong);
            Console.WriteLine(exempleFloat);
            Console.WriteLine(exempleDouble);
            Console.WriteLine(exempleDecimal);
            Console.WriteLine(g);
            Console.ReadLine();
            //Cette instruction attends une entrée utilisateur, elle
            nous permet donc de voir ce qui s'est affiché dans la
            console.
        }
    }
```

Ce qui nous donne l'affichage suivant :

```
True
w
une chaîne de caractères !
8
139
-345
35000
40000
79500
402340540607
345678905768
1.72
8765439876523.54
00.000124567234257347385903428 5
9.80665
```

2.2.3 Opérateurs

2.2.3.1 Opérateurs arithmétiques

Ces opérateurs nous permettent d'effectuer du calcul arithmétique sur nos variables.

Operateur	Signification
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo

Le code suivant présente des exemples de calculs :

```
//C#
using System;

    class Program
    {
        static void Main(string[] args)
        {
            int x;      //déclaration d'un entier x sans initialisation.
            int y = 5; //déclaration d'un entier y ayant pour valeur 7.
            x = 7;      //affectation de x

            //On initialise les résultats à 0
            int resultatAddition = 0;
            int resultatSoustraction = 0;
            int resultatMultiplication = 0;
            int resultatDivision = 0;
            int resultatModulo = 0;

            //On effectue les calculs et on les assignent aux résultats
            //correspondants.
            resultatAddition       = x + y;
            resultatSoustraction   = x - y;
            resultatMultiplication = x * y;
            resultatDivision       = x / y;
            resultatModulo         = x % y;

           //Ici on fait afficher nos résultats à la console.
           Console.WriteLine(resultatAddition);
           Console.WriteLine(resultatSoustraction);
           Console.WriteLine(resultatMultiplication);
           Console.WriteLine(resultatDivision);
           Console.WriteLine(resultatModulo);


           Console.ReadLine();//Cette instruction attend une entrée
                              //utilisateur, elle nous permet donc de
                              //voir ce qui s'est affiché dans la
console.
        }
    }
```

La console nous renvoie les résultats suivants :

```
12
2
35
1
2
```

Pour vous entrainer, vous pouvez changer les valeurs de x et y et observer les nouveaux résultats.

2.2.3.2 Opérateurs de test

Ces opérateurs nous permettent de tester des propositions. Ils renvoient donc une valeur booléenne (true ou false).

Operateur	Signification
>	Strictement supérieur
<	Strictement inférieur
>=	Supérieur ou égal
<=	Inférieur ou égal
!=	Différent de
==	Egal

Le code suivant présente des exemples de tests :

```
//C#
using System;

    class Program
    {
        static void Main(string[] args)
        {
            //l'utilisation de la virgule permet de condenser
            //les déclarations de variables de meme type.


            //On déclare a et b deux entiers que nous allons comparer
            int a = 3, b = 8;

            //On déclare nos résultats.
            bool resultat1, resultat2, resultat3, resultat4;

           //Ici on effectue nos tests.
           resultat1 = a > b;
           resultat2 = a < b;
           resultat3 = a != b;
           resultat4 = a == b;

           //Ici on fait afficher nos résultats à la console.
           Console.WriteLine(resultat1);
           Console.WriteLine(resultat2);
           Console.WriteLine(resultat3);
           Console.WriteLine(resultat4);

           Console.ReadLine();
            //Cette instruction attend une entrée utilisateur, elle
            nous permet donc de voir ce qui s'est affiché dans la
            console.
        }
    }
```

La console nous renvoie les résultats des tests effectués :

```
False
True
True
False
```

2.2.3.3 Opérateurs conditionnels

Operateur	Signification
||	OU
&&	ET
? : (opérateur ternaire)	*Condition* **?** *valeur_si_Vrai* **:** *valeur_si_Faux*
??	*??* *valeur si null*

```
//C#
using System;

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

            int a = 3;  //trois entiers pour les trois tests.
            int b = 6;
            int c = 1;
            string d = null;

            bool test1; //trois résultat des tests.
            bool test2;
            string test3;
            string test4;

            test1 = ((a == 2) || (a == 5)); //vrai si a=2 ou a=5.


            test2 = ((b >= 0) && (b <= 10)); //vrai si b est compris
            //entre zéro et dix.


            test3 = (c == 1) ? "c est égal à 1" : "c n'est pas égal à 1";
            //enregistre la premier chaîne si c==1 sinon enregistre la
            //deuxiemme chaîne.

