



SL228_POBJ Visual C# 2015

Chapitre 4

**Données, opérateurs et instructions de
contrôles**

**Christian HUBER (CHR)
Version 1.0 Mars 2016**

CONTENU DU CHAPITRE 4

4. Données, opérateurs et structures de contrôles	4-1
4.1. Aide du Visual C#	4-1
4.1.1. Référence C#	4-2
4.1.1.1. Spécification du langage C#	4-2
4.2. Quelques aspects du langage C#	4-3
4.2.1. Mots clés	4-3
4.3. Variables et constante	4-4
4.3.1. Les types de données du C#	4-4
4.3.1.1. Particularité de certain types C#	4-4
4.3.1.2. Post fixe pour typer les valeurs numériques	4-5
4.3.2. Déclaration des variables	4-5
4.3.2.1. Exemples de déclaration de variables	4-5
4.3.2.2. Noms des variables	4-5
4.3.2.3. Les Tableaux	4-6
4.3.3. Affectation et expression	4-6
4.3.4. Portées des variables	4-7
4.3.4.1. Noms des attributs	4-7
4.3.4.2. Accessibilité des attributs	4-7
4.3.5. Attributs constantes	4-8
4.3.5.1. Modification d'une constante	4-8
4.3.6. Les énumérations	4-8
4.3.6.1. Enumération, exemple	4-9
4.4. Opérateurs	4-11
4.4.1. Opérateurs principaux	4-11
4.4.2. Opérateurs unaires	4-11
4.4.3. Opérateurs multiplicatifs	4-12
4.4.4. Opérateurs additifs	4-12
4.4.5. Opérateurs de décalage	4-12
4.4.6. Opérateurs relationnels et de test de type	4-12
4.4.7. Opérateurs Logiques	4-13
4.4.7.1. Egalité et inégalité	4-13
4.4.7.2. And, Xor et Or bit à bit	4-13
4.4.7.3. And et Or conditionnels	4-13
4.4.7.4. Operateur de fusion de Null	4-13
4.4.7.1. Operateur conditionnel	4-13
4.4.8. Opérateurs d'assignation et lambda	4-13
4.4.9. Dépassement arithmétique	4-14
4.4.10. Priorité des opérateurs	4-15
4.5. Structures de contrôles: choix et boucles	4-17
4.5.1. Instruction de sélection	4-18
4.5.1.1. if..else	4-18
4.5.1.2. switch case default	4-18
4.5.2. Les instructions d'itérations	4-20

4.5.2.1.	L'instruction while	4-20
4.5.2.2.	L'instruction do while	4-20
4.5.2.3.	L'instruction for	4-21
4.5.2.4.	L'instruction foreach et in	4-21
4.5.3.	Instructions de saut	4-23
4.5.3.1.	L'instruction break	4-23
4.5.3.2.	L'instruction continue	4-24
4.5.3.3.	L'instruction goto	4-25
4.5.3.4.	L'instruction return	4-26
4.5.3.5.	L'instruction yield	4-26
4.5.4.	Instructions de gestion des exceptions	4-27
4.5.4.1.	Les instructions try catch finally	4-27
4.5.4.2.	try catch, exemple	4-27
4.5.4.3.	Ajout du finally	4-28
4.5.4.4.	L'instruction throw	4-29
4.6.	Conclusion	4-31
4.7.	Historique des version	4-31
4.7.1.	Version 1.0 Mars 2016	4-31

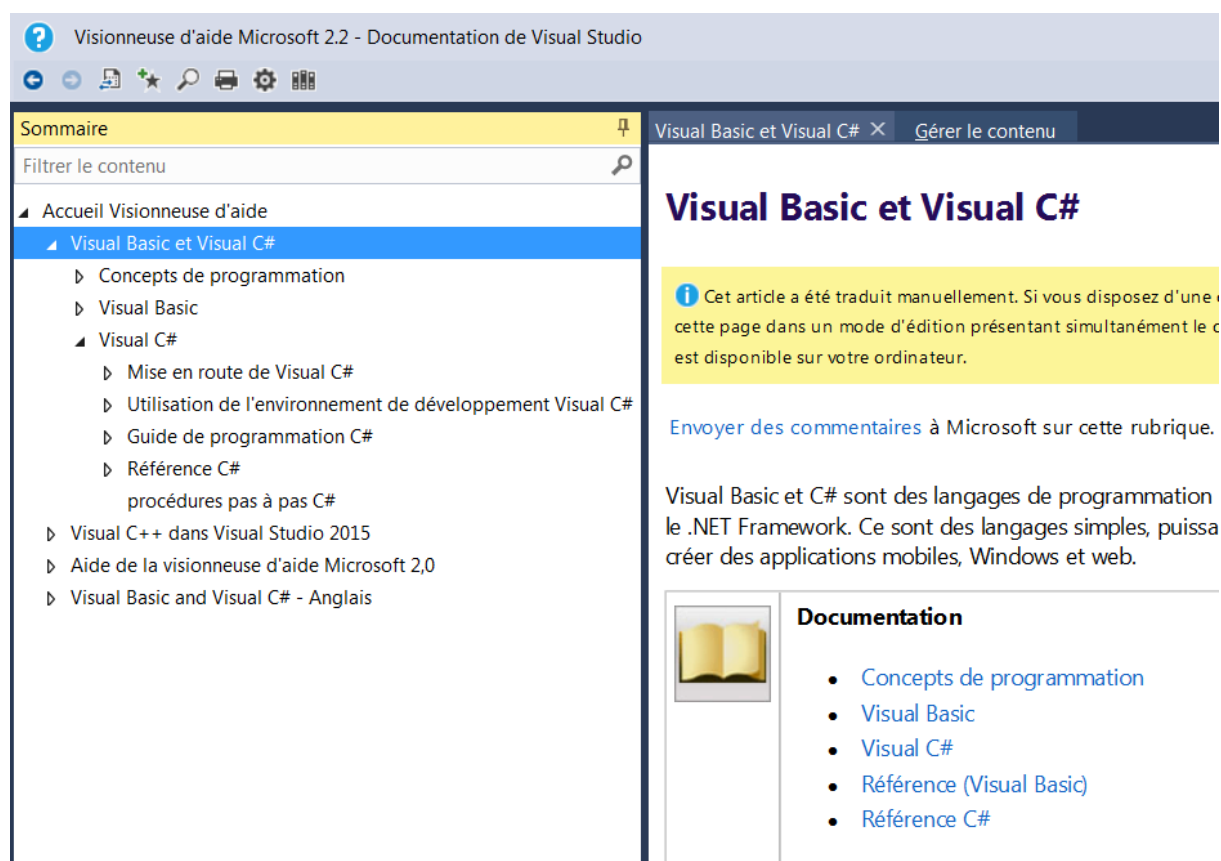
4. DONNÉES, OPÉRATEURS ET STRUCTURES DE CONTRÔLES

Ce chapitre traite de la partie langage du C#, donc de l'aspect syntaxe de la programmation.

Comme le C# est très proche du C/C++ pour une grande partie, ce chapitre met en évidence ce qui est différent.

4.1. AIDE DU VISUAL C#

Après configuration et choix d'utiliser la visionneuse d'aide, on constate que l'aide du Visual C# est assez liée à celle du VB.



The screenshot shows the Visual Studio 2015 Help Viewer interface. The title bar reads 'Visionneuse d'aide Microsoft 2.2 - Documentation de Visual Studio'. The left sidebar contains a 'Sommaire' (Table of Contents) with a search bar and a list of topics. The 'Visual Basic et Visual C#' section is expanded, showing sub-topics like 'Concepts de programmation', 'Visual Basic', and 'Visual C#'. The 'Visual C#' section is further expanded, showing 'Mise en route de Visual C#', 'Utilisation de l'environnement de développement Visual C#', 'Guide de programmation C#', and 'Référence C#'. The main content area displays the 'Visual Basic et Visual C#' page, which includes a notice about manual translation, a link to send comments, and a 'Documentation' section with a list of links: 'Concepts de programmation', 'Visual Basic', 'Visual C#', 'Référence (Visual Basic)', and 'Référence C#'.

Au niveau de l'aide du Visual Studio, nous allons nous référer principalement à la section **Référence C#**.

