Les mots clés de C# 1.0

par Jérôme Lambert (Mes articles)

Date de publication : 20/03/2007

Dernière mise à jour : 26/03/2007

Suite à l'article **Les mots réservés du langage Java** de Baptiste Wicht, il était temps de faire de même.

Au travers de cet article, vous découvrirez ce qu'est un mot clé avec une présentation de chacun des 77 mots clés du langage C#.

I - Introduction....4
II - Des mots pas si réservés que ça....4
II-A - Première technique - sensibilité à la casse....4
II-B - Seconde technique - préfixe @....4
III - Liste des mots clés....4
IV - Description des mots clés....5
abstract....5
as....5
base....6
bool....6
break....6
byte....7
case....7
catch....7
char....8
checked....8
class....8
const....9
continue....9
decimal....9
default....9
delegate....10
do....10
double....11
else....11
enum....11
event....12
explicit....12
extern....13
false....13
finally....13
fixed....14
float....14
for....14
foreach....15
goto....15
if....16
implicit....16
in....16
int....17
interface....17
internal....17
is....17
lock....18
long....18
namespace....18
new....19
null....19
object....19
operator....19
out....20
override....21
params....21
private....21
protected....22
public....22
readonly....22
ref....22

return....23
sbyte....23
sealed....23
short....23
sizeof....24
stackalloc....24
static....24
string....25
struct....26
switch....26
this....26
throw....27
true....27
try....27
typeof....27
uint....27
ulong....28
unchecked....28
unsafe....28
ushort....28
using....29
virtual....30
volatile....30
void....30
while....30
V - Remerciements....31

I - Introduction

Tout d'abord, un mot clé (ou keyword an anglais) est un mot réservé qui a une signification particulière pour le compilateur. Les mots clés ne pourront donc pas être utilisés pour identifier les variables, classes ou encore les fonctions.

Au cours de cet article, l'ensemble des mots clés de C# vous seront présentés. Vous verrez aussi comment les utiliser pour identifier des variables, méthodes voir classes contrairement à ce qui a été dit précédemment.

II - Des mots pas si réservés que ça...

Bien que déconseillé, il vous est tout à fait possible d'utiliser les mots clés pour identifier les variables, classes et autres.

II-A - Première technique - sensibilité à la casse

C# étant sensible à la casse, vous pouvez donc utiliser un mot clé en modifiant sa casse d'origine :

```
void While()
{
    int iF = 0;
}
```

II-B - Seconde technique - préfixe @

La seconde technique permet d'utiliser un mot clé en le préfixant de **@** :

```
void @while()
{
    int @if = 0;
}
```

III - Liste des mots clés

Voici la liste des mots clés qui seront détaillés par la suite :

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**	**int**	**interface**
internal	**is**	**lock**	**long**	**namespace**
new	**null**	**object**	**operator**	**out**
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**	**volatile**
void	**while**			

IV - Description des mots clés

abstract

abstract est utilisé avec les classes, méthodes, propriétés, indexeurs et évènements. En utilisant **abstract** dans la déclaration d'une classe, cela signifiera que cette classe est une classe de base qui ne pourra pas être instanciée directement. Pour ce qui est des membres marqués comme abstraits ou contenus dans une classe abstraite, ils devront être implémentés dans la classe qui dérive de la classe abstraite.

```
abstract public class Animal
{
    abstract public string Cri();
}

public class Chien : Animal
{
    public override string Cri()
    {
        return "Waf !!!";
    }
}

static void Main()
{
    // Animal monAnimal = new Animal(); -> Impossible
    Chien monChient = new Chien();
}
```

Plus d'infos sur le mot clé ici.

as

as permet de vérifier qu'un objet est bien d'un certain de type. Si la condition est vérifiée un casting implicite est effectué pour récupérer l'objet sous le bon type, dans le cas contraire, **null** sera renvoyé.

```
abstract public class Animal
{
    abstract public string Cri();
}

public class Chien : Animal
{
    public override string Cri()
    {
        return "Waf !!!";
    }
}
public class Chat : Animal
{
    public override string Cri()
    {
        return "Miaouuu !!!";
    }
}

static void Main()
{
    Object[] mesAnimaux = { new Chien(), new Chat() };

    foreach (Object AnimalCourant in mesAnimaux)
    {
        Chien monChien = AnimalCourant as Chien;
        Chat monchat = AnimalCourant as Chat;

        if (monChien != null)
```

```
            Console.WriteLine("Voici le cri du chien : {0}", monChien.Cri());
        else if (monchat != null)
            Console.WriteLine("Voici le cri du chat : {0}", monchat.Cri());
        else
            Console.WriteLine("Animal inconnu !");
    }
}
```

Plus d'infos sur le mot clé ici.

base

base permet d'accéder depuis une classe dérivée à la classe de base.

```
public class Chien : Animal
{
    public override string Cri()
    {
        return "Waf !!!";
    }
}

public class Labrador : Chien
{
    public string CriLabrador()
    {
        return base.Cri();
    }
}
```

Plus d'infos sur le mot clé ici.

bool

bool est un alias de la structure **System.Boolean**. Vous pourrez donc aussi bien utiliser **bool** que **Boolean** pour déclarer une variable booléenne.

```
bool monBoolAlias = true;
Boolean monBooleanStructure = monBoolAlias;
```

Plus d'infos sur le mot clé ici.

break

break permet de sortir d'une boucle ou d'un **switch**.

