# C#

# Les bases

# Table des matières


*Introduction* ..... *6*

**Présentation** ..... **6**

**Qu'est ce que le C# ?** ..... **6**

**.NET, qu'est ce que c'est ?** ..... **6**

**Ce qu'il vous faut pour écrire et lancer du code C#** ..... **7**

**Conventions** ..... **7**

*Chapitre 1* ..... *8*

*Premier programme* ..... *8*

**Premier programme avec Visual C#** ..... **8**

**Examen du code source** ..... **11**

**Présentation rapide des espaces de noms** ..... **12**

**Les blocs** ..... **12**

**Le squelette du programme** ..... **13**

**Les commentaires** ..... **13**

**Exercices d'application** ..... **13**

Exercice 1 ..... 13

Exercice 2 ..... 13

Exercice 3 ..... 13

Exercice 4 ..... 13

*Chapitre 2* ..... *14*

*Les variables* ..... *14*

**Définition** ..... **14**

**Explications** ..... **14**

**Déclaration et affection d'une variable** ..... **14**

**Exemple** ..... **15**

**Examen du code** ..... **15**

**Exercice d'application** ..... **16**

Exercice 5 ..... 16

**Réutilisation d'une variable** ..... **16**

**Caractères spéciaux** ..... **16**

**Exercices d'application** ..... **17**

Exercice 6 ..... 17

Exercice 7 ..... 17

**Les chaînes de caractères** ..... **17**

**Exercices** ..... **18**

Exercice 8 ..... 18
Exercice 9 ..... 18
**Saisie de variables ..... 18**
**Opérations sur les nombres ..... 19**
**Concaténation ..... 20**
**Exercices ..... 21**
Exercice 10 ..... 21
Exercice 11 ..... 21
Exercice 12 ..... 21
Exercice 13 ..... 21
***Chapitre 3 ..... 22***
***Les conditions ..... 22***
**If et else ..... 22**
**Les opérateurs logiques ..... 22**
**Opérateurs logiques purs ..... 23**
**Vrai ou faux ..... 23**
**Combinaison ..... 23**
**Un exemple ..... 24**
**Accolades ..... 24**
**Else if ..... 25**
**Exercices ..... 25**
Exercice 14 ..... 25
Exercice 15 ..... 25
Exercice 16 ..... 25
**Switch / case ..... 26**
Syntaxe ..... 26
Sémantique ..... 26
Exemples ..... 27
Exercice 17 ..... 27
**L'opérateur ternaire ..... 27**
Syntaxe ..... 27
Exemple ..... 27
Sémantique de l'exemple ..... 27
***Chapitre 4 ..... 29***
***Les boucles ..... 29***
**Faire… tant que ..... 29**
Syntaxe ..... 29
Sémantique ..... 29
Exemple ..... 29
Contre-exemple : une boucle interminable ..... 29
**Tant que… ..... 30**
Syntaxe ..... 30
Exemple ..... 30
**La boucle for ..... 30**
Syntaxe ..... 30
Exemple ..... 31

Remarque ... 31
**Des boucles dans des boucles ... 31**
**Exercices ... 32**
Exercice 18 ... 32
Exercice 19 ... 32
***Chapitre 5 ... 33***
***Les classes ... 33***
**Ce que vous savez sur les classes ... 33**
**Déclaration d'une classe ... 33**
**Les méthodes ... 34**
Syntaxe ... 34
Exemple ... 35
**Débuguer un programme ... 36**
**Exercices ... 37**
Exercice 20 ... 37
Exercice 21 : Calculatrice ... 37
**Les champs ... 37**
Syntaxe ... 37
Exemple ... 38
**Exercice d'application directe ... 38**
Exercice 22 ... 38
**Les propriétés ... 38**
Syntaxe ... 38
Exemple ... 39
**Constructeurs de classes ... 40**
Syntaxe ... 41
Exemple ... 41
**This ... 42**
**Exercice ... 43**
Exercice 23 ... 43
***Chapitre 6 ... 44***
***Les tableaux ... 44***
**Syntaxe ... 44**
**Travailler avec des tableaux ... 46**
**Foreach ... 46**
**Exercice d'application ... 48**
Exercice 24 ... 48
**Les tableaux multidimensionnels ... 49**
Exemple de déclaration d'une matrice ... 49
Syntaxe ... 49
Exemple ... 49
**Exercices ... 50**
Exercice 25 ... 50
Exercice 26 ... 51
***Projet ... 52***

**Introduction** ..... **52**

**Conventions de nommage** ..... **52**

**Projet** ..... **53**

Préambule ..... 53

Projet ..... 53

Partie codeur ..... 54

Remarques ..... 54

**Pour aller plus loin** ..... **54**

# Introduction

## Présentation

Ce tutorial a pour objectif de vous apprendre à maîtriser les bases de la programmation en C#. Il s'adresse aux débutants et ne nécessite aucune connaissance préalable en informatique.

Les notions d'interfaces graphiques, de services, de gestion de données ou autres ne sont pas traitées dans cette section.
De même les notions ou explications dont l'intérêt s'avère limité pour le débutant ne seront pas abordées ici.

## Qu'est ce que le C# ?

Le C# (se prononce C Sharp) est un nouveau langage de programmation créé par Microsoft et publié officiellement en 2002. Il a été spécialement conçu pour tirer tous les avantages de la technologie .NET. Il permet d'en exploiter toutes les fonctionnalités.

C# étant basé sur .NET, il bénéficie d'une excellente conception et se trouve être, dans de nombreux cas, bien plus agréable et plus efficace que les anciens langages C++ et VB.

C# est un langage orienté objet dont les concepts et la syntaxe viennent de C++ et Java.

Un programme C# doit répondre à une syntaxe[1] très stricte. Cette syntaxe est très proche de celle du C++.

C# est un langage compilé. Pour compiler du C#, il faut un compilateur[2]. Cette étape s'appelle la compilation.
Vous pouvez réaliser un grand nombre d'applications grâce à C#, notamment des :

- Applications Windows
- Pages ASP.NET
- Services Web
- Services Windows

## .NET, qu'est ce que c'est ?

.NET (ou DotNet) est une bibliothèque recouvrant l'ensemble des fonctions Windows.
C'est aussi un environnement d'exécution pour vos applications.

Les programmes ne sont plus compilés pour un système d'exploitation particulier mais pour .NET. C'est-à-dire que n'importe quel système d'exploitation pourra exploiter votre programme du moment que .NET y est implémenté.

.NET est sécurisé et facilite l'interopérabilité[3] des programmes.

Les composants de .NET sont appelés le .NET Framework.

---

[1] Ensemble des règles d'écriture.
[2] Programme qui traduit un langage, le langage source, en un autre, appelé langage cible, en préservant la signification du texte source.
[3] Il s'agit de la communication des programmes entre eux.

Le .NET Framework est une sorte de machine virtuelle vous permettant d'exécuter vos applications .NET (et donc C#) et d'interagir avec elles

*Remarque : Le .NET Framework ne fonctionne que sous Windows. Pour programmer sous Linux ou MacOs, il vous faut utiliser Mono ou DotGNU qui ont un fonctionnement similaire au Framework mais pour d'autres plateformes.*

# Ce qu'il vous faut pour écrire et lancer du code C#

.NET fonctionne sur Windows 98, 2000, XP et versions plus récentes…
Pour écrire du code C#, le bloc note suffit. Il vous faut ensuite un compilateur C# et le .NET Framework pour lancer vos programmes.

Il est toutefois conseillé et préférable d'utiliser un EDI[4] qui regroupe un logiciel de traitement de texte spécialisé au langage (reconnaissance des mots clés[5], etc.), un compilateur, un débugger, et autres outils facilitant le développement d'applications.
Parmi les EDI, les plus connus sont :

- Microsoft Visual C# *payant ou version Express moins développée*
- Borland C# Builder *payant ou version d'essai limité dans le temps*
- SharpDevelop *gratuit mais moins poussé que les précédents*

Nous utiliserons ici Visual C# Express 2005, qui est un excellent EDI, dans sa version anglaise pour vous habituer dès le début au fait que l'univers de la programmation fonctionne principalement en anglais.
Vous ne pourrez l'installer que sous Windows XP minimum. Le Framework et autres composants nécessaires sont automatiquement installés.

# Conventions

Voici quelques exemples des conventions de ce tutorial, ceci afin d'en simplifier la lecture et la compréhension :

- Les termes importants sont en **gras**.
- Les mots clés tels que `if`, `else`, `while`, `class`… apparaissent comme ceci.

- Un bloc de code que vous pouvez exécuter est placé dans une fenêtre de ce style :

```
static void Main()
{
    Fonction();
}
```

- *Les conseils, les suggestions et les informations sont présentés comme ceci.*

- Les informations importantes sont mentionnées de cette manière :

| **Ceci est une information importante, ne l'oubliez pas !** |
|---|

- Pour des raisons évidentes de mise en page, lorsque le code sortira de la seule ligne d'où il aurait du se trouver, le caractère ⇒ signalera que la ligne précédente et courante ne forment qu'une.

[4] Editeur de Développement Intégré
[5] Ensemble des mots prédéfinis par le langage de programmation.

# Chapitre 1
# Premier programme

## Premier programme avec Visual C#

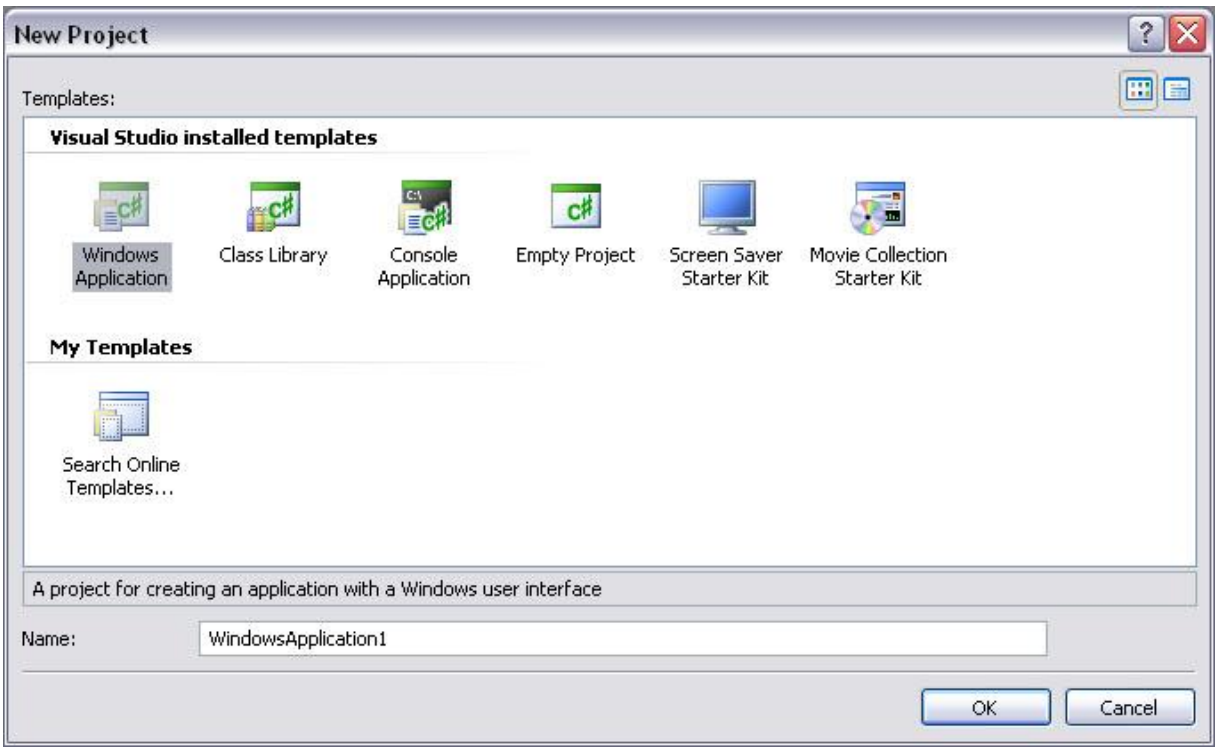
Après avoir installé Visual C#, vous disposez de tous les composants nécessaires pour réaliser votre premier programme.

Commencez par ouvrir Visual C#.
Par défaut, dans Démarrer → Tous les programmes → Microsoft Visual C# 2005 Express Edition

Vous arrivez alors sur la page d'accueil de Visual C#.