4.1.1. RÉFÉRENCE C#

Voici le contenu de cette section :

Référence C# x Gérer le contenu

Cette section fournit des informations de référence sur les mots clés, les opérateurs, les erreurs du compilateur et les avertissements en langage C#.

Dans cette section

Mots clés C#
Propose des liens vers des informations relatives aux mots clés et à la syntaxe C#.

Opérateurs C#
Propose des liens vers des informations relatives aux opérateurs et à la syntaxe C#.

Directives de préprocesseur C#
Propose des liens vers des informations relatives aux commandes de compilateur à incorporer au code source C#.

Options du compilateur C#
Induit des informations sur les options du compilateur et les façons de les utiliser..

Erreurs du compilateur C#
Induit des extraits de code qui montrent la cause et la correction des erreurs et des avertissements du compilateur C#.

Spécification du langage C#
Propose des pointeurs vers la dernière version de la spécification du langage C# au format Microsoft Word.

4.1.1.1. SPÉCIFICATION DU LANGAGE C#

Cette section conduit à la possibilité de télécharger un document au format .docx qui regroupe toute les spécification du langage C#.

Spécification du langage C#


Cet article a été traduit manuellement. Si vous disposez d'une connexion Internet, sélectionnez Afficher cette version en ligne pour consulter cette page dans un mode d'édition présentant simultanément le contenu original en anglais. [Afficher la version anglaise](#) de cette rubrique qui est disponible sur votre ordinateur.

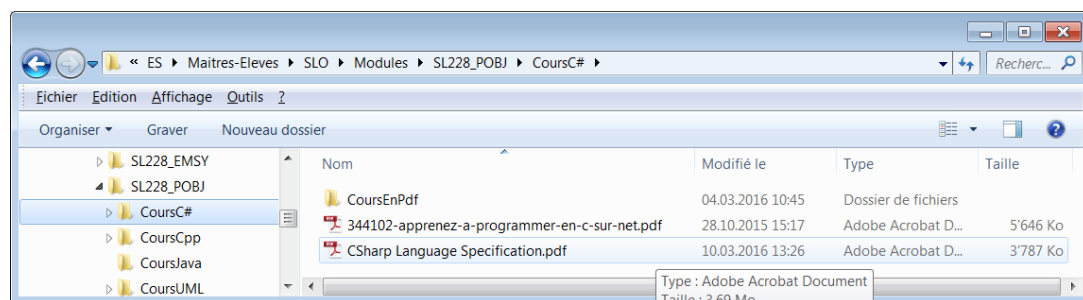
[Envoyer des commentaires](#) à Microsoft sur cette rubrique. [Afficher cette rubrique en ligne](#) dans le navigateur par défaut.

La spécification du langage C# est la source de référence pour la syntaxe C# et son utilisation. Cette spécification contient des informations détaillées sur tous les aspects du langage, notamment de nombreux éléments qui ne sont pas décrits dans la documentation pour Visual C#.

Vous pouvez télécharger cette spécification à partir du [Centre de téléchargement Microsoft](#). Si vous avez installé Visual Studio 2013, cette spécification se trouve sur votre ordinateur dans le dossier Program Files (x86)/Microsoft Visual Studio 12.0/VC#/Specifications/1033. Cependant, cette spécification n'est pas incluse dans les installations de Visual Studio Express 2013.

C'est finalement ce document de 500 pages environ qui nous permettra de puiser des détails.

La version transformée en .pdf est ajoutée sous K:\ES\Maitres-Eleves\SLO\Modules\SL228_POBJ\CoursC#



4.2. QUELQUES ASPECTS DU LANGAGE C#

Voici quelques aspect du langage C#.

4.2.1. MOTS CLÉS

Voici obtenu de la section Mots clés C# (mauvaise traduction ?)

[Mots clés C#](#)

Propose des liens vers des informations relatives aux mots clés et à la syntaxe C#.

Un tableau avec la liste des mots clés.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
Do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in (modificateur générique)	int	interface
internal	is	lock	long
namespace	new	null	object
Opérateur .	out	out (modificateur générique)	override
params	private	Protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	String	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
void	volatile	while	

4.3. VARIABLES ET CONSTANTE

Le C# est fortement typé, ce qui oblige assez souvent à utiliser des type cast et d'utiliser des post fixes pour les valeurs numériques.

4.3.1. LES TYPES DE DONNÉES DU C#

C# utilise les mêmes types que C++, mais n'accepte pas le mot clé **unsigned**, pour cela il y a des types spécifique qui correspondent aux type du :NET Framework.

Le tableau (source OpenClassroom), ci-dessous résume les types disponibles:

Type C#	Type .NET	Signification	Taille en mémoire (en octets)	Domaine de valeurs
<code>char</code>	Char	character (caractère)	2	caractère Unicode (UTF-16) allant de U+0000 à U+ffff
<code>string</code>	String	chaîne de caractères	variable	référence sur une séquence de caractères Unicode
<code>int</code>	Int32	integer (nombre entier)	4	$[-2^{31}; 2^{31}-1]$
<code>uint</code>	UInt32	unsigned integer (nombre entier non signé)	4	$[0; 2^{32}-1]$
<code>long</code>	Int64	nombre entier long	8	$[-2^{63}; 2^{63}-1]$
<code>ulong</code>	UInt64	unsigned long (nombre entier long non signé)	8	$[0; 2^{64}-1]$
<code>sbyte</code>	SByte	signed byte (octet signé)	1	$[-2^7; 2^7-1]$
<code>byte</code>	Byte	octet	1	$[0; 2^8-1]$
<code>short</code>	Int16	nombre entier court	2	$[-2^{15}; 2^{15}-1]$
<code>ushort</code>	UInt16	unsigned short (nombre entier court non signé)	2	$[0; 2^{16}-1]$
<code>float</code>	Single	flottant (nombre réel)	4	$\pm 1,5 \cdot 10^{-45}$ à $\pm 3,4 \cdot 10^{+38}$ (7 chiffres de précision)
<code>double</code>	Double	double flottant (nombre réel)	8	$\pm 5,0 \cdot 10^{-324}$ à $\pm 1,7 \cdot 10^{+308}$ (15 à 16 chiffres de précision)
<code>decimal</code>	Decimal	nombre décimal	16	$\pm 1,0 \cdot 10^{-28}$ à $\pm 7,9 \cdot 10^{+28}$ (28 à 29 chiffres significatifs)
<code>bool</code>	Boolean	booléen	1	true / false
<code>object</code>	Object	référence d'objet	variable	référence d'objet

Chacun des types C# présentés est un **alias** du type .NET associé, cela signifie que lorsque que vous écrivez par exemple `int`, c'est comme si vous aviez écrit `System.Int32`. Pour raccourcir et par habitude, on privilégie le type C# au type .NET.

4.3.1.1. PARTICULARITÉ DE CERTAIN TYPES C#

Le type **char** est 16 bits et n'est pas signé. Il supporte les caractères unicode. Un caractère unicode s'exprime par `"\uCCCC"` avec CCCC qui représente le code du caractère exprimé en hexadécimal.

Le type **byte** remplace le **unsigned char** du C++.

4.3.1.2. POST FIXE POUR TYPER LES VALEURS NUMÉRIQUES

Post fixe	Type de donnée	Exemple
L	long	1000000 L
U	uint	0xFFFF U
UL	ulong	0x80000000 UL
F	float	1.25 F
D	double	1.33333 D
M	decimal	50000 M

4.3.2. DÉCLARATION DES VARIABLES

Les variables C# se déclarent de la même manière qu'en C/C++, soit :

type NomVariable;

4.3.2.1. EXEMPLES DE DÉCLARATION DE VARIABLES

Voici un exemple de déclaration et affectations de valeur à des variables de différent types.

```
public partial class Form1 : Form
{
    int toto = 25;
    short titi = 234;
    char c1 = 'A';
    char c2 = (char)0x41;
    char c3 = '\u0041';
    byte b = 0xFF;
    long big = 123456789000000L;
    bool flag = true;
    float x = 1.25F;
    double y = 1.25;
    string Mess = "langage C#";
}
```

4.3.2.2. NOMS DES VARIABLES

le nom d'une variable est obligatoirement écrit avec des caractères alphanumériques (de préférence sans accent), ainsi que le underscore '_' (sans les apostrophes) :

abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789 ;

le nom doit forcément commencer par une lettre (underscore '_' compris) ;

le nom ne doit pas être le nom d'un type ou d'une classe déjà définis, d'un mot-clef du langage, ...(il ne doit pas être déjà pris). Bref, il ne faut pas que la confusion soit possible.

