C# : L'essentiel en concentré

Sommaire

1 Introduction 3
1.1 Qu'est-ce qu'un langage de programmation 3
1.2 Pourquoi l'orienté objet 3
1.3 Un peu d'histoire 3
1.4 Un exemple 4
1.5 Gestion de la mémoire vive en .NET 5
2 La syntaxe procédurale : logique C/C++ appliquée en C# 5
2.1 Variables, opérateurs numérique et types 6
2.1.1 Les types 6
2.1.2 Les opérateurs 7
2.1.3 Exemple de calculs et de déclaration de variable 7
2.1.4 Les chaines de caractères : quelques détails 8
2.1.5 Les valeurs clefs et les types nullable 9
2.1.6 Le Cast 9
2.2 La condition: if – else – switch/case – opérateur ternaire 10
2.3 Les boucles: while – do – for – goto 12
2.3.1 While 12
2.3.2 Do while 13
2.3.3 For 13
2.3.4 Goto 13
2.3.5 Sortie de boucles : Break-Continue 14
2.4 Array – enum – struct 15
2.4.1 foreach 16
2.5 Gestion des erreurs : try-catch – finally 16
2.6 Instructions préprocesseurs 18
2.7 Le code « unsafe » et « checked / unchecked » 18
3 L'orienté Objet en C# 20
3.1 Introduction 20
3.2 Using 20
3.3 Instanciation 20

3.3.1 L'attribut et le modificateur d'accès 21
3.4 La propriété 22
3.5 Static 23
3.6 Les méthodes 24
3.6.1 Retour sur Main 24
3.6.2 Constructeur / destructeur 25
3.6.3 La surcharge 27
3.6.4 Delegate 28
3.6.5 Les événements 29
3.6.6 Méthodes anonymes et Expressions lambda 31
3.6.7 Méthode d'extension. 33
3.6.8 Itérateurs 34
3.7 L'héritage, le polymorphisme et les interfaces 35
3.7.1 Introduction 35
3.7.2 Exemples d'héritage : les exceptions et throw 36
3.7.3 Redéfinition de méthodes et d'attributs 37
3.7.4 Les interfaces 38
3.7.5 Les attributs 40
4 Conclusion 41

1 Introduction

Comme ce chapitre résume de manière synthétique le langage de programmation, si vous n'avez aucune connaissance en orienté objet, java ou programmation je vous recommande de lire au moins deux fois ce cours pour donner une meilleure cohésion à l'ensemble des connaissances qui en ressortent. Vous verrez qu'il y a beaucoup de références à des sous-parties qui seront traitées en aval. Il m'arrivera souvent de coder plusieurs fois des exemples identiques ou très proche avec des mots clefs différents pour vous permettre de comprendre par analogie et vous éviter de relire l'ensemble de la source avec attention pour comprendre. Dans les premières parties il m'arrivera de dire « fonction » à la place de « méthode », c'est parce que c'est plus parlant pour beaucoup, cet abus de langage volontaire sera éclairci dans la partie réservée aux méthodes.

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

C#, je ne pense pas vous l'apprendre est un langage de programmation. Ces langages ne sont pas le mode de pensé de la machine qui elle ne comprend que des instructions binaires dépendantes du processeur. Ces langages ont pour but d'être plus lisible et plus pratique que l'assembleur pour simplifier la vie des développeurs. Le C et le C++ sont des langages compilés qui sont donc transformés directement en instructions processeur. En .NET, ce n'est pas le cas la compilation donne un code (MSIL/Microsoft Intermediate Language) qui est interprété par un logiciel (interpréteur). Il faut donc retenir que les langages sont des normes de développement qui cherchent souvent un compromis entre le confort, la rapidité de développement et la performance. Ces normes sont accompagnées d'outils tel un compilateur.

Du fait que le code compilé MSIL soit commun au langage .NET utilisé, on peu dire que le C# et le VB.NET sont deux langages de programmation ayant un confort différent mais des performances très similaires.

1.2 Pourquoi l'orienté objet

L'orienté objet à été introduit par le langage SmallTalk (développé par la société Xerox) en 1972. L'orienté objet est un ensemble de règles, de concepts, d'outils du langage de programmation pour rendre le développement plus aisé grâce au groupement d'éléments. Les objets sont donc des entités qui permettent de grouper ensemble des fonctions et des variables en facilitant leurs gestions par groupe et en ouvrant de multiples possibilités. Définir un objet peut paraitre comme ajouter un outil, Le framework .NET apporte lui un set imposant d'objets existants. Un langage Orienté objet n'est pas les objets, mais une syntaxe et des mots clefs pour les manipuler et les créer. Bien que nous allons utiliser des objets du framework ce cour visera à vous apprendre le langage.

L'orienté objet permet la gestion d'ensemble, le groupement, l'isolation des données et des fonctions. Cet ensemble de concepts et d'outils permet par exemple de grouper dans un coin les objets pour accéder aux données distinctement des objets qui servent a faire l'interface avec l'utilisateur et des objets qui servent à appliquer les calculs et procédure logiques de fond. De ce fait nous pourrons disposer d'un code simple et lisible.

1.3 Un peu d'histoire

Le C# est un langage récent apparu en 2001, il est principalement inspiré par le Java (1995) mais aussi par le C++. Ce langage dont le nom se prononce « C-Sharp » à été créé par Microsoft à été normalisé par l'ECMA l'année de sa sortie (puis par l'ISO deux ans plus tard). Il y a eu trois version du C# implémentant toujours quelques fonctionnalités supplémentaires. Le C# étant un produit du framework .NET leurs évolutions sont très liées. Le C# 2.0 est sorti avec le framework 2.0, le C#3.0 lui

bien qu'utilisable sur le framework2.0 arrive nativement et prend vraiment tout son intérêt dans le framework 3.5 avec le LINQ. Pour utiliser la syntaxe du C#3.0 il faut utiliser le compilateur adapté dans un environnement cible 2.0 ou supérieur. Ce compilateur est notamment disponible et implémenté avec VisualStudio 2008.

1.4 Un exemple

Ce tutorial est axé sur le langage et la syntaxe du C#, dans le cas ou vous seriez sous Windows, vous pouvez essayer Visual Studio Express ou SharpDevelop (voire le mode C# de Eclipse). Dans le cas ou vous seriez sous Linux ou Mac, je vous encourage à lire dès à présent les deux premiers tutoriels sur Mono. Il est important de connaitre ses outils et de les avoir fonctionnels pour pouvoir se concentrer sur la logique de programmation.

Remarque : *Pour les inconditionnels de la ligne de commande, ajoutez à votre variable d'environnement Path : « C:\WINDOWS\Microsoft.NET\Framework\v3.5 », vous pourrez compiler les exemples que vous aurez enregistré dans des fichiers texte en faisant « csc.exe monpremiercode.cs » puis exécuter le programme « monpremiercode.exe », vous pourrez aussi ajouter des options de compilation, pour voir celles disponible exécutez : « csc.exe /? » .*

Voici un code affichant dans un terminal « Bonjour ! ». Le code proposé est réduit à sa plus simple expression. Même les projets console par défaut des environnements de développement intégrés sont plus conséquent.
Pour l'instant nous nous concentrerons sur le contenu des accolades qui suivent « Main( ) ». Les accolades sont précédées de mots clefs et noms qui servent à former une structure d'exécution. Cette structure relevant de l'orienté objet est nécessaire à l'exécution mais elle ne sera expliquée seulement qu'après avoir acquis les bases de la logique procédurale.

C#

```
static class Program
{
    static void Main()
    {
        // Dans les exemples qui suivent nous placerons notre code ici
        System.Console.WriteLine("Bonjour !");
    }
}
```

Vous avez remarqué que sur la première ligne de la zone étudiée (la ligne verte) j'ai écrit un texte libre en français. Cette ligne commence par « // ». Ce symbole est un symbole de commentaire qui dit au compilateur de ne pas interpréter la fin de la ligne. Il m'arrivera souvent de vous faire des remarques dans ces commentaires qui sont partie intégrante du tutoriel.

Il existe aussi une autre convention de commentaire qui au lieu d'échapper la fin de la ligne, échappe jusqu'à un symbole de fermeture. Ces commentaires là s'ouvrent avec « /* » et se ferment avec « */ » (vous aurez un exemple dans le prochain extrait de code). Il y a une syntaxe qui permet de générer de la documentation à partir des commentaires, si vous devez travailler en équipe ou pour une entreprise, générer une documentation qui permettra à votre successeur de prendre la relève s'avère important (page sur la documentation du code en C#). Si vous vous intéressez à cette syntaxe, cette page vous expliquera l'essentiel. Cela dit le contenu de cette page contient des données assez techniques de ce fait il pourrait être pertinent d'y jeter un œil seulement après avoir fini de lire ce tutorial.

System.Console.WriteLine est appelé avec « Bonjour ! » en argument, on ajoute à cela un « ; » pour spécifier « exécute ça puis on passe à la suite ». On appelle chaque élément qui suivi d'un « ; » une instruction. Il peut êtres sur une ou plusieurs lignes. Lorsque Main a exécuté la dernière instruction qu'il contient le programme prend fin. Si vous avez lancé graphiquement votre application, la fenêtre console se fermera car la fin du programme entraine la fin de la console. Je sais que ça n'explique pas tout l'extrait de code mais venons y progressivement avec cette architecture en tête. Retenez bien que chaque imbrication de commandes finissant par un « ; » s'appelle « instruction ».

Notez bien que les prochains exemples seront souvent juste la procédure entre les accolades de la fonction Main

1.5 Gestion de la mémoire vive en .NET

Pour comprendre un peu mieux certains phénomènes qui seront abordés plus tard, nous allons expliquer rapidement la gestion de la mémoire par le .NET framework. La mémoire est un point essentiel pour pouvoir faire de la logique. Retenir des résultats intermédiaire pour les réutiliser ou autres. Pour enregistrer temporairement une valeur lui donne un nom. Ce nom sera utilisé comme un lien vers un espace mémoire. Cette association nom, espace mémoire et donnée contenue est un tout que l'on appelle variable. Dans certains langages de programmation comme le C, le C++ ou le C# nous avons des « variables liens ». Ces variables contiennent l'adresse d'une donnée qui éventuellement pourrait ne pas être nommée. Ces variables liens sont en réalité appelées pointeurs, elles sont souvent cachées dans le .NET bien qu'elles soient utilisées au cœur du framework.

.NET sépare la mémoire en deux parties isolées, la pile et le tas. Quand on fait un programme en .NET, le framework nous réserve un espace mémoire. Les autres programmes ne peuvent normalement pas y accéder. Les données dites « type valeur » sont celles qui ont leurs valeurs dans la pile. Les variables de type valeur sont le plus souvent inferieure à 32 octets. Bien que performant pour les petites variables le type valeur est complété par un autre type : le type référence. Les variables de type référence ont un pointeur enregistré dans la pile qui pointe vers les données de la variable. Ces données sont dans la zone mémoire appelée le tas.

Un outil appelé « Garbage Collector » est chargé de libérer et défragmenter dynamiquement la mémoire du tas. Ainsi, si des données n'ont plus de pointeurs associées, la mémoire est libérée. Si les données étaient au milieu du tas, l'espace libéré sera réutilisé par les données du « fond du tas ». Ce procédé permet de redimensionner notre espace mémoire réservé et donc de libérer de la mémoire pour le système et les autres processus.

2 La syntaxe procédurale : logique C/C++ appliquée en C#

Comme nous le disions, la syntaxe du C# est très inspirée du C/C++, on peut même faire de l'arithmétique de pointeurs pour peu que l'on définisse notre code en tant que « Unsafe ». Les personnes ayant de bonnes bases en C/C++ ne verront rien d'extraordinaire dans cette partie qui risque de paraitre dense pour ceux qui partent de zéro.

Remarque : *Si c'est votre tout premier cours de programmation, ce chapitre va vous paraitre dense, je vous encourage à essayer chaque exemple et essayer de prendre du recul avec cette partie dans votre bagage pour mieux revenir sur la partie traitant de l'orienté objet.*

2.1 Variables, opérateurs numérique et types.

Les opérateurs numériques sont la base des calculs et de la logique. La mémoire est un point clef de la programmation. Pour interpréter le code binaire dans une variable et donc la donnée en mémoire il faut en connaitre le type. La déclaration des variables, l'attribution de leurs valeurs et les opérateurs numériques de base sont similaire en C# et en C/C++, voyons comment les utiliser.