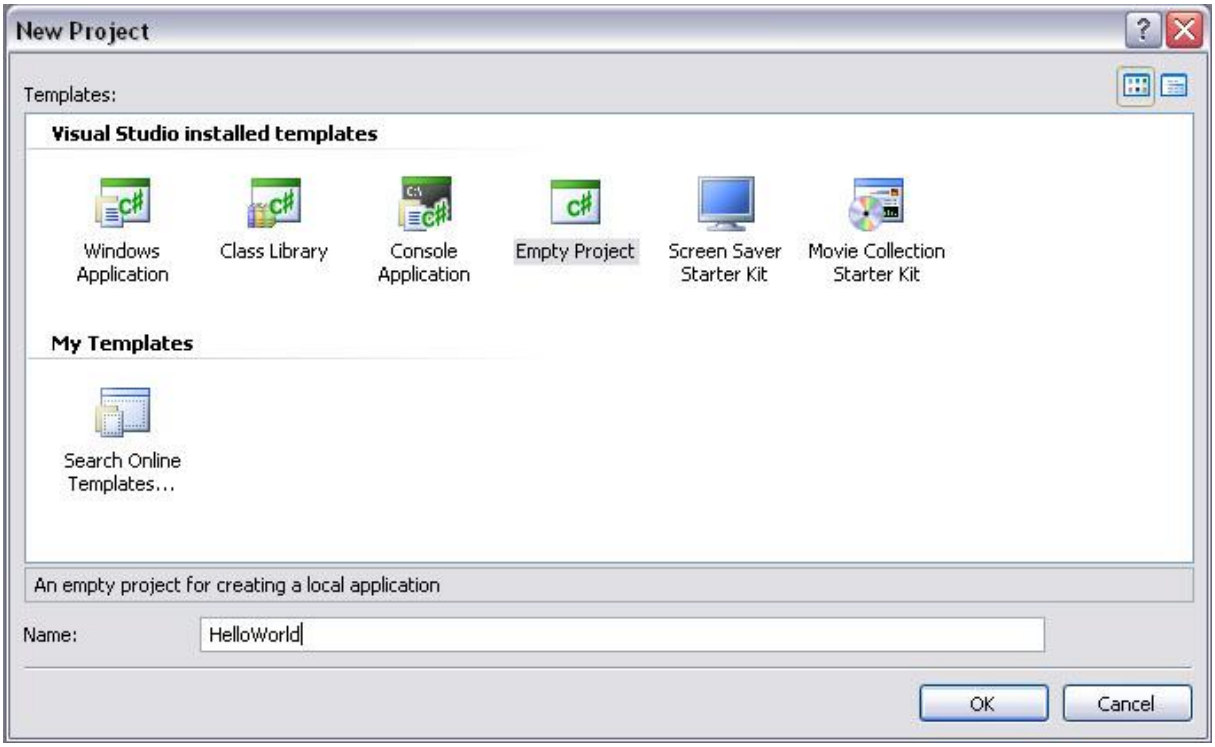
Faites File → New Project…

Vous arrivez alors sur cette fenêtre :



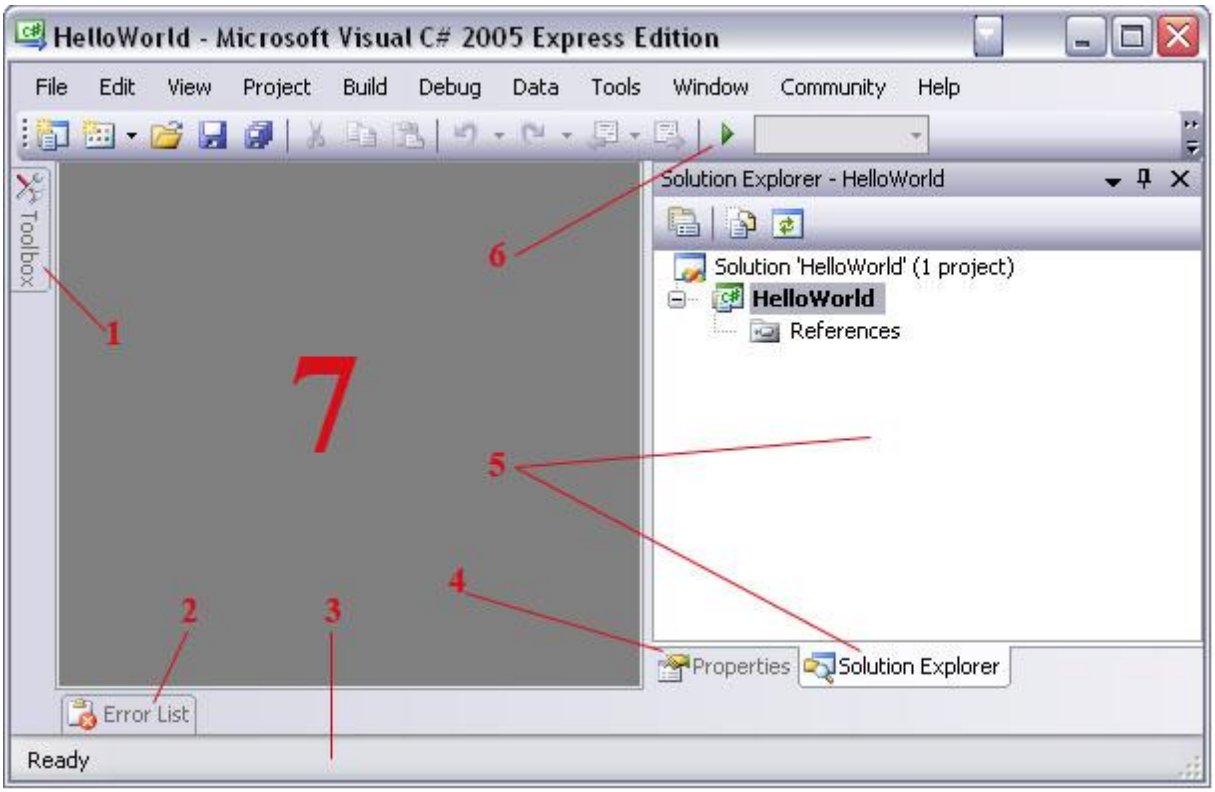

*Faites File → New Project…*

Sélectionnez Empty Project, littéralement projet vide, renommez le HelloWorld et validez.




*Sélectionnez « Empty Project », littéralement projet vide, renommez le HelloWorld et validez.*

Le projet a été créé. Voyons un peu à quoi ressemble notre EDI.



En 1, il s'agit de la boîte à outils, c'est ici que seront regroupées tous les composants pour créer très rapidement des applications Windows avec interface graphique… nous n'en feront donc pas l'usage dans cette section.

En 2 nous avons notre rapport d'erreur : il s'agit d'un outil nous permettant de savoir si notre programme ne présente pas d'erreurs et nous facilitant ainsi leurs corrections.

En 3, il s'agit tout simplement de la barre d'état, qui nous indique la plupart du temps ce que fait notre EDI.

En 4, il s'agit de l'éditeur de propriétés des fenêtres Windows… nous n'en aurons pas l'utilité ici.

En 5, il s'agit de l'un des principaux composants de Visual C# : l'explorateur de solutions littéralement. C'est ici que viendront se placer tous les fichiers, images, ou autres composants que vous ajouterez à votre solution.
La différence entre une solution et un projet est d'ordre hiérarchique : un projet ne peut contenir que des fichiers alors qu'une solution peut contenir plusieurs projets différents.

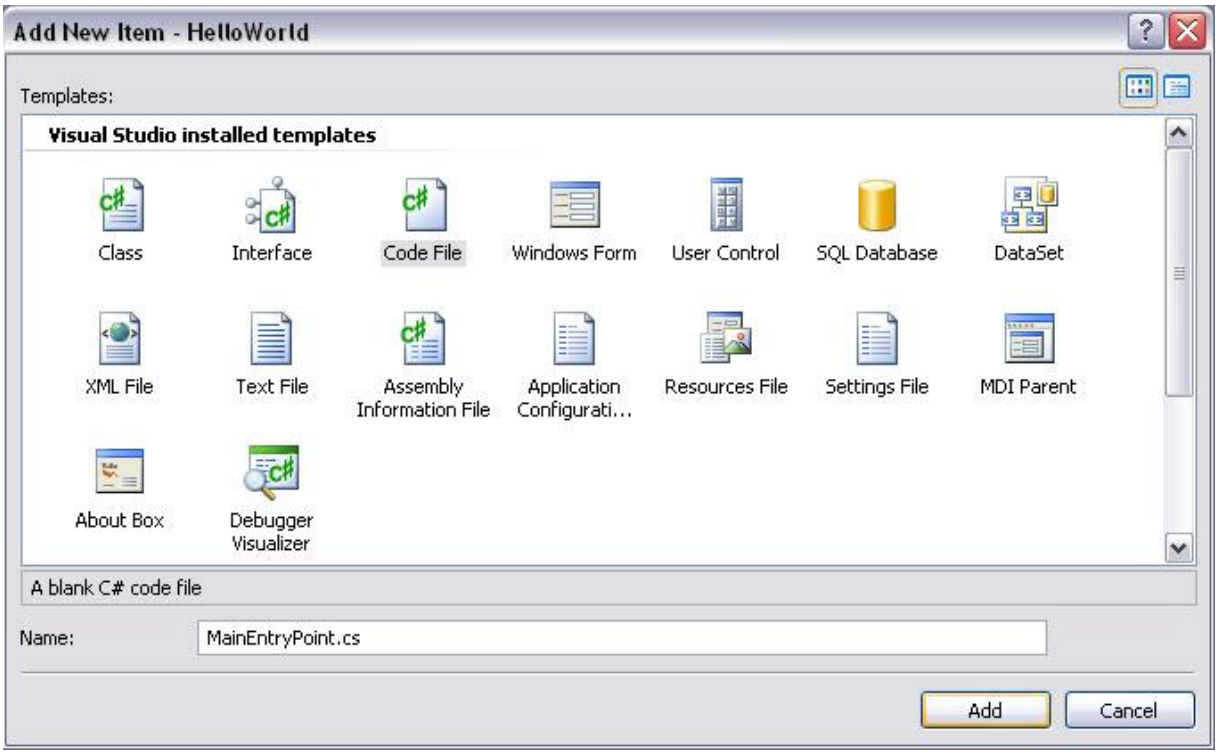
En 6e position nous avons un raccourci très pratique, nous permettant d'exécuter notre programme.

Et enfin, le n° 7… c'est ici que vous taperez votre code, ici qu'apparaîtront tous les fichiers que vous ouvrirez depuis le Solution Explorer.

*Un projet est représenté par l'icône*  *et une solution, par l'icône*  .

Revenons sur notre petit programme.
Le but de ce programme est simple : afficher « Hello World » dans la Console.
Vous devez donc créer un fichier C# pour y placer le code. Un fichier C Sharp a pour extension .cs.

Pour ajouter un fichier à votre projet, vous devez faire un clic droit sur le projet en question, ici le projet HelloWorld, et faites Add → New Item.
Là, une multitude de templates s'offrent à vous. Choisissez le template « Code File », renommez le MainEntryPoint.cs et validez en appuyant sur Add.




Le fichier MainEntryPoint.cs est alors ajouté à votre projet, dans le Solution Explorer et une page blanche s'affiche dans l'écran principal de Visual C#.

Copiez-y alors le code suivant :

```
class MainEntryPoint
{
    static void Main()
    {
        System.Console.Write("Hello World");
        System.Console.Read();
        return;
    }
}
```

Exécutez alors le programme : soit en cliquant sur le raccourci (item n° 6 dans la présentation précédente), soit en allant dans le menu Debug et en cliquant ensuite sur Start Debugging, ou encore, en appuyant sur F5.

Résultat : Une jolie fenêtre s'ouvre, c'est la Console, et affiche Hello World nous laissant gentiment le temps de lire jusqu'à ce que nous appuyions sur une touche. La fenêtre se ferme alors.

# Examen du code source[6]

La première ligne du programme déclare une classe : MainEntryPoint. Inutile d'en savoir plus pour le moment car il s'agit d'une notion orienté objet accessible à un niveau plus élevé. Nous y reviendrons plus loin dans ce tutorial.

Nous avons ensuite une fonction[7] `Main()`. C'est ici que commence véritablement notre programme.

**Tout programme en C# commence par une fonction `Main()`.**

En effet, il faut que lors de l'exécution du programme, celui-ci sache par où commencer… Il commence donc par cette fonction `Main()`.

Cette fonction `Main()` se termine par `return;`. Cela signifie qu'elle ne doit rien retourner : elle doit retourner `void` (vide en anglais).
Cette fonction doit être déclarée comme statique (`static`). Nous reviendrons plus tard sur ce mot clé.

Le reste du code est assez explicite.
`System.Console.Write("Hello World");` signifie que nous allons écrire dans la console. Nous passons à la fonction `Write` le texte correspondant au texte que nous voulons afficher, entre guillemets.
`System.Console.Read();` signifie que nous allons lire dans la console. Ou pour être plus clair, nous attendons que l'utilisateur entre quelque chose dans la console pour pouvoir lire ce qu'il y a entré. Ici, cela équivaut à attendre que l'utilisateur appuie sur la touche Entrer.
L'expression suivante étant le `return;`, notre programme s'arrête ici.

Le mot `System` qui se trouve devant le mot `Console` signifie que nous allons chercher les fonctions permettant d'accéder à la console dans `System`.
Autrement dit, si nous voulons accéder à la console, il faut que le compilateur sache que nous allons la chercher dans `System` et pas ailleurs.
Le problème, c'est que si nous voulons afficher 300 lignes dans notre console en attendant à chaque fois que l'utilisateur appuie sur une touche entre chaque nouvelle ligne, çà va nous faire un paquet de code à rajouter devant tout nos `Console.Write` et autres…

[6] Le code source représente le programme sous sa forme textuelle (en C#).
[7] Une fonction est constituée d'éléments qui s'exécutent les uns après les autres. On peut la considérer comme un groupe d'autres fonctions.

A ce problème, une solution, dire au compilateur que les fonctions que nous employons se trouvent dans `System`. Nous avons pour cela à notre disposition le mot clé `using`. Notre petit programme devient alors :

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        Console.Write("Hello World");
        Console.Read();
        return;
    }
}
```

Il s'agit du même programme mais en version allégée… Vous pouvez tester…

*Remarque : Le nom du fichier et de la classe peuvent être différents, cela n'a aucune incidence sur le programme.*

# Présentation rapide des espaces de noms

Un espace de noms peut se définir comme étant un groupement logique de fonctions (contenues dans des classes).

Exemples :

- L'espace de noms `System.Windows.Forms` contiendra toutes les fonctions nécessaires à la bonne réalisation d'une application Windows qui utilise des Form.
- L'espace de noms `System.Design.Drawing2D` permettra de dessiner des éléments simples (un rectangle, un cercle, etc.).

Pour accéder aux fonctions appartenant à un espace de noms donné, il suffit de l'inclure dans le fichier source en question par l'intermédiaire du mot clé `using` (cf. le code ci-dessus) et selon un schéma simple :

```
using <nom de l'espace de noms>;
```

`System` est un espace de noms.

# Les blocs

La partie de programme située entre deux accolades, une ouvrante et une fermante, est appelée bloc.

Je conseille de prendre l'habitude de faire une tabulation après le retour à la ligne qui suit l'accolade. Puis retirer cette tabulation après l'accolade fermante du bloc.