☞ C# fait la différence entre majuscules et minuscules (on dit que ce langage est case sensitive), donc les variables test et Test sont différentes.

Si le nom d'une variable se compose de 2 mots, on écrit le 2^{ème} mot avec une majuscule. Par exemple :

double tauxConversion;

4.3.2.3. LES TABLEAUX

Les tableaux en C# diffèrent du C/C++ car il doivent être alloués. Voici le principe de déclaration d'un tableau..

```
type[] nomTableau = new type[n];
```

Avec n qui donne le nombre d'élément du tableau. L'indice varie de 0 à n-1.

Par exemple :

```
int[] tableEch = new int[32];
```

Il est possible d'attribuer des valeurs aux éléments du tableau lors de la déclaration:

Par exemple :

```
short[] table2 = new short[] {10, 20, 30,40};
```

☞ dans ce cas on ne donne pas la dimension.

4.3.3. AFFECTATION ET EXPRESSION

Pour illustrer le typage assez fort du C# voici un exemple d'affectation d'une expression à une variable de type short.

```
short x1, x2;  
const short K = 10;
```

```
x1 = 1;  
x2 = (short)(10 * x1);  
x2 = (short)(K * x1);
```

☹ dans le 1er cas on peut comprendre le besoin du cast car 10 est considéré comme int, mais avec K qui est du type short c'est moins compréhensible.

4.3.4. PORTÉES DES VARIABLES

La portée d'une variable définit les parties du code qui en connaissent l'existence. Quand vous déclarez une variable dans une méthode, seul le code de cette méthode peut accéder à la valeur de cette variable ou la modifier. On dit que cette variable a une portée locale, c'est-à-dire limitée à cette méthode.

Comme en C# on utilise en général une classe, lorsque l'on veut une variable visible de toutes les méthodes de la classe, il s'agit de déclarer cette variable en tant qu'attributs de cette classe.

☹ Si on veut que cet attribut soit visible des autres classe il faut le déclarer public, ce qui est discutable au niveau des concepts d'encapsulation. il vaut mieux maintenir l'attribut **private** et introduire les méthodes public get et set.

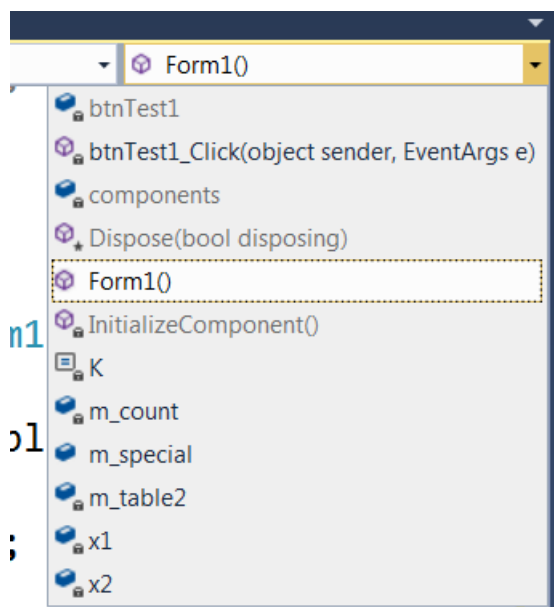
4.3.4.1. NOMS DES ATTRIBUTS

Pour éviter de confondre un attributs avec une variable locale ou un paramètre d'une méthode il est recommandé d'utiliser le m_nomAttribut. Par exemple :


```
public partial class Form1 : Form
{
    private short[] m_table2 = new short[] { 10, 20, 30, 40 };
    int m_count = 0;
    public int m_special;
```

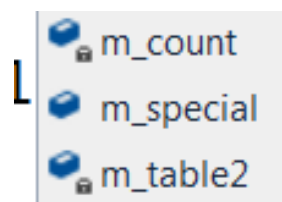
4.3.4.2. ACCESSIBILITÉ DES ATTRIBUTS

On dispose d'un menu déroulant listant les élément de la classe Form1.



On constate que le mot clé **private** ou son absence donne le même résultat !

 indique **private**



4.3.5. ATTRIBUTS CONSTANTES

En général si on a besoins d'une constante, c'est pour l'utiliser dans l'ensemble de la classe.

Avec le mot clé **const** il est ainsi possible de déclarer un attribut dont on ne pourra pas modifier ou lui affecter une nouvelle valeur.

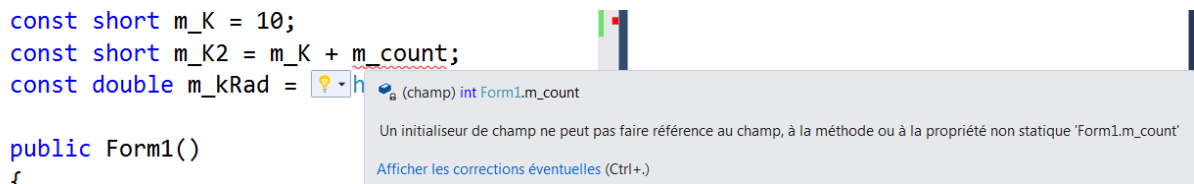
Vous déclarez un attribut constante à l'aide de la syntaxe suivante :

```
[public|private] const m_constantName = expression ;
```

Par exemple :

```
const short m_K = 10;
const double m_kRad = Math.PI / 180.0;
```

☞ L'expression ne peut pas contenir de variable ou d'attribut non statique !



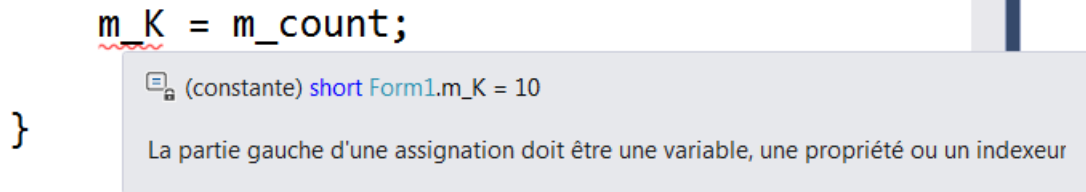
```
const short m_K = 10;
const short m_K2 = m_K + m_count;
const double m_kRad = ...
```

Un initialiseur de champ ne peut pas faire référence au champ, à la méthode ou à la propriété non statique 'Form1.m_count'

Afficher les corrections éventuelles (Ctrl+.)

4.3.5.1. MODIFICATION D'UNE CONSTANTE

L'affectation d'une valeur à une constante provoque une erreur de compilation :



```
m_K = m_count;
```

(constante) short Form1.m_K = 10

La partie gauche d'une assignation doit être une variable, une propriété ou un indexeur

4.3.6. LES ÉNUMÉRATIONS

Un type énumération (également nommé une énumération ou un **enum**) offre un moyen efficace pour définir un jeu de constantes intégrales nommées qui peuvent être assignées à une variable.

Par exemple, supposons que vous devez définir une variable dont la valeur représente un jour de la semaine. Cette variable ne stockera jamais plus de sept valeurs précises. Pour définir ces valeurs, vous pouvez utiliser un type énumération, déclaré en utilisant le mot clé **enum**. Voici l'exemple fournis par Microsoft.

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday,
            Friday, Saturday };
enum Months : byte { Jan, Feb, Mar, Apr, May, Jun, Jul,
                    Aug, Sep, Oct, Nov, Dec };
```

Par défaut, le type sous-jacent de chaque élément de l'enum est int. Vous pouvez spécifier un autre type numérique intégral en utilisant un deux-points, comme l'illustre l'exemple précédent.

4.3.6.1. ENUMÉRATION, EXEMPLE

Création d'un attribut de type enum.