Cas de la boucle

```
for (int i = 0; i < 5; i++)
{
    if (i == 2)
    {
        Console.WriteLine("On peut quitter la boucle !");
        break;
    }
}
```

Cas du switch

```
Console.WriteLine("Voulez-vous quitter ? [O pour oui - N pour non]");
string ret = Console.ReadLine();
```

Cas du switch

```
switch (ret.ToUpper()[0])
{
    case 'O':
        Console.WriteLine("Au revoir");
        break;
    case 'N':
        Console.WriteLine("On continue alors...");
        break;
    default:
        Console.WriteLine("Commande inconnue !");
        break;
}
```

Plus d'infos sur le mot clé ici.

byte

byte est un alias de la structure **System.Byte**. Vous pourrez y stocker une valeur entière non signée codée sur 8 bits (de 0 à 255).

```
byte monByteAlias = 5;
Byte = monByteStructure = monByteAlias;
```

Plus d'infos sur le mot clé ici.

case

case représente un des aiguillages de l'instruction **switch**.

```
Console.WriteLine("Voulez-vous quitter ? [O pour oui - N pour non]");
string ret = Console.ReadLine();

switch (ret.ToUpper()[0])
{
    case 'O':
        Console.WriteLine("Au revoir");
        break;
    case 'N':
        Console.WriteLine("On continue alors...");
        break;
    default:
        Console.WriteLine("Commande inconnue !");
        break;
}
```

Plus d'infos sur le mot clé ici.

catch

catch permet de traiter une exception qui a été levée dans le bloc **try** précédent. Si les instructions dans le bloc **try** peuvent lever plusieurs exceptions différentes, il est possible de les intercepter en faisant suivre le bloc **try** par plusieurs bloc **catch**.

```
try
{
    int i = 0;
    int j = 5;
```

```
    int ret = j / i;
}
catch (DivideByZeroException exc)
{
    Console.WriteLine("Voici le message de l'erreur : {0}", exc.Message);
}
```

Remarque : Si vous désirez intercepter n'importe quelle exception dans un unique bloc **catch**, il vous suffit d'intercepter l'exception **Exception** qui est la classe de base de toute exception.

Plus d'infos sur le mot clé ici.

char

char est un alias de la structure **System.Char**. Il vous permet de déclarer un caractère unicode.

```
char monCharAlias = 'A';
Char monCharStructure = monCharAlias;
```

Plus d'infos sur le mot clé ici.

checked

Par défaut, C# ne signale aucune erreur en cas de dépassement de capacité sur des entiers. La valeur résultante sera donc erronée dans un tel cas. **checked** permet de générer une exception en cas de dépassement de capacité lors d'opérations arithmétiques, conversions ou encore division par 0.

```
int i = 0;
int j = 1;

// i = -2147483648
i = Int32.MaxValue + j;

try
{
    int k = 0;
    int l = 1;
    k = checked(Int32.MaxValue + l);
}
catch (Exception)
{
    Console.WriteLine("Dépassement de capacité !");
}
```

Plus d'infos sur le mot clé ici.

class

class, comme son l'indique, permet de déclarer une classe.

```
class maClasse
{ }
```

Plus d'infos sur le mot clé ici.

const

const permet de déclarer une variable en tant que constante. L'initialisation devra se faire en même temps que la déclaration car il ne sera plus possible de modifier la valeur de la variable par après.

```
const int maConstante = 2007;
```

Plus d'infos sur le mot clé ici.

continue

continue va de paire avec break. **continue** permet de passer à l'itération suivante dans une boucle contrairement à **break** qui permet de sortir de la boucle.

```
for (int i = 0; i < 5; i++)
{
    if (i == 1)
    {
        // On passe directement à l'itération suivante sans afficher le message
        continue;
    }

    Console.WriteLine("Numéro de l'itération : {0}", i);
}
```

Plus d'infos sur le mot clé ici.

decimal

decimal est un alias de la structure **System.Decimal**. Ce type permet d'avoir une plus grande précision que le type **float**. On peut y stocker une valeur de type flottante codée sur 128 bits (environ 8*10^28).

```
decimal monDecimalAlias = 9.1m;
Decimal monDecimalStructure = monDecimalAlias;
```

Plus d'infos sur le mot clé ici.

default

default a 2 significations :
Dans un bloc **switch**, **default** représente le bloc par défaut lorsque aucun aiguillage ne correspond.

```
switch (3)
{
    case 1: Console.WriteLine("Valeur 1"); break;
    case 2: Console.WriteLine("Valeur 2"); break;
    default: Console.WriteLine("Valeur par defaut"); break;
}
```