```
{
   Tabulation
   Tout le code est frappé à cette hauteur
}

Retrait de la tabulation
Tout le code est frappé à cette hauteur
```

Cette méthode permet de contrôler la fermeture des accolades et de leurs correspondances.

# Le squelette du programme

On peut définir le squelette d'un programme C# de la façon suivante :

```
//Ajout des espaces de noms principaux
class MaClass
{
    static void Main()
    {
        //Appels de fonctions
        Console.Read(); /*Facultatif mais permet d'attendre que
                          l'utilisateur valide (en mode console)*/
        return;         //Fin du programme, tous c'est bien déroulé
    }
}
```

# Les commentaires

Les commentaires doivent permettre à quelqu'un qui ne connait pas le C# de pouvoir lire et comprendre ce qui se passe. Les commentaires sont indispensables à tout bon programme.
Il y a deux façons de placer des commentaires : soit en utilisant `/*Commentaire*/`, soit en utilisant `//Le reste de la ligne est un commentaire` où le commentaire se termine à la fin de la ligne (cf. le squelette du programme).

# Exercices d'application

Exercice 1
Réaliser un programme qui écrit « Coucou ».

Exercice 2
Ecrire un programme qui :
- affiche « Bonjour, appuyez sur une touche SVP. »
- attend qu'on appuie sur une touche
- affiche « Merci d'avoir appuyé sur une touche. Au revoir. »

Exercice 3
Commenter le programme de l'exercice 2.

Exercice 4
Ecrire un programme qui écrit « Je fais du C# maintenant ! ».

*Astuces : Utilisez `Console.WriteLine("monTexte");` pour afficher tu texte et revenir à la ligne ensuite.*
*Vous pouvez forcer la console à attendre que vous appuyez sur une touche avant de fermer en faisant Debug → Start Without Debugging ou Ctrl+F5.*

*Conseil : Ne faites pas de copier/coller, reprenez tout à zéro pour vérifier que vous êtes bien capables de faire ces petits exercices.*

# Chapitre 2
# Les variables

## Définition

Une variable associe un nom à une valeur qui peut éventuellement varier au cours du temps. Plus précisément une variable dénote une valeur.

## Explications

Comme son nom l'indique, une variable varie.

Vous pouvez vous représenter une variable comme étant une boîte dans laquelle on met quelque chose, c'est-à-dire qu'on écrit quelque chose, ou dont on « apprend » quelque chose, c'est-à-dire qu'on lit ce quelque chose.

Pour lire une variable, on peut utiliser la fonction `Console.Write` (ou `Console.WriteLine`).
Pour donner une valeur à une variable, on utilise l'opérateur d'affection =.

Une variable ne peut contenir qu'une seule chose à la fois. Si vous mettez une seconde donnée dans une variable, la précédente est effacée.

Le nom d'une variable est appelé identificateur.

## Déclaration et affection d'une variable

Pour déclarer une variable, on utilise la syntaxe suivante :

```
<type> <identificateur>;
```

Un identificateur ne peut commencer par un chiffre et ne peut contenir de tirets - .
La syntaxe pour affecter une valeur à une variable est la suivante :

```
<identificateur> = valeur;
```

Il est également possible de déclarer et d'affecter une variable :

```
<type> <identificateur> = valeur;
```

Exemples de types :

| | |
|---|---|
| `int` | : entier |
| `char` | : caractère |
| `double` | : réel |

# Exemple

Voici un petit programme illustrant tout ce qui a été vu précédemment.

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        //Etape 1 : Déclaration de l'entier premiereLettre
        int premiereLettre;
        //Affection de l'entier premiereLettre à la valeur 67
        premiereLettre = 67;

        //Etape 2 : Déclaration du caractère secondeLettre
        char secondeLettre;
        //Affectation du caractère secondeLettre à la valeur de O
        secondeLettre = 'O';

        /*Etape 3 : Déclaration et affection du caractère dernieреLettre
à la valeur de Q*/
        char dernieреLettre = 'Q';

        //Etape 4 : Affiche la valeur contenue dans la variable
premiereLettre
        Console.Write("{0}", premiereLettre);

        //Affiche la valeur contenue dans secondeLettre
        Console.Write("{0}", secondeLettre);

        /*Affiche la valeur contenue dans dernieреLettre et va à la
ligne*/
        Console.WriteLine("{0}", dernieреLettre);

        //Etape 5 : Affiche Au revoir et va à la ligne
        Console.Write("Au revoir\n");

        //Fin du programme
        return;
    }
}
```

Lancez ce petit programme grâce à l'astuce signalée dans le chapitre précédant et essayez de comprendre ce qui se passe.

# Examen du code

A l'étape 1, on met `67` dans la variable `premiereLettre`.
A l'étape 2, on met la valeur de `O` dans la variable `secondeLettre`.
A l'étape 3, on met la valeur de `Q` dans la variable `troisiemeLettre`.

*Remarque : En informatique, tout n'est que nombre, c'est pour çà que je dis la valeur de O et non O. En effet, on stocke dans la variable la valeur Unicode correspondante à la lettre O. Nous reviendrons sur ce point plus tard.*

A l'étape 4, on affiche `premiereLettre` : `Console.Write("{0}", premiereLettre);` va remplacer le `{0}` par la valeur contenue dans `premiereLettre`.

De même pour les autres valeurs.
A l'étape 5, on affiche Au revoir et on va à la ligne. Le retour à la ligne est garanti par un caractère spécial : \n. Ainsi, `Console.Write("Au revoir\n");` est identique à `Console.WriteLine("Au revoir");`.

Le « {*rang*} » signifie que le programme doit remplacer ce « {*rang*} » par la variable n° *rang* qui se trouve dans la suite des instructions.

Exemple :

```
Console.WriteLine("{0}{1}{2}", var1, var2, var3); /*{0} sera remplacé par la valeur de var1, {1} par celle de var2 et {2} par celle de var3*/
```

revient au même que

```
Console.Write("{0}", var1); //{0} sera remplacé par la valeur de var1
Console.Write("{0}", var2); //{0} sera remplacé par la valeur de var2
Console.WriteLine("{0}", var3); //idem mais avec var3
```

*Remarque : L'affectation d'un caractère se fait toujours grâce à l'opérateur d'affectation = mais il faut placer le caractère entre guillemets simples, sinon le compilateur indiquera une erreur.*

# Exercice d'application

Exercice 5

En utilisant ce qui a été fait précédemment, afficher 62 369 12 e 15 c.

# Réutilisation d'une variable

Dans l'explication de ce qu'est une variable, il est dit : « Une variable ne peut contenir qu'une seule chose à la fois. Si vous mettez une seconde donnée dans une variable, la précédente est effacée. ».
Par conséquent

```
i = 5;
i = 19;
i = 17;
```

i contient la valeur 17 à la fin.

# Caractères spéciaux

Nous avons vu que pour faire un retour chariot, c'est-à-dire un retour à la ligne, nous pouvions utiliser soit `Console.WriteLine`, mais le retour chariot n'a lieu qu'en fin de ligne obligatoirement, soit `\n` que nous placions dans notre texte là où nous voulions faire un retour à la ligne.

`\n` est un caractère spécial.

Les caractères spéciaux commencent tous par le caractère de suffixe \ . Cela permet au compilateur de faire la différence par exemple entre `\n` et `n`.

Il existe environ une dizaine de caractères spéciaux dont voici les principaux :

| | |
|---|---|
| `\n` | fait un retour chariot |
| `\t` | fait une tabulation |
| `\'` | permet de créer un caractère contenant la valeur du *quote*. |

\"  affiche un guillemet.

Exemples d'utilisations :

```
Console.WriteLine("Je cite : \"Ceci est une citation\"");
```

affichera : Je cite : "Ceci est une citation".

```
char c = '\'';
```

stocke la valeur de ‘ (se dit *quote* en informatique, et non pas apostrophe qui est du français) dans la variable c de type caractère.

# Exercices d'application

Exercice 6

En utilisant les variables, écrire un programme qui affiche :
Bonjour
666

Exercice 7

De la même manière, écrire un programme qui affiche :
B
o
n
BonBon
4
2
1
421

*Remarque : Si vous n'avez qu'une seule variable à passer à afficher dans la Console et rien d'autre, vous pouvez écrire directement : `Console.Write(maVariable);` et ceux, quelque soit le type de la variable : que ce soit un entier (`int`), une chaîne de caractères (`string`), un caractère (`char`)…*

# Les chaînes de caractères

Les chaînes de caractères sont tout simplement des caractères qui se suivent les uns les autres.

Exemple : « Coucou, c'est moi ! » est une chaîne de caractères.

Côté mémoire, on pourrait représenter une chaîne de caractère comme étant un tableau de caractères (la notion de tableau sera abordée plus loin) :

| C | o | u | c | o | u | , |  | c | \' | e | s | t |  | m | o | i |  | ! | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

*Remarque : Le caractère \0 est un caractère spécial marquant la fin d'une chaîne de caractères.*

Créer une chaîne de caractères est très simple en C#. On utilise pour cela le type `string`.
Pour donner une valeur (ici du texte) à notre chaîne de caractères, on place le texte entre des guillemets.

Exemple :

```
string chaîne = "Coucou, c'est moi !";
```

Voici un programme simple illustrant l'utilisation d'une `string` :

```
using System;

class MainEntryPoint
{
```

```
    static void Main()
    {
        //Déclare une chaîne de caractères contenant Coucou, c'est moi !
        string maChaine = "Coucou, c'est moi !";
        //Affiche : Coucou, c'est moi !
        Console.WriteLine(maChaine);
        Console.Read();
        return;
    }
}
```

Il existe plusieurs fonctions permettant de convertir un type `string` en un autre type (`int` par exemple).

Retenez en une pour l'instant :
`Convert.ToInt32(maChaine);` fonction qui retourne une variable de type `int`.

Exemple d'utilisation :

```
int a = Convert.ToInt32("12");
```

ou encore

```
string maChaine = "12";
int a = Convert.ToInt32(maChaine);
```

**Si la chaîne de caractère ne contient pas une valeur susceptible d'être convertie en entier, cette fonction retourne une erreur et le programme s'arrête.**

Exemple d'un programme erroné :

```
int a = Convert.ToInt32("Coucou");
```

Ici, il est impossible de transformer Coucou en un entier, le programme s'interrompt en vous rapportant qu'une erreur s'est produite.

# Exercices

Exercice 8

Ecrire un programme utilisant le type `string` pour afficher une chaîne de caractères : « Bonjour, comment allez-vous ! »

Exercice 9

De la même manière, écrire un programme qui :

- stocke dans une variable « Salut ! »
- affiche cette chaîne de caractères
- la remplace par une valeur de type entière (exemple : 42 où 42 est une chaîne de caractères)
- affiche la chaîne de caractères suivie du message : « est stocké dans une variable de type string ».
- convertie la chaîne de caractères en un entier et stocke sa valeur dans une nouvelle variable
- affiche cette valeur suivie du message : « est stocké dans une variable de type int ».

# Saisie de variables

Nous avons vu précédemment que lorsque nous voulions marquer une pause jusqu'à ce que l'utilisateur appuie sur la touche Entrer, nous devions utiliser la fonction `Console.Read();`. Cette fonction nous

permet de saisir des données et de les placer dans des variables. Elle nous renvoie effectivement une variable de type entier.

Exemple :

```
int a = Console.Read();
```

Le problème c'est que cette fonction retourne absolument tout ce que l'utilisateur entre, y compris la valeur du retour chariot lorsque celui-ci valide.

Il ne s'agit donc pas d'une très bonne solution pour saisir des variables via la console.

Alors comment faire ?

Eh bien, vous avez peut-être remarqué que lorsque vous saisissiez `Console.Read`, votre EDI vous proposait automatiquement d'autres fonctions, et notamment, `Console.ReadLine`.
La différence entre `Console.Read` et `Console.ReadLine` est subtile mais très simple : `Console.Read` renvoie un entier alors que `Console.ReadLine` renvoie une `string`. Côté utilisateur, on ne voit pas la différence : lorsqu'on appuie sur la touche Entrer, on termine en même temps la saisie de notre variable.

Exemple d'utilisation de `Console.ReadLine` :

```
string myString = Console.ReadLine();
```

Application :

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        //Déclare une chaîne de caractères
        string myString;
        Console.Write("Entrez quelque chose : ");
        /*Affecte la valeur entrée par l'utilisateur à la chaîne de
caractères myString*/
        myString = Console.ReadLine();
        //Affiche ce que l'utilisateur a tapé avec un petit message
        Console.WriteLine("Vous avez entré : {0}", myString);
        Console.Read();
        return;
    }
}
```

# Opérations sur les nombres

Les opérateurs de bases en programmation sont les mêmes que les opérateurs mathématiques que vous employez depuis votre plus tendre enfance, à savoir :

- l'opérateur + pour l'addition : additionne 2 entiers entre eux
- l'opérateur – pour la soustraction : soustrait 2 entiers entre eux
- l'opérateur * pour la multiplication : multiplie 2 entier entre eux
- l'opérateur / pour la division : divise le 1$^{er}$ entier par le 2$^{nd}$ et tronque le résultat
- l'opérateur % qui permet de connaître le reste d'une division euclidienne (que vous n'avez probablement jamais vu)

Exemple d'utilisation :

```
int a = 2;
int b = 3;
int c = a + b;  //c = 5
int d = c + 12; //d = 17
int e = a*b;   //e = 3*2 = 6
int f = c%a;   //5%2 = 1 car 5 = 2*2 + 1
```

De même avec les autres opérateurs.

Les priorités établies en mathématiques sont respectées.
C'est-à-dire que : 3 + 5*2 ⇔ 3 + (5*2) ≠ (3+5)*2

Il existe des raccourcis sur les opérateurs permettant d'alléger un peu le code et d'en faciliter la lecture.

Il est ainsi possible d'affecter une variable et d'effectuer une opération en même temps (d'un point de vue uniquement visuel).
Ces opérateurs sont +=, -=, *=, /=, %=, ++ et --.

Exemples :

```
int i  = 0;    //La variable doit avoir été initialisé au préalable.
i += 12;       //⇔ i = i + 12;
i -= 12;       //⇔ i = i - 12;
i *= 5;        //⇔ i = i * 5;
i /= 5;        //⇔ i = i / 5;
i %= 2;        //⇔ i = i % 2;
i++;           //⇔ i += 1; ⇔ i = i + 1;
i--;           //⇔ i -= 1; ⇔ i = i - 1;
```

*Remarque : L'opérateur ++ est appelé opérateur d'incrémentation et l'opérateur --, opérateur de décrémentation.*

# Concaténation

La concaténation est une opération visant à regrouper plusieurs chaînes de caractères entre elles.

Elle s'effectue par l'intermédiaire de l'opérateur + et ne peut avoir lieu qu'entre types `string`.

Les autres opérateurs ne s'emploient pas sur les types `string`.

On peut se représenter une concaténation comme étant une opération mettant bout à bout les chaînes de caractères (ou `string`) entre elles.

Exemple :