```
namespace DemoVariables
{
    enum e_formes { Sinus, Triangle, Carre, DentDeScie };

    public partial class Form1 : Form
    {
        e_formes m_forme = e_formes.Sinus;
```

L'expression `enum e_formes` définit un type comme un typedef, d'où la possibilité de le placer en dehors de la classe dans le namespace. Pour l'attribut on utilise le type définit `e_formes`.

Exemple utilisation de l'attribut (`m_forme`) dans une méthode avec un switch. A chaque click sur le bouton on affiche `m_forme` et on utilise le switch, ensuite on incrémente `m_forme`. 📌 à noter comment les valeurs de l'énumération sont utilisées dans le case.

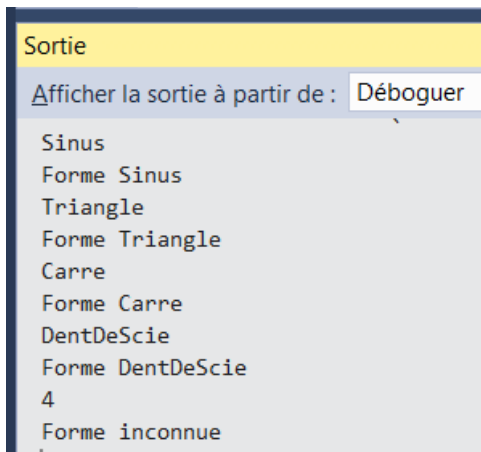
```
private void btnTest1_Click(object sender, EventArgs e)
{
    lblTest1.Text = m_forme.ToString();
    System.Console.WriteLine(m_forme);
    switch (m_forme)
    {
        case e_formes.Sinus:
            System.Console.WriteLine("Forme Sinus");
            break;
        case e_formes.Triangle:
            System.Console.WriteLine("Forme Triangle");
            break;
        case e_formes.Carre:
            System.Console.WriteLine("Forme Carre");
            break;
        case e_formes.DentDeScie:
            System.Console.WriteLine("Forme DentDeScie");
            break;
        default:
            System.Console.WriteLine("Forme inconnue");
            break;
    }
    m_forme++;
}
```

Résultat au niveau du label :



😊 `lblTest1.Text = m_forme.ToString();` fourni le nom de la valeur de l'enum.

Résultat au niveau de la console :