Dans une quête de synthèse, j'ai fait peu d'exemple dans cette partie, si vous trouvez dur à comprendre les types ou les opérateurs trop détachez de la pratique avancez avec en parallèle l'exemple en 2.1.3. Tous les opérateurs et tous les types n'y sont pas illustrés. Durant l'ensemble du cours vous verrez l'usage de chaque operateur au moins une fois, mais pour ce qui est des types j'utiliserai principalement des entiers.

2.1.1 Les types

Voici quelques types de base utilisable en C#. Les types présentés dans ce tableau à l'exception de « string » et les structures sont les seuls types à être stockées par valeur.

Classe	***Alias***	***Octets***	***Valeur codées***
System.SByte	sbyte	1	-128 à 127
System.Byte	byte	1	0 à 255
System.Int16	short	2	-32768 à 32767
System.Int32	int	4	-2147483648 à 2147483647
System.Uint32	uint	4	0 à 4294967295
System.Int64	long	8	-9223372036854775808 à 9223372036854775807
System.Single	float	4	$-3,4^{E}+38$ à $3,4^{E}+38$
System.Double	double	8	$-1,79^{E}+308$ à $1,79^{E}+308$
System.Decimal	decimal	16	$-7,9^{E}+29$ à $7,9^{E}+29$
System.Char	char	2	Caractère Unicode (UTF-16)
System.Boolean	bool	4	Valeur `true` ou `false`
System.String	string	*	Chaîne de caractères

Remarque : *Ce tableau est adapté d'un tableau que vous trouverez dans le chapitre 1 des cours sur le Framework, ce chapitre pourrait vous aider à approfondir vos connaissances techniques des bases du typage, de l'orienté objet et vous y trouverez une explication plus détaillée et plus imagée de l'usage de la mémoire vive en .NET.*

Contrairement à certains langages (comme le python) tout objet doit être explicitement typé. Chaque variable a un type défini et on est obligé de spécifier un type pour chaque argument de chaque méthode. Pour allouer de la mémoire à une variable, on crée une instruction avec d'une part le type et d'autre part le nom de la variable. Si vous voulez trouver facilement la valeur maximale et la valeur minimale d'un type numérique, faites « System.Int32.MaxValue » pour retourner le plus grand des int ou MinValue pour trouver le plus petit. Vous n'avez pas encore vu comment faire usage de ses valeurs mais notez que les valeurs limites sont accessible dans le framework et de ce fait vous n'êtes pas tenu de les retenir.

Si on ajoute à une variables « .GetType() », cela retournent le type de la variable. Vous verrez un contrôle de type utilisant cette méthode dans la partie sur la gestion des erreurs.

2.1.2 Les opérateurs

Pour traiter la variable on utilise des opérateurs. Voici les opérateurs dans leur ordre de priorité (on pourra utiliser des parenthèses pour redéfinir les priorités comme en mathématique.) :

Opérateur C#	***Signification***
Calculs	***Ces opérateurs permettent de faire des calculs***
*	Multiplication.
/	Division.
%	Modulo.
+	Addition.
-	Soustraction.
Tests	***Ces opérateurs permettent de retourner des booléens (vrai ou faux)***
is	Teste le type d'un objet (utilisé parti 2.5)
<	Inferieur à.
>	Supérieur à.
<=,>=	Inferieur ou égal, supérieur ou égal.
!=	Différent de.
==	Est égal à
Logique	***Ces opérateurs traitent des variables et en retournent une.***
&&	ET logique (vrai si tous les membres sont vrai)
||	OU logique (vrai si au moins un des membres est vrai)
??	Retourne l'opérande de gauche si non-null ou bien celui de droite. (2.1.3)
cond ***?*** var1 ***:*** var2	Renvoie var1 si cond est vrai ou alors var2 (Voir partie 2.2)
Attribution	***Ces opérateurs permettent d'attribuer une valeur à une variable***
=	Attribution (permet de donner une valeur à une variable).
+=, -=,*=,/=,%=,&=,|=	Permet de contracter la variable de l'opérateur et de l'attribution *.
***Incrémentation ** **	***Ces opérateurs réattribuent à une variable sa valeur modifiée.***
++	Ajoute 1 à la valeur.
--	Enlève 1 à la valeur.

__ :__ Ces opérateurs d'attributions ne servent qu'à faire une contraction pratique. Avec l'opérateur += on ajoutera la valeur actuelle de la variable à la valeur à attribuer. Variable <opérateur>= 12 équivaut à Variable = Variable<opérateur>12. Vous aurez des exemples dans l'extrait de code suivant.*

*__** :__ Les opérateurs d'incrémentation se placent sur une variable dans une instruction seule ou dans un calcul. Si le signe est à gauche de la variable ce sera exécuté en priorité absolue dans l'instruction, s'il est à droite la variable on incrémentera à la toute fin de l'instruction.*

Remarque : *On ne peut pas avoir plus d'un opérateur d'attribution par instruction.*

2.1.3 Exemple de calculs et de déclaration de variable

Voici un exemple d'utilisation :

```
C#

int a;       // on déclare a de type int
int b = 5; //  on déclare b et y attribu la valeur 5
a = 0;     //   on attribu la valeur 0 dans a

System.Console.WriteLine("a=\t"+a.ToString()+"\nb=\t" + b.ToString() + "\n");
/*************************************************************************
```

```
 *  Toutes les variables objets ont la méthode ".ToString()".
 *  Appliquée à un int, ".ToString()" retourne une chaine de
 *  texte contenant la valeur.
 *
 *  L'opérateur "+" entre des string les concatènes c'est à dire les fusione
 *  en les mettant à la suite.
 *
 *  "\n" et "\t" sont des caractères spéciaux : voir partie 2.1.5
 ***************************************************************************/
a += a + ++b;     // b=b+1; a=a+a+b;
System.Console.WriteLine("a=\t"+a.ToString()+"\nb=\t" + b.ToString() + "\n");
a /= b;           // a=a/b
a -= 2*b++*a;    //  a=a-2*b*a; b+=1;

System.Console.WriteLine("a=\t"+a.ToString()+"\nb=\t" + b.ToString() + "\n");
System.Console.ReadKey(); // attend que l'utilisateur appuie sur une touche
```

Retour Console:

```
a=   0
b=   5

a=   6
b=   6

a=   -11
b=   7
```

Cet exemple peut paraitre court, c'est vrais mais ne bloquez pas dessus, si ce n'est pas acquis maintenant, vous verrez des exemples partout dans ce court vu que c'est vraiment la base. Il faut retenir pour la déclaration des variables : le nom du type, le nom de la variable et le caractère de fin d'instruction (« ; »). L'exemple montre que l'on peut attribuer une valeur en même temps que l'on déclare une variable (deuxième ligne).

2.1.4 Les chaines de caractères : quelques détails

Dans le commentaire le plus imposant, on voit que pour les strings il existe des caractères spéciaux dont celui du retour à la ligne et de la tabulation.
Voici quelques caractères spéciaux :

- \n, \r : retour à la ligne
- \t : tabulation (pratique pour les alignements verticaux)
- \\ : permet d'afficher un antislash
- \" : permet d'afficher un guillemet
- \xXX ou « XX »est une valeur hexadécimale de deux caractères (chacun entre 0 et F) : permet d'afficher un caractère par son code hexa.

Si vous ne voulez pas bénéficier du support de ces caractères et avoir une chaine non interprétée, il suffit de précéder la chaine par un @ (pratique pour les adresses de fichier par exemple). Les chaînes précédée d'un arobase peuvent être appelées « chaîne Verbatim », ces chaînes peuvent contenir de réels retours à la ligne.

Au lieu de concaténer les chaines avec le signe « + » on peut ajouter des variables dans les strings avec un moyen simple de formatage.

C#

```
int arg1 =12;
string arg2 = "bonjour";
```

```
ulong arg3 = 1546471534;
Console.WriteLine("arg1={0}\narg2={1}\narg3={2}",arg1,arg2,arg3);
arg2 = string.Format("{0}, {1}, {2}", arg1, arg2, arg3);
Console.WriteLine(arg2);
Console.ReadKey();
```

Retour Console

```
arg1=12
arg2=bonjour
arg3=1546471534
12,bonjour,1546471534
```

Cette méthode permet une visibilité un peu plus claire et peut éviter un grand nombre de « .ToString() ». Les arguments n'attendent pas de type particulier.

2.1.5 Les valeurs clefs et les types nullable

Il existe des valeurs qui sont représentées seulement par des mots clefs. Ces mots clefs sont null, true et false. Pour pouvoir attribuer la valeur null à une variable (de type valeur) on utilise une structure qui contient un booléen et la valeur de type défini(le mot structure est défini partie 2.4). Ce booléen est appelé HasValue. Lorsque sa valeur est à « true » le nullable a une valeur, quand il est à false la variable vaut null.

Cet exemple vous montre les deux méthodes pour faire un nullable.

C#

```
uint? a = 2;
System.Nullable<uint> b = 15;
b = null;
a += b; // a = null + 2 c'est à dire a = null b = 12;
System.Console.WriteLine(b.ToString()+"-*-"+a.ToString()+"-*-"+(a??b).ToString());
System.Console.ReadKey();
```

Retour Console

```
12-*--*-12
```

Si l'on suffixe le type valeur d'un « ? » on obtient ce type mais nullable. On peut aussi mettre explicitement comme type de la variable nullable comme fait pour la variable b (Pour mieux comprendre l'utilité des chevrons qui dans se contexte ne veulent évidement pas dire « supérieur à », vous pourrez voir les classes génériques dans le chapitre traitant de l'instanciation). Ces variables sont susceptibles de contenir null qui correspond à « néant », « rien » (dans certains cas d'utilisation « non défini »). L'intérêt d'avoir une valeur null est le plus souvent pour mentionner que ce n'est pas défini.

L'autre utilisation de null est la des-allocation de mémoire. Dans le cas ou la variable est de type référence, si lui attribue null (exemple : « variable = null ; ») l'adresse stockée par le pointeur dans la pile sera mise à 0. Lorsque le Garbage Collector contrôlera dans le tas la variable, vu que rien ne pointe dessus elle sera supprimée.

2.1.6 Le Cast

Le cast est un moyen d'utiliser une variable qui n'est pas du type à utiliser pour le calcul. C'est à dire le cast permet de retourner une valeur du type attendu à partir de la variable. Ainsi on pourra par exemple caster un int en long pour utiliser une variable n'ayant pas les même limites de dépassement. Il y a deux moyens pour essayer de changer le type d'un objet, un évite les erreurs quand on n'est pas sur que l'objet puisse être casté et l'autre retype ou renvoie une erreur ou une

valeur aberrante (pour comprendre quand on a une erreur ou une valeur aberrante regardez la partie sur checked/unchecked).

Le mot clef « as » doit caster vers un type nullable ou référence (voir partie 2.7). Si la variable est du type requis ou « compatible », as retourne la valeur typée. Au cas contraire il retournera null. Le « as » est traduit avec d'autres opérateurs dans l'exemple. Pour rappel, l'opérateur « ?? » sert à retourner une autre valeur si le premier élément est null, il est parfois utilisé avec as pour avoir un retour en cas d'impossibilité de cast différent de null.

Le cast classique consiste à imposer à une variable de changer de type, si ce changement de type ne fonctionne pas une exception est levée.

C#

```
int a = int.MaxValue;
long b = long.MaxValue;
long c = 15;

// cast implicite
object o = c;

// cast explicite necessaire
a = (int)b;         //lève une overflowexeption si les dépassements sont contrôlés

// cast implicite
b = a;              //fonctionne

// usage de as et de l'opérateur ??
b = (o as long?) ?? 0 ;
// équivau à l'instruction:
b = ((o is long?) ? ((long?) o) : ((long?)null) ) ?? 0;
// l'opérateur ternaire sera détaillé dans la partie qui suit.
// ici, seuls "long" et "long?" fonctionnent pour le type de c

Console.WriteLine(b);
System.Console.ReadKey();
```

2.2 La condition: if – else – switch/case – opérateur ternaire

Le mot clef if (en français « si ») permet l'exécution conditionnelle de code. La condition soumise à if doit retourner un Booleen (True ou False), nous utiliserons souvent ici des opérateurs de test. Ensuite nous passons la procédure à exécuter si la condition est vraie entre accolades.

Souvent, et c'est vrai avec beaucoup de mots clefs, quand les accolades ne paraissent pas à la suite de la condition, le mot clef s'appliquera uniquement sur l'instruction qui suit.

Le mot clef else (sinon) est toujours précédé d'au moins un if. else comme if est suivi de code entre accolades. Ce code sera utilisé seulement si la dernière instruction if n'a pas eu à exécuter le bloc. Si l'on doit procéder à une série de test ou un seul doit être exécuté, vous verrez une suite du genre « if, [else if, else if…], else », seule la procédure associée de la première condition vraie sera exécutée.