Sa seconde utilité est de pouvoir récupérer la valeur par défaut d'un type valeur ou référence. On y trouve une utilité lors de l'utilisation de générique où on sait déterminer si on a affaire à un type valeur ou référence.

```
public class MaClasse<T>
{
    public MaClasse(T param)
    {
```

```
        Object obj = default(T);

        if (obj == null)
            Console.WriteLine("T est un type référence !");
        else
            Console.WriteLine("T est un type valeur !");
    }
}

static void Main()
{
    MaClasse<int> test1 = new MaClasse<int>(5);
    MaClasse<string> test2 = new MaClasse<string>("Salut");
}
```

Plus d'infos sur le mot clé ici.

delegate

delegate permet de déclarer un type référence qui permettra d'encapsuler plusieurs méthodes nommées ou anonymes.

```
delegate void D();

static void MethodA()
{
    Console.WriteLine("Méthode A");
}
static void MethodB()
{
    Console.WriteLine("Méthode B");
}

static void Main()
{
    D monDelegue = new D(MethodA);
    // Affiche :
    // Méthode A
    monDelegue();

    monDelegue += new D(MethodB);
    // Affiche :
    // Méthode A
    // Méthode B
    monDelegue();

    monDelegue -= new D(MethodA);
    // Affiche :
    // Méthode B
    monDelegue();


    D monDelegueAnonyme = delegate { Console.WriteLine("Ceci est une méthode anonyme !"); };
    // Affiche :
    // Ceci est une méthode anonyme !
    monDelegueAnonyme();
}
```

Plus d'infos sur le mot clé ici.

do

do permet d'exécuter en boucle un bloc d'instructions tant que la condition renvoie vrai. A noter que la condition ne sera vérifiée qu'après avoir exécuté une première fois le bloc d'instruction.

```
static void Main()
{
    int i = 0;
    do
    {
        i++;
    } while (i < 5);
}
```

Plus d'infos sur le mot clé ici.

double

double est un alias de la structure **System.Double**. Vous pourrez stocker dans ce type une valeur à virgule flottante codée sur 64 bits (de 4.9*10^-324 à 1.8*10^308).

```
double monDoubleAlias = 13.7;
Double = monDoubleStructure = monDoubleAlias;
```

Plus d'infos sur le mot clé ici.

else

else va de paire avec **if**. Si la condition du **if** n'est pas vérifiée, on passera au bloc **else**.

```
Boolean valeur = true;
if (valeur == true)
{
    Console.WriteLine("La condition est vraie");
}
else
{
    Console.WriteLine("La condition est fausse");
}
```

Plus d'infos sur le mot clé ici.

enum

enum permet de déclarer une énumération qui représente un ensemble fini de constantes nommées.

```
enum Jours
{
    Lundi,
    Mardi,
    Mercredi,
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}
```

Plus d'infos sur le mot clé ici.

event

event permet de déclarer un évènement. Les évènements permettent de notifier des occurrences qu'un état donné a été modifié (clique sur un bouton par exemple).

```
delegate void TimeEventHandler();

class TestTime
{
    public event TimeEventHandler OnTime;

    public void time()
    {
        OnTime();
    }
}

static class Program
{
    static void Main()
    {
        TestTime monTime = new TestTime();

        // On s'abonne à l'évènement
        monTime.OnTime += new TimeEventHandler(monTime_OnTime);

        // Notification
        monTime.time();
    }

    /// <summary>
    /// Handler de l'évènement onTime
    /// </summary>
    static void monTime_OnTime()
    {
        MessageBox.Show("Evènement !");
    }
}
```

Plus d'infos sur le mot clé ici.

explicit

explicit permet de déclarer un opérateur explicite de conversion.

```
public class Franc
{
    public Double Argent;

    public Franc(Double param_Argent)
    {
        Argent = param_Argent;
    }
}

public class Euro
{
    public Double Argent;

    public Euro(Double param_Argent)
    {
        Argent = param_Argent;
    }

    public static explicit operator Euro(Franc param_Franc)
    {
```

```
        return new Euro(param_Franc.Argent / 6.55957);
    }
}

static class Program
{
    static void Main()
    {
        Franc mesFranc = new Franc(1000);
        Euro ConversionEuro = (Euro)mesFranc;

        Console.WriteLine("{0} FF = {1} €", mesFranc.Argent, ConversionEuro.Argent);
    }
}
```

Plus d'infos sur le mot clé ici.

extern

extern permet de déclarer une méthode comme étant implémentée en externe. C'est-à-dire dans une DLL non managée. **extern** sera le plus souvent utilisé avec l'attribut **DllImport**.

```
static class Program
{
    /// <summary>
    /// Permet d'afficher un MessageBox
    /// </summary>
    [DllImport("User32.dll")]
    public static extern int MessageBox(int h, string m, string c, int type);

    static void Main()
    {
        MessageBox(0, "Mon texte", "Mon titre", 0);
    }
}
```

Plus d'infos sur le mot clé ici.

false

false est un opérateur représentant la valeur faux d'un **bool** (ou **Boolean**).

```
bool monBool = false;
```

