

Programmation Orientée Objet sous C# et .NET

Table des matières

1	Introduction	4
2	Programmation Orientée Objet	5
2.1	Encapsulation	5
2.2	L' Héritage	7
2.3	Le Polymorphisme	9
3	Les bases du langage C#.....	10
3.1	Types de Données	10
3.1.1	Notation des données littérales	13
3.1.2	Déclaration des données.....	14
3.1.3	Les conversions entre nombres et chaînes de caractères	15
3.1.4	Les tableaux de données.....	17
3.2	Les instructions élémentaires de C#.....	20
3.2.1	Affectation de la valeur d'une expression à une variable	20
3.2.2	Expression arithmétique	20
3.2.3	Priorités dans l'évaluation des expressions arithmétiques.....	21
3.2.4	Expressions relationnelles.....	22
3.2.5	Expressions booléennes	23
3.2.6	Traitement de bits.....	24
3.2.7	Combinaison d'opérateurs.....	25
3.2.8	Opérateurs d'incrémentatation et de décrémentatation	25
3.2.9	L'opérateur ternaire « ? »	25
3.2.10	Priorité générale des opérateurs	26
3.2.11	Les conversions de type	26
3.2.12	Mise en Forme de Types Numériques	27
3.2.13	Mise en Forme de Dates	28
3.3	Les instructions de contrôle du déroulement du programme	29
3.3.1	Arrêt	29
3.3.2	Structure de choix simple (if)	29
3.3.3	Structure de cas (switch).....	31
3.3.4	Boucle While (tant que)	32
3.3.5	Boucle Do While (jusqu'à ce que)	32
3.3.6	Boucle For (pour)	33
3.3.7	Boucle Foreach (pour chaque)	34
3.3.8	La Gestion des Exceptions.....	35
3.4	Arguments du programme principal	40
3.5	Les énumérations	41
3.6	Les Collections.....	41
3.7	Passage de paramètres à une fonction	44
3.7.1	Passage par valeur	45
3.7.2	Passage par référence.....	45
3.7.3	Passage par référence avec le mot clé out	46
3.8	Membres Statiques	48
4	Classes .NET d'usage courant.....	50
4.1	String	50
4.1.1	String.Format	54
4.2	StringBuilder	54
4.3	Array	57
4.4	StreamReader	62

4.5	StreamWriter	65
4.6	Random	68
5	Interfaces Graphiques	70
5.1	Les bases des interfaces graphiques	70
5.1.1	Un premier projet	70
5.2	Les composants de base	76
5.2.1	Boîte de Dialogue	77
5.2.2	Propriétés, Méthodes et Evénements communs.....	77
5.2.3	Formulaire (« Form »)	79
5.2.4	Etiquette (« Label »).....	82
5.2.5	Boîte de saisie (« TextBox »)	83
5.2.6	Liste (« ListBox ») et Liste Déroulante (« ComboBox »)	86
5.2.7	Case à Cocher (« CheckBox ») et Bouton Radio (« RadioButton »).....	95
5.2.8	Menus	99
5.3	Composants non visuels	101
5.3.1	Boîtes de dialogue OpenFileDialog et SaveFileDialog	101
5.3.2	Timer	106
5.4	Événements souris	109
5.5	Ajout Dynamique de Composants	111
5.6	Gestionnaire d'Événement Commun	115
5.7	Application Ayant Plusieurs Formulaires	118
Annexe I.	Convention de Nommage	121
Annexe II.	Convention de Codage.....	122
Annexe III.	Chaînes de Format.....	123

1 Introduction

C# (prononcer « sicharp ») est un langage récent. Il a été disponible en versions beta successives depuis l'année 2000 avant d'être officiellement disponible en février 2002 en même temps que la plate-forme .NET 1.0 de Microsoft à laquelle il est lié. C# ne peut fonctionner qu'avec cet environnement d'exécution. Celui-ci rend disponible aux programmes qui s'exécutent en son sein un ensemble très important de classes. En première approximation, on peut dire que la plate-forme .NET est un environnement d'exécution analogue à une machine virtuelle Java. On peut noter cependant deux différences :

1. Java s'exécute sur différents OS (windows, unix, macintosh) depuis ses débuts. En 2002, la plate-forme .NET ne s'exécutait que sur les machines Windows. Depuis quelques années le projet Mono [<http://www.mono-project.com>] permet d'utiliser la plate-forme .NET sur des OS tels que Unix et Linux. La version actuelle de Mono (février 2008) supporte .NET 1.1 et des éléments de .NET 2.0.
2. La plate-forme .NET permet l'exécution de programmes écrits en différents langages. Il suffit que le compilateur de ceux-ci sache produire du code IL (Intermediate Language), code exécuté par la machine virtuelle .NET. Toutes les classes de .NET sont disponibles aux langages compatibles .NET ce qui tend à gommer les différences entre langages dans la mesure où les programmes utilisent largement ces classes. Le choix d'un langage .NET devient affaire de goût plus que de performances.