Le mot clef switch permet seulement de contrôler différentes valeurs que peut prendre une variable ou expression et d'agir en fonction. Les propositions ne peuvent pas être des variables, il ne peut pas y en voir 2 identiques, tous les types de valeurs ne peuvent y être utilisés et pour ce qui est des performances le switch est la moins optimisée des 3 solutions. Néanmoins certains développeurs trouvent sa syntaxe plus lisible et plus pratique que l'imbrication de « if ».

La syntaxe de l'opérateur ternaire est la suivante : « (booléen) ? retourSiVrai : retourSiFaux ». On peut bien évidemment en imbriquer plusieurs pour cumuler plusieurs conditions. Mais il faut noter que l'opérateur ternaire permet un retour conditionnel et non pas une exécution conditionnelle, c'est-à-dire qu'il ne peut rien exécuté et qu'il se place dans des expressions comme une valeur.

```
C#

int a; string b;
System.Console.WriteLine("Entrez 2");
a = int.Parse(System.Console.ReadLine());
/* int.Parse prend le texte entré par l'utilisateur (Readline())
 * et en retourne la valeur indiquée si ce texte contien un nombre.
 */

// Méthode if/else
if (a == 2)
{
    a = a;          // a=a est une instruction inutile à titre d'exemple
    b = "normal" ;  // on peut mêtre plusieurs instructions
}
else if (a == 1 || a == 3)
    b = "Pas loin"; // on peut se passer d'acollade pour une instruction seule

else                // on aurai pu remplacer "else" par "if(a < 1 || a > 3 )"
{
    b = "bizarre";  // une instruction seule passe même avec les acollades
}
// Méthode switch/case
switch (a)
{
    case 2:                 // le switch saute à cette condition si a vaut 2
        b = "Bien Joué !"; // puis exécute ses instruction
        break;              // Le break fait sortir du switch
    case 1:                 // en C# : pas de break = pas d'instructions
    case 3:                 // le cas 1 vien exécuter les instructions du cas 3
        b="Pas loin";
        break;              // donc on sort du switch
    default:                // default représente tous les autres cas
        b = "bizarre";
        break;
}
// Méthode opérateur de retour conditionel
b= (a == 2) ? "normal" : (a == 1 || a == 3) ? "Pas loin" :"bizarre" ;

System.Console.WriteLine(b);
System.Console.ReadKey();
```

Cet extrait de code montre trois manières de faire la même chose, Le programme demande à l'utilisateur d'entrer une valeur qui est affecté dans « a ». A la fin on affiche la chaine de caractère « b ». Entre trois façons pour mettre dans « b » :

- la chaine « Bien joué ! » si « a » est égal à 2
- « Pas loin » si « a » est égal à 1 ou 3
- « bizarre » si « a » ne répond à aucun de ces critères.

Comme expliqué antérieurement, on constate que suite au « if » il y a une condition elle-même suivie d'une ou plusieurs instructions. Dans le cas de l'exemple ci-dessus si « a » est égal à 2 « a==2 » retourne vrai ce qui engendre l'exécution de l'affectation de « Bien joué ! ». Lorsque l'on

utilise le mot clef if juste après le « else » c'est pour faire une liste de test, le premier vrai sera le seul exécuté.

Le switch permet donc ici de tester les cas qui nous intéressent individuellement c'est-à-dire 1, 2 et 3. Vu que le cas 1 et le cas 3 ont le même traitement on peu les réunir. Le mot clef break doit être mis en fin d'exécution avant la nouvelle valeur du switch. Si vous voulez plus de renseignement sur le switch ou en cas de problèmes avec ce mot clef ou si vous voulez voir les différences entre le switch C et le switch C#, je vous recommande cet article francophone.

Imbriqué, l'opérateur ternaire n'est pas un cadeau pour ce qui est de la lisibilité. Pour rendre plus clair à l'œil, nous aurions pu mettre des parenthèses autour de la sous expression ternaire ou la mettre à la ligne. Vous avez néanmoins pu constater l'aspect pratique de cet opérateur qui est celui qui a eu la syntaxe la plus rapide, ça n'aurai pas été pour l'exemple, j'aurai certainement mis l'expression directement en argument de la méthode WriteLine sans passer par « b ». Bien qu'il puisse paraitre pratique et assez optimisé pour ce qui est des performances, l'opérateur ternaire va par son manque de lisibilité à l'encontre des principes des langages récents qui visent le confort de développement et la lisibilité. Il sera principalement utilisé en argument de fonction vu qu'on ne peut pas vraiment y mettre des exécutions conditionnelles.

2.3 Les boucles: while – do – for – goto

Les boucles font parti des fondamentaux de la logique de la programmation, notre prochain exemple ressemblera enfin à quelque chose vu que l'on aura vu après ça l'essentiel de la partie structure logique des instructions dans une procédure.

2.3.1 While

While (en français « tant que ») permet de faire une boucle conditionnelle, probablement le type de boucle que vous croiserez le plus avec les boucles for. Niveau structure il ressemble au if, il a une condition entre parenthèses et un block de code nécessairement entre accolade pour peu qu'il ait plusieurs instructions. Si la condition du while est vraie le block de code est exécuté, à la fin de cette exécution, le block est exécuté à nouveau si la condition et vrai et ce jusqu'à ce que ce ne soit plus le cas.

```
C#

int a=2;
int b=15;
int temp;
while (b > 0)
{
    if (a == 1)
        a = 2;
    else a = 1;
    System.Console.WriteLine("Tour du joueur "+a.ToString()
        +"\nil reste "+b.ToString()
        +" allumettes\nprenez un nombre d'allumettes entre 1 et 3 (defaut 3)");
    temp = int.Parse(System.Console.ReadLine());
    if (temp > 0 && temp < 4)
        b-=temp;
    else b-= 3;
}
System.Console.WriteLine("joueur "+a.ToString()+" a perdu");
System.Console.ReadKey();
```

On peut voir dans cet exemple que tant qu'il reste des allumettes, les joueurs peuvent en prendre jusqu'à ce qu'il y en ait plus moment ou on sortira de la boucle pour afficher le nom du

perdant. Le fonctionnement est simple et commun aux autres. Il suffit de maitriser un type de boucle pour facilement comprendre les autres.

2.3.2 Do while

Très similaire au while normal, il impose un minimum d'une exécution avant de contrôler les conditions de maintien de la boucle.

```
C#

int a=0;
do
{
    System.Console.WriteLine(a++.ToString());
} while (a > 0 && a < 12) ;
System.Console.ReadKey();
```

Le do-while est l'équivalent du while mais la syntaxe et différente et le contrôle de condition se fait à la fin de la boucle. De ce fait le block est exécuté au moins une fois même si la condition est fausse en premier lieu. Dans ce cas « a » = 0, il exécute le bloc une première fois alors que la condition est fausse. Ensuite jusqu'à ce que « a » ne respecte plus la condition il l'incrémente et l'affiche. (Affiche les nombres de 0 à 11 inclus).

2.3.3 For

La boucle for est souvent très pratique mais n'est pas toujours aimée des débutants, c'est un while avec deux champs supplémentaires : une variable locale et une exécution. Le tout est dans les parenthèses du for, nous y verrons nécessairement deux « ; » pour séparer les 3 champs. Refaisons l'exemple du jeu des allumettes avec une boucle for :

```
C#

int a = 2;
int temp;
for (int b = 15; b > 0; b-=(temp>0 && temp<4)?temp:3 )
{
    if (a == 1)
        a = 2;
    else a = 1;
    System.Console.WriteLine("Tour du joueur " + a.ToString()
        + "\nil reste " + b.ToString()
        + " alumettes\nprenez un nombre d'allumettes entre 1 et 3 (defaut 3)");
    temp = int.Parse(System.Console.ReadLine());
}
System.Console.WriteLine("joueur " + a.ToString() + " a perdu");
System.Console.ReadKey();
```

La boucle for comme vous pouvez le constater met en valeur une variable locale qui sera le plus souvent utilisée dans la condition de la boucle (donc son instanciation), la condition même puis une instruction qui met en valeur le traitement fait à chaque itération (a part la première), vous verrez souvent « for(int a = 1 ; a<=10 ;a++){ //instructions ;} » pour les boucles qui doivent se répéter 10 fois.

2.3.4 Goto

Le goto est une alternative bien qu'à éviter qui permet des choses assez étonnantes. Le goto fait un saut jusqu'à un endroit nommé n' importe où dans la procédure. Pour nommer un endroit on

parle de label ou d'étiquette, on met son nom suivi de « : ». J'ai mis la méthode Main dans cet exemple pour montrer que par convention, ces labels sont indentés en retrait sur la gauche. La condition boucle réalisé avec if. Dans ce contexte le goto et le label sont à proscrire du fait qu'un do while permet la même chose. Il est important de noter que le goto peut aussi faire un saut vers un label qui est en aval dans le code ce qui peut permettre de sortir par exemple d'une imbrication de boucles.

C#

```
static void Main()
{
    int a = 2;
    int temp=0;
    int b = 15;

nouveauTour:
    if (a == 1)
        a = 2;
    else a = 1;
    System.Console.WriteLine("Tour du joueur " + a.ToString()
        + "\nil reste " + b.ToString()
        + " alumettes\nprenez un nombre d'allumettes entre 1 et 3 (defaut 3)");
    temp = int.Parse(System.Console.ReadLine());
    if (temp > 0 && temp < 4)
        b -= temp;
    else b -= 3;
    if (b > 0)
        goto nouveauTour;

    System.Console.WriteLine("joueur " + a.ToString() + " a perdu");
    System.Console.ReadKey();
}
```

Remarque : *Le goto est mal aimé de beaucoup de programmeurs car goto ne respecte pas la structure logique de la pensée de de l'indentation et du coup c'est moins lisible que les autres boucles. Certains programeurs vont jusqu'à interdire l'utilisation du goto ou développer des langages n'implémentant pas ce type de boucle. De ce fait dans la mesure du possible il faut privilégier l'utilfisation d'autres types de boucles.*

2.3.5 Sortie de boucles : Break-Continue

Il peut arriver dans le cas ou les boucles effectuent plusieurs opérations que l'on veuille sortir directement de la boucle ou sauter l'exécution de la fin du block, pour ça il existe les mots clefs break et continue. Le mot clef break permet de sortir brutalement de la boucle tandis que continue renvoie au test. Il va sans dire que ces deux instructions ne fonctionnent pas dans le cas de boucle faites au goto.

C#

```
int a = 2;
int temp=0;
int b = 15;
while(true)
{
    if (a == 1)
        a = 2;
    else a = 1;
    System.Console.WriteLine("Tour du joueur " + a.ToString()
        + "\nil reste " + b.ToString()
        + " alumettes\nprenez un nombre d'allumettes entre 1 et 3 (defaut 3)");
    temp = int.Parse(System.Console.ReadLine());
```

```
    if (temp > 0 && temp < 4)
        b -= temp;
    else b -= 3;
    if (b > 0) continue;
    break;
}
System.Console.WriteLine("joueur " + a.ToString() + " a perdu");
System.Console.ReadKey();
```

Evidemment ce cas est plus un cas d'étude qu'un cas pratique, vous aurez un autre exemple aussi explicite mais montrant plus l'intérêt de ces mots clefs dans la partie sur la gestion des erreurs.

2.4 Array – enum – struct

Pour grouper des variables ou des valeurs il y a plusieurs outils, les trois outils que je vous présente ici sont très différents.

- Struct : défini un type de variable à plusieurs champs.
- Enum : organise des valeurs clefs
- Les arrays sont des variables déclarées en tant que liste de valeurs de type commun.