            test4 = d ??  "d est null";
            //test4 vaut d ou 5 si d vaut null


            //on affiche les résultats dans la console.
            Console.WriteLine(test1);
            Console.WriteLine(test2);
            Console.WriteLine(test3);
            Console.WriteLine(test4);

            Console.ReadLine();

        }
    }
```

La console nous renvoie :

```
False
True
c est égal à 1
```

Je vous conseille vivement de reprendre ce code et de changer les valeurs de a, b et c afin de bien comprendre le fonctionnement de ces opérateurs.

2.2.3.4 Opérateurs d'attribution

Operateur	Signification
=	Permet d'assigner une valeur à une variable
+=, -=, *=, /=, %=	Opération puis réassignation

Nous allons tout de suite expliquer le concept des opérateurs de la deuxième ligne du tableau dont la signification doit être encore un peu floue.

Soit l'opération suivante :

`a +=4` équivaut à écrire `a = a+4`. La première expression nous permet donc de condenser la première dans le but de ne pas avoir à répéter le nom de la variable à « réassigner ».

Le code suivant va donner un exemple pour chacun d'eux :

```
//C#
using System;

class Program
{
    static void Main(string[] args)
    {
        int a; // on déclare un entier a.

        a = 3; //opérateur "=" assigne la valeur "3" à la variable "a".
        Console.WriteLine(a);

        a += 1; //équivaut à a = a+1, donc ici a = 4.
        Console.WriteLine(a);

        a -= 2; //équivaut à a = a-2,donc ici a = 2.
        Console.WriteLine(a);

        a *= 3; //équivaut à a = a*3,donc ici a = 6.
        Console.WriteLine(a);

        a /= 6; //équivaut à a = a/6,donc ici a = 1.
        Console.WriteLine(a);

        Console.ReadLine();
    }
}
```

Nous avons suivi chaque opération de l'instruction « `Console.WriteLine(a);` » ce qui va nous permettre de retracer les résultats prévus dans les commentaires. Cela nous donne l'affichage de la console suivant

```
3
4
2
6
1
```

Ce qui correspond parfaitement aux résultats prévus.

2.2.3.5 Opérateurs d'incrémentation

Les Opérateur d'incrémentation dérivent eux aussi d'une écriture condensée.

Operateur	Signification
variable++ , *variable*--	+1 ou -1 après utilisation de la variable
++*variable* , --*variable*	+1 ou -1 avant l'utilisation de la variable

Voici un exemple d'incrémentation :

```
//C#
using System;

class Program
{
    static void Main(string[] args)
    {
        //a++ équivaut à a+=1 qui équivaut à a=a+1.

        int a = 0;
      // on initialise a à 0.

        Console.WriteLine(a++);
        //affiche a,soit 0, puis l'incrémente.

        Console.WriteLine(a);
        //affiche la valeur courante de a, soit 1.

        Console.WriteLine(++a);
        //incrémente a, puis affiche sa valeur, soit 2.


        Console.ReadLine();
    }
}
```

Ce qui nous donne :

```
0
1
2
```

Le système de la décrémentation est totalement identique si ce n'est que l'opération effectuée est « -1 » au lieu de « +1 ».

2.2.4 Les conditions

Nous allons voir dans cette partie les mots-clés qui vont nous permettre d'effectuer des tests à l'intérieur de notre code. Encore une fois, la structure des ces conditions est à 95% identique à celle du langage C, C++ ou Java. Les personnes ayant déjà des connaissances dans ces langages n'y verront donc rien de nouveau.

2.2.4.1 If

Le mot clé if signifie « Si » en français. La syntaxe d'une structure if est détaillée ci-dessous :

```
if( condition )
{
 instruction à effectuer;
 instruction à effectuer;
}
```

L'instruction « `if` » traite une expression booléenne, si la condition placée entre parenthèse est vraie, alors on exécute le code entre accolades.

2.2.4.2 If else

Au simple « si », on peut également ajouter un « sinon si » qui sera testé après le premier si, puis un « sinon » qui s'exécute si aucune des clauses « si » n'est vraie.

Exemple :

```
if(conditionA) //si conditionA vraie.
{
      instructionA;   //alors j'effectue cette partie de code.
}

if else(conditionB) //sinon si la conditionB est vrai.
{
      instructionB;     //alors j'effectue cette partie de code.
}

else    // si aucune condition n'est vraie alors executer ce code.
{
      instructionC
}
```

On peut mettre autant de « `if else` » que l'on souhaite pour tester autant de conditions que l'on veut.

2.2.4.3 Switch

La structure d'un `switch` est la suivante :

```
switch (expression)     //définit la variable sur laquelle vont se baser
        {

            //les diférents cas.