Plus d'infos sur le mot clé ici.

finally

finally est le troisième bloc possible avec un **try/catch**. Il sera toujours exécuté peu importe ce qu'il arrive : en quittant le bloc **try** (normalement ou avec return) ou après le bloc **catch** suite à une exception.
Le bloc **finally** est la plupart du temps (pour ne pas dire tout le temps) utilisé pour libérer les ressources allouées dans le bloc **try**.

```
static void Main()
{
    ArrayList monTableau = new ArrayList();
    monTableau.Add("5");
    monTableau.Add("9");
    monTableau.Add("s");
```

```
    try
    {
        for (int i = 0; i < monTableau.Count; i++)
        {
            int monInt = Convert.ToInt32(monTableau[i]);
        }
    }
    catch (Exception)
    {
        Console.WriteLine("Il y a eu une exception !");
    }
    finally
    {
        monTableau = null;
    }
}
```

Plus d'infos sur le mot clé ici.

fixed

fixed est utilisé dans les contexte **unsafe**, c'est à dire avec utilisation de pointeurs. Dotnet peut à tout moment modifier l'emplacement en mémoire de telle ou telle autre variable ce qui pourrait poser problème si on désire parcourir un tableau à partir de l'adresse de la première cellule. **fixed** va permettre d'empêcher que le Garbage Collector déplace la variable dans la mémoire.

```
static class Program
{
    unsafe static void Main()
    {
        int[] tab = new int[2007];

        // p pointe sur la première case de tab
        fixed (int* p = tab)
        {
            Random random = new Random();
            for (int i = 0; i < tab.Length; i++)
                p[i] = random.Next();
        }
    }
}
```

Plus d'infos sur le mot clé ici.

float

float est un alias de la structure **System.Single**. Vous pourrez stocker dans ce type une valeur à virgule flottante codée sur 32 bits (de 1.4*10^-45 à 3.4*10^38).

```
float monFloatAlias = 13.7F;
Single monSingleStructure = monFloatAlias;
```

Plus d'infos sur le mot clé ici.

for

for permet de répéter une instruction ou un bloc d'instructions tant que l'expression évaluée renvoie vrai.

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

```
    Console.WriteLine("{0}", i);
}
```

La boucle **for** ci-dessus se déroule comme suit :
- i est initialisé à 0 ;
- i est évalué : si i est bien inférieure à 10, on exécute le bloc d'instructions ;
- Une fois le bloc exécuté, i est incrémenté de 1 ;
- Pour être une nouvelle fois évalué ;
- Cette séquence se répètera tant que i sera strictement inférieure à 10

Plus d'infos sur le mot clé ici.

foreach

foreach permet de parcourir une collection afin de récupérer un par un chaque élément afin d'exécuter un bloc d'instructions.

```
string[] stringTab = { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };

foreach (string str in stringTab)
{
    Console.WriteLine("{0}", str);
}
```

Plus d'infos sur le mot clé ici.

goto

goto permet de sauter à une instruction étiquetée. Cette pratique est vivement déconseillée car on peut très vite se retrouver avec un code dont il est difficile de savoir ce qui sera exécuté sous telle ou telle autre condition. On appelle ça la " programmation spaghetti ".

```
static class Program
{
    static void Main()
    {
        int i = (new Random()).Next();

        if (i % 2 == 0)
            goto Pair;
        else
            goto Impair;


    Pair:
        Console.WriteLine("i est un nombre pair ({0})", i);
        return;

    Impair:
        Console.WriteLine("i est un nombre impair ({0})", i);
        return;

    }
}
```

Plus d'infos sur le mot clé ici.

if

if permet de tester une ou plusieurs condition afin d'exécuter un bloc d'instructions dans le cas où la condition est vrai. En cas de condition qui renvoie faux, on passera au test de la condition du bloc suivant (**else if**) ou au bloc par défaut (**else**) s'il y en a un.

```
int i = 0;

if (i == 0)
{
    Console.WriteLine("i vaut {0}", i);
}
else if (i == 1)
{
    Console.WriteLine("i vaut {0}", i);
}
else
{
    Console.WriteLine("i ne vaut ni 0, ni 1");
}
```

Plus d'infos sur le mot clé ici.

implicit

implicit est utilisé pour déclarer un opérateur de conversion implicite défini dans une classe.

```
class NombreEntier
{
    public int valeur;
    public NombreEntier(int param)
    {
        valeur = param;
    }

    // Opérateur de conversion en int
    public static implicit operator int(NombreEntier a)
    {
        return a.valeur;
    }
}
static class Program
{
    static void Main()
    {
        NombreEntier monNombre1 = new NombreEntier(5);

        // Conversion
        int monInt = (int)monNombre1;
        Console.WriteLine("monInt vaut {0}", monInt);

    }
}
```

Plus d'infos sur le mot clé ici.

in

in fonctionne avec le mot clé **foreach** qui permet de parcourir une collection. **in** est utilisé pour spécifier la collection dans laquelle il faudra récupérer chaque élément.

```
string[] stringTab = { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

```
foreach (string str in stringTab)
{
    Console.WriteLine("{0}", str);
}
```

Plus d'infos sur le mot clé ici.

int

int est un alias de la structure **System.Int32**. Vous pourrez stocker dans ce type une valeur entière signée codée sur 32 bits (de -2^31 à 2^31 - 1).