En 2002, C# utilisait la plate-forme .NET 1.0. C# était alors largement une « copie » de Java et .NET une bibliothèque de classes très proche de celle de la plate-forme de développement Java. Dans le cadre de l'apprentissage du langage, on passait d'un environnement C# à un environnement Java sans être vraiment dépaycé. On trouvait même des outils de conversion de code source d'un langage vers l'autre. Depuis, les choses ont évolué. Chaque langage et chaque plate-forme de développement a désormais ses spécificités. Il n'est plus aussi immédiat de transférer ses compétences d'un domaine à l'autre.

C# 3.0 et le framework .NET 3.5 amènent beaucoup de nouveautés. La plus importante est probablement LINQ (Language INtegrated Query) qui permet de requêter de façon uniforme, d'une façon proche de celle du langage SQL, des séquences d'objets provenant de structures en mémoire telles que les tableaux et les listes, de bases de données (SQL Server uniquement pour le moment - février 2008) ou de fichiers XML.

Ce document n'est pas un cours exhaustif. Par exemple, LINQ n'y est pas abordé. Il est destiné à des personnes connaissant déjà la programmation et qui veulent découvrir les bases de C#.

2 Programmation Orientée Objet

La Programmation Orientée Objet est dirigée par 3 fondamentaux qu'il convient de toujours garder à l'esprit : **encapsulation**, **héritage** et **polymorphisme**.

Ce module se limite à la notion d'encapsulation et ne traite ni d'héritage, ni de polymorphisme au delà de cette section.

2.1 Encapsulation

Derrière ce terme se cache le concept même de l'objet : réunir sous la même entité les données et les moyens de les gérer, à savoir les champs et les méthodes.

L'**encapsulation** introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis

un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.

Si l'**encapsulation** est déjà une réalité dans les langages procéduraux (comme le Pascal non objet par exemple) au travers des unités et autres librairies, il prend une toute nouvelle dimension avec l'**objet**.

En effet, sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet et donc l'ensemble des procédures et fonctions destinées à la gestion *interne* de l'objet, auxquelles le programmeur final n'aura pas à avoir accès. L'**encapsulation** permet donc de masquer un certain nombre de champs et méthodes tout en laissant visibles d'autres champs et méthodes. Nous verrons ceci un peu plus loin.

L'**encapsulation** permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

Un objet est créé selon un modèle qu'on appelle une classe :

```
public class C1{
    Type1 p1;
    Type2 p2;
    ...
    Type3 m3(...) {
        ...
    }
    Type4 m4(...) {
    ... }
    ... }
}
```

A partir de la classe *C1* précédente, on peut créer de nombreux objets *O1*, *O2*,... Tous auront les champs *p1*, *p2*,... et les méthodes *m3*, *m4*, ... Mais ils auront des valeurs différentes pour leurs champs *pi* ayant ainsi chacun un état qui leur est propre.

On fait référence aux propriétés et méthode d'un objet au moyen d'un « . » placé entre l'identificateur de l'objet et celui de la propriété ou méthode. Ainsi donc, si *o1* est un objet de type *C1*, *o1.p1* désigne la propriété *p1* de *o1* et *o1.m1* la méthode *m1* de *O1*.

Exemple : Nous allons définir une classe « *Personne* » :

```
class Personne
{
    public int Age;           // Age de la personne
    public double Taille;    // Taille de la personne (en mètres)
    public void FeterAnniversaire() // Ajoute une année à l'âge de la personne
    {
        Age = Age + 1;
    }
    public void Grandir(double x) // Ajoute x mètres à la taille de la personne
    {
        Taille = Taille + x;
    }
}
```

Note : le mot-clé *public* indique que les attributs *Age* et *Taille*, ainsi que méthodes *FeterAnniversaire* et *Grandir* sont accessible depuis l'extérieur d'un objet de classe *Personne*. Inversement, le mot-clé *private* permet de spécifier qu'un attribut ou une méthode ne sont accessible que depuis le code de la classe elle-même. Dans ce cas, une méthode peut être considérée comme une simple fonction ou procédure.