```
string a = "Coucou,";
string b = " c'est moi !";
string c = a + b; //c contient "Coucou, c'est moi !"
```

*Remarque : Il est possible de concaténer autant de chaînes de caractères entre elles qu'on le veut.*

Ainsi,

```
Console.WriteLine("Vous avez entré : {0}", myString);
```

peut également s'écrire

```
Console.WriteLine("Vous avez entré : " + myString);
```

dans la précédente application.

# Exercices

Exercice 10

Réaliser un programme qui :

- demande à l'utilisateur d'entrer un morceau de texte
- affiche le morceau de texte avec un petit message de remerciement
- attend que l'utilisateur appuie sur une touche avant de quitter

Exercice 11

Réaliser un programme qui :

- demande à l'utilisateur d'entrer un entier
- stocke cet entier dans un type int
- affiche la valeur contenue dans la variable de type int

Exercice 12

Réaliser une petite calculatrice très simple qui :

- demande à l'utilisateur d'entrer une valeur a
- demande à l'utilisateur d'entrer une valeur b
- stocke ces valeurs dans des variables de types int
- effectue la somme de ces 2 valeurs
- affiche l'addition complète avec son résultat
- attend que l'utilisateur appuie sur une touche avant de quitter

Exercice 13

Reprendre le programme de l'exercice 12 en changeant l'opération pour la remplacer par chacune des 4 autres. C'est-à-dire que ce programme fera :

- une addition et affichera l'opération complète
- une soustraction et affichera l'opération complète
- une multiplication et affichera l'opération complète
- une division et affichera l'opération complète
- une division avec reste et affichera l'opération complète en précisant le reste.

# Chapitre 3
# Les conditions

## If et else

Ces conditions ont pour syntaxe :

```
if (condition vraie)
{
   instructions 1
}
else
{
   instructions 2
}
```

ce qui signifie :

```
si (la condition est vraie, est vérifiée)
{
   alors, faire instructions 1
}
sinon
{
   faire instructions 2
}
```

## Les opérateurs logiques

Ils servent à comparer deux nombres entre eux.

| Libellé | Opérateur |
|---|---|
| strictement inférieur | < |
| strictement supérieur | > |
| égal | == |
| différent | != |
| inférieur ou égal | <= |
| supérieur ou égal | >= |

**Ne pas confondre = et ==**
**= est l'opérateur d'affection et permet donc d'attribuer une valeur à une variable alors que == est l'opérateur d'égalité, il s'agit d'une condition vérifiant que deux variables ont la même valeur ou non.**

# Opérateurs logiques purs

Ce sont des opérateurs logiques permettant de combiner des expressions logiques.

| Libellé | Opérateur |
|---|---|
| And (et) | `&&` |
| Or (ou) | `\|\|` |
| Not (non) | ! |

`|` se nomme pipe en informatique (se prononce paillp).

# Vrai ou faux

Une condition retourne soit `true` (vrai) si elle est vérifiée, soit `false` (faux) si elle ne l'est pas.

`true` et `false` sont des valeurs de type booléen.

Le type booléen ne peut contenir que ces deux valeurs : `true` ou `false`.

Il s'emploie comme n'importe quel autre type excepté qu'il est impossible d'effectuer des opérations sur des booléens.

Le type booléen ne sert qu'à stocker une valeur : soit `true`, soit `false`.

Le mot clé permettant de déclarer une variable comme étant de type booléen est : `bool`.

`true` peut être assimilé à la valeur 1 ou à toute autre valeur non nulle.
`false` peut être assimilé à la valeur 0.

alors, l'opérateur logique Or (`||`) correspond à une addition :

| `\|\|` | `true` | `false` |
|---|---|---|
| `true` | `true` | `true` |
| `false` | `true` | `false` |

⇔

| + | 1 | 0 |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 1 | 0 |

et l'opérateur logique And (`&&`) correspond à une multiplication :

| `&&` | `true` | `false` |
|---|---|---|
| `true` | `true` | `false` |
| `false` | `false` | `false` |

⇔

| * | 1 | 0 |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 0 |

*Remarques :* *`!true = false` et `!false = true`*
*Toutes ces propriétés sont établies dans l'algèbre de Bool.*

# Combinaison

Toutes les opérations logiques peuvent se combiner entre elles.

Exemple : `if (a == 5 && b == 2)`

Vous pouvez placer chaque condition entre parenthèses…

Exemple : `if ((a == 5) && (b == 2))`

… et jouer sur les conditions à analyser en priorité…

Exemple :
`if ((a == 5) && (b == 2 || c == 3))`
est différent de
`if ((a == 5 && b == 2) || c == 3)`

En effet, pour que la première condition renvoie `true`, il faut que a vaille 5 et il suffit ensuite que b soit égal à 2 ou que c soit égal à 3. Dans ce cas, la condition renvoie `true`, sinon `false`.
Alors que la seconde condition vérifie si a vaut 5 et b vaut 2, si c'est le cas elle renvoie `true`, sinon elle vérifie que c vaut 3, si oui, alors la condition renvoie `true`, sinon `false`.

Résultats : En posant a ≠ 5, b ≠ 2 et c = 3, alors :
- la première condition renvoie `false`
- la seconde condition renvoie `true`

# Un exemple

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        //Demande une valeur
        Console.Write("Entrez une valeur : ");
        string input = Console.ReadLine();
        //Transforme la chaîne de caractères entrée en un entier
        int valeur = Convert.ToInt32(input);
        //Condition : si la valeur entrée est strictment supérieure à 10
        if (valeur > 10)
        {
            //alors afficher...
            Console.WriteLine("Vous avez entrée une valeur strictement
supérieure à 10.");
        }
        else
        {
            //sinon afficher
            Console.WriteLine("Vous avez entrée une valeur inférieure ou
égale à 10.");
        }
        //Histoire d'avoir le temps de lire la réponse ;)...
        Console.Read();
    }
}
```

*Remarque : Jusqu'à présent, tous les exemples avaient un `return` à la fin de la fonction Main. Celui ne retournant aucune valeur, il est possible de l'omettre sans qu'aucune erreur ou problème n'apparaisse.*

# Accolades

Les accolades entourant les blocs d'instructions d'une condition peuvent être omises si le bloc n'est constitué que d'une seule instruction.

Exemple :

```
if (a == 2)
   Console.Write ("a vaut 2");
else
   Console.Write ("a ne vaut pas 2");
```

# Else if

```
if (condition 1)
{
   instructions 1
}
else
{
   if (condition 2)
   {
         instructions 2;
   }
   else
   {
         instructions 3;
   }
}
```

peut s'écrire avec `else if` comme étant :

```
if (condition 1)
{
   instructions 1
}
else if (condition 2)
{
   instructions 2;
}
else
{
   instructions 3;
}
```

*Remarque : Vous pouvez imbriquer autant de `if`, de `else if`, de `else` que vous le souhaitez, il n'y a pas de limite.*

# Exercices

Exercice 14

Réaliser un programme qui :

- demande à l'utilisateur d'entrer une valeur
- affiche si cette valeur est positive ou négative ou égale à 0

Exercice 15

Réaliser un programme qui :

- demande à l'utilisateur d'entrer une valeur
- affiche si cette valeur est comprise entre 0 et 10 ou non.

Exercice 16

Réaliser un programme qui :

- demande à l'utilisateur d'entrer un caractère

- affiche si celui-ci est une voyelle ou une consonne

*Aide : Pour transformer une `string` en `char`, utilisez la fonction `Convert.ToChar(myString);` qui retourne un `char`. Pour comparer un caractère à un autre, rappelez-vous comment vous faisiez pour affecter une valeur à un type `char`.*

*Astuce : Il est plus rapide de vérifier si la lettre en question est une voyelle (6 cas possibles seulement) plutôt que de vérifier s'il s'agit d'une consonne (20 cas possibles).*

# Switch / case

Syntaxe :

```
switch (/*variable*/)
{
   case /*valeur 1*/:
         //instructions 1
         break;
   case /*valeur 2*/:
         //instructions 2
         break;
   case /*valeur 3*/:
         //instructions 3
         break;
   //etc.
   default:
         //instructions par défaut
         break;
}
```

revient au même que

```
if (/*variable*/ == /*valeur 1*/)
{
   //instructions 1
}
else if (/*variable*/ == /*valeur 2*/)
{
   //instructions 3
}
else if (/*variable*/ == /*valeur 3*/)
{
   //instructions 3
}
//etc.
else
{
   //instructions par défaut
}
```

Sémantique[8] :

On passe une variable à `switch`.
La valeur de cette variable sera comparée à chaque valeur se trouvant après chaque `case`. Dès que la valeur correspond, l'instruction se trouvant dans le `case` est lancée… le `break` signifie que l'on peut désormais sortir de l'instruction conditionnelle (c'est-à-dire, du `switch`).Tant qu'aucun break n'est rencontré, les

[8] Signification et objectif du code

conditions suivantes sont testées et si elles sont vérifiées, leurs instructions correspondantes sont également exécutées… jusqu'à ce que l'on rencontre un `break`.

Exemples :

```
switch (a)  //a est un entier d'une valeur quelconque
{
   case 1:
         Console.WriteLine("a vaut un");
         break;
   case 16:
         Console.WriteLine("a vaut seize");
         break;
   case 36:
         Console.WriteLine("a vaut trente six");
         break;
   default:
         Console.WriteLine("a ne vaut ni un, ni seize, ni trente six");
         break;
}
```

ou encore

```
switch (country)  //country est de type string
{
   case "uk":
   case "usa":
         Console.WriteLine("Your language is english"); /*si usa ou si
uk*/
         break;
   case "fr":
         Console.WriteLine("Votre langue est le français"); /*si fr
uniquement*/
         break;
   default:
         Console.WriteLine("Your language is neither english nor
=  french");
         break;
}
```

Petit exercice d'application directe :

Exercice 17

Réécrire l'exercice 16 avec un `switch` / `case`.

# L'opérateur ternaire

Syntaxe :

*<condition> ? <si vrai faire…> : <sinon faire…>;*

Exemple :

```
int a = 8;
int b = (a == 8) ? 2 : 3;
```

Sémantique de l'exemple:

On déclare une variable a et on l'initialise à une valeur de 8.

A la ligne suivante, on déclare b et on l'initialise à la valeur 2 si a vaut 8 sinon (si a vaut toute autre valeur que 8), à la valeur 3.

*Remarque :* *Cet exemple peut s'écrire avec un* `if` *et un* `else` *de la manière suivante :*

```
int a = 8;
int b = 0; /*Il est conseillé de toujours initialiser une variable lors
de sa déclaration*/
if (a == 8)
   b = 2;
else
   b = 3;
```

# Chapitre 4
# Les boucles

Une boucle est constituée d'un bloc de code contenant des instructions. L'intérêt d'utiliser une boucle est simple : il s'agit d'effectuer plusieurs fois la même opération sans pour autant avoir à la réécrire à chaque nouvelle fois que nous en avons besoin. Il s'agit d'une tâche répétitive.

Par exemple : Nous voulons implémenter un programme qui nous calcule un nombre exposant un autre nombre ($\alpha = \gamma^{\beta}$). Alors comment faire ? On ne va pas créer toutes les conditions (si $\beta = 0$, si $\beta = 1$, si $\beta = 2$, si $\beta = 3$…) et pour leur dire à chaque fois de faire la même chose (en l'occurrence multiplier $\gamma$ par lui-même une fois de plus à chaque fois qu'on passe à la condition suivante… le code serait interminable ou limité à quelques cas… Non, on ne va pas faire comme çà. On va utiliser une boucle qui va répéter ($\beta$-1) fois l'opération de multiplier $\gamma$ par lui-même.

## Faire… tant que

Syntaxe :

```
do
{
   //instructions
} while (condition);
```

Sémantique :

Le programme rentre dans la boucle pour la première fois lorsqu'il rencontre le mot clé `do`. Il exécute alors toutes les instructions se trouvant dans le bloc de code de la boucle. Lorsqu'il rencontre le mot clé `while`, il vérifie si la condition qu'il contient est vraie ou fausse. Si celle-ci est vraie, il revient au début de la boucle et réexécute toutes les instructions. En revanche, si elle est fausse, il sort de la boucle et continue d'exécuter le code se trouvant après la boucle.

Exemple :

```
int a = 0;           //On crée une variable a et on l'initialise à 0
do
{
   a++;              //On incrémente a (a = a + 1)…
} while (a != 10);   //… tant que a est différent de 10, on continue
```

Contre-exemple : une boucle interminable

```
int a = 0;           //On crée une variable a et on l'initialise à 0
do
{
   a += 2;           //On ajoute 2 à a
} while (a != 11);   //… cette condition ne sera jamais vraie.
```

**Faites attention à votre condition de fin de boucle, celle-ci doit être validée pour quitter la boucle.**

# Tant que…

Cette boucle est quasiment la même que la précédente avec une petite différence qui a son importance.

Syntaxe :

```
while (condition)
{
   //instructions
}
```

A première vue, c'est pareil. Il n'y a que la syntaxe qui change… eh ben non ! La principale différence fait toute la différence : c'est le fait que la condition se trouve au début de la boucle et non plus à la fin. Que se passe-t-il alors ?
Le programme arrive au début de notre boucle et rencontre notre condition. Il vérifie si celle-ci est vraie ou fausse. Si elle est fausse, les instructions sont exécutées. A la fin des instructions, on revient au début de la boucle : on vérifie sa condition et si… Si la condition est vraie, la boucle se termine en sautant les instructions et le programme continue d'exécuter le reste du code.

La principale différence entre un `do…while` et un simple `while`, c'est que le `do…while` oblige les instructions à être exécutées au moins une fois alors que le `while` (tout seul) peut très bien l'empêcher dès le départ.

Exemple :

```
//L'utilisateur entre une valeur entière a
while (a > 10)        //… tant que a est différent de 10, on…
{
   a--;               //…décrémente a
}
```

Dans cet exemple, la boucle ne s'exécutera que si l'utilisateur entre une valeur strictement supérieure à 10. Dans un autre cas (10, -2, 7…), les instructions contenues dans la boucle de s'exécuteront jamais.

En revanche, dans l'exemple qui suit, les instructions sont exécutées une fois minimum, la condition n'étant vérifiée qu'après.

```
//L'utilisateur entre une valeur entière a
do
{
   a--;               //…décrémente a
} while (a > 10) ;    //… tant que a est différent de 10, on…
```

# La boucle for

Celle-ci permet d'exécuter un nombre fini de fois les instructions se trouvant en son sein.