L'exemple propose un semblant d'organiseur de tâches et fait la somme de la durée des événements de la semaine. enum et struct sont pour faire des déclarations qui seront en dehors de Main.

enum et struct forment des éléments, une fois définis on ne peut plus les modifier. Dans la structure (struct) on définie chacune de ses sous variables et leur types comme si on déclarait des variables. Vous remarquerez le mot « public ». La structure est en réalité très proche de la classe (voir chapitre 1 sur le framework) mais n'est en général utilisé que pour ranger un petit nombre de variable.

```
C#

class truc
{
struct evenement
{
    public int[] jours;
    public string tache;
    public int duree;
}
enum semaine
{lundi=1, mardi, mercredi, jeudi, vendredi, samedi, dimanche}
enum longueurTemps
{tresCourt=15,court=30,moyen=60,assezlong=90,deuxheures=120,treslong=240}

static void Main()
{
    int temp=0;
    evenement[] cal = new evenement[]
    {
        // La syntaxe utilisée ci-dessous pour remplir les évènements = C#3.0
        new evenement{
            jours= new int[] {(int)semaine.dimanche,(int)semaine.mercredi},
            tache="faire la vaiselle",
            duree=(int)longueurTemps.court},
        new evenement{
            jours=new int[7], tache="lire ses
            flux RSS",
            duree=(int)longueurTemps.assezlong},
        new evenement{
```

```
            jours = new int[] {(int)semaine.lundi, (int)semaine.mercredi},
            tache = "prendre un bain",
            duree = (int)longueurTemps.moyen}
    };

    cal[1].tache += " et suivre quelques liens";

    for (int i=0; i < cal.Length; i++)
    {
        temp += cal[i].duree * cal[i].jours.Length;
        System.Console.WriteLine(((double)temp/60).ToString()+
            " heures de ta semaine occupée (tâche ajouté: " + cal[i].tache+" )");
    }
    System.Console.ReadKey();
}
}
```

Retour Console:

```
1 heures de ta semaine occupée (tâche ajouté: faire la vaiselle )
11,5 heures de ta semaine occupée (tâche ajouté: lire ses flux RSS et suivre
quelques liens )
13,5 heures de ta semaine occupée (tâche ajouté: prendre un bain )
```

Les principaux points à retenir ici sont :

- Lorsque l'on appelle l'enum il faut caster pour avoir le type désiré
- Les arrays sont à taille statique, vous ne pourrez pas ajouter d'éléments, pour faire des sortes d'« arrays dynamique » on utilisera une classe générique (évoqué dans le chapitre 3.7 sur l'instanciation).

2.4.1 foreach

Vous avez constaté que ma boucle for me permettait de parcourir la liste « cal » avec la variable i qui me sert d'index. De ce fait je traite tour à tour chaque élément de l'array. Pour faire cette démarche plus lisiblement et gagner en taille de code j'aurai pu refaire ma boucle comme suit :

C#

```
foreach (evenement e in cal)
{
    temp += e.duree * e.jours.Length;
    System.Console.WriteLine(((double)temp/60).ToString()+
        " heures de ta semaine occupée (tâche ajouté: " + e.tache+"
)"+semaine.lundi.GetType());
}
System.Console.ReadKey();
```

La liste cal est la même que dans l'exemple précédent, on fait passer tour à tour dans la variable locale chaque élément e. Au final on exécute les mêmes instructions qu'avec la boucle for dans l'exemple antérieur. Notez le mot clef « in » qui n'apparait que dans ce contexte. Pour qu'un foreach soit utilisable sur un objet ce dernier doit avoir un comportement adéquat (voir la partie sur les itérateurs).

2.5 Gestion des erreurs : try-catch – finally

Comme vous l'avez peut être déjà constaté, si vous entrez une chose qui n'est pas assimilable à du texte dans le jeu qui nous sert d'exemple, nous avons une erreur. Pour gérer les erreurs nous pouvons isoler le code propice à l'erreur et l'isoler dans un bloc de code « try ». Nous pouvons regarder la documentation de la méthode propice à l'erreur pour voir les différents types d'erreurs

qui peuvent advenir en ce contexte. La méthode propice à l'erreur dans notre code est « Parse », la surcharge prenant en argument un string : les exceptions de Parse(string) sur MSDN. Pour pouvoir traiter l'erreur, on va ranger le code en cas d'erreur dans un bloc « catch ». Si on met catch seul (sans arguments ni parenthèses) le traitement sera le même quelque soit l'erreur et on ne pourra pas retourner d'informations relatives aux erreurs. En mettant les parenthèses et un argument général (comme dans notre exemple) on a une variable interne au catch (« e ») qui contiens les données de l'erreur et qui permet de es afficher et de réagir.

Les exceptions sont des classes qui seront défini en exemple sur l'héritage. Exception regroupe toutes les catégories c'est-à-dire tous les types d'exceptions. Dans notre exemple nous comparerons les types de l'exception à ceux que l'on a trouvés sur MSDN pour voir la comparaison des types. Nous retraiterons les erreurs dans la partie « Exemple d'héritage ».

Le mot clef finally le plus souvent utilisé pour libérer la mémoire est à placé après le try et le catch. Ce block sera exécuté qu'il y ait erreur ou non.

Remarque: *voici quelques principaux attributs communs à toutes les Exceptions permettant d'obtenir des informations pour signaler, trouver et corriger les bugs :*

- *Message est un String décrivant la cause de l'erreur de manière relativement explicite pour un utilisateur.*
- *HelpLink retourne l'adresse (ex :URL) du fichier d'aide associé au type de l'exception.*
- *Source donne le nom de la référence, l'assembly ou du fichier source qui est à l'origine.*
- *TargetSite permet d'avoir le type de retour de la méthode qui à levé l'erreur et le type des arguments.*
- *StackTrace donne la ligne de l'erreur et remonte du Main à la méthode qui lève l'exception et fourni la ligne.*

C#

```
uint a = 2;
uint temp = 0;
uint b = 15;

while (b > 0)
{
    if (a == 1)
        a = 2;
    else a = 1;
    System.Console.WriteLine("Tour du joueur " + a.ToString()
        + "\nil reste " + b.ToString()
        + " alumettes\nprenez un nombre d'allumettes entre 1 et 3\n(defaut 3,
nombre négatif pour abandoner)");
    try
    {
        temp = uint.Parse(System.Console.ReadLine());
    }
    catch (System.Exception e)
    {
        System.Console.WriteLine(e.Message);
        if (e.GetType().Equals(typeof(System.ArgumentNullException)))
            System.Console.WriteLine("impossible: retour minimal= \"\" (non-
null)");
        else if (e is System.OverflowException)
            break;
        else System.Console.WriteLine("message non parsable");

        if (a == 1) a = 2;
        else a = 1;
        continue;
```

```
    }
    if (temp < 4)
        b -= temp;
    else b -= 3;
}

System.Console.WriteLine("joueur " + a.ToString() + " a perdu");
System.Console.ReadKey() ;
```

Remarque : *typeof est un mot clef qui prend en argument une classe et qui retourne un type, ici il nous permet de savoir si le type de l'erreur « e » relève bien des classes en argument.*

2.6 Instructions préprocesseurs

Je ne mettrai pas d'exemple pour l'instruction préprocesseur, c'est assez peu utilisé bien que ce soit la seule manière pour pouvoir définir des instructions à compilation conditionnelle. En général ce ne sera pas utilisé, d'autan que contrairement au C/C++ classique, « #define » ne permet pas de faire de macros. Une liste des mots clefs des instructions préprocesseurs est disponible à cette page :

http://msdn.microsoft.com/fr-fr/library/ed8yd1ha(VS.80).aspx

2.7 Le code « unsafe » et « checked / unchecked »

Cette partie aura du sens particulièrement pour ceux qui ont déjà manipulé des pointeurs en C/C++, bien que cela permettes des choses assez puissante, on peu parfaitement se passer de ce mode à part pour le hacking. Le mode unsafe (pourrait être traduit par non sûr) permet de manipuler directement les adresses mémoire et donc les pointeurs. Nous allons évoquer ici quelques mots clefs : unsafe, stackalloc, fixed.

Pour pouvoir compiler du code unsafe, vous devez le spécifier, depuis Visual Studio sélectionnez : projet > propriétés du projet, vous devrez cocher une checkbox dans l'onglet générer (projet>options>Options du compilateur pour MonoDevelop ou le paramètre de commande /unsafe dans le terminal) pour spécifier explicitement votre autorisation.

Ces déplacements de données étant gênants si l'on utilise des adresses mémoire, on a deux moyens de solutionner le problème : stackallock (rappelle un peu le malloc) qui permet d'allouer de la mémoire dans la pile et fixed qui empêche au garbage collector de déplacer certaines données du tas.

Illustrons par un exemple utilisant les pointeurs :

C#

```
unsafe static void Main()
{
    int* variable = stackalloc int[100];
    // stackalloc ne s'utilise que pour des arrays types valeur
    for (int i = 0; i < 100; i++)
    {
        *variable++ = i;
        Console.WriteLine( (int)variable );
    }
    variable-=100;
    for (int i = 0; i < 100; i++)
        System.Console.WriteLine(*variable++);
```

```
    char[] texte = new char[] {'t','a','m','t','l','e','m','o','u','s','s','e'};
    fixed (char* pointeur = &texte[0])
    {
        pointeur[3] = 'p';
        *pointeur = pointeur[3];
        for (int i = 0; i <= texte.Length; i++)
            System.Console.Write(pointeur[i]);
    }
    System.Console.ReadKey();
}
```

int* est un type de donnée pointeur. C'est un pointeur ver un objet de type int.
*variable correspond au contenu de la variable. C'est la donnée à proprement parlée.
variable correspond à l'adresse mémoire de la variable.
&variable correspond à l'adresse mémoire de la variable.

Stackalloc est l'équivalent de _alloca de la bibliothèque runtime C, il a alloué 400 octets et retourné un pointeur. Derrière on peut manipuler le pointeur comme en C. fixed permet de faire un pointeur fixe, la variable « pointeur » ne peut être modifié ni par notre code ni par le Garage Collector. Unsafe défini des blocks de code, il peut être mis seul avant un block comme try (unsafe{ /*instructions*/ }) ou il peut s'ajouter aux spécifications d'un block existant comme dans l'exemple.

Le code mis en « unchecked » (un calcul ou un block) est un peut plus performant mais les résultats peuvent changer, unchecked consiste à ne plus contrôler les dépassements de valeurs. De ce fait le comportement est de retourner à la valeur minimale quand on est trop grand et inversement. (Application des opérations binaire avec mise à l'écart des retenues). Par défaut avec Visual Studio tous les codes sont en unchecked, de ce fait, pour que ces blocks aient vraiment un sens, il faut avoir le contrôle de dépassement (checkbox dans « projet>propriété du projet>générer>options avancées » dans Visual Studio ou sous MonoDevelop dans « projet>options>configuration>debug>Options du compilateur »).

C#

```
int var;
unchecked
{
    var = int.MaxValue;
    var *= 2;
    System.Console.WriteLine("\n" +var.ToString() + "\n" + ((int)(int.MaxValue
+1)).ToString());
}
System.Console.ReadKey();
```

Retour Console:

```
-2
-2147483648
```

Si vous n'avez pas dans les options du compilateur le contrôle de dépassement de coché par contre vous pouvez l'activer sur un calcul ou un block avec le mot clef « checked ».

C#

```
int a = int.MaxValue;
int b = 12;
b=checked(a * b); // ce code génèrera ici une erreur de type OverflowExeption .
Console.WriteLine(b);
```

3 L'orienté Objet en C#

3.1 Introduction

Comme nous le disions dans l'introduction, l'orienté objet est un moyen de regrouper des variable pour pouvoir traiter des ensemble comme des entités propres. C# est un langage fortement orienté objet. Les objets correspondants à quelque chose, il faut les définir avant d'en utiliser. Par chance le framework .NET propose déjà une belle liste d'objets. La « définition » est appelée la classe et la variable contenant l'objet est appelées l'instance. Avant de s'amuser à instancier n'importe quoi nous allons voir comment on crée des classes simples (qui rangent des variables), nous verrons ensuite comment utiliser ces classes..

3.2 Using

Dans la partie 2 vous avez certainement trouvé que l'on utilisait beaucoup « System. ». System est un NameSpace que l'on peut aussi appeler « espace de nom ». Les NameSpaces servent à ranger les classes un peu comme dans des dossiers. Si l'on veut s'économiser d'écrire System à chaque fois que l'on veut accéder à un objet de ce NameSpace, il suffit de mettre au début du fichier « using System ; ». Ces imports d'espaces de nom permettent d'accéder directement à tous les éléments contenus (appart les NameSpaces qu'il contient) sans avoir à mentionné ou il est rangé. Plus vous ferrez de using plus vous aurez de propositions à l'auto complétion, d'où l'intérêt de limiter l'usage des using car cela permet une auto complétion plus rapide et plus pertinente.

3.3 Instanciation

Nous allons définir des classes mais pour savoir comment on les instancie nous allons d'abord regarder cette classe intégrée dans le Framework que nous avons déjà utilisée avec une instanciation simple : le String

C#

```
System.String var= new System.String(new char[] {'l','e','t','t','r','e'});
```

Décomposons cette instruction. Nous avons dans un premier temps le type et le nom de la variable pour la déclaration. En suite nous avons le mot clef new, on utilise à nouveau le type que l'on appelle comme une fonction. Cette notion de fonction et d'argument à la déclaration seront expliqués dans la partie 3.8.2. Nous attendons aussi un argument. Dans cet exemple l'argument est un Array de char, comme on compte en mettre un qui n'existe pas, qui n'est pas déclaré, bref un nouveau comme pour le String on met le mot clef new et on met de la donnée.