```
int monIntAlias = 2007;
Int32 monInt32Structure = monIntAlias;
```

Plus d'infos sur le mot clé ici.

interface

interface permet de déclarer une interface.

```
interface MonInterface
{
    void MethodeInterface();
}
```

Plus d'infos sur le mot clé ici.

internal

internal est un qualificatif qui s'applique aux classes ainsi qu'aux champs et méthodes d'une classe. Dans ce cas, seules les méthodes de la même **assembly** peuvent avoir accès à un champ ou une méthode qualifiée de **internal**.

```
internal class maClasse
{ }
```

Plus d'infos sur le mot clé ici.

is

is permet de tester si un objet est bien d'un type donné.

```
object[] mesObjets = { 1, "Salut", 13.7F };

foreach (object obj in mesObjets)
{
    if (obj is int)
        Console.WriteLine("int");
    else if (obj is string)
        Console.WriteLine("string");
    else
        Console.WriteLine("Type inconnu");
}
```

Plus d'infos sur le mot clé ici.

lock

lock permet de déclarer une section critique dans le cas d'une application multithreads en verrouillant l'accès à un objet. Tout thread qui tentera d'accéder à cette section critique et utiliser par la même occasion l'objet verrouillé se verra bloqué tant que la ressource ne sera pas libérée.

```
object obj = new object();

lock (obj)
{
    // Début section critique

    // instructions

    // Fin section critique
}
```

Plus d'infos sur le mot clé ici.

long

long est un alias de la structure **System.Int64**. Vous pourrez stocker dans ce type une valeur entière signée codée sur 64 bits (de -9.2*10^18 à 9.2*10^18).

```
long monLongAlias = 1000000000;
Int64 monInt64Structure = monLongAlias;
```

Plus d'infos sur le mot clé ici.

namespace

namepace sert à spécifier une portée afin d'organiser ses classes. C'est comme si on rangeait les classes dans des dossiers pour avoir une organisation qui permet de s'y retrouver facilement.
Répertoire "Opérations" dans le projet :

```
namespace MonNamespace.Operations
{
    class Operations
    { }
}
```

Avec 2 sous répertoires "Entiers" et "Reels" :

```
namespace MonNamespace.Operations.Entiers
{
    class Entiers
    { }
}

namespace MonNamespace.Operations.Reels
{
    class Reels
    { }
}
```

Le Framework Class Library (FCL) de Dotnet est organisée en **namespace** afin d'y avoir un accès intuitif avec des noms de **namespace** significatifs (comme **System.xml** pour les classes gérant le **Xml**).

Plus d'infos sur le mot clé ici.

new

new consiste à allouer sur le **heap** la mémoire nécessaire pour l'objet en question. Les variables allouées sur le **heap** sont les chaînes de caractères, tableaux et objets tandis que les types valeurs (**int**, **long**, **bool**,) et les structures sont alloués sur la pile.

```
int[] monTableauInt = new int[5];
```

Plus d'infos sur le mot clé ici.

null

null permet de mettre une référence à nulle. Par défaut, une référence vaut **null**.

```
// obj1 et obj2 valent null
object obj1;
object obj2 = null;
```

Plus d'infos sur le mot clé ici.

object

object est un alias de **System.Object** et représente la classe de base aussi bien pour les objets de type référence que valeur.

```
class Oiseau
{ }

static class Program
{
    static void Main()
    {
        // Casting d'un type valeur en object
        int monInt = 10;
        object monIntDansObject = monInt;

        // Casting d'un type référence en object
        Oiseau monOiseau = new Oiseau();
        object monOiseauDansObject = monOiseau;
    }
}
```

Plus d'infos sur le mot clé ici.

operator

operator est utilisé pour définir le comportement d'une classe par rapport à un opérateur arithmétique, de conversion ou encore logique.

```
class NombreEntier
```

```
{
    public int valeur;
    public NombreEntier(int param)
    {
        valeur = param;
    }

    // Opérateur arithmétique +
    public static NombreEntier operator +(NombreEntier a, NombreEntier b)
    {
        return new NombreEntier(a.valeur + b.valeur);
    }

    // Opérateurs logiques == et !=
    public static bool operator ==(NombreEntier a, NombreEntier b)
    {
        return a.valeur == b.valeur;
    }
    public static bool operator !=(NombreEntier a, NombreEntier b)
    {
        return a.valeur != b.valeur;
    }

    // Opérateur de conversion en int
    public static implicit operator int(NombreEntier a)
    {
        return a.valeur;
    }
}
static class Program
{
    static void Main()
    {
        NombreEntier monNombre1 = new NombreEntier(5);
        NombreEntier monNombre2 = new NombreEntier(6);

        // Opération +
        NombreEntier monNombre3 = monNombre1 + monNombre2;
        Console.WriteLine("monNombre3 vaut {0}", monNombre3.valeur);

        // Comparaison
        if (monNombre1 != monNombre3)
            Console.WriteLine("monNombre1 et monNombre3 sont différents");
        else
            Console.WriteLine("monNombre1 et monNombre3 sont égaux");

        // Conversion
        int monInt = (int)monNombre1;
        Console.WriteLine("monInt vaut {0}", monInt);

    }
}
```

Plus d'infos sur le mot clé ici.

out

out permet de passer un paramètre par référence sans qu'il ny ait besoin de l'initialiser.