Ensuite, nous créons deux objets distincts de la classe *Personne* :

```
Personne Pierre = new Personne();
Personne Paul = new Personne();
```

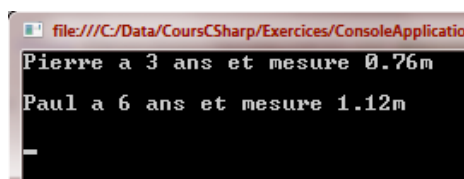
Nous pouvons maintenant utiliser et modifier nos deux objets indépendamment l'un de l'autre :

```
Pierre.FeterAnniversaire();  
Pierre.FeterAnniversaire();  
Pierre.FeterAnniversaire();  
Pierre.Grandir(0.76);  
Paul.FeterAnniversaire();  
Paul.FeterAnniversaire();  
Paul.FeterAnniversaire();  
Paul.FeterAnniversaire();  
Paul.FeterAnniversaire();  
Paul.FeterAnniversaire();  
Paul.Grandir(1.12);
```

Suite à ces appels, on peut (dans une application console) afficher l'état de nos deux personnages :

```
Console.Out.WriteLine("Pierre a {0:D} ans et mesure {1:F}m\n", Pierre.Age,  
Pierre.Taille);  
Console.Out.WriteLine("Paul a {0:D} ans et mesure {1:F}m\n", Paul.Age, Paul.Taille);
```

Ce qui produit le résultat suivant:



```
file:///C:/Data/CoursCSharp/Exercices/ConsoleApplication  
Pierre a 3 ans et mesure 0.76m  
Paul a 6 ans et mesure 1.12m  
-
```

2.2 L' Héritage

Si l'encapsulation pouvait se faire manuellement (grâce à la définition d'une unité par exemple), il en va tout autrement de l'**héritage**. Cette notion est celle qui s'explique le mieux au travers d'un exemple. Considérons un objet *Bâtiment*. Cet objet est pour le moins générique, et sa définition reste assez vague. On peut toutefois lui associer divers champs, dont par exemple :

- Les murs
- Le toit
- Une porte
- L'adresse
- La superficie

On peut supposer que cet objet *Bâtiment* dispose d'un ensemble de méthodes destinées à sa gestion. On pourrait ainsi définir entre autres des méthodes pour :

- Ouvrir le Bâtiment
- Fermer le Bâtiment
- Agrandir le Bâtiment

Grâce au concept d'**héritage**, cet objet *Bâtiment* va pouvoir donner naissance à un ou des *descendants*. Ces descendants vont tous bénéficier des caractéristiques propres de leur *ancêtre*, à savoir ses champs et méthodes. Cependant, les descendants conservent la possibilité de posséder leur propres champs et méthodes. Tout comme un enfant hérite des caractéristiques de ses parents et développe les siennes, un objet peut hériter des caractéristiques de son ancêtre, mais aussi en **développer de nouvelles**, ou bien encore se **spécialiser**.

Ainsi, si l'on poursuit notre exemple, nous allons pouvoir créer un objet *Maison*. Ce nouvel objet est toujours considéré comme un *Bâtiment*, il possède donc toujours des murs, un toit, une porte, les champs *Adresse* ou *Superficie* et les méthodes destinées par exemple à *Ouvrir le Bâtiment*.

Toutefois, si notre nouvel objet est toujours un *Bâtiment*, il n'en reste pas moins qu'il s'agit d'une *Maison*. On peut donc lui adjoindre d'autres champs et méthodes, et par exemple :