Remarque : les classes anonymes sont une nouveauté de C# 3.0, si vous développez sur un logiciel existant, il vous faudra spécifier de compiler avec la syntaxe C# 3.0 pour pouvoir faire ce qui vas suivre. Le code C#2.0 compilera normalement en C#3.0 sans modifications.

Parfois on doit créer une classe pour ranger des variables, mais pour une utilisation locale et qui ne sortira pas de la méthode en court. Dans ce cas il existe une syntaxe de déclaration de classe précise qui génère des « Classes anonymes ». Les classes anonymes ont un nom de classe généré à la volée sans informer le développeur de ce fait on ne peut pas spécifier explicitement le type de la variable qui vas contenir cette classe. Pour déclarer une variable dont on ne connait pas le type, il existe un mot clef approprié : « var ». Le mot clef « var » permet de laisser le compilateur typer la variable à notre place. Une classe anonyme se remplie un peu comme un array, exemple :

C#

```
var classeanonyme = new { a=1, b=12,
                          c="texte",
                          d= "quelquechose"};
Console.WriteLine(classeanonyme.b.ToString()+classeanonyme.c);
```

Comme dit précédemment, le framework est un ensemble de classes organisées dans des espaces de noms, pour l'instant nous avons uniquement utilisé des éléments de System. Je vais vous présenter des objets de System.Collection.Generic. Ce NameSpace contient de quoi faire des listes dynamiques et de type de votre choix, ce type doit être nommé. Le type des éléments contenus dans la liste doit correspondre au type entre les chevrons.

C#

```
public List<int> liste = new List<int>();
liste.Add(12);
```

Ce type de liste vous permet de ranger, ajouter, enlever, modifier, trier et chercher les éléments. Vous pourrez utiliser ces listes pour ranger des objets de même type ou de type de même forme. De ce fait pour faire des listes de types anonyme, bien qu'il y ait des astuces, que ce soit décompiler les assemblies pour connaitre le nom de la classe anonyme ou autre elles s'avèrent relever de la bidouille et ne sont pas explicites à la lecture du code.

3.3.1 L'attribut et le modificateur d'accès

On a défini le modèle objet comme un moyen conceptuel pour organiser des variables et les traiter par groupe. Certaine variables internes aux objets ne requièrent pas d'interactions depuis d'autres parties applicatives. Des accès plus restreints entrainent plus de facilité pour repérer les erreurs, en réduire le nombre, une auto complétion plus pertinente, plus de facilité si quelqu'un doit utiliser la classe et de cibler la documentation sur les variables, méthodes et propriétés accessibles.

Les attributs dans les classes comme dans les structures doivent avoir de défini un degré d'accès. Ces niveaux d'accès sont définis par un de ces quatre mots clefs:

- Public : donne tout l'accès à toutes les méthodes de toutes les classes.
- Private : Restreint l'accès à la classe même, il sera souvent préférable d'utiliser Protected.
- Protected : Comme Private, la différence étant mince elle sera expliquée dans la partie sur l'héritage.
- Internal : internal retire l'accès à toutes les méthodes et tous les éléments qui ne partagent pas la même assembly (qui ne font pas parti du même fichier compilé).

Ces modificateurs sont communs pour les classes, les méthodes, les attributs, les accesseurs, c'est donc important à connaitre et vous aller les croiser souvent (surtout les trois premiers).

L'attribut est donc la variable simple dans la classe, vous en avez déjà vus dans la structure. Refaisons en Partial une petite déclaration d'attribut que l'on va instancier.

C# : fichier 2

```
using System;

namespace ConsoleApplication
{
    partial class Class
    {
        public int a =12;
        public const ulong constante = (ulong) 128;
```

```
    }
}
```

C# : fichier 1

```
using System;
using System.Collections.Generic;
namespace ConsoleApplication
{
    partial class Class
    {
        public string b = constante.ToString();
        public List<int> liste = new List<int>();
    }
    static class Program
    {
        static void Main()
        {
            Class var = new Class();
            var.liste.Add(var.a);
            var.b += "  " + var.liste[0].ToString();

            Console.WriteLine(var.b);
            Console.ReadKey();
        }
    }
}
```

La variable déclarée avec le mot clef « const » doit être attribuée dans la même instruction et ne pourra plus être modifiée. On verra d'autres moyens qui peuvent paraitre plus pratique pour limiter en écriture les variables.

3.4 La propriété

Lorsque l'on a des variables assimilées à des objets, elles peuvent avoir des contraintes autres que les contraintes informatiques, pour éviter qu'une variable enfreigne ces contraintes, on peut en définir à l'attribution. De plus, on peut vouloir contrôler l'accès en lecture de certaines variables ou même les mettre dans un format plus propice que celui du fonctionnement interne à l'objet.

Voici un exemple de code, notez bien les mots clefs « get », « set » et « value ».

C#

```
class Class
{
    protected uint _var = 0 ;
    public uint a
    {   get
        {   return _var+1; }
        set
        {   if (value < 100)
                _var = value;
            else
                _var = 100; }
    }
}
class Program
{
    static void Main()
    {
        Class objet = new Class();
        for (int i=0; i < 120; i++)
        {
            objet.a = i;
```

```
            Console.WriteLine(objet.a);
        }
        Console.ReadKey();
    }
}
```

Ainsi, quand on appelle l'attribut « a » d'un objet de classe « Class », en réalité on accède à _var. Ce procédé sécurise l'application du fait que _var soit en « protected » et du coup inaccessible par du code ne venant pas de la classe. Dans notre exemple, on empêche à la variable de dépasser 100 en valeur interne, elle pourra retourner entre 1 et 101. On pourrait tout à fait mettre plusieurs instructions dans le get (qui correspond à l'accès en lecture de la variable spéciale). On peut faire un accesseur avec seulement get ou seulement set pour l'avoir en lecture seule ou écriture seule. Les attributs ne sont pas forcément liés à une variable interne, elle peut tout a fait manipuler plusieurs ou aucune variable (dans ce dernier cas l'intérêt sera limité par rapport à une méthode).

Comparativement au mot clef « const » qui empêche toute écriture, le couple variable privée/get (sans set) permet de modifier en interne les valeurs.

3.5 Static

Normalement pour accéder à un attribut, une propriété ou une méthode, on doit forcément créer un objet de la classe pour pouvoir l'utiliser. Le mot clef « static » permet d'avoir des variables et méthodes utilisable non pas d'une instance mais de la classe elle-même

C#

```
class Class
{
    static protected int _var = 0 ;
    static public int attribut
    {
        get
        {
            return _var+1;
        }
        set
        {
            if (value < 100)
                _var = value;
            else
                _var = 100;
        }
    }
}
class Program
{
    static void Main()
    {
        for (int i=0; i < 120; i++)
        {
            Class.attribut = i;
            Console.WriteLine(Class.attribut);
        }
        Console.ReadKey();
    }
}
```

Si je veux une classe qui serve de « boite à outils » je peux créer des méthodes statiques qui ne nécessitent pas que la classe soit instanciée. Si on déclare la classe entière en statique, cela entraine qu'on ne pourra pas en faire d'instances et qu'il n'y aura pas de constructeur. Tous les éléments d'une classe statique doivent être spécifiés en statique.

Remarque : Les classes statiques sont un moyen simple de faire un Singleton. Un singleton est un objet qui ne peut être instencié qu'une seule fois, ici la classe statique ne peut avoir qu'une « instance ».

Dans le cas d'une classe statique, il faudra utiliser private au lieu de protected, ce sera détaillé dans la partie sur l'héritage.

3.6 Les méthodes

Le mot fonction vous paraitrait certainement plus clair et c'est pour ça que jusqu'à ce point du chapitre il m'arrivait de l'employer, mais les fonctions qui sont liés à des objets ont une autre appellation : ce sont des méthodes. Pour l'instant nous n'avons vu que la méthode Main qui sera traité et expliquée en premier lieu. Pour comprendre les méthodes il faut savoir qu'elles sont définies par : un degré d'accès, un nom, un type de retour et des types et un nombre (pouvant être nul) d'arguments. Cette définition est appelée « prototype » ou « signature » et a une syntaxe particulière. Dans cette partie essentielle sur les méthodes, nous allons :

- commencer par expliquer la méthode Main, nous allons aussi voir comment passer des paramètres à l'exécution, comment une méthode retourne une valeur… Bref l'essentiel de ce que l'on a à comprendre sur les méthodes sera expliqué sur la méthode Main que l'on a beaucoup utilisée sans vraiment l'expliquer.
- Nous verrons ensuite comment créer une méthode qui permette de générer proprement le contenu de la classe avec des arguments (les constructeurs).
- Puis nous aborderons comment proposer plusieurs « signatures » avec la même méthode (la surcharge).
- Comment entrer une méthode existante dans une variable
- Comment faire des « mini-méthodes » pratiques en argument
- Et enfin nous aborderons les Itérateurs.

3.6.1 Retour sur Main

Le C# est très orienté objet, du coup même la méthode qui s'exécute au début du programme est rangée dans une classe. Comme on a déjà beaucoup utilisé Main regardons-le directement pour pouvoir expliquer clairement les choses.

C#

```
class Program
{
    static int Main(string[] args)
    {
        foreach (string s in args)
            Console.WriteLine(s);
        Console.ReadKey();
        return 0;
    }
}
```

Retour Console: exécuté manuellement : « C:\adresse\application.exe avec des arguments »

```
avec des
arguments
```

Comme vous l'aurez remarqué, Main est Statique, c'est tout bonnement car lorsque le programme s'exécute, la classe contenant Main n'est pas instanciée. Antérieurement, Main n'avait

pas d'argument et à la place de « int » il y avait le mot void. Le type spécial void que nous utilisions sert à dire que la méthode ne retourne rien. void est l'allias de System.Void, il n'est utilisé que pour mentionner que la méthode ne retourne pas de valeur et en unsafe pour faire pointeur vers données de type inconnu. De ce fait, dans cet exemple, Main retourne une valeur : 0. Le mot clef return met fin à la méthode et lui fait retourner la valeur qui suit. Faire retourner une valeur de type int à un programme sert à spécifier un code d'erreur. Par convention, le 0 signifie que tout c'est bien dérouler, ensuite on peut créer plein de valeur de retour pour signaler les différentes erreurs possibles. Usuellement, il est publié avec le logiciel une correspondance entre la valeur de retour en cas d'erreur et les erreurs elles mêmes.

Contrairement au C, une méthode devant retourner une valeur sans mot clef return ne passe pas la compilation, de ce fait une méthode qui a un type de retour autre que void finira systématiquement par un return. Parfois pour les procédures, il peut être plus pratique de faire un return true quand il n'y a pas de problème et false à la place de lever une exception, on mettra bool pour le type de retour.

Main a 4 Signatures possibles :

C#

```
public static int  Main()
public static void Main()
public static int  Main(string[] arg)
public static void Main(string[] arg)
```

L'Array de string est composé de l'ensemble des chaines qui suivent la commande en séparant par des espaces.