```
static class Program
{
    static void Main()
    {
        int monInt;

        maFonction(out monInt);

        // monInt vaut 2007
        Console.WriteLine("monInt vaut {0}", monInt);
```

```
    }

    static void maFonction(out int param_int)
    {
        param_int = 2007;
    }
}
```

Plus d'infos sur le mot clé ici.

override

override permet de ré implémenter une méthode virtuelle ou abstraite d'une classe de base dans la classe dérivée.

```
public class classA
{
    public virtual void ExempleMethode()
    {
        Console.WriteLine("ExempleMethode de classA");
    }
}

public class classB : classA
{
    public override void ExempleMethode()
    {
        Console.WriteLine("Surcharge de ExempleMethode de classB");
    }
}
```

Plus d'infos sur le mot clé ici.

params

params est utilisé pour passer une nombre variable de paramètres à une fonction.

```
static void Main()
{
    MaFonctionVariable("Bonjour");
    MaFonctionVariable("Bonjour", "Au revoir");
}

static void MaFonctionVariable(params String[] MesParams)
{
    foreach (String courantString in MesParams)
        Console.WriteLine("Valeur du paramètre : {0}", courantString);
}
```

Plus d'infos sur le mot clé ici.

private

private est un qualificatif qui s'applique aux classes ainsi qu'aux champs et méthodes d'une classe. Dans ce cas, seules les méthodes de la même classe peuvent y avoir accès ce qui signifie qu'une classe dérivée ne pourra avoir accès à un champ ou une méthode qualifiée de **private**.

```
private class maClasse
{ }
```

Plus d'infos sur le mot clé ici.

protected

protected est un qualificatif qui s'applique aux classes ainsi qu'aux champs et méthodes d'une classe. Dans ce cas, seules les méthodes de la même classe ou des classes dérivées peuvent avoir accès à un champ ou une méthode qualifiée de **protected**.

```
protected class maClasse
{ }
```

Plus d'infos sur le mot clé ici.

public

public est un qualificatif qui s'applique aux classes ainsi qu'aux champs et méthodes d'une classe. Dans ce cas, n'importe quelle méthode de n'importe quelle classe peut avoir accès à un champ ou une méthode qualifiée de **public**.

```
public class maClasse
{ }
```

Plus d'infos sur le mot clé ici.

readonly

readonly permet de qualifier un champ en tant que constante qui pourra être initialisé au plus tard dans le constructeur de la classe.

```
class TestConstante
{
    readonly string monReadOnly;

    public TestConstante(string monParam)
    {
        monReadOnly = monParam;
    }
}
```

Plus d'infos sur le mot clé ici.

ref

ref permet de passer un paramètre par référence.

```
static class Program
{
    static void Main()
    {
        int monInt = 2006;

        // monInt vaut 2006
        Console.WriteLine("monInt avant l'appel de fonction vaut {0}", monInt);

        maFonction(ref monInt);

        // monInt vaut 2007
```

```
        Console.WriteLine("monInt après l'appel de fonction vaut {0}", monInt);
    }

    static void maFonction(ref int param_int)
    {
        param_int++;
    }
}
```

Plus d'infos sur le mot clé ici.

return

return permet de terminer l'exécution d'une méthode voir de retourner une valeur à la méthode appelante.

```
void ExampleReturn(int param)
{
    if (param == 0)
        // Met fin à l'exécution de la méthode
        return;
}

int ExampleReturnValeur(int param)
{
    // retourne le carré de la valeur passée en paramètre
    return param * param;
}
```

Plus d'infos sur le mot clé ici.

sbyte

sbyte est un alias de la structure **System.SByte**. Vous pourrez y stocker une valeur entière signée codée sur 8 bits (de -128 à 127).

```
sbyte monsByteAlias = -5;
SByte = monSByteStructure = monsByteAlias;
```

Plus d'infos sur le mot clé ici.

sealed

sealed permet d'empêcher une classe d'être dérivée.

```
sealed class ClassImpossibleAHeriter
{
    // ...
}
```

Plus d'infos sur le mot clé ici.

short

short est un alias de la structure **System.Int16**. Vous pourrez stocker dans ce type une valeur entière signée codée sur 16 bits (de -2^15 à 2^15 - 1).

```
short monShortAlias = 2007;
Int16 monInt16Structure = monShortAlias;
```

Plus d'infos sur le mot clé ici.

sizeof

sizeof permet de récupérer la taille en octet d'un type valeur.