Syntaxe :

```
for (i = point de départ; i < point d'arrivé; i+pas)
{
   //instructions
}
```

i est une variable. On lui passe une valeur en entrant dans la boucle. Cette valeur est sa valeur de départ. Les instructions sont ensuite exécutées. On revient au début de la boucle. La condition est testée. Si vraie, la boucle est terminée et les instructions qu'elle contient ne sont pas réexécutées. Si fausse, on ajoute à i la valeur `pas` et on exécute à nouveau les instructions…

Exemple :

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        Console.WriteLine("Un message va vous être affiché 10 fois !");
        Console.Read();

        /*Voici notre boucle. On crée un variable entière i qu'on
initialise à 0 afin de traiter 10 fois la boucle comme annoncé
précedemment. Cette boucle continuera tant que i sera inférieur 10.*/
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Fois n° " + i.ToString());
            Console.WriteLine("Bonjour !");
        }

        Console.Read();
    }
}
```

Regardez ce qui se passe lors de l'exécution.

*Remarque : Les variables déclarées dans les boucles for, ou pour les boucles for, sont généralement appelées i, j, k, l… Cette pratique remonte aux années 60 où les chercheurs avaient alors décidés d'utiliser ces noms de variables en tant que compteurs de boucles.*

*Astuce : Vous pouvez omettre, comme pour le `if`, les accolades s'il n'y a qu'une seule une instruction.*

```
for (int i = 0; i < 10; i++)
   Console.WriteLine("Bonjour !");
```

# Des boucles dans des boucles

Vous avez parfaitement le droit d'imbriquer des boucles dans d'autres boucles, c'est sans limite.

Exemples :

```
while (a > 1)
{
   for (int i = 0; i < 10; i++)
         b++;
}
```

ou encore

```
for (int i = 0; i < 10; i++)
   for (int j = 0; j < 15; j++)
         b++;
```

Etc.

# Exercices

Exercice 18

Ecrivez de 3 façons différentes le programme présenté dans l'introduction de ce chapitre. C'est-à-dire, écrivez un programme qui :

- demande à l'utilisateur d'entrer une valeur $\gamma$ et une valeur $\beta$
- affiche la valeur $\alpha$

Pour calculer $\alpha$, vous vous servirez de ce qui a été vu précédemment. C'est-à-dire que vous calculerez $\alpha$ d'abord avec une boucle `do...while`, puis avec une boucle `while`, puis avec une boucle `for` sans oublier d'anticiper le cas extrême où $\beta = 0$ ($\beta$ est un entier naturel).

Exercice 19

Ecrire un programme qui :

- demande à l'utilisateur d'entrer un entier : soit 1, soit 2, soit 3, soit 4
- si l'utilisateur appuie sur 1, affiche "Vous avez entré 1." puis reviens au début.
- si l'utilisateur appuie sur 2, affiche "Vous avez entré 2." puis reviens au début.
- si l'utilisateur appuie sur 3, affiche "Vous avez entré 3." puis reviens au début.
- si l'utilisateur appuie sur 4, affiche "Le programme va se fermer." et attend que l'utilisateur appuie sur une touche avant de quitter.

*Astuce : Pour l'exercice 18, vous serez très vite limité par le choix de vos nombres et de vos exposants si vous utilisez des entiers, car ils ont une limite, un nombre maximal à ne pas dépasser (qui se fixe par $2^{31}$ pour les `int` et par $2^{32}$ pour les `uint`). Je vous conseille d'utiliser des `double` dans l'exercice 18 pour stocker $\alpha$, pour $\beta$ et $\gamma$ vous utilisez le type `uint` car il ne peut contenir que des entiers positifs (entiers naturels), ce que vous nous voulons (les négatifs ne seront pas tolérés ici). Vous vous servirez donc de la fonction `UInt32.TryParse` pour les conversions vers le type `uint` de la même manière que vous vous serviez de `Int32.TryParse` (confère correction des exercices précédents).*

*Aide : Pour l'exercice 19, songez à ce que signifie "si 1, si 2, si 3, revient au début". Vous pouvez également le voir par la négation du 4 ou en utilisant un booléen pour dire si le programme doit se terminer ou pas… et vous aurez la solution. Pensez à prendre en compte le fait que l'utilisateur peut entrer autre chose que ce qui lui est demandé.*

# Chapitre 5
# Les classes

## Ce que vous savez sur les classes

A vrai dire, rien… et pourtant, vous vous en servez depuis le début de ce tutorial, sans en connaître le fonctionnement ni la place centrale qu'elles occupent en C#.

Revoyons un peu notre tout premier programme :

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        Console.Write("Hello World");
        Console.Read();
        return;
    }
}
```

Que faisons-nous ici ?

A la ligne 3, on déclare une classe dont le nom est MainEntryPoint. On ouvre ensuite les accolades de notre classe puis on les ferme tout à la fin. On crée ensuite la fonction Main, fonction principale et obligatoire à tout programme C#. C'est donc dans la classe, entre ses accolades qu'on peut déclarer des fonctions.

A partir de ces éléments, on peut déjà établir un schéma présentant la structure d'une classe.

```
class <nom de la classe>
{
    //fonctions
}
```

## Déclaration d'une classe

Syntaxe :

La syntaxe d'une classe est ni plus ni moins celle décrite au dessus, c'est-à-dire :

```
class <nom de la classe>
{
    //fonctions
}
```

Vous pouvez déclarer autant de classes que vous voulez dans votre fichier source. Par exemple :

```
class <nom de la classe 1>
{
    //fonctions
}

class <nom de la classe 2>
{
    //fonctions
}

class …
```

# Les méthodes

« Super je sais déclarer une classe maintenant sauf que çà sert à rien puisqu'il n'y a rien dedans. En plus je ne sais même pas à quoi cela pourrait servir. »

C'est vrai que pour l'instant çà ne sert à rien. Tout ce que vous avez appris à mettre dans une classe, c'est une fonction Main. Mais il ne peut y avoir qu'une seule fonction Main par programme, donc vous êtes déjà bloqué.

Pas de panique, j'arrive… on va d'abord remplir un peu notre classe avec des méthodes. Une méthode est une fonction, c'est pareil, mais on va voir par la suite, que, parmi les fonctions qu'une classe peut implémenter, il y en a qui n'ont pas la même syntaxe que les méthodes et qui vont donc porter un nom différent.

Syntaxe :

```
<modificateur d'accès> <type de retour> <nom de la méthode>(<arguments>)
{
   //instructions
}
```

Les classes peuvent agir entre elles. Elles peuvent appeler les fonctions d'autres classes. Pour éviter qu'elles n'appellent tout et n'importe quoi, il faut leur donner des autorisations d'accéder directement à telle ou telle fonction. C'est le rôle du modificateur d'accès de permettre ou de refuser à une classe étrangère d'accéder à certaines fonctions, ou au contraire de l'autoriser à le faire.
Parmi les quelques modificateurs d'accès, nous allons en retenir deux pour l'instant : `public` et `private`. `public` autorise une autre classe à accéder à la fonction alors que `private` le lui interdit.

Le type de retour est un type, comme ceux que nous avons vu auparavant, qui va déterminer quelle valeur la méthode devra renvoyer lorsqu'elle se terminera. Si le type de retour est déclaré comme étant un entier (`int`), la méthode devra obligatoirement renvoyer un entier. Pour renvoyer une valeur, ou rien du tout, on utilise le mot clé `return`, suivi de la valeur qu'il doit retourner. Le mot clé `return` indique, quoi qu'il en soit, que la fonction est sur le point de se terminer.

Le nom de la méthode est simplement le nom que vous souhaitez donner à votre méthode. Généralement celui-ci indique ce que celle-ci va effectuer.

Les arguments sont des variables que l'ont passe à la méthode lors de son appel dans le programme. Il peut y en avoir aucun, un seul ou plusieurs. Il n'y a pas de limites. Les limites seront celles que vous fixerez au moment de la saisie de votre méthode (« pfou ! y'en a marre, 42$^{e}$ argument, j'arrête… »)…
Une fonction commence par une accolade ouvrante et se termine par une accolade fermante.

Si ce cours n'était que théorique, il vous resterait encore probablement toute une page d'explications sur comment utiliser, lancer une fonction. Comment créer une classe et l'appeler, etc., mais nous allons voir tout cela tout de suite à l'aide d'un exemple.

Exemple :

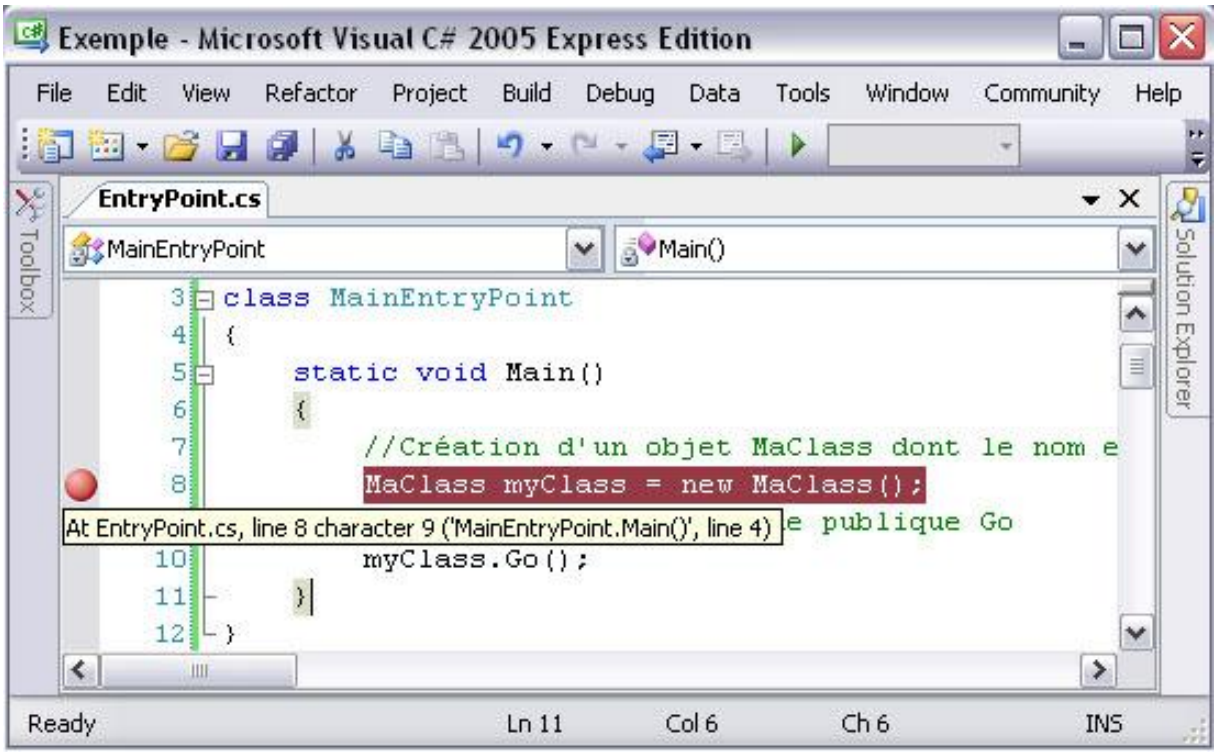
```
using System;

class MainEntryPoint
{
    static void Main()
    {
        //Création d'un objet MaClass dont le nom est myClass
        MaClass myClass = new MaClass();
        //On fonce sur la méthode publique Go
        myClass.Go();
    }
}

class MaClass
{
    /*La fonction principale de la classe MaClass, c'est elle qui fait
tout !*/
    public void Go()
    {
        Console.WriteLine("Programme somme : ");
        string val1, val2;
        Console.Write("Saisissez la 1ere valeur : ");
        val1 = Console.ReadLine();
        Console.Write("Saisissez la 2nd  valeur : ");
        val2 = Console.ReadLine();
        int valeur1 = Convert.ToInt32(val1), b = Convert.ToInt32(val2);
        Console.Write(valeur1.ToString() + " + " + b.ToString() + " =
⇒ ");
        //Jusque là, pas de problème, c'est du déjà vu.

        /*Ici, on appelle la méthode privée SommeDe qui n'est accessible
que lorsqu'on se trouve dans MaClass. A partir d'ici, le programme
bifurque et va dans la méthode SommeDe en lui passant les valeurs de
valeur1 pour a et de b pour b*/
        int somme = sommeDe(valeur1, b);

        /*Ici somme vaut s. On affiche somme et effectivement, il
s'affiche la même valeur que celle de s.*/
        Console.WriteLine(somme.ToString() + "\nMerci...");
        Console.Read();
    }

    /*Cette méthode est privée, on ne peut y accéder que dans cette
classe*/
    private int sommeDe(int a, int b)
    {
        /*Lorsqu'on arrive ici, la valeur de a est la même que celle de
valeur1 (vue au dessus) et la valeur de b est la même que celle de b
(vue au dessus)*/

        //Somme de a et b stocké dans s
        int s = a + b;
        //On renvoie s
        return s;
    }
}
```

# Débuguer un programme

Le débogage vous permet de suivre pas à pas l'évolution de votre programme et de contrôler ainsi ce qu'il s'y passe.

Nous allons débuguer ici l'exemple précédent.

Pour ce faire, commencez par poser un breakpoint sur la 1ere ligne de notre fonction Main (ce qui doit correspondre à la ligne 8 du programme). Pour poser un breakpoint, cliquez dans la marge grise sur la ligne correspondant à l'endroit où vous vous voulez le placer. Un point rouge s'affiche dès que le breakpoint est fixé.




Puis exécutez le programme (Start Debugging ou F5). Celui-ci s'interrompt quasi immédiatement marquant d'une flèche jaune le breakpoint que vous venez de poser. La flèche jaune indique la ligne où vous vous trouvez lors de votre débogage.
Vous constaterez également qu'un nouveau panneau est apparu en bas de votre EDI. Notamment les panneaux Locals et Watch qui vous permettent de voir en temps réel ce qui se passe dans votre programme.

Allez dans le panneau Locals si vous n'y êtes pas déjà. Vous pouvez déjà constater qu'il affiche une ligne, et en lisant le tableau vous savez que la variable myClass a pour valeur `null` et qu'elle est de type MaClass.