3.6.2 Constructeur / destructeur

Vous trouverez souvent dans les classes non statiques des méthodes dites "constructeurs". Ce sont des méthodes qui servent à mettre des valeurs par défaut, à créer des éléments… Si on voulait par exemple faire une classe qui range un jeu d'allumette proche de celui que l'on a fait antérieurement :

C#

```
using System;
class Jeu
{
    readonly public int nbreBatonets;// readonly = variable ne peut être accédée
    readonly public int nbreJoueurs;//  en écriture que dans le constructeur.
    protected int joueur;
    protected int tour;
    protected int batonets;
    public string[] noms;
    public static uint nbreDInstances;

    /************************************************
     *  Voici le Constructeur, c'est une méthode spéciale sans type de retour
     *  et qui porte le même nom que la classe.
     *  On initialise ici les variable de Jeu propre aux « options »
     ************************************************/
    public Jeu(int nbreBnts, int nbreJrs,params string[] noms)
    {
        nbreDInstances++;
        nbreJoueurs = nbreJrs;
        nbreBatonets = nbreBnts;
        this.noms = noms;
        this.initGame();
    }
    // !! Le prochain extrait de code sera à ajouter ici !! //
    public void initGame() //Cette méthode permet d'initialiser une nouvelle partie
```

```
    {
        joueur = 0;
        tour = 0;
        batonets = nbreBatonets;
    }
    public bool Tour() //exécute un tour et renvoie faux si défaite.
    {
        int tmp;
        tour++;
        if (joueur < nbreJoueurs-1)
            joueur ++;
        else joueur = 0;
        System.Console.WriteLine("Tour "+tour.ToString());
        System.Console.WriteLine("A " + noms[joueur]
            + "\nil reste " + batonets.ToString()
            + " alumettes\nprenez un nombre d'allumettes entre 1 et 3 (defaut 3)");
        int.TryParse(System.Console.ReadLine(),out tmp);
        if (tmp > 0 && tmp < 3)
            batonets -= tmp;
        else batonets -= 3;
        if (batonets < 1) return false;
        return true;
    }
    // La méthode Perdu sera remplacée dans la partie sur la redéfinition
    public string Perdu()
    {
        return noms[joueur]+" a perdu au bout de " +tour.ToString()+" Tours.";
    }
    ~Jeu()
    {
        nbreDInstances--;
    }
}

class Program
{
    static void Main()
    {
        Jeu jeu = new Jeu(15,2,"Crovax","Popi");
        while (jeu.Tour()); // exécutera le test jusqu'à ce qu'il retourne faux
        Console.WriteLine(jeu.Perdu());
        /*
        for (int i =0 ; i < 10000; i++)
        {
            Console.WriteLine(Jeu.nbreDInstances);
            jeu = new Jeu(i, i, new string[] { "joueur1", "joueur2" });
            System.Threading.Thread.Sleep(10);
        }
        */
        Console.ReadKey();
    }
}
```

Remarque : *Voilà un extrait de code susceptible d'éclaircir l'intérêt du mot clef static. Si vous décommentez la boucle for et essayez plusieurs fois en modifiant quelques valeurs, vous pourrez constater le fonctionnement du garbage collector.quand le garbage collector veut effacer un objet il appelle de destructeur puis supprime les données de l'instance qui ne l'ont pas été.*

Un mot clef important n'a pas été abordé : this. Il est ici employé pour éviter un conflit de noms. « this » correspond à l'instance en cours de traitement donc this.noms correspond à la propriété public quand noms seul correspond à la valeur passé en argument de la méthode.

Le mot clef params est utilisé ici dans le constructeur mais est optionnel et peut être utilise pour n'importe quelle méthode. Il est utilisé pour le dernier argument, il est suivi d'un type array et

d'un nom de variable (« noms »). Tous les derniers arguments à l'appel de la méthode devront être des strings et rentrerons dans l'array. Sans le mot clef « params », l'instanciation de la classe « Jeu » n'aurait pas était comme vu dans l'extrait (« `Jeu jeu = new Jeu(15,2,"Crovax","Popi");` ») mais avec un véritable array en argument (« `Jeu jeu = new Jeu(15,2,new string[] {"Crovax","Popi"});` »). Params accepte aussi la syntaxe avec array en argument.

Nous utilisons le mot clef « out » dans la méthode statique « TryParse » d'Int32, ce mot clef apparait pour dire que l'argument passe par référence. C'est-à-dire que la variable pourra être modifiée dans la méthode, car les types « valeurs » ne sont jamais modifiées quand ils sont passés en argument sans mots clefs spécifiant le contraire. Un mot clef très proche de out existe : ref. La seule différence est que l'on utilise out pour une méthode qui vas donner une valeur sans tenir compte de l'actuelle tant disque ref sera plus utilisé pour traiter une variable faire en sorte qu'elle ait une nouvelle valeur à partir de sa valeur actuelle. De ce fait, ref attend une variable ayant une valeur attribuée tandis que out une variable déclarée même sans attributions fera l'affaire. De ce fait on n'est obligé de passer des variable et non pas des valeurs pour que la méthode puisse mettre ses données quelque part. TryParse procède comme Parse sauf qu'il entre 0 dans la variable au lieu de générer des erreurs. Les mots clefs param, ref et out paraissent dans les prototypes/signature des méthodes qui les utilisent.

Le Destructeur n'a ni de type de retour ni de degrés d'accès ni d'arguments. Son nom est celui de la classe et est précédé d'un tilde « ~ ».

3.6.3 La surcharge

Vous trouvez peut être qu'il n'est pas pratique de ne pas pouvoir faire un objet par défaut, d'avoir à spécifier à chaque fois tous les arguments… Pour pouvoir avoir le choix entre plusieurs possibilité d'arguments en entrée il existe la surcharge des méthodes. Nous allons procéder à la surcharge du constructeur. La surcharge d'une méthode permet de changer la signature en gardant le même nom. Une surcharge permet de changer les paramètres et le type de retour.

La surcharge peut aussi s'appliquer à un opérateur comme le « * », le « / », il suffit de précéder l'opérateur du mot clef operator. Du fait que ce soit vous qui définissiez l'exécution vous pouvez l'utiliser avec la syntaxe et la priorité des opérateurs sans pour autan bien que ce soit recommandé faire une exécution qui correspond à une addition ou qui retourne une valeur.

On peut aussi surcharger l'opérateur unaire de cast, il y a deux types de définition de cast : implicit et explicit. La surcharge peut être déclarée dans la classe d'instance source pour aller vers un type cible ou dans la classe cible source de type quelconque et retourne un objet de classe. Pour améliorer la lisibilité du code il est encouragé d'utiliser explicit dont le comportement est plus prévisible.

```
C#

class Jeu
{
    …
    public Jeu()
    {
        nbreDInstances++;
        nbreJoueurs = 2;
        nbreBatonets = 15;
        this.noms = new string[] {"Crovax","Popi"};
        this.initGame();
    }

    public static Jeu operator+(Jeu un,Jeu deux)
```

```
    {
        string[] noms=new string[un.noms.Length+deux.noms.Length];
        int p =0;
        for (int i=0 ; i<un.noms.Length;i++)
            noms[p++]=un.noms[i];
        for (int i=0 ; i<deux.noms.Length;i++)
            noms[p++]=deux.noms[i];

        return new Jeu(un.nbreBatonets + deux.nbreBatonets , un.nbreJoueurs
            + deux.nbreJoueurs, noms);
    }

    public static Jeu operator- (Jeu b,int a)
    {
        return new Jeu(b.nbreBatonets - a, b.nbreJoueurs, b.noms);
    }
    public static implicit operator int(Jeu a){return a.nbreJoueurs;}
    public static explicit operator Jeu(int a){return new Jeu(a,2,"Pop", "Crov");}
  // ( Remettre initGame, Tour, Perdu et ~Jeu de l'exemple précédent ) …
}
class Program
{
    static void Main()
    {
         int c;
        Jeu jeu = new Jeu();
        Jeu a = jeu + new Jeu(25, 3, "Riri", "Fifi", "Knight who say NI!");
        a = a - 12;
        c = a;
        a += (Jeu)c;
        while (a.Tour());
        Console.WriteLine(a.Perdu());
        Console.ReadKey();
    }
}
```

Ainsi implémenté, l'addition permet de créer un jeu ayant les joueurs en commun, le nombre de bâtonnets initial additionné (du fait qu'ils soient en readonly je suis obligé de recréer une instance). On peut songer par exemple dans une éventuelle évolution du jeu en réseau fusionner les salles de jeux par un simple opérateur +, avec ce même opérateur en cast implicite vers un int connaitre le nombre de joueurs dans les instances d'un serveur. On pourrait aussi à partir de l'option « nbreBatonets » générer en un cast une instance appropriée par défaut.

3.6.4 Delegate

Le delegate est comme un pointeur de fonction, c'est-à-dire une variable qui indique une fonction pouvant s'utiliser comme telle. Ce principe permet de passer une méthode en argument et aussi de pouvoir modifier l'exécution suivant l'état d'une variable delegate. On commence par générer un type de délégué, puis ou peut déclarer des délégués avec un degré d'accès inferieur ou égal à celui du type. Le type de délégué correspond à une signature, un type délégué indique les types des arguments et le type de retour de la méthode que le délégué associé va contenir. L'intérêt de créer des types de délégués est avant tout de sécuriser le code et de savoir comment utiliser le délégué. De plus si on entre dans un délégué une méthode qui a plusieurs surcharges, l'interpréteur déduira la quelle prendre pour que les signatures entre la surcharge et le type de délégué correspondent.

L'exemple ici montre plus un exemple d'utilisation que l'utilité réelle des délégués, mais d'autres exemples pourront vous éclairer sur l'utilité dès la partie prochaine.

```
C#
```

```
using System;

class testDelegate
{
    public delegate int typeDeDelegue(ref int a);//Déclaration : type de délégué
    //les méthodes associées devront répondre à sa signature : int nom(ref int var)
    public static int incrementer(ref int Nombre)
    {
        return ++Nombre;
    }
}
class Program
{
    static void Main()
    {
        int Valeur = 0;
        //ci dessous on déclare une instance du délégué
        testDelegate.typeDeDelegue a = testDelegate.incrementer;
        // le délégué sans parenthèse correspond à la variable
        // la méthode sans parenthèse correspond à l'adresse
        while (a(ref Valeur) < 100) ;
        // avec parenthèses la méthode est exécutée.
        Console.WriteLine(Valeur);
        Console.ReadKey();
    }
}
```

Les types de délégués sont accessibles comme des éléments statiques. Dans cet exemple simple nous avons utilisé a comme un pointeur ver la méthode incrémenter. Nous pouvons aussi utiliser des délégués en paramètres, nous verrons cela dans la partie qui suit. Notez le point virgule après le while : seule la condition sera exécutée jusqu'à être fausse.

3.6.5 Les événements

Les évènements font parti d'un type de programmation à part entière, on parle de programmation évènementielle qui s'oppose partiellement à la programmation procédurale. La programmation événementielle consiste à réagir en fonction des évènements. Les applications à interfaces graphiques sont souvent associées aux concepts de programmation événementielle. Par exemple, un utilisateur qui clique sur un des boutons de l'interface lance une procédure. Le fait que l'utilisateur puisse à tout moment agir sur n'importe quel bouton entend que ce n'est pas une procédure à variable qui se déroule de manière uniforme. C'est bien agir en fonction des interactions. Dans nos exemples nous allons utiliser sans quitter notre console les évènements pour voir leur fonctionnement.

C#

```
this.button1 = new System.Windows.Forms.Button()
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Les évènements sont des délégués voire des listes de délégués qui s'appellent à peut prêt comme de simples méthodes. Dans l'extrait généré de WinForm ci-dessus nous voyons que le bouton a par défaut un évènement existant. Le plus souvent ce type de délégué sera celui du framework : EventHandler (Nous verrons quelle signature correspond à ce type de délégué).

L'évènement n'étant pas une notion très simple à intégrer j'ai proposé deux exemples. Les évènements se déclarent avec le mot clef « event » et un type de délégué. Un évènement ne peut être exécuté que depuis la classe il est déclaré mais peut être modifié depuis une autre s'il est public.

C#

```
using System;

class exemple1
{
    // EventHandler est un délégué de même signature que celui-ci-dessous.
    // public delegate void elem(object sender,EventArgs e) ;
    // public event elem evenement;
    public event EventHandler evenement;

    public void add_evenement(int i)
    {
        // evenement(this, EventArgs.Empty); (lève une exception si aucune méthode
assiciées)
        switch (i)
        {
            case 1:
                evenement+=new EventHandler(exemple_evenement1);
                break;
            case 2:
                evenement += new EventHandler(exemple_evenement2);
                break;
            case 3:
                evenement += new EventHandler(exemple_evenement1);
                evenement += new EventHandler(exemple_evenement2);
                break;
            default:
                evenement -= new EventHandler(exemple_evenement1);
                break;
        }
    }
    public void execute()
    {evenement(this, EventArgs.Empty);}
    public static void exemple_evenement1(object sender, EventArgs e)
    {Console.WriteLine("execution de la méthode 1");}
    public static void exemple_evenement2(object sender, EventArgs e)
    {Console.WriteLine("execution de la méthode 2");}
}

class Program
{
    static void Main()
    {
        exemple1 a = new exemple1();
        for (int i=1; i < 6; i++)
        {
            Console.WriteLine(i.ToString() + "        \n");
            a.add_evenement(i);
            a.execute();
        }
        System.Console.ReadKey();
    }
}
```

Dans notre premier exemple l'évènement est utilisé de manière très procédurale. Dans le deuxième exemple bien que ce ne soit pas évident en console nous allons montrer quelque chose de plus « évènementiel », nous allons loguer l'état d'une variable.