```
static class Program
{
    static void Main()
    {
        Console.WriteLine("byte vaut {0} octets", sizeof(byte));
        Console.WriteLine("short vaut {0} octets", sizeof(short));
        Console.WriteLine("int vaut {0} octets", sizeof(int));
        Console.WriteLine("long vaut {0} octets", sizeof(long));
    }
}
```

Plus d'infos sur le mot clé ici.

stackalloc

stackalloc est utilisé dans un contexte **unsafe** et permet d'allouer un bloc de mémoire sur la pile.

```
int* psa = stackalloc int[100];
```

Plus d'infos sur le mot clé ici.

static

static permet de qualifier des champs comme des méthodes d'une classe qui existent sans instance de la classe. Ils sont donc communs à toutes les instances d'une classe.
Pour y accéder, on utilisera le nom de la classe.

```
static class Program
{
    static void Main()
    {
        // "Le compteur vaut 0"
        ExempleStatic.AfficheCompteur();

        ExempleStatic monObj1 = new ExempleStatic();

        // "Le compteur vaut 1"
        ExempleStatic.AfficheCompteur();

        ExempleStatic monObj2 = new ExempleStatic();

        // "Le compteur vaut 2"
        ExempleStatic.AfficheCompteur();
    }
}

/// <summary>
/// Classe qui compte le nombre d'instance
/// </summary>
class ExempleStatic
{
```

```
    static int Compteur = 0;

    public ExempleStatic()
    {
        Compteur++;
    }

    public static void AfficheCompteur()
    {
        Console.WriteLine("Le compteur vaut {0}", Compteur);
    }
}
```

Plus d'infos sur le mot clé ici.

string

string est un alias de **System.String** et permet de stocker une chaîne de caractères. Malgré le fait que ce soit un type référence, à chaque opération, un nouveau **string** est créé.

```
static class Program
{
    static void Main()
    {
        string monString = "Bonjour";
        System.String monStringAlias = monString;

        // "Le string vaut Bonjour"
        Console.WriteLine("Le string vaut {0}", monStringAlias);

        // "Le string vaut Bonjour tout le monde !!!"
        AfficheString(monStringAlias);

        // La référence n'a pas été modifiée
        // "Le string vaut Bonjour"
        Console.WriteLine("Le string vaut {0}", monStringAlias);
    }

    static void AfficheString(string param_String)
    {
        param_String = param_String + " tout le monde !!!";
        Console.WriteLine("Le string vaut {0}", param_String);
    }
}
```

La technique pour contourner ce problème est de passer la référence de la référence elle-même avec **ref** ou **out** :

```
static void Main()
{
    //...

    // La référence a enfin été modifiée
    // "Le string vaut Bonjour tout le monde !!!"
    Console.WriteLine("Le string vaut {0}", monStringAlias);
}

static void AfficheStringRef(ref string param_String)
{
    param_String = param_String + " tout le monde !!!";
}
```

Plus d'infos sur le mot clé ici.

struct

struct sert à déclarer une structure.

```
struct ExempleStructure
{
    int monInt;
    long monLong;
}
```

A noter qu'une structure est un type valeur étant donnée qu'elle dérive de **System.ValueType**.

Plus d'infos sur le mot clé ici.

switch

switch permet de faire un aiguillage vers une ou plusieurs instructions en fonction d'une variable de contrôle. La variable de contrôle peut être une valeur entière (**byte**, **short**, **int**, **long**), un caractère (**char**), une chaîne de caractères (**string**) ou encore une valeur d'énumération.

```
switch ((new Random()).Next(5))
{
    case 0: Console.WriteLine("La valeur vaut 0"); break;
    case 1: Console.WriteLine("La valeur vaut 1"); break;
    case 2: Console.WriteLine("La valeur vaut 2"); break;
    case 3: Console.WriteLine("La valeur vaut 3"); break;
    default: Console.WriteLine("La valeur n'est pas gérée"); break;
}
```

Plus d'infos sur le mot clé ici.

this

this est utilisé pour faire référence à l'objet en cours et ainsi accéder aux membres de l'objet en question.

```
class ExempleThis
{
    private String MonString;

    public ExempleThis()
    {
        MonString = "Hello";
    }

    private String ReturnWorld()
    {
        return "World";
    }

    public void DisplayHelloWorld()
    {
        Console.WriteLine(this.MonString + " " + this.ReturnWorld());
    }
}
```

Plus d'infos sur le mot clé ici.

throw

throw permet de lever une exception afin de signaler un problème.

```
void ExempleMethode(int param_Int)
{
    if (param_Int == 0)
        throw new Exception("Le paramètre ne peut être egal à 0 !!!");
}
```

Plus d'infos sur le mot clé ici.

true

true est un opérateur représentant la valeur vrai d'un **bool** (ou **Boolean**).

```
bool monBool = true;
```

Plus d'infos sur le mot clé ici.

try

try permet d'englober un bloc d'instructions qui est dit "protégé". C'est-à-dire que si une exception est levée dans ce bloc, elle sera traitée dans un des blocs **catch** qui suit.