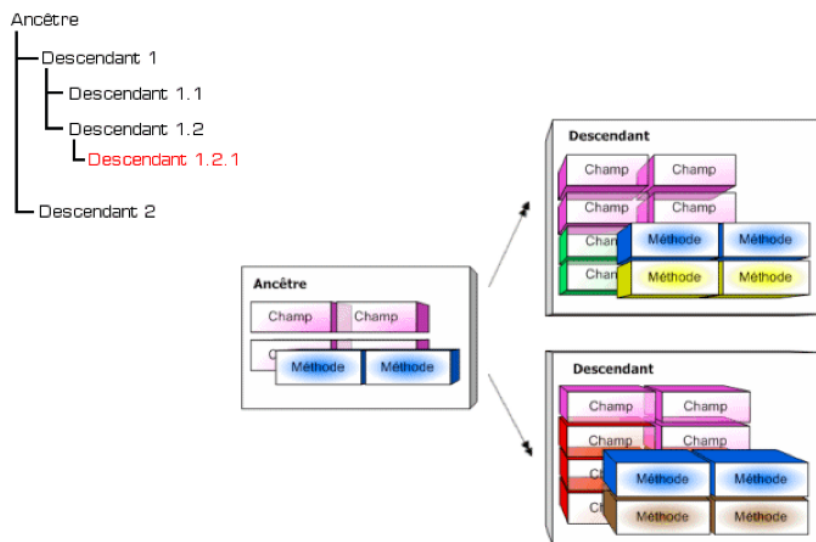
- Nombre de fenêtres
- Nombre d'étages
- Nombre de pièces
- Possède ou non un jardin
- Possède une cave

Notre *Bâtiment* a ici bien évolué. Il s'est **spécialisé**. Avec notre *Maison*, nous sommes allés plus avant dans les détails, et elle est à même de nous offrir des services plus évolués. Nous avons complété ce qui n'était qu'un squelette.

Ce processus d'héritage peut bien sûr être répété. Autrement dit, il est tout à fait possible de déclarer à présent un descendant de *Maison*, développant sa spécialisation : un *Chalet* ou encore une *Villa*. Mais de la même manière, il n'y a pas de restrictions théoriques concernant le nombre de descendants pour un objet. Ainsi, pourquoi ne pas déclarer des objets *Immeuble* ou encore *Usine* dont l'ancêtre commun serait toujours *Bâtiment*.

Ce concept d'**héritage** ouvre donc la porte à un nouveau genre de programmation.

On notera qu'une fois qu'un champ ou une méthode est définie, il ou elle le reste pour tous les descendants, quel que soit leur degré d'éloignement.



2.3 Le Polymorphisme

Le terme **polymorphisme** est certainement celui que l'on appréhende le plus. Afin de mieux le cerner, il suffit d'analyser la structure du mot : *poly* comme *plusieurs* et *morphisme* comme *forme*. Le **polymorphisme** traite de la capacité de l'objet à posséder *plusieurs formes*.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, comme on le sait déjà, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir **redéfinir une méthode** afin de la réécrire, ou de la **compléter**.

On voit donc le concept de **polymorphisme** : choisir en fonction des besoins la méthode ancêtre appeler, et ce au cours de l'exécution. Le comportement de l'objet devient modifiable à volonté.

Le **polymorphisme**, en d'autres termes, est donc la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours. Ainsi, si l'on considère un objet *Véhicule* et ses descendants *Bateau*, *Avion*, *Voiture* possédant tous une méthode *Avancer*, le système appellera la fonction *Avancer* spécifique suivant que le véhicule est un *Bateau*, un *Avion* ou bien une *Voiture*.

3 Les bases du langage C#

Ce chapitre décrit les éléments de base du langage, à savoir :

- Les types de données
- Les instructions élémentaires (affectation de valeur, évaluation d'expressions, opérateurs)
- Les structures de contrôle de déroulement d'un programme

Le langage est ici présenté pour l'instant comme un langage « classique », avec des exemples écrits pour une exécution en mode console. Les exercices de ce module ne se font que dans un mode graphique, mais la syntaxe présentée dans ce chapitre est naturellement valable dans ce mode.

Notation :

Dans les chapitres qui suivent, les exemples de code sont encadrés, en police « courier new », sur fond blanc :

```
Console.WriteLine ("Le programme démarre")
```

Les résultats d'un programme sont encadrés, en police « courier new », blanc sur fond bleu

```
Le programme démarre
```

Les définitions de syntaxe sont encadrées, en police « courier new », sur fond gris:

```
Variable = Expression
```

3.1 Types de Données

C# utilise les types de données suivants:

Type C#	Type .NET	Donnée représentée	Suffixe	Codage[octets]	Domaine de valeurs
char	Char	caractère		2	caractère Unicode (UTF-16)
string	String	chaîne de caractères		4+n	référence sur une séquence de caractères Unicode
int	Int32	nombre entier		4	[-2147483648, 2147483647]
uint	UInt32	nombre entier	U	4	[0, 4294967295]
long	Int64	nombre entier	L	8	[-9223372036854775808, 9223372036854775807]
ulong	UInt16	nombre entier	UL	8	[0, 18446744073709551615]
sbyte		nombre entier		1	[-128,+127]
byte	Byte	nombre entier		1	[0,255]
short	Int16	nombre entier		2	[-32768, 32767]
ushort		nombre entier		2	[0,65535]
float	Single	nombre réel	F	4	[1.5 10 ⁻⁴⁵ 3.4 10 ⁺³⁸] en valeur absolue