```
C#

using System;

class exemple2
{
    public delegate void evenementHandler(exemple2 sender, EventArgs e);
    public event evenementHandler lit;
```

```
    public event evenementHandler ecrit;
    protected string _logable;
    public string Logable
    {
        get
        {
            if(lit != null)lit(this,EventArgs.Empty);
            return _logable;
        }
        set
        {
            _logable=value;
            if (ecrit != null) ecrit(this, EventArgs.Empty);
        }
    }
    public bool switchLog()
    {
        if (lit == null && ecrit == null)
        {
            lit = new evenementHandler(accesLecture);
            ecrit = new evenementHandler(accesEcriture);
            return true;
        }
        lit = null;
        ecrit = null;
        return false;
    }
    public static void accesLecture(exemple2 sender, EventArgs e)
    {
        Console.WriteLine("Accès en lecture:" +  sender._logable);
    }
    public static void accesEcriture(exemple2 sender, EventArgs e)
    {
        Console.WriteLine("Accès en ecriture:" + sender._logable);
    }
}

class Program
{
    static void Main()
    {
        exemple2 b = new exemple2();
        b.switchLog();
        b.Logable = "test";
        b.Logable = b.Logable;
        Console.WriteLine(b.Logable);
        b.switchLog();
        b.Logable = "sans log c'est moins verbeux";
        b.Logable = b.Logable;
        Console.WriteLine(b.Logable);
        System.Console.ReadKey();
    }
}
```

Le terminal n'est pas la plus adapté des interfaces pour parler d'évènements mais j'espère que ces exemples vous suffiront pour vous approprier les notions d'évènements qui seront utilisés dans de vrais programmes .NET.

3.6.6 Méthodes anonymes et Expressions lambda

Les méthodes anonymes fait partie des nouveautés du C#2.0. Cet outil permet de créer une méthode pointé par un délégué sans avoir à la mettre dans une classe : L'utilisation est simple :

```
C#

using System;
```

```
class testDelegate
{
    public delegate void NoArg ();
    public delegate void ArgDelegate(NoArg a);
    public void executer10Fois(NoArg fonction)
    {
        for(int i=0;i<10;i++) fonction();
    }
    public void executer10Fois(ArgDelegate fonction,NoArg argument)
    {
        for (int i = 0; i < 10; i++) fonction(argument);
    }
}
class Program
{
    static void Main()
    {
        testDelegate var = new testDelegate();
        testDelegate.NoArg afficheA = delegate() { Console.Write("a"); };
        var.executer10Fois(afficheA);
        Console.WriteLine("\n                          \n");
        testDelegate.ArgDelegate a = var.executer10Fois;
        // var.executer10Fois(a,afficheA) ; ou plus instructif :
        var.executer10Fois
        (
          delegate(testDelegate.NoArg fonction){for(int i=0;i<10;i++) fonction();},
          delegate() { Console.Write("a"); }
         );
        Console.ReadKey();
    }
}
```

Voici un exemple assez simple et tordu à la fois, il nous montre une partie du potentiel des délégués quels qu'ils soient. Notez que les délégués anonymes sont typés automatiquement pour entrer dans la variable ou en argument. Vous pouvez facilement imaginer une attribution conditionnelle dans la quelle le formatage dépendra de la méthode pointé par la variable déléguée.

Remarque : la partie qui suit sur l'expression lambda et les méthodes d'extension est composée essentiellement de nouveautés du C# 3.0, si vous développez sur un logiciel existant, il vous faudra spécifier de compiler avec la syntaxe C# 3.0 qui est rétro compatible pour pouvoir faire ce qui vas suivre.

Les expressions lambda sont des délégués anonymes limités à une instruction, la syntaxe est claire, rapide et lisible mais on ne peut mettre de boucles. Si le type de délégué attendu a un type de retour, la seule « instruction » de l'expression lambda sera retournée.

La syntaxe de l'expression lambda est simple : les paramètres du délégué anonyme, le signe « => » puis l'instruction.

```
C#

(int arg1, int arg2) => arg1+ arg2 //delegate(int arg1,int arg2){return arg1+arg2;}
arg => arg + 4                     // delegate(int arg){return arg+4;}
                                   //  Le type du paramètre précédent dépend du type
                                   //   délégué attendu (donc pas forcément int)
() => Console.Write("a")           //    delegate(){Console.Write("a");}
```

Voici trois exemples hors contexte. Si on ne spécifie pas le type il est spécifié automatiquement et s'il y a un seul paramètre typé implicitement pas besoin des parenthèses pour l'isoler. Refaisons la même chose que tout à l'heure avec ces lambda expressions.

C#

```
using System;
class testDelegate
{
    public delegate void NoArg ();
    public delegate void ArgDelegate(NoArg a);
    //ici on déclare une instance du délégué
    public void executer10Fois(NoArg fonction)
    {
        for(int i=0;i<10;i++) fonction();
    }
    public void executer10Fois(ArgDelegate fonction,NoArg argument)
    {
        for (int i = 0; i < 10; i++) fonction(argument);
    }
}
class Program
{
    static void Main()
    {
        testDelegate var = new testDelegate();
        testDelegate.ArgDelegate a = var.executer10Fois;
        var.executer10Fois
        (
            //(testDelegate.NoArg fonction) => a(fonction),
            fonction => a(fonction) ,
            () => Console.Write("a")
        );
        Console.ReadKey();
    }
}
```

Nous avons vu l'essentiel sur les méthodes et les délégués, nous allons désormais voir comment on peut lier nos méthodes à des objets du framework.

3.6.7 Méthode d'extension.

Les méthodes d'extensions sont des méthodes statiques qui permettent de s'utiliser comme si elles étaient propre à une classe autre que celle dans la quelle elle est définie. Je m'explique, si vous voulez faire une méthode de traitement de string, vous n'allez pas pouvoir modifier la classe string du framework pour ajouter une méthode de traitement, donc vous allez devoir créer une classe avec une méthode statique puis faire appel à cette classe pour le traitement. La méthode d'extension permet d'appliquer la méthode statique comme si elle était dans la classe. Par exemple ci dessous on crée une méthode qui permet de n'avoir à écrire que var.Write() avec var variable de type string au lieu de faire Console.WriteLine(var).

C#

```
using System;
static class StringToolBox
{
    static int titleIndex = 0;
    public static void Write(this string s)
    {
        Console.WriteLine(s);
    }
    public static string titleFormat(this string s,int i)
    {
        return
            "\n" + "_" + (++titleIndex).ToString() + "_" +
            s.ToUpper() + "_" + i.ToString() + "_" + "\n" ;
    }
```

```
}
class Program
{
    static void Main()
    {
        for(int i=0;i<10;i++)
            ("bonjour "+"Monsieur").titleFormat(i).Write();
        Console.ReadKey();
    }
}
```

Le mot clef qui sert à spécifier que la méthode est une méthode d'extension est le mot clef this, on mettra toujours ce this avec le premier argument (qui est l'objet qui précède le point lors de l'appel).

3.6.8 Itérateurs

Certaines méthodes sont appelées des itérateurs, elles utilisent le mot clef yield. Ces méthodes retournent des objets IEnumerable ou IEnumerator (Nous privilégierons IEnumerable qui peut être utilisée dans un foreach). Nous allons voir comment faire avec et sans le foreach pour que vous puissiez comprendre comment sont les objets que nous utilisions. IEnumerable et IEnumerator ne contiennent pas le résultat mais le moyen d'y accéder, pour preuve, l'exemple modifiera le résultat après la déclaration de l'énumérable et le parcourra deux fois avec deux résultats différents.

C#

```
using System;
using System.Collections;
static class StringTB
{
    public static int I = 0;
    public static void Write(this string s)
    {
        Console.WriteLine(s);
    }
    public static IEnumerable titleFormat(this string s, int limit)
    {
        for (int i =1;i<=limit;i++)
            yield return
                "\n"+"_"+i.ToString()+"_"+s.ToUpper()
                +"_"+(++I).ToString()+"_"+"\n";
        yield break;
    }
}

class Program
{
    static void Main()
    {
        IEnumerable enumerable = "bonjour Monsieur".titleFormat(100);
        // on a créé l'énumérable avec titleFormat

        IEnumerator enumerateur = enumerable.GetEnumerator();
        // on extrait l'énumérateur

        StringTB.I = 5;
        // on procède à une modification qui changera le résultat
        // conclusion rien n'as encore été calculé.

        for (int i = 0; i < 100; i++)
        {
      // si on ne commençait pas par un MoveNext le premier résultat serait "null"
            enumerateur.MoveNext();
```

```
            string s = enumerateur.Current as string ?? "erreur";
            // caste en string (si impossibles contiendra "erreur")

            StringTB.I = StringTB.I % 10 == 0 ? StringTB.I * 2:StringTB.I;
            // multiplie par 2 toutes les 10 itérations I
            // conclusion : les strings sont calculés au fur et à mesure

            s.Write();
        }
        StringTB.I = 0;
        Console.ReadKey();
        // parcour avec foreach:
        foreach (string s in enumerable)
            s.Write();
        Console.ReadKey();
    }
}
```

Yield est un mot clef qui est central dans le linq bien que relativement caché, cette technologie vous permet de jouer avec des énumérables, de les imbriqués avec une facilité étonnante, et bien-sur à l'accès en lecture, ça exécute le nécessaire pour accéder à la donnée.

Yield n'est utilisable qu'avec un des deux mots clefs qui suit : Return et Break. Yield return ajoute une entrée dans l'énumérateur et yield break arrête l'exécution du block. Yield break employé dans notre exemple, il ne sert à rien.

3.7 L'héritage, le polymorphisme et les interfaces

3.7.1 Introduction

L'héritage est un moyen d'offrir aux classes toutes les méthodes et tous les attributs public et protected d'une classe existante. Contrairement au C++, le C# ne permet pas d'hériter de plusieurs classes bien que les interfaces permettent de faire un équivalent. L'héritage conduit à des similitudes entre les objets et donc de la simplicité pour apprendre. Nous ne l'avions pas vu mais tous les objets héritent de l'objet « object ». Object est la classe à la base de tout, c'est object qui implémente la méthode ToString que l'on a beaucoup utilisé. Cette méthode étant de la classe mère « universelle », elle est implémentée par défaut par toutes les classes.

Le polymorphisme dans le framework est un outil pratique qui permet par exemple de ranger dans une liste typée des objets de classes différentes pour peu qu'ils se comportent pareil. Tous les objets peuvent êtres appelés comme des objets d'un de leurs types parents. De ce fait toutes variables quelque soient leur type peuvent être utilisé comme des variable de type object. Toute variable peut être passée en argument d'une méthode qui attend un objet de type « object ».

Les interfaces sont des espèces de classes qui ne servent qu'à être « héritées » (dans le cas d'une interface, on dit qu'elle est « implémentée »). Quand on implémente une interface on garanti un comportement, si on implémente par exemple l'interface IEnumerable, notre objet pourra bénéficier de linq et être énuméré dans un foreach. Dans l'exemple précédant on utilisait IEnumerable et IEnumerator comme les types des variables mais c'est juste grâce au fait que le polymorphisme marche avec les interfaces en plus des classes mères. De ce fait peut importe le type de retour qui entrera la donnée dans ma variable dès lors que ce type implémente l'interface IEnumerable car ce retour aura forcément le comportement attendu.

L'orienté objet initie des nouveaux mots-clefs de restriction de droits, commençons par réexpliquer les mots que nous connaissons déjà dans ce contexte d'héritage :

- public : Les éléments public d'une classe sont légués aux classes enfantes.
- protected : Les éléments protected d'une classe sont légués aux classes enfantes.

- private : Les éléments private d'une classe ne sont pas hérité aux classes enfantes.
- static : On ne peut pas faire hériter une classe spécifiée en static. Les éléments statiques d'une classe non statique qui sont en public ou protected peuvent êtres accédés par les classes enfante mais ne sont pas à proprement parlé héritées. Le mot clef « base » permet d'accéder aux éléments de la classe mère sans se soucier de son nom (s'utilise comme « this »). Comme les classes statiques ne peuvent hériter il est vu comme un non sens que de spécifier des éléments en protected, de ce fait on utilisera public ou private.

Voici une liste de quelques nouveaux mots clefs initiés par la notion d'héritage:

- sealed : Aucune classe de peut hériter d'une classe d'une méthode ou d'une propriété sealed (si on veut qu'un élément soit public sans que les classes filles en hérite par exemple).
- abstract : Les classes abstraites ne sont pas utilisables directement, elles ne servent que de classes mère pour faciliter et regrouper la construction d'objets de même forme. Les éléments abstraits d'une classe eux doivent être défini, ils sont comme des contraintes de méthodes et variable interne à définir. (Une classe abstract ne peut être instancié)
- virtual : virtual est fonctionnellement comme abstract à une différence prêt : dans la classe mère il y a une implémentation par défaut que l'on peut ne pas redéfinir.

Le framework utilise beaucoup l'héritage et ce modèle de polymorphisme, nous allons revenir sur les exceptions qui héritent toutes de la classe exception qui elle même hérite de Object.

3.7.2 Exemples d'héritage : les exceptions et throw

Cet exemple est à couplé avec celui de la partie 2.5. Dans ce deuxième exemple je crée un nouveau type d'exception dont je ne redéfini que le Constructeur. Ce type d'exception a les mêmes attributs et les mêmes propriétés que l'exception par default au détail prêt qu'il n'a qu'un constructeur. Pour définir les variables comme dans une exception standard simplement, j'ajoute à mon constructeur le constructeur de la classe Exception.

C#