Appuyez ensuite sur F10 pour avancer d'un pas dans votre programme (la liste des commandes de débogage se trouve dans le menu Debug). Nous avons désormais passé la 1ere ligne qui crée un objet MaClass et le met dans la variable myClass. Vous noterez qu'à présent, notre variable myClass a changé de valeur. Sa valeur est de type {MyClass} maintenant. Tout c'est bien déroulé jusqu'à présent.

La prochaine ligne est myClass.Go(). Si vous faites F10 sur cette ligne, vous allez passer par-dessus, mais on a envie de voir ce qu'il a dedans, donc on va faire un Step Into (F11) et… bingo on a changé de classe et on se retrouve dans notre méthode publique Go qui appartient à la classe MaClass.
Vous constaterez en même temps que dans le panneau Locals, la variable myClass a disparu mais qu'elle a cédée sa place à d'autres biens plus nombreuses. Ce sont en fait toutes les variables accessibles depuis la fonction en cours (d'où le nom du panneau : Locals). On en déduit que les variables ont une portée limitée.

En poursuivant le débogage, vous pourrez voir que les valeurs vont changées au fur et à mesure que vous avancerez dans la méthode.
Notez bien le fonctionnement de la méthode sommeDe. Elle calcule une valeur, la renvoie et cette valeur renvoyée est stockée dans la variable somme de la méthode Go.

Vous pouvez vous déplacer à volonté dans une méthode en bougeant la flèche jaune où vous souhaitez reprendre le débogage. Vous pouvez également modifier les valeurs de vos variables à tout instant. Il vous suffit pour cela de cliquer dans la colonne valeur sur la ligne de la variable dont vous voulez modifier le contenu puis de valider une fois la modification terminée.

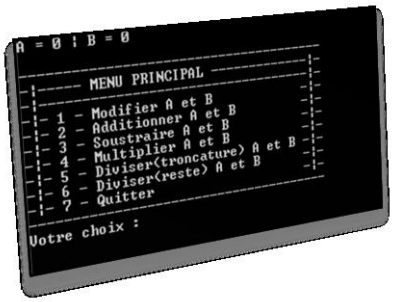
# Exercices

Exercice 20

Créez un programme qui demande à l'utilisateur d'entrer son année de naissance puis qui calcule, à partir de l'année courante, l'âge de la personne en question et l'affiche.
Votre fonction Main ne fait aucune des tâches citées ci-dessus. Dans la classe effectuant tout le travail, vous séparerez chacune des actions exigées par cet exercice dans une méthode différente (il y en a donc 3 en tout).

Exercice 21 : Calculatrice

Cette calculatrice, dont le menu correspond à l'image ci-dessous, devra faire en sorte que :




- A et B soient deux entiers
- chaque choix possible (sauf le 1 et le 7) soit traité et affiché dans une méthode différente
- la fonction Main n'effectue aucune des tâches du menu, ni même ne l'affiche
- l'affichage du menu est géré par une méthode qui renvoie un entier correspondant au choix de l'utilisateur
- avant d'afficher le menu, on nettoie la console (on la vide)
- l'utilisateur doit avoir le temps de pouvoir lire les résultats, qu'ils soient bons ou mauvais.

*Aide : Dans l'exercice 20, pour connaître l'année actuelle lorsque le programme est exécuté, utilisez* `DateTime.Today.Year` *qui renvoie un entier.*
*Dans l'exercice 21, pour nettoyer la console, utilisez la méthode* `Clear` *que celle-ci propose. Pour traiter le choix n°1 dans un* `switch/case`*, vous devrez créer de nouvelles variables, temporaires ; le* `switch/case` *ne le tolèrera que si vous placez le code, entre le case et le break de celui-ci, dans un bloc.*

# Les champs

Vous avez remarqué lors du débogage que les variables avaient une portée limitée à la fonction dans laquelle on les déclarait. Il est donc impossible d'appeler une même variable A déclarée dans une fonction F depuis une autre fonction G.

Le problème qui se pose est donc : Comment faire pour que la variable soit accessible partout dans notre classe, et plus encore…

La solution à ce problème est simple : nous devons utiliser des champs (*fields* en anglais).

Syntaxe :

```
<modificateur d'accès> <type> <identificateur>;
```

Pas grand-chose de nouveau donc. Vous vous servez désormais couramment de types et d'identificateurs lorsque vous déclarez une variable dans une fonction. Quant aux modificateurs d'accès, ce sont les mêmes que ceux utilisés pour les méthodes. Leur utilisation est identique : ils servent à déterminer si telle ou telle classe à le droit d'accéder à la variable.

Ils se déclarent directement dans la classe, au même titre qu'une fonction. Lorsque le modificateur d'accès n'est pas précisé, la variable est privée (`private`).

Exemple :

```
class MaClass
{
   private int maVariable;
   private string maChaîne;
   public double monReel;
   //fonctions
}
```

# Exercice d'application directe

Exercice 22

Reprendre l'exercice 21 en faisant en sorte de n'avoir plus aucun paramètre à passer aux méthodes de calculs ni à celle de l'affichage du menu et réécrivez le choix n°1 dans une méthode, exploitant dorénavant les champs.

# Les propriétés

Bien qu'il puisse paraître tentant de donner à tous ses champs un accès `public`, cela est en réalité très déconseillé, car dans ce cas, un module externe pourrait effectuer des manipulations interdites sur la variable ce qui risquerait de faire boguer le programme.

Par exemple, imaginez que vous vous trouvez dans un musée et que tous les objets qui s'y trouvent sont « `public` ».Cela voudrait dire, que vous, visiteur, avez tout à fait le droit de manipuler l'objet comme bon vous semble…. Heureusement ce n'est pas le cas, vous n'avez que le droit de voir, de regarder, mais pas de modifier…

Les propriétés ont le même but. Elles servent à dire si la variable à le droit d'être lu ou d'être écrite ou les deux.

Syntaxe :

```
<modificateur d'accès> <type> <nom de la propriété>
{
   get
   {
         //instructions
   }
   set
   {
         //instructions
   }
}
```

Vous remarquez qu'une propriété est différente d'une méthode :

- une propriété n'a jamais d'arguments, d'où la disparition des parenthèses
- une propriété contient au moins un des deux accesseurs (`get` ou `set`) suivi d'au moins une instruction

Si vous utilisez l'accesseur `get`, celui-ci doit obligatoirement retourner une valeur du même type que celui définit pour la propriété. Ce type sera donc forcément différent de `void` (vide).

Si vous utilisez l'accesseur `set`, vous n'aurez pas de valeur à retourner. Vous pourrez au contraire modifier une variable grâce à la valeur passée par la méthode `set`. Cette valeur est stockée dans un mot clé : `value`. Ce mot clé n'est disponible que dans l'accesseur set d'une propriété.

Par conséquent le type d'une propriété sera toujours différent de `void`.

Exemple :

Voici un petit exemple qui illustre comment utiliser les propriétés conjointement avec des méthodes.
Nous avons ici une classe Animal. Cette classe contient le nom de l'animal et son sexe que nous allons stocker sous forme booléenne.

Nous créerons notre animal grâce à une méthode à deux arguments.
Nous voulons avoir le droit de modifier le nom de l'animal (ce que l'on peut faire aussi couramment) mais pas son sexe (difficile de changer le sexe d'un animal en principe).
Nous utiliserons une méthode dans le but de récapituler toutes ses informations sous la forme d'une chaîne de caractères.

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        //On crée une instance d'Animal, contenue dans ani.
        Animal ani = new Animal();
        //On crée un animal qui s'appele Norbert et qui est un mâle.
        ani.Créer("Norbert", true);
        /*On demande le récapitulatif de l'animal que l'on stocke dans
animalString.*/
        string animalString = ani.Récapitulatif();
        //On affiche animalString
        Console.WriteLine(animalString);

        //On change le nom de l'animal
        ani.Nom = "Nestor";
        //On affiche directement le récapitulatif
        Console.WriteLine(ani.Récapitulatif());

        //Admirez le résultat.
        Console.Read();
    }
}

class Animal
{
    //Champs
    private string nom;
    private bool estUnMale;

    //Propriétés

    /*Cette propriété nous permet d'obtenir et de modifier le nom de
l'animal*/

    public string Nom
    {
        get
        {
            return nom;
```

```
        }
        set
        {
            /*Ici nous sommes sur que value est de type string car
c'est le type de la propriété.*/
            nom = value;
        }
    }

    /*Cette propriété ne permet que de connaître le sexe de l'animal,
mais pas de le modifier*/
    public bool EstUnMale
    {
        get
        {
            return estUnMale;
        }
    }
    /*Cette méthode à deux argument permet d'affecter les deux champs
de la classe animal*/
    public void Créer(string nomDuBestio, bool estMale)
    {
        nom = nomDuBestio;
        estUnMale = estMale;
    }

    //Cette méthode nous renvoie une chaine de caractères
    public string Récapitulatif()
    {
        //Début de la chaine...
        string recap = "Animal : " + nom + " de sexe ";

        /*... et sa fin, répondant d'une condition ternaire : s'il
n'est pas male, alors c'est une femelle*/
        recap += estUnMale ? "male." : "femelle.";

        //Retourne la chaine de caractères
        return recap;
    }
}
```

Cet exemple est également autoguidé. Si vous avez du mal à comprendre ce qui se passe, faites un petit débogage, çà aide toujours.

Remarquez que la variable `nom` n'est pas confondue avec la propriété `Nom`, et de même pour la variable `estUnMale` et la propriété `EstUnMale`. Cela est du au fait que C# est sensible à la casse[9]. Par conséquent, pour le programme, une variable qui s'appelle `variable` est différente d'une autre qui s'appelle `Variable` ou encore `VARIABLE`, etc. Nous reviendrons sur la casse et les conventions de nommage plus loin dans ce tutorial.

# Constructeurs de classes

Vous aurez probablement remarqué dans l'exemple précédent un petit détail un peu embêtant : il s'agit de la méthode Créer. Cette méthode étant disponible tout le temps pour notre Animal, nous pouvions en réalité changer son sexe dès que nous en avions envie. Ce n'est pas une super bonne technique donc.

[9] Le fait de différencier les lettres minuscules des lettres majuscules

Ce dont nous avons besoin c'est d'une méthode (pour passer quelques paramètres) qui ne soit accessible qu'une seule fois (lors de la création de la classe par exemple) et qui puisse remplir nos champs.

Cette méthode particulière existe et a pour nom : constructeur.

Un constructeur de classe est une méthode en tout point identique aux autres méthodes sauf :
- qu'il doit porter le nom exact de sa classe (en respectant la casse)
- qu'il n'a pas de type de retour

Syntaxe :

```
class MaClass
{
    //champs

    public MaClass(<arguments>)
    {
        //code de construction
    }

    //propriétés, méthodes
}
```

Un constructeur est appelé lors de la création de la classe, jamais à un autre moment.
Un constructeur peut être déclaré `private` si vous souhaitez le rendre invisible aux autres classes.
Si vous ne spécifiez pas de constructeur, le compilateur vous en crée un par défaut. Celui-ci se contente généralement d'initialiser tout les champs de sa classe : les `string` à une chaîne vide, les type valeurs (`int` et autres) à 0 et les booléens à `false`.
Pour toute autre manipulation vous devez écrire votre propre constructeur.

Exemple :

Reprenons l'exemple précédent en y modifiant quelques lignes.

Modifiez la fonction Créer en la transformant en un constructeur :

```
    /*Ce constructeur à deux argument permet d'affecter les deux champs
de la classe animal*/
    public Animal(string nomDuBestio, bool estMale)
    {
        nom = nomDuBestio;
        estUnMale = estMale;
    }
```

Maintenant retournez dans la fonction Main et modifiez le début :

```
        //On crée une instance d'Animal, contenue dans ani.
        Animal ani = new Animal();
        //On crée un animal qui s'appele Norbert et qui est un mâle.
        ani.Créer("Norbert", true);
```

en

```
        /*On crée une instance d'Animal, contenue dans ani. Cette
instance est une Animal qui s'appele Norbert et qui est mâle.*/
        Animal ani = new Animal("Norbert", true);
```

Vous aurez remarqué qu'en retapant `new Animal(` , Visual C# vous a automatiquement proposé de rajouter une chaîne de caractères (string) et un booléen pour construire votre Animal…

Désormais, il n'y a plus aucun litige possible : nous avons un seul animal d'un seul sexe et il nous est impossible de le changer sauf si l'on change d'animal, ce qui correspond à un modèle réaliste.

Débuggez cet exemple en plaçant deux breakpoint :
- un sur la 1ere ligne du constructeur
- un sur la ligne qui instancie `ani` (`Animal ani = new Animal("Norbert", true);`)

# This

`this` est un mot clé disponible depuis n'importe quel endroit dans une classe.

Il permet d'avoir accès avec certitude aux membres d'une classe.

Les membres d'une classe sont tout ce qui se trouve dans la classe, c'est-à-dire :
- des constructeurs
- des champs
- des méthodes
- des propriétés

Exemple d'utilisation :

Reprenons encore une fois notre exemple, désormais favoris, Animal.

Le début de notre classe ressemble à ceci :

```
private string nom;
private bool estUnMale;

public Animal(string nomDuBestio, bool estMale)
{
   nom = nomDuBestiaux;
   estUnMale = estMale;
}
```

Il apparaît que « `nomDuBestio` » n'est pas un identificateur très approprié et trop familier. Si un chef de projet passe derrière il risque de ne pas trop apprécier…
Tout en restant explicite sur ce qu'il est demandé d'entrer pour le « `nomDuBestio` », il faut trouver un autre identificateur, et, n'ayant pas beaucoup d'imagination, nous choisissons de reprendre tout simplement le mot `nom` comme identificateur.

Sauf que : `nom = nom;`
Tout le monde s'y perd et surtout le compilateur. Pas de panique, vous venez d'apprendre un nouveau mot clé, spécialement conçu pour cibler les membres d'une classe. Or ici, c'est le **champ** `nom` que nous voulons cibler.

Ce bout de code devient donc :