Type C#	Type .NET	Donnée représentée	Suffixe	Codage[octets]	Domaine de valeurs
double	Double	nombre réel	D	8	$[-1.7 \cdot 10^{+308}, 1.7 \cdot 10^{+308}]$ en valeur absolue
decimal	Decimal	nombre entier	M	16	$[1.0 \cdot 10^{+28}, 7.9 \cdot 10^{+28}]$ en valeur absolue avec 28 chiffres significatifs
bool	Boolean	valeur booléenne		1	true, false

Tableau 1 Les Types de Données

On découvre qu'il y a deux types possibles pour un entier sur 32 bits : *int* et *Int32*. *Int32* est une structure appartenant à l'espace de noms *System*. Son nom complet est ainsi *System.Int32*. Le type *int* est un alias C# qui désigne la structure .NET *System.Int32*. De même, le type C# *string* est un alias pour le type .NET *System.String*. *System.String* est le seul type du Tableau 1 qui est une classe et non une structure. Les deux notions sont proches avec cependant la différence fondamentale suivante :

- Une variable de type Structure se manipule via sa valeur
- Une variable de type Classe est un objet et il se manipule via son adresse (référence en langage objet).

Une structure comme une classe sont des types complexes ayant des attributs et des méthodes. Ainsi, on pourra écrire :

```
string nomDuType=3.GetType().FullName;;
```

Ci-dessus le littéral **3** est par défaut de type C# *int*, donc de type .NET *System.Int32*. Cette structure a une méthode *GetType()* qui rend un objet encapsulant les caractéristiques du type de données *System.Int32*. Parmi celles-ci, la propriété *FullName* rend le nom complet du type. On voit donc que le littéral **3** est un objet plus complexe qu'il n'y paraît à première vue.

Voici un programme illustrant ces différents points :

```
1. using System;
2. namespace Chap1 {
3. class P00 {
4. static void Main(string[] args) {
5. // exemple 1
6. int ent = 2;
7. float fl = 10.5F;
8. double d = -4.6;
9. string s = "essai";
10. uint ui = 5;
11. long l = 1000;
12. ulong ul = 1001;
13. byte octet = 5;
14. short sh = -4;
15. ushort ush = 10;
16. decimal dec = 10.67M;
17. bool b = true;
18. Console.WriteLine("Type de ent[{1}] : [{0},{2}]", ent.GetType().FullName,
    ent, sizeof(int));
19. Console.WriteLine("Type de fl[{1}]: [{0},{2}]", fl.GetType().FullName, fl,
    sizeof(float));
20. Console.WriteLine("Type de d[{1}] : [{0},{2}]", d.GetType().FullName, d,
    sizeof(double));
21. Console.WriteLine("Type de s[{1}] : [{0}]", s.GetType().FullName, s);
22. Console.WriteLine("Type de ui[{1}] : [{0},{2}]", ui.GetType().FullName, ui,
    sizeof(uint));
23. Console.WriteLine("Type de l[{1}] : [{0},{2}]", l.GetType().FullName, l,
    sizeof(long));
24. Console.WriteLine("Type de ul[{1}] : [{0},{2}]", ul.GetType().FullName, ul,
    sizeof(ulong));
25. Console.WriteLine("Type de b[{1}] : [{0},{2}]", octet.GetType().FullName,
    octet,
26. sizeof(byte));
27. Console.WriteLine("Type de sh[{1}] : [{0},{2}]", sh.GetType().FullName, sh,
    sizeof(short));
28. Console.WriteLine("Type de ush[{1}] : [{0},{2}]", ush.GetType().FullName,
    ush,
29. sizeof(ushort));
30. Console.WriteLine("Type de dec[{1}] : [{0},{2}]", dec.GetType().FullName,
    dec,
31. sizeof(decimal));
32. Console.WriteLine("Type de b[{1}] : [{0},{2}]", b.GetType().FullName, b,
    sizeof(bool));
33. } }
34. }
```

- **ligne 7 : déclaration d'un entier *ent***

- ligne 19 : le type d'une variable *v* peut être obtenue par *v.GetType().FullName*. La taille d'une structure *S* peut être obtenue par *sizeof(S)*. L'instruction *Console.WriteLine("... {0} ... {1} ...", param0, param1, ...)* écrit à l'écran, le texte qui est son premier paramètre, en remplaçant chaque notation *{i}* par la valeur de l'expression *parami*.
- ligne 22: le type *string* désignant une classe et non une structure, on ne peut utiliser l'opérateur *sizeof*.