```
Sortie
Afficher la sortie à partir de : Déboguer
Sinus
Forme Sinus
Triangle
Forme Triangle
Carre
Forme Carre
DentDeScie
Forme DentDeScie
4
Forme inconnue
```

On constate que lorsque la valeur de `m_forme` dépasse les valeurs prévues dans l'énumération, on obtient la valeur numérique.

4.4. OPÉRATEURS

Au niveau de la référence C#, sous-section **Opérateurs C#** voici ce que nous trouvons concernant les opérateurs.

4.4.1. OPÉRATEURS PRINCIPAUX

Ce sont les opérateurs dont la priorité est la plus élevée.

x.y : accès au membre.

x?.y : accès au membre conditionnel null. Retourne null si l'opérande de gauche a la valeur null.

f(x) : appel de fonction.

a[x] : indexation de l'objet d'agrégation.

a?[x] : indexation conditionnelle null. Retourne null si l'opérande de gauche a la valeur null.

x++ : incrément suffixé. Retourne la valeur de x et met à jour l'emplacement de stockage avec la valeur de x augmentée de un (ajoute généralement l'entier 1).

x-- : décrément suffixé. Retourne la valeur de x et met à jour l'emplacement de stockage avec la valeur de x diminuée de un (soustrait généralement l'entier 1).

new : instanciation de type.

typeof : renvoie l'objet System.Type qui représente l'opérande.

checked : active le contrôle de dépassement de capacité pour les opérations sur les entiers.

unchecked : désactive le contrôle de dépassement de capacité pour les opérations sur les entiers. Il s'agit du comportement de compilateur par défaut.

default(T) : retourne la valeur initialisée par défaut de type T, la valeur null pour les types de référence, zéro pour les types numériques et zéro/null renseigné dans les membres pour les types de struct.

delegate : déclare et retourne une instance de délégué.

sizeof : retourne la taille en octets de l'opérande du type.

-> : déréréférencement de pointeur associé à l'accès au membre.

4.4.2. OPÉRATEURS UNAIRES

Ces opérateurs ont une priorité supérieure à celle de la section suivante et une priorité inférieure à celle de la section précédente.

+x : renvoie la valeur de x.

-x : négation numérique.

!x : négation logique.

~x : complément au niveau du bit.

++x : incrément préfixé. Retourne la valeur de x après avoir mis à jour l'emplacement de stockage avec la valeur de x augmentée de un (ajoute généralement l'entier 1).

--x : décrémentation préfixée. Retourne la valeur de x après avoir mis à jour l'emplacement de stockage avec la valeur de x diminuée de un (ajoute généralement l'entier 1).

(T)x : cast de type.

await : attend un Task.

&x : adresse de.

***x** : déréférencement.

4.4.3. OPÉRATEURS MULTIPLICATIFS

Ces opérateurs ont une priorité supérieure à celle de la section suivante et une priorité inférieure à celle de la section précédente.

x * y : multiplication.

x / y : division. Si les opérandes sont des entiers, le résultat est un entier tronqué vers zéro (par exemple, $-7 / 2$ is -3).

x % y : modulo. Si les opérandes sont des entiers, cet opérateur renvoie le reste de la division de x par y. Si $q = x / y$ et $r = x \% y$, alors $x = q * y + r$.

4.4.4. OPÉRATEURS ADDITIFS

Ces opérateurs ont une priorité supérieure à celle de la section suivante et une priorité inférieure à celle de la section précédente.

x + y : addition.

x - y : soustraction.

4.4.5. OPÉRATEURS DE DÉCALAGE

Ces opérateurs ont une priorité supérieure à celle de la section suivante et une priorité inférieure à celle de la section précédente.

x << y : décalage des bits vers la gauche et remplissage avec zéro à droite.

x >> y : décalage des bits vers la droite. Si l'opérande de gauche est **int** ou **long**, alors les bits de gauche sont remplis avec le bit de signe. Si l'opérande de gauche est **uint** ou **ulong**, alors les bits de gauche sont remplis avec zéro.

4.4.6. OPÉRATEURS RELATIONNELS ET DE TEST DE TYPE

Ces opérateurs ont une priorité supérieure à celle de la section suivante et une priorité inférieure à celle de la section précédente.

x < y : inférieur à (true si x est inférieur à y).

x > y : supérieur à (true si x est supérieur à y).

x <= y : inférieur ou égal à.

x >= y : supérieur ou égal à.

is : compatibilité du type. Retourne true si l'opérande de gauche évalué peut être converti en type spécifié dans l'opérande de droite (type statique).

as : conversion de type. Retourne l'opérande de gauche converti en type spécifié par l'opérande de droite (type statique), mais **as** retourne **null** où (T)x lèverait une exception.

4.4.7. OPÉRATEURS LOGIQUES

4.4.7.1. EGALITÉ ET INÉGALITÉ

x == y : égalité. Par défaut, pour les types de référence autres que **string**, cet opérateur retourne l'égalité de référence (test d'identité). Toutefois, des types peuvent surcharger **==**, donc si votre objectif est de tester l'identité, il est préférable d'utiliser la méthode `ReferenceEquals` sur **object**.

x != y : différent. Consultez le commentaire sur **==**. Si un type surcharge **==**, alors il doit surcharger **!=**.

4.4.7.2. AND, XOR ET OR BIT À BIT

x & y : AND logique ou au niveau du bit. L'utilisation avec des types entiers et des types **enum** est généralement autorisée.

x ^ y : XOR logique ou au niveau du bit. Vous pouvez l'utiliser généralement avec des types entiers et des types **enum**.

x | y : OR logique ou au niveau du bit. L'utilisation avec des types entiers et des types **enum** est généralement autorisée.

4.4.7.3. AND ET OR CONDITIONNELS

x && y : AND logique. Si le premier opérande a la valeur false, alors C# n'évalue pas le second opérande.

x || y : OR logique. Si le premier opérande a la valeur true, alors C# n'évalue pas le second opérande.

4.4.7.4. OPERATEUR DE FUSION DE NULL

x ?? y : retourne x si non-**null** ; sinon, retourne y.

4.4.7.1. OPERATEUR CONDITIONNEL

t ? x : y : si le test t a la valeur true, alors évalue et retourne x ; sinon, évalue et retourne y.

4.4.8. OPÉRATEURS D'ASSIGNATION ET LAMBDA

Ces opérateurs ont une priorité supérieure à celle de la section suivante et une priorité inférieure à celle de la section précédente.

x = y : assignation.

x += y : incrément. Additionne la valeur de y à la valeur de x, stocke le résultat dans x et retourne la nouvelle valeur. Si x désigne un **event**, alors y doit être une fonction appropriée que C# ajoute en tant que gestionnaire d'événements.

x -= y : décrémentation. Soustrait la valeur de y de la valeur de x, stocke le résultat dans x et retourne la nouvelle valeur. Si x désigne un **event**, alors y doit être une fonction appropriée que C# supprime en tant que gestionnaire d'événements.

x *= y : assignation de multiplication. Multiplie la valeur de y par la valeur de x, stocke le résultat dans x et retourne la nouvelle valeur.

x /= y : assignation de division. Divise la valeur de x par la valeur de y, stocke le résultat dans x et retourne la nouvelle valeur.

x %= y : assignation de modulo. Divise la valeur de x par la valeur de y, stocke le reste dans x et retourne la nouvelle valeur.

x &= y : assignation AND. Assigne l'opérateur AND à la valeur de y et la valeur de x, stocke le résultat dans x et retourne la nouvelle valeur.

x |= y : assignation OR. Assigne l'opérateur OR à la valeur de y et la valeur de x, stocke le résultat dans x et retourne la nouvelle valeur.

x ^= y : assignation XOR. Assigne l'opérateur XOR à la valeur de y et la valeur de x, stocke le résultat dans x et retourne la nouvelle valeur.

x <<= y : assignation de décalage vers la gauche. Décale la valeur de x vers la gauche de y places, stocke le résultat dans x et retourne la nouvelle valeur.

x >>= y : assignation de décalage vers la droite. Décale la valeur de x vers la droite de y places, stocke le résultat dans x et retourne la nouvelle valeur.

=> : déclaration lambda. **Nouveau à découvrir !**

4.4.9. DEPASSEMENT ARITHMETIQUE

Les opérateurs arithmétiques (+, -, *, /) peuvent produire des résultats qui sont en dehors de la plage de valeurs possibles pour le type numérique concerné. Reportez-vous à la section d'un opérateur particulier pour obtenir plus d'informations, mais en règle générale, les points suivants s'appliquent :

- Le dépassement arithmétique d'un entier lève soit une exception [OverflowException](#) ou ignore les bits les plus significatifs du résultat. La division d'un entier par zéro lève toujours une exception **DivideByZeroException**.
- Le dépassement arithmétique ou la division par zéro d'une virgule flottante ne lèvent jamais d'exception, car les types à virgule flottante se basent sur IEEE 754 et peuvent représenter l'infini et NaN (n'est pas un nombre).
- Le dépassement arithmétique d'un nombre [décimal](#) lève toujours une exception [OverflowException](#). La division d'un nombre décimal par zéro lève toujours une exception [DivideByZeroException](#).

En cas de dépassement d'un entier, ce qui se produit dépend du contexte d'exécution, lequel peut être [checked](#) ou [unchecked](#). Dans un contexte checked, une exception [OverflowException](#) est levée. Dans un contexte unchecked, les bits les plus significatifs du résultat sont ignorés et l'exécution se poursuit. Ainsi, C# vous donne la possibilité de traiter ou d'ignorer le dépassement.

En plus des opérateurs arithmétiques, des casts entre types intégraux peuvent générer un dépassement, par exemple, un cast de [long](#) en [int](#). Ils sont sujets à une exécution checked ou unchecked. En revanche, les opérateurs de bits et les opérateurs de décalage ne génèrent jamais de dépassement.

4.4.10. PRIORITÉ DES OPÉRATEURS

Lorsque plusieurs opérations sont contenues dans une même expression chacune d'elles est évaluée et résolue dans un ordre prédéfini, appelé priorité des opérateurs.

Voici un tableau obtenu de Wikipédia qui établit la liste des opérateurs et les classe par priorité décroissantes. Les opérateurs situés dans le même bloc ont la même priorité.

Opérateurs	Description	Associativité
::	Qualificateur d'alias d'espace de noms	de gauche à droite
() [] . ->	Parenthèses pour évaluer en priorité Tableau Sélection d'un membre par un identificateur (structures et objets) Sélection d'un membre par un pointeur (structures et objets)	
++ -- + - ! ~ (type) * & as is typeof sizeof new	Incrémentation post ou pré-fixée Opérateur moins unaire (change le signe de l'opérande) Non logique et Non binaire Conversion de type Déréférencement Référencement (adresse d'une variable) Conversion de type référence (pas d'exception lancée) Test de type Type d'une variable / expression Taille d'une variable / d'un type Allocation mémoire	
* / %	Multiplication, division, et modulo (reste d'une division)	
+ -	Addition et soustraction	de droite à gauche
<< >>	Décalage de bits vers la droite ou vers la gauche	
< <= > >=	Comparaison “ inférieur strictement ” et “ inférieur ou égal ” Comparaison “ supérieur strictement ” et “ supérieur ou égal ”	
== !=	Condition “ égal ” et “ différent ”	
&	ET binaire	
^	OU exclusif binaire / logique	
	OU binaire	

&&	ET logique booléen	
	OU logique booléen	
<i>c?t:f</i>	Opérateur ternaire de condition	de droite à gauche
=	Affectation	
<i>+= -=</i>	Affectation avec somme ou soustraction	
<i>*= /= %=</i>	Affectation avec multiplication, division ou modulo	
<i><<= >>=</i>	Affectation avec décalage de bits	
<i>&= ^= =</i>	Affectation avec ET, OU ou OU exclusif binaires	
,	Séquence d'expressions	de gauche à droite

Lorsqu'une même expression comprend une multiplication et une division, chaque opération est évaluée dans l'ordre d'apparition, de gauche à droite. Il en est de même des expressions contenant une addition et une soustraction. L'utilisation de parenthèses permet de modifier l'ordre de priorité afin qu'un élément d'une expression soit évalué avant les autres. Les opérations situées à l'intérieur de parenthèses sont toujours traitées avant les autres. La priorité des opérateurs s'applique cependant à l'intérieur des parenthèses.

Recommandation : utiliser les parenthèses cela facilite la lecture et spécifie exactement l'action à accomplir.

4.5. STRUCTURES DE CONTRÔLES: CHOIX ET BOUCLES

Au niveau de l'aide du C#, au niveau du *Guide de programmation C#*, se référer à la sous-section :

▲ Instructions, expressions et opérateurs (guide de programmation C#)

Instructions (Guide de programmation C#)

Expressions (Guide de programmation C#)

Opérateurs (guide de programmation C#)

▷ Fonctions anonymes (Guide de programmation C#)

Opérateurs surchargeables (Guide de programmation C#)

▷ Opérateurs de conversion (Guide de programmation C#)

Et finalement au niveau de [Instructions \(Guide de programmation C#\)](#) on trouve la liste suivante :

Instructions de sélection	Les instructions de sélection permettent de se brancher à différentes sections de code, en fonction d'une ou plusieurs conditions spécifiées. Pour plus d'informations, voir les rubriques suivantes : if , else , switch , case
Instructions d'itération	Les instructions d'itération permettent d'effectuer une boucle à travers des collections telles que des tableaux, ou d'exécuter à plusieurs reprises le même jeu d'instructions jusqu'à ce qu'une condition spécifiée soit remplie. Pour plus d'informations, voir les rubriques suivantes : do , for , foreach , in , while
Instructions de saut	Les instructions de saut transfèrent le contrôle à une autre section de code. Pour plus d'informations, voir les rubriques suivantes : break , continue , default , goto , return , yield
Instructions de gestion des exceptions	Les instructions de gestion des exceptions vous permettent d'effectuer une récupération « propre » suite à des conditions d'exception qui se produisent au moment de l'exécution. Pour plus d'informations, voir les rubriques suivantes : throw , try-catch , try-finally , try-catch-finally



Remarque : pour obtenir directement de l'aide sur un sujet, sélectionnez le mot clé dans le code et pressez sur F1 !

4.5.1. INSTRUCTION DE SÉLECTION

Instructions de sélection	Les instructions de sélection permettent de se brancher à différentes sections de code, en fonction d'une ou plusieurs conditions spécifiées. Pour plus d'informations, voir les rubriques suivantes : <i>if, else, switch, case</i>
---------------------------	---

4.5.1.1. IF..ELSE

La syntaxe des sélection **if else** est identique à celle du C standard.

Voici un exemple obtenu de la documentation :

L'exemple suivant détermine si un caractère d'entrée est une minuscule, une majuscule, ou un nombre. Si les trois conditions ont la valeur false, le caractère n'est pas un caractère alphanumérique. L'exemple affiche un message pour chaque cas.