```
using System;
class MonException : Exception
{
    public MonException() : base("Le problème est de type custom")
    {
        Console.WriteLine("Erreur spéciale avec mon exécution");
    }
}
class Program
{
    public static void Main()
    {
        Random r = new Random(DateTime.Now.Millisecond);
        while (true)
        {
            if (1 >= r.Next(0,100000000))
            {
                MonException e = new MonException();
                try
                {
                    throw e;
                }
                catch (MonException except)
```

```
                {
                    Console.WriteLine(except.ToString());
                }
                finally
                {
                    e = null;
                }
            }
        }
    }
}
```

La nouvelle classe est instanciée et levée comme n'importe quelle exception. Le mot clef throw permet de lever une instance d'exception, il sera plus commun de voir « throw new MonException() ; » mais c'était pour justifier l'usage d'un block finally pour la dés-allocation de mémoire.

Dans l'exemple de la partie 2.5 on perçoit plus nettement le polymorphisme mais nous aurons l'occasion d'y revenir.

3.7.3 Redéfinition de méthodes et d'attributs

Le constructeur est une redéfinition particulière vu que le nom n'est pas le même mais pour les classes ou les éléments abstract on se verra obligé de redéfinir des éléments qui existent déjà ce qui normalement est impossible. Nous allons utiliser deux mots clefs très similaires dans ce contexte : override et new.

C#

```
using System;
abstract class mere
{
    public abstract int longueur { get;}
    public abstract int largeur { get; set; }
    public virtual int aire() { return longueur*largeur;}
    public virtual void printAire() { Console.WriteLine(longueur*largeur); }
}
class fille : mere
{
    private int la;
    private int lo;

    public override int largeur
    {
        get { return la; }
        set { la = value; }
    }
    public override int longueur
    {
        get { return lo; }
    }
    public fille()
    {
        la = 6;
        lo = 5;
    }
    public new int aire() {return (largeur+longueur)*2;}
    public override void printAire()
    {
        Console.WriteLine((largeur + longueur) * 2);
        // base.printAire();
    }
    public override string ToString()
    {
```

```
        return "on peut définir comment réagi la méthode .ToString comme ceci" ;
    }
}
class Program
{
    public static void Main()
    {
        mere m = new fille();
        fille f = new fille();
        Console.WriteLine("affichage casté en mère puis en fille (override)");
        m.printAire();
        f.printAire();
        Console.WriteLine("retour d'aire casté en mère et en fille (new)");
        Console.WriteLine(m.aire() + "\n" + f.aire());
        Console.ReadKey();
    }
}
```

Retour Console:

```
affichage casté en mère puis en fille (override)
22
22
retour d'aire casté en mère et en fille (new)
30
22
```

Dans cet exemple on a une classe abstraite mère et une classe fille. Quand vous tentez d'implémenter une propriété ou une méthode, vous verrez que votre IDE (Visual Studio ou MonoDevelop) vous propose directement la structure de base. La syntaxe des propriétés de la classe mère est encore une nouveauté du C#3.0, on définie ce qui sera à implémenter. La ligne commentée montre comment j'aurais pu par exemple appeler la méthode par défaut implémentée par la classe mère.

L'exemple risque d'être un bon complément d'explication sur la différence entre la redéfinition avec override et la redéfinition avec new.

- New : permet de créer une méthode propre à la classe fille qui sera appelée normalement sur une instance de cette classe fille. Si l'instance de la classe fille est rangée dans une variable de type de la classe mère si la méthode homonyme est appelée ce sera la méthode par défaut de la classe mère qui sera appelée et non celle de la classe fille.
- Override : permet quand à lui de remplacer réellement la méthode de la classe mère pour l'instance de la classe fille quelque soit le contexte.

3.7.4 Les interfaces

Les interfaces s'utilisent comme des classes abstraites, on peut les implémentées comme on hériterait d'une classe. On défini dans l'interface des méthodes et propriétés à avoir pour implémenter l'interface pour pouvoir bénéficier du polymorphisme lié. Si on caste l'objet avec l'interface, seules les méthodes qui relèvent de l'interface seront accessibles.

C'est donc une forme d'héritage multiple. Si l'on repense aux méthodes d'extensions, je vous laisse comprendre l'intérêt du polymorphisme, pouvoir réduire le code et ne pas avoir à le refaire pour chaque classe. Certaines interface comme IEnumerable sont au cœur du framework, cette interface doit être implémentée que ce soit pour utiliser le mot clef yield, pour pouvoir utiliser foreach, pour pouvoir utiliser Linq2object…

Les méthodes des interfaces peuvent êtres implémentées de deux façons : implicitement ou explicitement. Les interfaces on toujours tous leurs attributs publics, c'est pourquoi le degré d'accès

n’est pas demandé dans la déclaration de l’interface. On ne peut pas faire de méthode virtual, les interfaces établissent une structure à respecter

```C#

using System;
interface Imere
{
    int longueur { get;}
    int largeur { get; set; }
    void aire();
    void printAire();
}

class fille : Imere
{   private int la;
    private int lo;
    public fille()
    {la = 6;lo = 5;}

    #region Imere Membres
        // implémentée explicitement
        int Imere.longueur
        {   get { return lo; }}
        // implémentée implicitement
        public int largeur
        {   get { return la; }
            set { la = value; }}

        // implémentée explicitement
        void Imere.aire()
        {Console.WriteLine((largeur+ ((Imere)this).longueur)*2);}

        //Implémenté implicitement
        public void printAire()
        {Console.WriteLine((largeur + ((Imere)this).longueur) * 2);}

    #endregion
}

class Program
{
    public static void Main()
    {

        Imere m2 = new fille();
        fille f2 = new fille();
        Console.WriteLine("affichage casté en interface puis en fille (implicit)");
        m2.printAire();
        f2.printAire();

        Imere m = new fille();
        fille f = new fille();
        Console.WriteLine("retour d'aire casté en Imère et en fille (explicit)");
        m.aire();

/*****************************************************************
 * la ligne qui suit ce commentaire ne passe pas la compilation.
 * Erreur 1:
 * 'fille' ne contient pas une définition pour 'aire' et aucune méthode d'extension
 * 'aire' acceptant un premier argument de type 'fille' n'a été trouvée
 * (une directive using ou une référence d'assembly est-elle manquante ?)
 *****************************************************************/
        // f.aire();
        Console.WriteLine("erreur\n\nutilisons une liste maintenant");
        Imere[] list = new Imere[] { f, m, m2, f2 };
        foreach (var e in list)
```

```
            e.aire();
        Console.ReadKey();
    }
}
```

3.7.5 Les attributs

Les attributs sont des classes héritant de attribut, leurs usage sert à modifier le comportement d'éléments (classes, enums, structs, méthodes). Les attributs ont une syntaxe particulière utilisant les crochets (« [ ] »).

Un exemple pratique ou les attributs sont utilisés c'est la sérialisation. L'usage d'un simple attribut permettra de ranger assez facilement dans un fichier XML toutes les données d'un objet. Et avec un simple attribut on peut demander à ce qu'un des champs ne soit pas sérialisé. Je ne vais pas vous insister sur la syntaxe pour sérialiser (où il faut créer un flux vers un fichier texte) mais montrer comment rendre sérialisable une classe. Si vous voulez exécuter cet exemple vous aurez besoin de la référence à `System.Runtime.Serialization.Formatters.Soap`.

C#

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

[Serializable()]
public class ExempleSerialisation
{
    public int serialise;

    [NonSerialized()]
    public string nonserialise;

    public ExempleSerialisation()
    {

        serialise = 1337;
        nonserialise = "mon nom est Jirfongrifengrafterg";
    }
    public void affichage()
    {
        Console.WriteLine("élément sérializé = "+ serialise.ToString());
        Console.WriteLine("élément non sérializé = "+ nonserialise);
   }
}

public class Program
{
    public static void Main()
    {
        ExempleSerialisation variable = new ExempleSerialisation();
        variable.affichage();

        // La partie qui suit permet de sérialiser et désérialiser,
        // je ne m'atarderait pas dessus, le sujet étant les attributs.
        Stream stream = File.Open("variable.xml", FileMode.Create);
        SoapFormatter formatter = new SoapFormatter();
        formatter.Serialize(stream, variable);
        stream.Close();
        Console.WriteLine("sérialisation finie\ndésérialisation:");
        variable = null;
        stream = File.Open("variable.xml", FileMode.Open);
        formatter = new SoapFormatter();
        variable = (ExempleSerialisation)formatter.Deserialize(stream);
        stream.Close();
```

```
            Console.WriteLine("désérialisation finie.");
            variable.affichage();
            Console.ReadKey();
        }
}
```

Retour Console:

```
élément sérialisé = 1337
élément non sérialisé = mon nom est Jirfongrifengrafterg
sérialisation finie
désérialisation:
désérialisation finie.
élément sérialisé = 1337
élément non sérialisé =
```

L'attribut [Serializable()] a automatiquement implémenté le nécessaire pour que la classe puisse être sérialisée. Nous ne voulions pas retenir un des champs, nous avons mis [NonSerialized()]. L'attribut ce place au début de l'élément et s'applique sur sa globalité.

C#

```
using System;
static class StringToolBox
{
    static int titleIndex = 0;
    public static void Write(this string s)
    {
        Console.WriteLine(s);
    }
    [Obsolete()]
    public static string titleFormat(this string s, int i)
    {
        return
            "\n" + "_" + (++titleIndex).ToString() + "_" +
            s.ToUpper() + "_" + i.ToString() + "_" + "\n";
    }
}
```

Voilà un autre attribut simple pour mentionner qu'une méthode est dévalué. L'environnement de développement précisera les méthodes que l'on compte retirer aux développeurs pour les inciter à en utiliser d'autres s'ils prévoient de mettre à jour la librairie contenant la classe. L'attribut Obsolete peut se mettre sur n'importe quel élément. Tous les attributs sont des classes héritant de System.Attribute.

4 Conclusion

Nous avons traité de manière très synthétique un sujet quasiment inépuisable, le framework .NET vous permettra d'utiliser les connaissances de ce tutorial pour créer des sites web, créer des applications fenêtrées, faire des applications graphiques, des librairies… Bref, vous pourrez faire quasiment presque tout ce qui est programmable avec le framework et le C#. Nous avons abordé de près ou de loin tous les mots clefs du C# à l'exception de quelques uns (volatile, lock, external et where).

Pour rappel nous avons vu :

- L'histoire et fonctionnement du C#
- Les types valeurs et string
- Les opérateurs

- Les conditions
- Les boucles
- Les Arrays
- Comment gérer les erreurs
- Comment utiliser les pointeurs hérités du C++
- Les espaces de nom
- Les modificateurs d'accès, les attributs et les propriétés
- Les méthodes dont Main, les constructeurs et les destructeurs
- La surcharge des méthodes
- Les délégués, évènements, et expressions lambda
- Les itérateurs
- L'héritage
- Le polymorphisme
- Les interfaces
- Les attributs

Pour tirer pleinement partie de ce premier tutoriel, vous devriez songer à aborder d'autres cours de DotNet-France, vous avez désormais le niveau requis. Voici les cours ver les quels vous pourriez vous orienter.

- Vous vous intéressez aux applications fenêtrées ? vous pouvez vous diriger vers le tutoriel traitant des WinForm.
- Vous voulez faire des sites web ? le tutoriel ASP.NET WebForm.
- Vous voulez faire par exemple un jeu vidéo, intéressez vous peut être au tutoriel XNA.
- Vous vous intéressez à l'accès aux bases de données et fichiers XML, le tutoriel ADO.NET est aussi là pour vous.
- Vous vous intéressez à l'interopérabilité et aux applications Unix ? peut être devriez vous vous diriger ver le tutoriel Mono.
- Vous voulez améliorer vos connaissances globales sur les rouages du framework ? Le tutoriel traitant du framework devrait vous satisfaire.

Cette liste vous donne déjà un avant goût de ce que vous pourrez faire avec le C#. J'espère que ce chapitre ne vous a pas perdu, et même ouvert une envie au développement .NET, n'hésitez pas a garder ce tutoriel sous le coude comme manuel de référence, car je ne pense pas que l'on puisse tout intégrer d'une seule traite.