Voici le résultat de l'exécution :

```
Type de ent[2] : [System.Int32,4]
Type de fl[10,5]: [System.Single,4]
Type de d[-4,6] : [System.Double,8]
Type de s[essai] : [System.String]
Type de ui[5] : [System.UInt32,4]
Type de l[1000] : [System.Int64,8]
Type de ul[1001] : [System.UInt64,8]
Type de b[5] : [System.Byte,1]
Type de sh[-4] : [System.Int16,2]
Type de ush[10] : [System.UInt16,2]
Type de dec[10,67] : [System.Decimal,16]
Type de b[True] : [System.Boolean,1]
```

L'affichage produit les types .NET et non les alias C#.

3.1.1 Notation des données littérales

Type C#	Notation
entier (32 bits)	145, -7, 0xFF (hexadécimal)
entier (64 bits) - suffixe L	100000L
Réel double	10134.789, -45E-18 (-45 10 ⁻¹⁸)
Réel - suffixe F	134.789F, -45E-18F (-45 10 ⁻¹⁸)
Réel decimal- suffixe M	100000M
Caractère	'A', 'b'
Chaîne de caractères	"aujourd'hui" "c:\\chap1\\paragraph3" @"c:\chap1\paragraph3"
Booléen	true, false
Date	new DateTime(1954,10,13) (an, mois, jour) pour le 13/10/1954

Tableau 2 Notations

On notera les deux chaînes littérales : *"c:\\chap1\\paragraph3"* et *@"c:\chap1\paragraph3"*. Dans les chaînes littérales, le caractère ** est interprété. Ainsi *"\n"* représente la marque de fin de ligne et non la succession des deux caractères ** et *n*. Si on voulait cette succession, il faudrait écrire *"\\n"* où la séquence ** est interprétée comme un seul caractère **. On pourrait écrire aussi *@"\\n"* pour avoir le même résultat. La syntaxe *@"texte"* demande que *texte* soit pris exactement comme il est écrit. On appelle parfois cela une chaîne **verbatim**.

3.1.2 Déclaration des données

3.1.2.1 Rôle des déclarations

Un programme manipule des données caractérisées par un nom et un type. Ces données sont stockées en mémoire. Au moment de la traduction du programme, le compilateur affecte à chaque donnée un emplacement en mémoire caractérisé par une adresse et une taille. Il le fait en s'aidant des déclarations faites par le programmeur.

Par ailleurs celles-ci permettent au compilateur de détecter des erreurs de programmation. Ainsi l'opération

```
x=x*2;
```

sera déclarée erronée si x est une chaîne de caractères par exemple.

3.1.2.2 Déclaration des constantes

La syntaxe de déclaration d'une constante est la suivante :

```
const type nom=valeur; //définit constante nom=valeur
```

Par exemple :

```
const float myPI=3.141592F;
```

Pourquoi déclarer des constantes ?

1. La lecture du programme sera plus aisée si l'on a donné à la constante un nom significatif :

```
const float taux_tva=0.186F;
```

2. La modification du programme sera plus aisée si la "constante" vient à changer. Ainsi dans le cas précédent, si le taux de tva passe à 33%, la seule modification à faire sera de modifier l'instruction définissant sa valeur :

```
const float taux_tva=0.33F;
```

Si l'on avait utilisé 0.186 explicitement dans le programme, ce serait alors de nombreuses instructions qu'il faudrait modifier.

3.1.2.3 Déclaration des variables

Une variable est identifiée par un nom et se rapporte à un type de données. C# fait la différence entre majuscules et minuscules. Ainsi les variables **FIN** et **fin** sont différentes.

Les variables peuvent être initialisées lors de leur déclaration. La syntaxe de déclaration d'une ou plusieurs variables est :

```
Identificateur_de_type variable1[=valeur1],variable2=[valeur2],...;
```

où *Identificateur_de_type* est un type prédéfini ou bien un type défini par le programmeur. De façon facultative, une variable peut être initialisée en même temps que déclarée.