```
Console.WriteLine("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}
```

4.5.1.2. SWITCH CASE DEFAULT

La construction **switch case default** a la même syntaxe que le C standard. Comme on dispose d'un type string il est possible en C# de réaliser un switch avec une variable de ce type.

Il est possible dans le case d'avoir une expression combinant des constantes, par exemple : **case 7 - 4:** ce qui fournit 3.

Voici un exemple obtenus de la documentation, mais adapté en ajoutant un bouton et un NumericUpDown. Les Console.WriteLine sont remplacé par l'affectation à un Label.

```
private void btnSwitch_Click(object sender, EventArgs e)
{
    int switchExpression = (int)nudT2.Value;
```

```
switch (switchExpression)
{
    // A switch section can have more than one case label.
    case 0:
    case 1:
        lblT2.Text = "Case 0 or 1";
        // Most switch sections contain a jump statement,
        // such as a break, goto, or return.
    break;
    case 2:
        lblT2.Text = "Case 2";
    break;
    // The following line causes a warning.
    lblT2.Text = "Unreachable code";
    // 7 - 4 in the following line evaluates to 3.
    case 7 - 4:
        lblT2.Text = "Case 3";
    break;
    // If the value of switchExpression is not 0, 1, 2, or 3,
    // the default case is executed.
    default:
        lblT2.Text = "Default case (optional)";
        // You cannot "fall through" any switch section,
        // including the last one.
    break;
} // end switch
}
```

```
case 2:
    lblT2.Text = "Case 2";
    break;
    // The following line causes a warning.
    lblT2.Text = "Unreachable code";
// 7 - 4 in the following line evaluates to 3.
case 7 - 4:
    lblT2.Text = "Case 3";
    break;
```

(champ) Label Form1.lblT2
Impossible d'atteindre le code détecté

☞ une ligne de code entre 2 section case break; est du code qui ne sera jamais exécuté.

Voici 2 résultats :

<div>Test switch</div> <div>1</div> <div>Case 0 or 1</div>	<div>Test switch</div> <div>4</div> <div>Default case (optional)</div>
--	--

4.5.2. LES INSTRUCTIONS D'ITÉRATIONS

Instructions d'itération	Les instructions d'itération permettent d'effectuer une boucle à travers des collections telles que des tableaux, ou d'exécuter à plusieurs reprises le même jeu d'instructions jusqu'à ce qu'une condition spécifiée soit remplie. Pour plus d'informations, voir les rubriques suivantes : <code>do</code> , <code>for</code> , <code>foreach</code> , <code>in</code> , <code>while</code>
--------------------------	--

4.5.2.1. L'INSTRUCTION WHILE

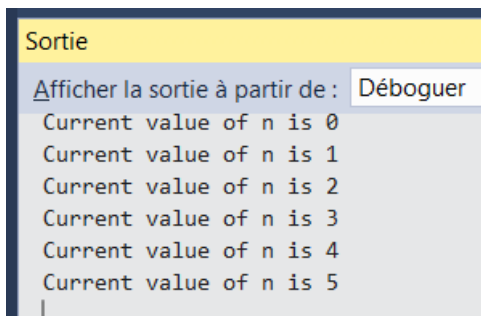
L'instruction **while** répète une instruction ou un bloc d'instructions jusqu'à ce qu'une expression spécifique corresponde à la valeur false.

Exemple :

```
int n = 0;

while (n < 6)
{
    Console.WriteLine("Current value of n is {0}", n);
    n++;
}
```

⚠ à noter l'expression `{0}` pour obtenir l'affichage de la valeur de `n` dans la console.



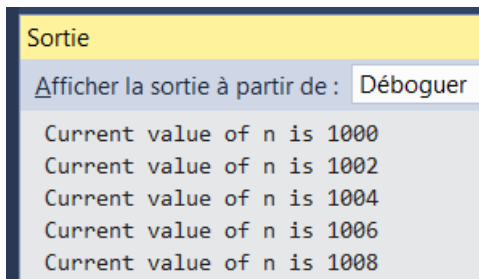
4.5.2.2. L'INSTRUCTION DO WHILE

L'instruction **do** répète une instruction ou un bloc d'instructions jusqu'à ce qu'une expression spécifique corresponde à la valeur false. Le corps de la boucle doit être placé entre accolades `{ }` à moins qu'il se compose d'une instruction unique. Dans ce cas, les accolades sont facultatives.

```
int n = 1000;

do
{
    Console.WriteLine("Current value of n is {0}", n);
    n += 2;
} while (n < 1010);
```


Résultat :

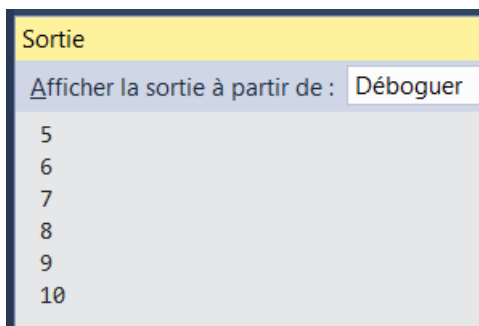


4.5.2.3. L'INSTRUCTION FOR

À l'aide d'une boucle **for** vous pouvez exécuter une instruction ou un bloc d'instructions à plusieurs reprises tant que l'expression spécifiée a la valeur true. Ce genre de boucle est utile pour itérer au sein des tableaux.

```
for (int i = 5; i <= 10; i++)
{
    Console.WriteLine(i);
}
```

☺ le C# permet la déclaration de la variable dans la () du for.



4.5.2.4. L'INSTRUCTION FOREACH ET IN

L'instruction **foreach** répète un groupe d'instructions incorporées pour chaque élément d'un tableau ou d'une collection d'objets qui implémente l'interface [System.Collections.IEnumerable](#) ou [System.Collections.Generic.IEnumerable<T>](#).

L'instruction foreach sert à itérer au sein de la collection pour obtenir les informations souhaitées, mais elle ne peut pas être utilisée pour ajouter ou supprimer des éléments dans la collection source, ceci afin d'éviter des effets secondaires imprévisibles.

Si vous devez ajouter ou supprimer des éléments dans la collection source, utilisez une boucle **for**.

4.5.2.4.1. Exemple avec un tableau

```
private void btnTforeach_Click(object sender, EventArgs e)
{
    int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };

    System.Console.WriteLine("Test foreach");
}
```

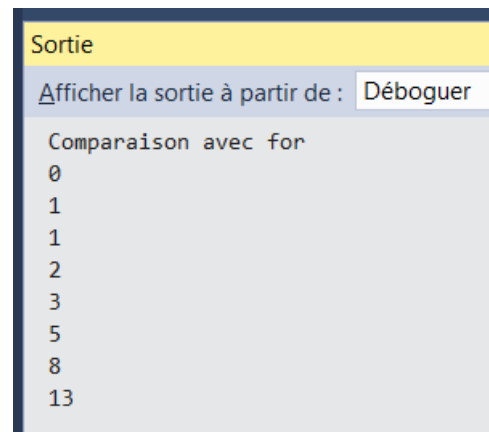
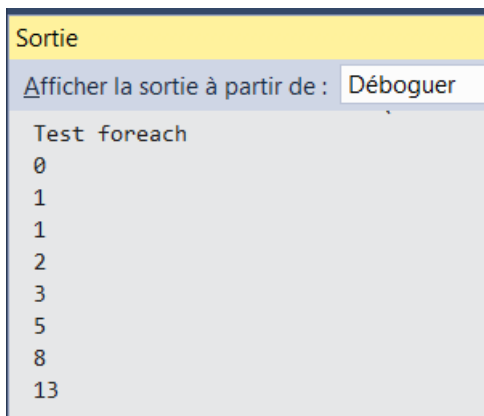
```

foreach (int element in fibarray)
{
    System.Console.WriteLine(element);
}
System.Console.WriteLine();