// Si expression == constanteEntière1 alors instruction1.
            case a :
                instruction_a;
                break;

//Si expression == constanteEntière2 alors instruction2.
            case b :
                Instruction_b;
                break;

//Si expression == constanteEntière3 alors instruction3.
            case c :
                  Instruction_c;
                break;

//Sinon effectuer ce code par défaut.
            default:
                instructionParDefaut;
                break;
        }
```

Évidemment, on peut placer autant de « case » que l'ont souhaite.

2.2.5 Les boucles

Nous allons à présent étudier les structures itératives en langage C#.

2.2.5.1 While

« `While` » signifie « Tant que ».Cette boucle « tourne » tant que l'expression qu'on lui donne est vraie.

Sa structure est la suivante :

```
while (expression) //la boucle tourne tant que l'expression est vraie
{
    instruction;
}
```

Nous allons voir un exemple simple, une boucle qui demande à l'utilisateur d'entrer « 0 » pour sortir, et qui se répète tant que l'utilisateur ne tape pas 0.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        //on déclare x un entier que notre boucle va tester.
        int x;

       //on lui assigne la valeur 1 pour pouvoir rentrer dans la boucle.
        x = 1;

        while (x!=0) //la boucle tourne tant que x est différent de 0
        {
            Console.WriteLine("Entrez 0 pour sortir de la boucle");

            //On stocke la valeur entrée par l"utilisateur dans x
            x = Convert.ToInt16(Console.ReadLine());
            }
      }
}
```

Rien de bien difficile dans ce code, mais une ligne en particulier mérite des explications, la ligne :

« `x = Convert.ToInt16(Console.ReadLine());` »

Pour stocker l'entrée au clavier dans la variable x, pourquoi n'a-t-on pas tout simplement écrit :

« `x = Console.ReadLine();` ».

Si vous écrivez cela, vous obtiendrez une erreur de Visual Studio, car vous essayer de stocker une chaîne de caractères dans un entier. En effet, la fonction « `Console.ReadLine()` » renvoi une chaine de caractère, on ne peut donc pas la stocker dans une variable entière.

C'est pour cela que l'on utilise la fonction « `Convert.ToInt16()` » afin de transformer la chaine de caractère récupérée en entier.

2.2.5.2 Do while

La boucle « do while » signifie « Faire tant que ».Elle est identique à la boucle « `While` » à deux détails près :

- Le code à l'intérieur de la boucle est effectué au moins une fois, car l'expression est testée à la fin.
- On met un « ; » après l'expression.

Voici la structure d'une structure itérative « `do while` » :

```
do  //Exécuter ce code.
  {

   instruction;

   }while(expression); //Puis si cette expression est vraie,
                       //repartir dans la boucle.
```

2.2.5.3 For

La boucle « for » correspond à la structure itérative « Pour » en algorithmique.

Sa structure est la suivante :

```
for (initialisation; expression; pas)
  {
                //code de la boucle
  }
```

Prenons par exemple un programme qui affiche Bonjour autant de fois que l'on veut. Nous allons d'abord déclarer une variable i initialisée à 1, et l'incrémenter à chaque tour de boucle jusqu'au nombre de « bonjour » désirés :

```
using System;

class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 10; i++) // On veut afficher 10 "Bonjour"
        {
            Console.WriteLine("Bonjour " + i + " fois"); //le "+" met les chaines
bout à bout (concatenation).
        }
        Console.ReadLine(); //Met le programme en pause à la fin de l'affichage.
    }
}
```

Ce qui nous renvoie le code suivant :

```
Bonjour 1 fois
Bonjour 2 fois
Bonjour 3 fois
Bonjour4 fois
Bonjour 5 fois
Bonjour 6 fois
Bonjour 7 fois
Bonjour 8 fois
Bonjour 9 fois
Bonjour 10 fois
```

3 Conclusion

Dans ce chapitre, nous avons abordé des concepts fondamentaux qui nous permettrons de poursuivre sereinement l'apprentissage du C# et de ses notions plus avancées. Nous avons étudié le fonctionnement du Framework .NET, appris à déclarer des variables et a implémenter des structures sélectives (*if else, switch case …*) et itératives (*les boucles*).

Nous avons également vu que le langage C# encore jeune, était un langage qui bénéficiait de fréquentes modifications visant à améliorer ses services et qu'il possédait tous les critères d'un langage d'avenir. En effet c'est un langage, simple et performant, parfaitement adapté à la plateforme .NET.

Dans le prochain chapitre, nous aborderons des notions avancées du langage C# tel que des collections, la gestion des exceptions, ainsi que le concept d'objets et de classes.