3.1.3 Les conversions entre nombres et chaînes de caractères

Pour convertir		Utiliser
De	En	
Nombre	Chaîne de caractères	nombre.ToString()
Chaîne de caractères	int	int.Parse(chaine) int.TryParse(chaine, out int)
Chaîne de caractères	long	long.Parse(Chaine) long.TryParse(chaine, out long)
Chaîne de caractères	double	double.Parse(Chaine) double.TryParse(chaine, out double)
Chaîne de caractères	float	float.Parse(Chaine) float.TryParse(chaine, out float)

Tableau 3 Conversions

La conversion d'une chaîne vers un nombre peut échouer si la chaîne ne représente pas un nombre valide. Il y a alors génération d'une erreur fatale appelée exception. Cette erreur peut être gérée par la clause try/catch (voir 3.3.8).

Une autre manière de gérer une erreur de conversion consiste à faire appel à la méthode TryParse(chaine, out variable). Celle-ci fonctionne différemment de Parse(chaine) : au lieu de retourner la valeur numérique, elle retourne une valeur booléenne qui indique si la conversion a réussi (true) ou échoué (false). Dans le cas où elle a réussi, la valeur numérique est alors stockée dans la variable passée comme second paramètre – et obligatoirement précédé du mot-clé « out »

Voici un programme présentant quelques techniques de conversion entre nombres et chaînes de caractères.

```

// Définissons des valeurs numériques stockées sous forme textuelle
string ValeurEntiere = "10";
string ValeurReelle = "3.14";
string ValeurNonNumérique = "123ab";

// Définissons des variables numériques
int Entier; // Une variable entière
double Reel; // Une variable réelle

// Définissons une variable booléenne
bool Resultat;

//-----
// Ce que l'on peut faire:
//-----
Entier = int.Parse(ValeurEntiere); // La variable Entier reçoit la
// valeur numérique 10
Reel = int.Parse(ValeurEntiere); // La variable Reel reçoit la
// valeur numérique 10.0
Reel = double.Parse(ValeurEntiere); // La variable Reel reçoit la
// valeur numérique 10.0
Reel = double.Parse(ValeurReelle); // La variable Reel reçoit la
// valeur numérique 3.14
Resultat=int.TryParse(ValeurEntiere, out Entier); // La variable Entier reçoit la
// valeur numérique 10
// La variable Resultat reçoit la
// valeur VRAI (conversion réussie)
Resultat= int.TryParse(ValeurReelle, out Entier); // La variable Resultat reçoit la
// valeur FAUX (conversion échouée)
// Entier est indéterminée
Resultat=double.TryParse(ValeurEntiere,out Reel); // La variable Reel reçoit la
// valeur numérique 10.0
// La variable Resultat reçoit la
// valeur VRAI (conversion réussie)
Resultat=double.TryParse(ValeurReelle, out Reel); // La variable Reel reçoit la
// valeur numérique 3.14
// La variable Resultat reçoit la
// valeur VRAI (conversion réussie)

//-----
// Ce que l'on NE PEUT PAS faire:
//-----
Entier = int.Parse(ValeurReelle); // Conversion impossible, le
// programme plantera à l'exécution
Reel = int.Parse(ValeurReelle); // Pareil
Entier = double.Parse(ValeurEntiere); // Types incompatibles, le
// programme ne compilera pas
Entier = double.Parse(ValeurReelle); // Pareil
Resultat = int.TryParse(ValeurEntiere, out Reel); // Pareil
Resultat=double.TryParse(ValeurEntiere, out Entier); // Pareil

```


On remarquera que les nombres réels sous forme de chaîne de caractères dans cet exemple utilisent point et non une virgule. Il est possible d'utiliser une virgule, mais alors la conversion pourrait provoquer une erreur. En effet, si l'ordinateur est configuré – par exemple – pour la France, la notation des nombres réels se fait bien avec une virgule et la conversion réussira.

3.1.4 Les tableaux de données

Un tableau C# est un objet permettant de rassembler sous un même identificateur des données de même type. Sa déclaration est la suivante :

```
Type[] tableau[]=new Type[n]
```

n est le nombre de données que peut contenir le tableau. La syntaxe `Tableau[i]` désigne la donnée n° *i* où *i* appartient à l'intervalle `[0,n-1]`. Toute référence à la donnée `Tableau[i]` où *i* n'appartient pas à l'intervalle `[0,n-1]` provoquera une exception. Un tableau peut être initialisé en même temps que déclaré :

```
int[] entiers=new int[] {0,10,20,30};
```

ou plus simplement :

```
int[] entiers={0,10,20,30};
```

Les tableaux ont une propriété **Length** qui est le nombre d'éléments du tableau.