// Compare the previous loop to a similar for loop.
System.Console.WriteLine("Comparaison avec for");
for (int i = 0; i < fibarray.Length; i++)
{
    System.Console.WriteLine(fibarray[i]);
}
System.Console.WriteLine();
}

```

👉 le mécanisme de **foreach** avec **in** permet d'explorer un tableau sans connaître sa taille. Ceci est dû au fait que l'on peut connaître la taille du tableau par la méthode `.Length`, ce qui permet aussi l'utilisation de la boucle `for`.



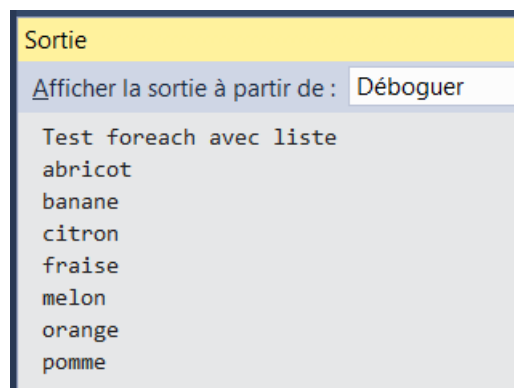
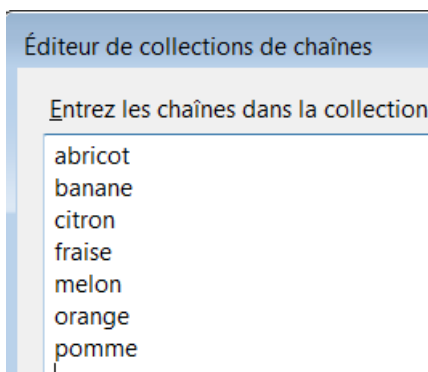
4.5.2.4.2. Exemple avec une liste

Cet exemple montre l'utilisation d'une boucle **foreach** pour lister le contenu d'une liste.

```

System.Console.WriteLine("Test foreach avec liste");
foreach (string element in cboFruits.Items)
{
    System.Console.WriteLine(element);
}

```



4.5.3. INSTRUCTIONS DE SAUT

Instructions de saut	Les instructions de saut transfèrent le contrôle à une autre section de code. Pour plus d'informations, voir les rubriques suivantes : break , continue , default , goto , return , yield
----------------------	---

4.5.3.1. L'INSTRUCTION BREAK

L'instruction **break** termine la boucle englobante la plus proche ou l'instruction [switch](#) dans laquelle elle apparaît. Le contrôle est transmis à l'instruction qui suit l'instruction terminée, s'il y en a une.

Exemple de la documentation C#.

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
1
2
3
4
*/
```

4.5.3.2. L'INSTRUCTION CONTINUE

L'instruction **continue** passe le contrôle à l'itération suivante d'un **while** ou d'un **for** englobant le continue.

Exemple de la documentation C#.

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Output:
9
10
*/
```

Comme on peut l'observer dans le résultat, lorsque l'instruction continue est appelée, l'action Console.WriteLine(i); n'est pas exécutée car le traitement saute directement au for.

4.5.3.3. L'INSTRUCTION GOTO

L'instruction **goto** transfère directement le contrôle du programme à une instruction étiquetée. le goto est utilisé couramment pour transférer le contrôle à une étiquette switch case spécifique ou à une étiquette default dans une instruction switch. L'instruction goto sert aussi à sortir des boucles profondément imbriquées.

☞ L'instruction **goto** ne peut effectuer un branchement que vers des lignes qui appartiennent à la procédure dans laquelle elle est utilisée.

L'exemple suivant illustre l'utilisation de goto pour sortir des boucles imbriquées. Reprise de l'exemple Microsoft en le transférant dans la méthode événementielle d'un bouton.

```
private void btnTgoto_Click(object sender, EventArgs e)
{
    int x = 200, y = 4;
    int count = 0;
    string[,] array = new string[x, y];

    // Initialize the array:
    for (int i = 0; i < x; i++)
        for (int j = 0; j < y; j++)
            array[i, j] = (++count).ToString();

    // Input a string: PAS OK
    // string myNumber = Console.ReadLine();

    string myNumber = nudT2.Value.ToString();
    Console.WriteLine("The number to search for is " +
                      myNumber);

    // Search:
    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            if (array[i, j].Equals(myNumber))
            {
                goto Found;
            }
        }
    }

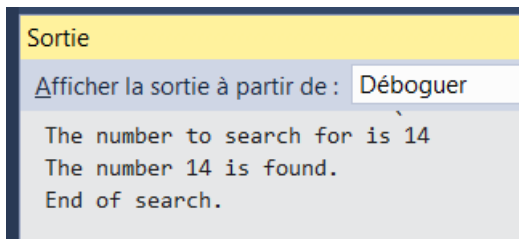
    Console.WriteLine("The number {0} was not found.", myNumber);
    goto Finish;

Found:
    Console.WriteLine("The number {0} is found.", myNumber);

Finish:
    Console.WriteLine("End of search.");
}
```

☞ à noter l'écriture des étiquette des goto, par exemple : **Found:**

Exemple de résultat avec la valeur du NumericUpDown à 14.



4.5.3.4. L'INSTRUCTION RETURN

L'instruction **return** termine l'exécution de la méthode où elle apparaît et rend le contrôle à la méthode appelante. Elle peut aussi retourner une valeur optionnelle. Si la méthode a un type de retour void, l'instruction return peut être omise.

Si l'instruction return est à l'intérieur d'un bloc try, le bloc finally, s'il existe, est exécuté avant que le contrôle retourne à la méthode appelante.

4.5.3.5. L'INSTRUCTION YIELD

Lorsque vous utilisez le mot clé **yield** dans une instruction, vous indiquez que la méthode, l'opérateur, ou l'accessor **get** dans lequel elle apparaît est un itérateur. L'utilisation de **yield** pour définir un itérateur rend une classe explicite supplémentaire inutile (la classe qui contient l'état d'une énumération ; pour obtenir un exemple, consultez [IEnumerator< T>](#) lorsque vous implémentez les modèles [IEnumerable](#) et [IEnumerator](#) pour un type de collection personnalisé.

L'exemple suivant montre les deux formes de l'instruction yield.

```
yield return <expression>;  
yield break;
```

☹ on notera l'existence de l'instruction yield, mais nous ne l'utiliseront pas pour l'instant car son rôle est peu compréhensible.

4.5.4. INSTRUCTIONS DE GESTION DES EXCEPTIONS

Instructions de gestion des exceptions	Les instructions de gestion des exceptions vous permettent d'effectuer une récupération « propre » suite à des conditions d'exception qui se produisent au moment de l'exécution. Pour plus d'informations, voir les rubriques suivantes : throw , try-catch , try-finally , try-catch-finally
--	---

4.5.4.1. LES INSTRUCTIONS TRY CATCH FINALLY

La construction **try ... catch**, permet de gérer (catch) les exception levée (throw) par l'action dans le bloc try.

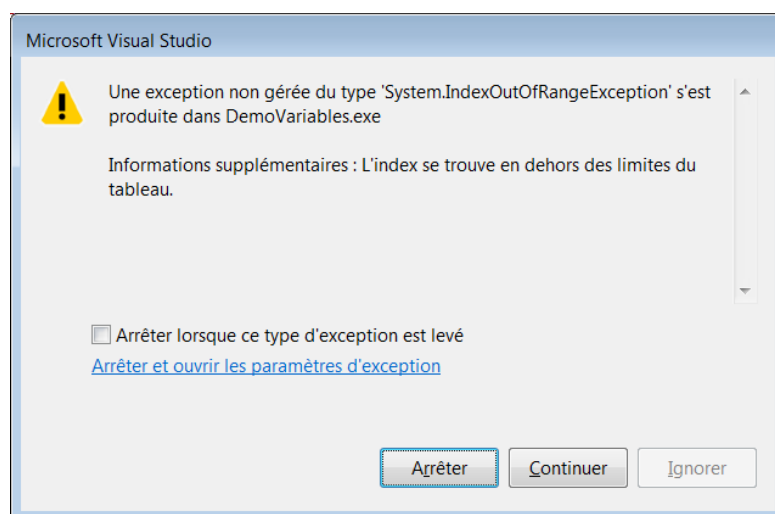
4.5.4.2. TRY CATCH, EXEMPLE

Dans cet exemple on utilise un tableau contenant un certain nombre d'élément et on accède à un élément en utilisant la valeur d'indice fournie par un NumericUpDown, ceci sans contrôle.

Voici la version de base de l'exemple :