```
private string nom;
private bool estUnMale;

public Animal(string nom, bool estUnMale)
{
   this.nom = nom;
   this.estUnMale = estUnMale;
}
```

Notez que j'en ai profité pour modifier l'identificateur du booléen, ce qui n'était pas du tout obligé.

De cet exemple, on déduit que la variable « préférée » dans une fonction est toujours la variable locale (aidez vous du débogage pour le vérifier). Une variable locale n'a qu'une portée limitée, elle se limite au bloc dans lequel elle est créée.

# Exercice

Exercice 23

En utilisant tout ce qui a été vu précédemment, créez un programme répondant aux contraintes suivantes :

- une classe doit lancer le programme, elle ne fait rien d'autre
- une classe s'occupe de répartir les tâches entre affichage et traitement des données
- une classe contient toutes les données

Ce programme permet à l'utilisateur d'entrer le nom d'un album de musique ainsi que son artiste. Un menu lui proposera :

- (1) d'entrer les données
- (2) d'afficher les données
- (Q) de quitter

Vous devrez utiliser : au moins un constructeur, au moins une propriété, au moins une méthode. Vos champs seront privés.

*Astuce :* *Vous pouvez considérer une classe comme les autres types (`int`, `double`, `float`, `char`, `string`, etc.). Cela signifie que vous pouvez déclarer une classe Y dans l'un des champs d'une classe X et manipuler comme bon vous semble votre classe Y depuis votre classe X. Exemple :* `private MyClass myClass;`

# Chapitre 6
# Les tableaux

## Syntaxe

Les tableaux servent à stocker une suite d'éléments consécutifs. Ils peuvent être unidimensionnel (à une seule ligne) ou multidimensionnel.

Déclaration d'un tableau unidimensionnel :

```
<type> [] <identificateur> = new <type>[<taille du tableau>];
```

Exemples :

```
//Un tableau d'entiers à 4 cases
int[] tableauEntiers = new int[4];
//Un tableau de caractères à 20 cases
char[] tableauCars = new char[20];
//Un tableau de chaînes de caractères à 10 cases
string[] tableauStrings = new string[10];
```

Vous n'êtes pas obligé de fixer la taille du tableau avant la compilation. Une variable peut la définir.

Exemple :

```
//Cette variable peut être modifiée pendant l'exécution...
int tailleDuTableau = 12;
/*...il s'en suivra que le tableau aura toujours la taille que contient
la variable tailleDuTableau au moment de sa déclaration (du tableau)*/
string[] tableauDeStrings = new string[tailleDuTableau];
```

Il existe deux types de tableaux multidimensionnels : les matrices qui sont des tableaux comportant le même nombre de colonnes que de lignes et les autres tableaux dont on peut déterminer précisément la taille pour chaque ligne et pour chaque colonne.

Déclaration d'une matrice :

```
<type> [,] <identificateur> = new <type>[<taille de la matrice>,<taille
de la matrice>];
```

Le nombre de virgules entre les crochets indique si le tableau est bidimensionnel, tridimensionnel, quadridimensionnel, etc. A chaque virgule, une nouvelle dimension est créée.

Ce qui a été vu pour un tableau unidimensionnel, vaut pour toute matrice.

Exemples :

```
//Une matrice 2D d'entiers à 4x4 cases
int[,] tableauEntiers = new int[4,4];
```

```
//Une matrice 5D de 20x20x20x20x20 caractères
char[,,,,] tableauCars = new char[20,20,20,20, 20];
//Une matrice 3D de 10x10x10 chaînes de caractères
string[,,] tableauStrings = new string[10,10,10];
//et…
//Cette variable peut être modifiée pendant l'exécution...
int tailleDeLaMatrice = 6;
/*...il s'en suivra que la matrice aura toujours la taille que contient
la variable tailleDeLaMatrice*/
string[,] matriceDeStrings = new string[tailleDeLaMatrice,
tailleDeLaMatrice];
```

Vous pouvez créer des matrices d'autant de dimensions que vous le voulez.

Les tableaux multidimensionnels non matriciels seront traités plus loin.

Pour accéder à une valeur dans un tableau, la syntaxe est la suivante :

```
<identifiant du tableau>[<position>];
```

Exemple :

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        /*Stockage de la taille du tableau dans une variable de manière
à pouvoir s'en servir facilement par la suite*/
        int tailleDuTableau = 10;
        //Création de notre tableau, un tableau d'entiers
        int[] tableau = new int[tailleDuTableau];

        //Quelques consignes
        Console.WriteLine("N'entrez que des nombres N (" +
⇒ tailleDuTableau + " au total) sinon le programme buggera.");
        Console.WriteLine("{0} < N < {1}.", int.MinValue,
⇒ int.MaxValue);

        /*La boucle qui va permettre à l'utilisateur de remplir le
tableau avec des valeurs*/
        for (int i = 0; i < tailleDuTableau; i++)
        {
            //Affichage de la position actuelle dans le tableau
            Console.Write("N[{0}] = ", i);
            //Demande à l'utilisateur d'entrer quelque chose
            string chaîne = Console.ReadLine();
            /*Rempli la case en cours avec ce quelque chose en le
transformant en un entier (si possible, sinon : bug)*/
            tableau[i] = Int32.Parse(chaîne);
        }

        Console.WriteLine("\nAffichage des valeurs entrée.\n");

        //Boucle d'affichage des valeurs contenues dans le tableau
        for (int i = 0; i < tailleDuTableau; i++)
        {
            Console.Write("Valeur de la case {0} : {1}", i,
⇒ tableau[i]);
            Console.ReadLine();
        }
    }
}
```

Pour davantage de détails, le débuggeur est votre ami…

**En informatique, on compte toujours à partir de 0.**
**Cela signifie qu'un tableau de 10 cases s'étend de la case 0 à la case 9 (ce qui fait bien 10 cases) et que la case 10 n'existe pas.**

# Travailler avec des tableaux

Les tableaux sont traités en C# comme des types à part entière. Il est donc très facile de travailler avec.

Par exemple, la méthode Length vous permettra de connaître la taille de votre tableau.

Exemple :

```
monTableau.Length;   //renvoi un entier indiquant la taille du tableau
```

La classe `Array` possède également un bon nombre de méthodes statiques permettant effectuant des opérations sur les tableaux.

Si les éléments du tableau font partis des types prédéfinis, il est possible de trier ces éléments par ordre ascendant ou descendant. Nous utilisons pour cela la méthode `Sort`.

Exemple :

```
Array.Sort(monTableau);
```

De même, pour inverser l'ordre de ces éléments, la méthode statique `Reverse` de la classe Array nous le fait bien gentiment.

Exemple :

```
Array.Reverse(monTableau);
```

# Foreach

Exemple :

```
using System;

class MainEntryPoint
{
    static void Main()
    {
        string[] noms = new string[]{ "Nicolas", "Jean", "Fabrice",
⇒ "Michel", "Maxime" };
        Console.WriteLine("Avant le \"Reverse\"");
        foreach (string nom in noms)
            Console.WriteLine("Nom : " + nom);
        Console.ReadLine();

        Array.Reverse(noms);

        Console.WriteLine("Après le \"Reverse\"");
        foreach (string nom in noms)
            Console.WriteLine("Nom : " + nom);
        Console.ReadLine();
    }
}
```

Cet exemple se passe de commentaires. Vous devriez être capable, en exécutant le programme ou même en le débuggant si besoin est, de comprendre ce qui s'y passe.

Ici, nous apprenons deux nouvelles choses :

- il est possible de remplir un tableau en même temps qu'il est déclaré
- `foreach` est une boucle mais cette boucle différente des autres rencontrées jusqu'à présent

Commençons par la première :

```
string[] noms = new string[]{ "Nicolas", "Jean", "Fabrice", "Michel",
"Maxime" };
```

Nous créons un tableau de chaînes de caractères dont l'identifiant est nom, et, dans le même temps, nous informons le compilateur que `"Nicolas"` doit occuper la 1ere case de ce tableau, `"Jean"` la seconde, `"Fabrice"` la troisième, etc. jusqu'à notre dernier prétendant, celui fixant la taille du tableau. Dans notre exemple, la taille du tableau est donc fixée à 5.

Cette technique de remplissage de tableau s'applique à tous les types de tableaux (entiers, réel, etc.) de toutes dimensions.

Exemples :

```
//une matrice à trois dimension, pas évident du premier coup
int[, ,] valeurs = new int[2, 2, 2]{
   {           //1er item
      {100, 101},    //1er item du 1er item
      {110, 111}     //2e item du 1er item
   },
   {           //2e item
      {200, 201},    //1er item du 2e item
      {210, 211}     //2e item du 2e item
   }
};

/*trois dimension, cela complique un peu quand même, voyons un exemple
plus compréhensible */
char[,] cars = new char[3, 3]{ //une matrice de 3 colonnes par 3 lignes
   {'a','z','e'},  //1ere ligne
   {'r','t','y'},  //2e ligne
   {'u','i','o'}   //3e ligne
};
```

Quoi qu'il en soit, la règle est toujours la même : les crochets contiennent la taille du tableau, ses dimensions… les accolades contiennent ses items. Les items sont séparés par des virgules.

Vous n'êtes pas obligé de mentionner la taille du tableau lorsque vous le créez de cette manière, le compilateur se charge à votre place de calculer le nombre d'items qu'il contient. Toutefois, si vous précisez que ce tableau est une matrice, il faut respecter le fait qu'il doit comporter autant de ligne que de colonnes et ceux, dans toutes ses dimensions.

De plus, les mots clés `new <type>` ne sont pas obligatoires lorsque le tableau est instancié de cette façon. Si vous précisez les données que doit contenir le tableau, le compilateur se chargera pour vous de mettre convenablement en forme votre tableau ou matrice.

Du coup, on peut réécrire notre liste de noms :

```
string[] noms = { "Nicolas", "Jean", "Fabrice", "Michel", "Maxime" };
```

**Une fois que la taille du tableau est fixée, celle-ci n'est plus modifiable. Il faudra créer un nouveau tableau et y copier les données de l'ancien afin d'agrandir ou de rétrécir l'original.**

Venons en maintenant au deuxième point : la boucle `foreach`

Cette boucle permet de parcourir par itération une collection. Pour simplifier, on peut dire qu'une collection est un objet contenant d'autres objets. C'est le cas de notre tableau de chaînes de caractères `noms`. `foreach` se déplace dans `noms` et à chaque itération, à chaque objet rencontré dans cette variable, `foreach` retourne cet objet. Ici nous savons qu'il s'agit d'une chaîne de caractères `string`. Nous demandons donc à ce que cet objet retourné, sous forme de `string`, soit stocké dans la variable `nom` (sans s), pour pouvoir l'exploiter.

Exemple :

```
int[] table; //Tableau d'entiers
//On le remplie quelque part...
//...et on affiche tout ce qu'il contient
foreach (int Z in table)
   //Affichage de Z
```

`foreach (int Z in table)` veut dire littéralement : pour chaque entier Z dans table (qui est un tableau d'entiers).

Je pense que cette traduction devrait bien vous aider à comprendre le fonctionnement de cette boucle.

`foreach` est très pratique, mais attention, il y a des restrictions. Il est en effet impossible d'affecter une variable créée par cette boucle.

Ainsi :

```
foreach (int Z in table)
   Z = 10;
```

est interdit. Ne comptez donc pas dessus pour remplir vos tableaux. Pensez y plutôt lors qu'il s'agira de les afficher…

# Exercice d'application

Exercice 24

Créez un programme qui décompose un nombre entré par l'utilisateur en un produit de nombres premiers. Une propriété devra renvoyer le tableau d'entiers en question. Ce tableau aura été rempli par une méthode se trouvant dans la même classe. Ce programme aura l'aspect suivant :

- l'utilisateur entre un nombre (évitez d'en entrer un trop grand du style à 100 chiffres)
- la factorisation en nombres premiers de son nombre est affichée sous la forme `monNombre = 2*3*5*7*…*N` où `N` est le plus grand nombre premier de la multiplication.

*Aide : Utilisez une boucle `for` et/ou `while` commençant à 2 et servant vous en pour effectuer la division euclidienne (% = modulo) du nombre. Si celle-ci renvoie 0, le nombre est divisible par le module en question et vous pouvez stocker le module et diviser le nombre par celui-ci. N'oubliez pas de vérifier à nouveau que le nombre n'est plus divisible par ce même module avant de l'incrémenter.*

*Exemple : Oula, compliqué ce qu'il dit. Voyons ceci avec un exemple : 10374 (que nous appellerons N) sous la forme d'un algorithme qui ne mérite pas ce titre et qui n'est pas terminé (je vous laisse ce soin là).*

*Début de la boucle*

*N modulo 2 = 0 donc 2 divise N donc on stocke 2 dans la 1ere case de notre tableau*

*On divise N par 2 et on stocke la valeur dans N*

*On recommence : N modulo 2 : est différent de 0, on passe…*

*Et on recommence avec 3, on refait tous les tests précédents et puis on continue avec 4, 5, etc., jusqu'à ce que N soit égal à 1 : c'est fini, on sort de la boucle.*

*Astuce : Etant donné que la taille du tableau ne peut être connu avant que celui-ci soit rempli, créez en un de taille suffisante, plus grand que nécessaire. Vous pourrez ensuite en créer un autre en épurant le précédant de toutes les cases contentant un 0 (étant vides).*

# Les tableaux multidimensionnels

Vous avez déjà vu au début de ce chapitre comment créer des matrices. Les matrices sont des tableaux multidimensionnels particuliers car elles comportent autant de lignes, de colonnes dans chacune de leurs dimensions.

Exemple de déclaration d'une matrice :

```
int[,] tableauEntiers = new int[4,4];
```

Les autres tableaux, ceux qui ne sont pas matriciels se créent de la même manière que les tableaux unidimensionnels : chaque groupe de crochets représente une dimension.

Syntaxe :

```
<type>[]…[] tableau;
```

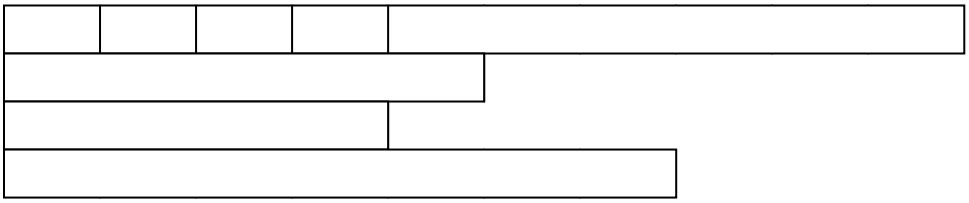
Les « … » signifient que vous pouvez placer autant de dimensions (de crochets) que vous le voulez.