```
try
{
    int i = 0;
    int j = 5;

    int ret = j / i;
}
catch (DivideByZeroException exc)
{
    Console.WriteLine("Voici le message de l'erreur : {0}", exc.Message);
}
```

Plus d'infos sur le mot clé ici.

typeof

typeof permet de récupérer un objet **System.Type** sur base d'un type.

```
Type monType = typeof(int);
```

Plus d'infos sur le mot clé ici.

uint

uint est un alias de la structure **System.UInt32**. Vous pourrez stocker dans ce type une valeur entière non signée codée sur 32 bits (de 0 à 4294967295).

```
uint monUIntAlias = 2007;
UInt32 monUInt32Structure = monUIntAlias;
```

Plus d'infos sur le mot clé ici.

ulong

ulong est un alias de la structure **System.UInt64**. Vous pourrez stocker dans ce type une valeur entière non signée codée sur 64 bits (de 0 à 18*10^18).

```
ulong monULongAlias = 1000000000;
UInt64 monUInt64Structure = monULongAlias;
```

Plus d'infos sur le mot clé ici.

unchecked

unchecked permet de désactiver le contrôle de dépassement de capacité sur des entiers ou lors de conversion.

```
unchecked
{
    int i = int.MaxValue + 1;
}
```

Sans le mot clé **unchecked**, l'instruction ne passerait même pas à la compilation car il s'agit d'une opération sur des constantes.

Plus d'infos sur le mot clé ici.

unsafe

unsafe permet de déclarer un bloc d'instructions pouvant utiliser les pointeurs.

```
unsafe
{
    int i = 2006, j;
    int* k;

    // k pointe sur i
    k = &i;

    // j contient 2007
    j = *k + 1;
}
```

A noter que pour utiliser les pointeurs dans un projet, il faut au préalable les activer. Pour cela allez dans les propriétés du projet, sélectionnez l'onglet "Build" et cochez la case "allow unsafe code".

Plus d'infos sur le mot clé ici.

ushort

ushort est un alias de la structure **System.UInt16**. Vous pourrez stocker dans ce type une valeur entière non signée codée sur 16 bits (de 0 à 65535).

```
ushort monUShortAlias = 2007;
UInt16 monUInt16Structure = monUShortAlias;
```

Plus d'infos sur le mot clé ici.

using

Il y a 3 utilisations pour le mot clé **using**.
Première utilisation: permettre l'utilisation de types dans un espace de noms :

```
using System;

namespace WindowsApplication3
{
    static class Program
    {
        static void Main()
        {
            System.Object obj1 = new System.Object();
            // ou
            Object obj2 = new Object();
        }
    }
}
```

Seconde utilisation : créer un alias en rapport avec un espace de noms :

```
using System;

using MonAlias = System.Object;

namespace WindowsApplication3
{
    static class Program
    {
        static void Main()
        {
            System.Object obj1 = new System.Object();
            // ou
            MonAlias obj2 = new MonAlias();
        }
    }
}
```

Troisième utilisation : définir la portée d'existence d'un objet. Pour être plus précis, la méthode **Dispose** de l'objet en question sera appelée automatiquement une fois que le bloc d'instructions sera terminé. Ce qui implique donc que l'objet utilisé dans un **using** devra implémenter l'interface **IDisposable** :

```
class Oiseau : IDisposable
{
    public void Dispose()
    { }
}

static class Program
{
    static void Main()
    {
        Oiseau monOiseau = new Oiseau();

        using (monOiseau)
        { }
    }
}
```

Plus d'infos sur le mot clé ici.

virtual

virtual est utilisé pour les méthodes, indexeurs, propriétés et évènements signifiant que ces derniers peuvent être ré implémentés dans une classe dérivée.

```
public class classA
{
    public virtual void ExempleMethode()
    {
        Console.WriteLine("ExempleMethode de classA");
    }
}

public class classB : classA
{
    public override void ExempleMethode()
    {
        Console.WriteLine("Surcharge de ExempleMethode de classB");
    }
}
```

Plus d'infos sur le mot clé ici.

volatile

volatile est utilisé sur les champs qui sont accédés par plusieurs threads différents. Cela permet que la dernière valeur du champ soit toujours présente.

```
public volatile string monStringVolatile = "Ma chaîne";
```

Plus d'infos sur le mot clé ici.

void

void est utilisé pour signaler qu'une méthode ne renvoie aucune valeur.

```
void ExempleMethode()
{ }
```

Plus d'infos sur le mot clé ici.

while

while représente une boucle. Tant que la condition renvoie vrai, le bloc d'instructions est exécuté. La condition est vérifiée avant d'exécuter le bloc d'instructions pour la première fois.

```
int i = 0;
while (i < 5)
{
    Console.WriteLine("i vaut {0}", i);

    i++;
}
```

Plus d'infos sur le mot clé ici.

V - Remerciements

Je tiens à remercier **Ditch** pour son aide et sa patience ainsi que **Nicolas Jolet** pour ses corrections.