```
private void btnTtryCatch_Click(object sender, EventArgs e)
{
    int[] myArray = new int[] { 10, 20, 30, 40, 50, 60,
                                70, 80, 90 };
    int idx = (int)nudT2.Value;
    // accède à un élément sans contrôle de la valeur
    // de l'indice
    lblValArray.Text = myArray[idx].ToString();
}
```

Si la valeur de l'indice dépasse 9 on aboutit à une exception non gérée :

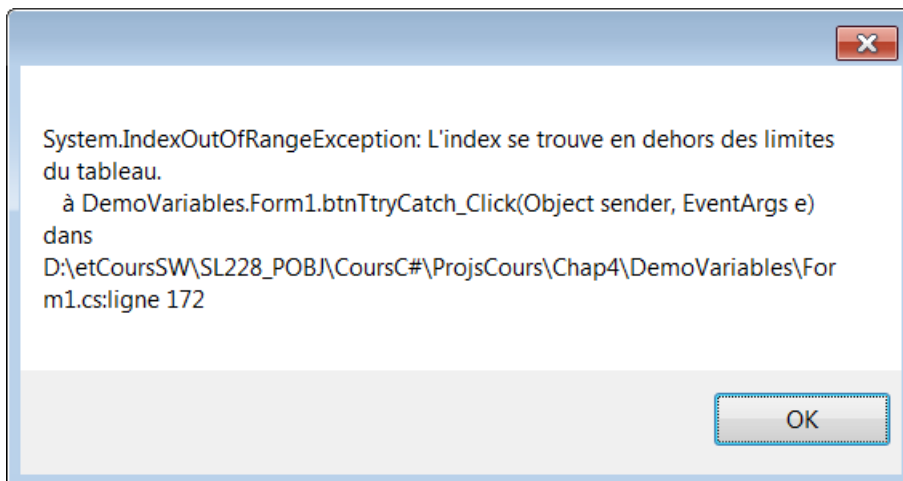


Nous allons introduire le try catch pour résoudre ce problème.

```
private void btnTtryCatch_Click(object sender, EventArgs e)
{
    int[] myArray = new int[] { 10, 20, 30, 40, 50, 60,
                                70, 80, 90 };
    int idx = (int)nudT2.Value;

    try
    {
        // accède à un élément avec indice inconnu
        lblValArray.Text = myArray[idx].ToString();
    }
    catch (Exception ex )
    {
        MessageBox.Show(ex.ToString());
    }
}
```

L'exception est maintenant gérée, lorsqu'elle se produit on obtient un affichage par une MessageBox. L'utilisateur est averti mais le programme continue son exécution.



4.5.4.3. AJOUT DU FINALLY

L'instruction finally permet d'accomplir une action dans les 2 situations (cas normal et avec exception).

Par exemple :

```
private void btnTtryCatch_Click(object sender, EventArgs e)
{
    int[] myArray = new int[] { 10, 20, 30, 40, 50, 60,
                                70, 80, 90 };
    int idx = (int)nudT2.Value;

    try
    {
```



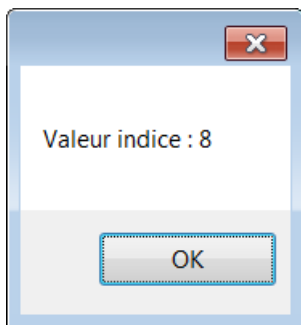
```

        // accède à un élément avec indice inconnu
        lblValArray.Text = myArray[idx].ToString();
    }
    catch (Exception ex )
    {
        MessageBox.Show(ex.ToString());
    }
    finally
    {
        MessageBox.Show("Valeur indice : " + idx);
    }
}

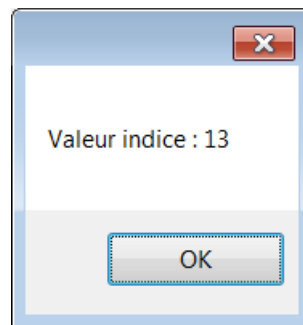
```

La section finally permet par exemple d'afficher la valeur de l'indice qu'il y ait eu ou non une exception.

cas normal



après message exception



4.5.4.4. L'INSTRUCTION THROW

L'instruction **throw** permet de lancer une exception. Cela permet par exemple de réaliser une méthode qui effectue un test et lance une exception pour avertir.

Voici un exemple réalisant la division de deux valeurs entières obtenues de deux TextBox. Malgré que au contraire du C++ une division par 0 génère une exception que l'on peut gérer, nous allons tout de même réaliser une méthode `safeDivInt` qui lance une exception en testant le diviseur.

4.5.4.4.1. La méthode `safeDivInt`

Elle montre la préparation et le lancement de l'exception.




```

private int safeDivInt(int num, int div)
{
    int res = 0;
    Exception ex;
    if ( div != 0 ) {
        res = num / div;
    } else {
        ex = new Exception ("safeDivInt : Erreur division par 0 !");
        throw ex;
    }
    return res;
}

```

4.5.4.4.2. Exception, constructeurs

Comme on peut le voir ci-dessous il y a 3 possibilité pour construire une exception. L'exemple précédant utilise la forme `Exception(string)`,

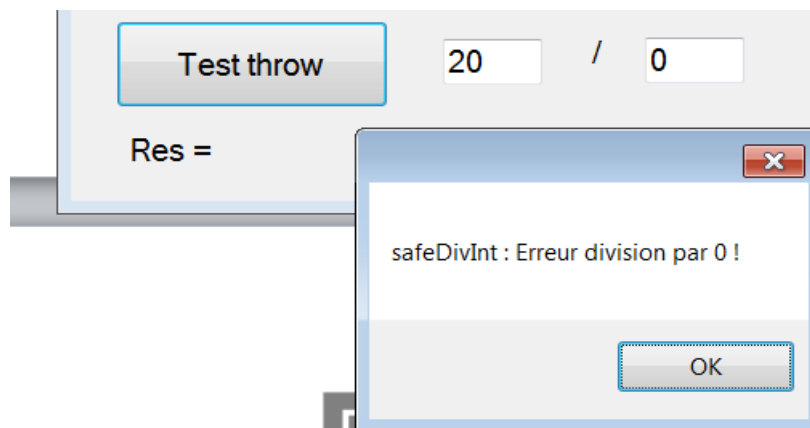
	Nom	Description
	<code>Exception()</code>	Initialise une nouvelle instance de la classe <code>Exception</code> .
	<code>Exception(String)</code>	Initialise une nouvelle instance de la classe <code>Exception</code> avec un message d'erreur spécifié.
	<code>Exception(String, Exception)</code>	Initialise une nouvelle instance de la classe <code>Exception</code> avec un message d'erreur spécifié et une référence à l'exception interne ayant provoqué cette exception.

4.5.4.4.3. Utilisation de la méthode `safeDivInt`

Voici dans l'événement d'un bouton l'utilisation de la méthode `safeDivInt` avec le `try catch`.

```
private void btnTthrow_Click(object sender, EventArgs e)
{
    int num = int.Parse(txtNum.Text);
    int div = int.Parse(txtDivi.Text);
    try {
        lblRes.Text = safeDivInt(num, div).ToString();
    }
    catch (Exception ex) {
        MessageBox.Show(ex.Message);
    }
}
```

On notera l'utilisation dans le message du `catch` de `ex.Message` pour obtenir uniquement le message de l'exception sans les info de ou elle s'est produite.



4.6. CONCLUSION

Ce chapitre a présenté l'essentiel de la syntaxe du Visual C# concernant les données, opérateurs et instructions. Il reste encore à découvrir comment réaliser des méthodes et surtout comment utiliser les nombreux éléments fournies par les classes du .NET Framework dans le cadre du C#.

4.7. HISTORIQUE DES VERSION

4.7.1. VERSION 1.0 MARS 2016

Création de ce chapitre en transformant (beaucoup) le chapitre correspondant du cours VB. Quelques simplification par le fait que la syntaxe du C# correspond en grande partie à celle du C.