Si vous aviez le droit avec une matrice ou un tableau unidimensionnel de préciser la taille de chacune de leurs dimensions en même temps que leur déclaration, c'est bien parce que ces tableaux-ci sont des cas particuliers. Pour un tableau multidimensionnel « classique », vous ne pouvez préciser la taille que de la 1ere dimension. Pour les autres dimensions, il faudra le faire plus loin dans le code.

Exemple :

```
int[][] tableauEntiers = new int[4][];
tableauEntiers[0] = new int[10];
tableauEntiers[1] = new int[5];
tableauEntiers[2] = new int[4];
tableauEntiers[3] = new int[7];
```

Ce tableau peut donc se représenter comme ceci en fin de compte :



Par contre il est **interdit** de faire :

```
int[][] tableauEntiers = new int[4][4];
```

L'avantage de ce genre de tableaux est que vous pouvez en tirez autant de sous tableaux que le désirez. Voyons cela par un morceau de code :

```
int[][] tableauEntiers = new int[5][];
//on remplie ce tableau…
foreach (int[] tableauNumOne in tableauEntiers)
   foreach (int valeur in tableauNumOne)
         //traitement
```

Il est possible de répéter ce schéma pour les tableaux tridimensionnel, quadridimensionnels, etc.

Autre exemple :

```
int[][][][] tableauEntiers = new int[5][][][];
//on remplie ce tableau…
foreach (int[][][] tableauNumOne in tableauEntiers)
   foreach (int[][] tableauNumTwo in tableauNumOne)
         foreach (int[] tableauNumThree in tableauNumTwo)
               foreach (int valeur in tableauNumThree)
                     //traitement
```

Par contre, lorsqu'il s'agit d'une matrice, le traitement est différent :

```
int[, ,] tableauEntiers = new int[5,5,5];
foreach (int valeur in tableauEntiers)
   //traitement
```

Vous ne pourrez donc tirer aucun tableau d'une matrice.

Pour savoir combien de dimensions comporte un tableau, utilisez la propriété `Rank`.

Exemple :

```
int nombreDeDimensions = tableau.Rank;
```

Pour connaître la taille d'une des dimensions du tableau, servez vous de la méthode `GetLength(int dimension)`.

Exemple :

```
int nombreDeDimensions = tableau.GetLength(3); /*donne la taille de la 4e dimension (on compte toujours à partir de 0)*/
```

# Exercices

Exercice 25

En vous basant sur l'exercice 23, réécrire ce programme en y ajoutant quelques fonctions :

- lorsque l'utilisateur lance le programme, celui-ci lui demande combien d'albums il compte ajouter. L'utilisateur doit entrer un nombre qui servira à déterminer la taille du tableau contenant les données.
- le menu devra proposer deux nouveaux choix : celui de pouvoir accéder directement aux informations d'un album par l'intermédiaire de son numéro. Les données de cet album pourront être soit modifiées, soit simplement affichées (ce qui nous fait bien 2 items de plus dans notre menu).
- lorsque l'utilisateur entre les données (lien (1) de l'ancien menu), il devra entrer toutes les données et remplir donc intégralement le tableau ; il en va de même lorsqu'il voudra les afficher (lien (2)).

Vous devrez anticiper les éventuels bugs générés par l'utilisateur.

*Aide : Passons en revue les problèmes qui vous sont posés :*

- *demande de valeur à l'utilisateur, en tenant compte des erreurs possibles (l'utilisateur entre un mot ou une phrase par exemple).*

*Pour remédier à ce problème utiliser la fonction `TryParse` de la classe `Int32`. Un petite exemple : `Int32.TryParse(maChaine, out monEntier);` N'oubliez pas le mot clé `out`, sans quoi vous aurez une erreur à la compilation.*

- *le menu, plus un problème pour vous maintenant…*

*Pensez bien à créer une fonction à part entière pour afficher le menu*

- *un tableau contenant un certain nombre d'albums (définit par son nom et son artiste), ce nombre étant fixé par l'utilisateur dès le début du programme*

*C'est ici qu'un problème plus sérieux apparaît. Dans l'exercice 23 vous utilisiez deux champs différents dans une classe créée spécialement pour remplir le rôle d'album, et bien qu'il puisse paraître tentant de transformer ces champs, qui sont de simples variables, en tableaux, il ne faut pas, ce n'est pas la bonne méthode.*

*Revoyons le problème : « tableau contenant un certain nombre d'albums ». Si on suit ce qui est écrit là, cela signifie que nous n'avons ni un tableau de noms d'albums, ni un tableau de noms d'artistes, mais un tableau regroupant le tout : un tableau d'albums. Mais qu'est qu'un album ? Vous l'avez déjà définit dans l'exercice 23 et depuis, un album n'a toujours pas changé. Il suffit donc de se dire que le nom de la classe représentant un album (que nous nommeront* `Album` *ici) est un type, et qu'on peut créer à partir d'un type un tableau contenant un certain nombre de fois ce genre de données. Par conséquent un tableau d'albums aura la forme suivante :* `Album[] myAlbums;` *Le fonctionnement de tableaux de classes (ici la classe* `Album`*) est exactement le même que celui des autres tableaux rencontrés auparavant.*

- *le programme doit anticiper les entrées utilisateurs erronées.*

*Pensez à utilisez les conditions, les booléens et les boucles* `while`*. Exemple : tant que l'utilisateur n'entre pas un nombre, on lui redemande de…*

Exercice 26

Ce programme-ci consiste à calculer la moyenne d'une classe. Il doit permettre à l'utilisateur de :

- changer le nombre d'élèves (limité à 10 lors du lancement du programme), la liste des anciens élèves est alors supprimée
- rentrer toutes les notes de chaque élève (il peut y avoir plusieurs notes pour un seul élève)
- changer les notes d'un élève en particulier (il devra rentrer toutes les notes de celui-ci à nouveau)
- afficher la moyenne de la classe
- afficher la moyenne de chaque élève, puis terminer par celle de la classe
- afficher la moyenne d'un élève en particulier
- quitter le programme bien entendu

Vous devrez gérer les éventuelles erreurs provoquées par l'utilisateur et apporter une interface utilisateur conviviale (toujours sur la console) en utilisant des tableaux de notes ou autres…

Pensez bien à définir, avant de commencer, ce qu'est un élève, une classe (≠ `class`) dans cet exercice et comment les implémenter.

Vous êtes libre de personnaliser ce programme du moment que les conditions établies ci-dessus sont remplies.

*Aide : Une ligne dans la console fait 80 (caractères) de large.*

*Remarques : Pour davantage de précision, utilisez un type flottant pour les moyennes.*

# Projet

## Introduction

Ce projet a pour but d'attester que tous les points traités auparavant dans ce tutorial ont bien été acquis et que vous êtes capable devant un problème donné, de le résoudre et d'arriver à créer le logiciel correspondant.

## Conventions de nommage

Afin que votre projet soit lisible et facilement compréhensible par d'autres programmeurs de la communauté C#, il faut respecter certaines conventions.

Casse des noms :

Dans la plupart des cas, utilisez la casse Pascal pour les noms. La casse Pascal signifie que la première lettre de chaque nom est une majuscule : `AfficherCustomer`, `SupprimerEmploye`, `AfficherEmployeSupprime`. Vous remarquerez que les espaces de noms et les classes de bases du Framework respectent cette casse.
Il est fortement déconseillé de séparer les mots par des traits de soulignement (*underscore*) : `Afficher_Customer` est donc déconseillé.
Dans d'autres langages, il était habituel de nommer les constantes en utilisant seulement des majuscules. Ce n'est pas conseillé en C#, ces noms sont plus difficiles à lires. Utilisez la casse Pascal :

```
const int MaConstante;
```

Soyez cohérents lorsque vous nommez vos méthodes, champs, classes… Par exemple, ne faîtes pas `EmployeAfficher` et `SupprimerEmploye` mais plutôt `AfficherEmploye` et `SupprimerEmploye` ou vice versa. Ou encore, faîtes `AfficherErreurEmploye`, `AfficherErreurCustomer` plutôt que `AfficherEmployeErreur` et `AfficherErreurCustomer`. Etc.

Une autre convention est également utilisée. Il s'agit de la casse chameau que l'on emploie dans les cas où les variables sont locales, privées. La casse chameau est quasiment identique à la casse Pascal sauf que la première lettre est une minuscule :

```
public int AdditionStringFormat(string format, int a, int b);
```

ou

```
private string key;

public string Key
{
    get { return key; }
}
```

Dans les autres cas, ceux où les variables sont publiques ou protégées, la casse pascale sera utilisée de préférence.

C# est sensible à la casse. Par conséquent `variable` est différent de `Variable` ou encore de `VARIABLE` mais n'oubliez pas que votre assemblage (votre code) peut être appelé par d'autres langages comme VB.NET. Or VB.NET n'est pas sensible à la casse. Par conséquent en VB. NET, `variable`, `Variable` ou `VARIABLE` sont identiques. Prenez donc bien soin de ne pas rendre accessible hors de l'assemblage des variables dont les identifiants ne diffèrent que par la casse. Dans l'exemple précédant `key` est privé et n'est donc pas accessible hors de l'assemblage, seul `Key` le sera.

# Projet

Préambule :

Vous aurez besoin dans ce projet d'utiliser certaines classes que vous n'avez jusqu'alors jamais utilisées. C'est comme çà. Vous ne pourrez, de toute façon, probablement jamais connaître l'intégralité des classes proposées par C#, ni comment les utiliser, tellement il y en a.
Cependant, vous avez à votre disposition le merveilleux outil qu'est Internet et tout ce qu'il offre par la même occasion.
Vous pourrez notamment trouver de l'aide sur le site de MSDN (Microsoft Developer Network) dédié à la programmation dans la partie Library (bibliothèque) ou encore, si vous utilisez Visual Studio, cette aide est disponible directement depuis l'EDI en cliquant sur l'un des liens proposés par le menu d'aide.
L'aide proposée par MSDN devrait vous permettre de trouver réponses à vos questions si vous cherchez correctement. Cependant si vous ne trouvez pas ce que vous cherchez sur MSDN, alors vous pourrez toujours aller rechercher (ou poster si besoin est) sur les nombreux forums de programmation disponibles sur internet.

Projet :

Le projet ici est très simple : réaliser un logiciel permettant de lister ses DVD. Ce logiciel sera accessible uniquement par la console.

Le menu doit proposer les liens suivants :

- Ajouter un DVD
- Editer un DVD
- Supprimer un DVD
- Afficher la liste complète
- Rechercher un DVD
- Réinitialiser la liste
- Enregistrer la liste
- Ouvrir une liste

Un DVD est décrit par :

- une clé, unique pour chaque DVD
- un nom, le titre du DVD
- un réalisateur (son nom)
- une petite liste des acteurs principaux
- un ou des genres

Pour éditer ou supprimer un DVD, il faudra obligatoirement utiliser sa clé, le désignant ainsi avec certitude.

L'affichage de la liste complète doit être le plus compact possible tout en restant clair. Celui-ci doit prendre un minimum de place.

La recherche d'un DVD propose à l'utilisateur d'entrer un nom de DVD. Celui-ci peut entrer le nom exact ou simplement les premières lettres. Quoi qu'il en soit, lorsqu'il valide, la liste des DVD - dont les titres commencent ou correspondent avec ce qu'il a entré - doit s'afficher.

Réinitialiser la liste signifie la remettre à zéro (*reset*), la liste est donc vidée.

Les fonctions enregistrer et ouvrir demandent à l'utilisateur d'entrer le chemin complet du fichier dans lequel il veut enregistrer sa liste et la liste est ensuite soit chargée, soit sauvée suivant un algorithme personnel (l'intérêt ici n'est pas de créer le meilleur algorithme possible mais simplement d'enregistrer la liste pour pouvoir la récupérer plus tard).

Partie codeur :

Une seule classe devra s'occuper de l'interface graphique (limitée ici à la console). La ou les autres classes nécessaires n'auront donc aucun appel de dessin dans la console à faire. Cela permettra de réutiliser facilement votre code par la suite pour développer, par exemple, une interface plus poussée.

Ce qui a été traité dans ce tutorial est suffisant pour réaliser ce programme excepté au sujet de quelques classes qui vous seront nouvelles. Cependant, une bonne conception, pratique et facilement réutilisable, peut nécessiter des connaissances plus poussées dans certains domaines du langage. Faîtes au mieux. Vous pouvez bien sur vous documenter pour améliorer la conception du logiciel.

**Vous n'aurez pas le droit d'utiliser l'espace de nom System.Collections ni ses nœuds sous-jacents. Autrement dit, la classe ArrayList, les Generics ou tout autre classe appartenant à ces espaces de noms sont interdit ici. Il en va de même pour l'espace de nom System.Data. La classe Array est également interdite. Vous n'avez pas non plus le droit d'utiliser XML pour sauvegarder.**

*Remarques :*

- *Vous pourrez (et même devrez) séparer vos différentes classes dans différents fichiers.*
- *Vous êtes totalement libre en ce qui concerne l'affichage dans la console. A vous de voir ce qui vous plait le mieux. Vous pouvez également perfectionner ce petit programme. Par exemple en ajoutant des options dans le mode recherche, en ajoutant de nouveaux liens dans le menu principal, etc. Ne perdez cependant pas de vue que ce programme est censé pouvoir être remis entre les mains d'un simple utilisateur de votre PC et que par conséquent, ce logiciel doit rester le plus clair et le plus facile d'utilisation possible.*
- *Une fois que vous penser avoir terminé ce projet, fermez Visual Studio et lancez directement votre application comme un utilisateur normal et regardez alors si tout se présente et fonctionne comme prévu.*
- *Pour info pratique : la console peut loger 80 caractères en tout sur sa largeur.*

# Pour aller plus loin

Réécrivez ce programme en utilisant la classe `System.Collections.Array`.

Créez vos propres espaces de noms pour ce programme.

Réécrivez ce programme en utilisant les Generics.

Un peu plus difficile :

Optimisez votre code afin d'en minimiser la quantité (surcharge de méthodes, réutilisation du code, etc.).

Bonne prog !