Un **tableau à deux dimensions** pourra être déclaré comme suit :

```
Type[,] tableau=new Type[n,m];
```

où *n* est le nombre de lignes, *m* le nombre de colonnes. La syntaxe `Tableau[i,j]` désigne l'élément *j* de la ligne *i* de *tableau*. Le tableau à deux dimensions peut lui aussi être initialisé en même temps qu'il est déclaré :

```
double[,] réels=new double[,] { {0.5, 1.7}, {8.4, -6}};
```

ou plus simplement :

```
double[,] réels={ {0.5, 1.7}, {8.4, -6}};
```

Le nombre d'éléments dans chacune des dimensions peut être obtenue par la méthode **GetLength(i)** où *i*=0 représente la dimension correspondant au 1er indice, *i*=1 la dimension correspondant au 2ième indice, ...

Le nombre total de dimensions est obtenu avec la propriété **Rank**, le nombre total d'éléments avec la propriété **Length**.

Un **tableau de tableaux** est déclaré comme suit : `Type[][] tableau=new Type[n][];`

La déclaration ci-dessus crée un tableau de *n* lignes. Chaque élément *tableau[i]* est une référence de tableau à une dimension. Ces références *tableau[i]* ne sont pas initialisées lors de la déclaration ci-dessus. Elles ont pour valeur la référence *null*.

L'exemple ci-dessous illustre la création d'un tableau de tableaux :

```
1. // un tableau de tableaux
2. string[][] noms = new string[3][];
3. for (int i = 0; i < noms.Length; i++)
4.     noms[i] = new string[i + 1];
5. // initialisation
6. for (int i = 0; i < noms.Length; i++)
7.     for (int j = 0; j < noms[i].Length; j++)
8.         noms[i][j] = "nom" + i + j;
```

- ligne 2 : un tableau *noms* de 3 éléments de type *string[][]*. Chaque élément est un pointeur de tableau (une référence d'objet) dont les éléments sont de type *string[]*.
- lignes 3-5 : les 3 éléments du tableau *noms* sont initialisés. Chacun "pointe" désormais sur un tableau d'éléments de type *string[]*. *noms[i][j]* est l'élément *j* du tableau de type *string []* référencé par *noms[i]*.
- ligne 9 : initialisation de l'élément *noms[i][j]* à l'intérieur d'une double boucle. Ici *noms[i]* est un tableau de *i+1* éléments. Comme *noms[i]* est un tableau, *noms[i].Length* est son nombre d'éléments.

Voici un exemple regroupant les trois types de tableaux que nous venons de présenter :

```
using System;
namespace Chap1 {
// tableaux
using System;
// classe de test
public class P02 {
    public static void Main() {
        // un tableau à 1 dimension initialisé
        int[] entiers = new int[] { 0, 10, 20, 30 };
        for (int i = 0; i < entiers.Length; i++) {
            Console.Out.WriteLine("entiers[{0}]= {1}", i, entiers[i]);
        }
        // un tableau à 2 dimensions initialisé
        double[,] réels = new double[,] { { 0.5, 1.7 }, { 8.4, -6 } };
        for (int i = 0; i < réels.GetLength(0); i++) {
            for (int j = 0; j < réels.GetLength(1); j++) {
                Console.Out.WriteLine("réels[{0},{1}]= {2}", i, j, réels[i, j]);
            }
        }
        // un tableau de tableaux
        string[][] noms = new string[3][];
        for (int i = 0; i < noms.Length; i++) {
            noms[i] = new string[i + 1];
        }
        // initialisation
        for (int i = 0; i < noms.Length; i++) {
            for (int j = 0; j < noms[i].Length; j++) {
                noms[i][j] = "nom" + i + j;
            }
        }
        // affichage
        for (int i = 0; i < noms.Length; i++) {
            for (int j = 0; j < noms[i].Length; j++) {
                Console.Out.WriteLine("noms[{0}][{1}]= {2}", i, j, noms[i][j]);
            }
        }
    }
}
}
```

A l'exécution, nous obtenons les résultats suivants :

```
entiers[0]=0
entiers[1]=10
entiers[2]=20
entiers[3]=30
réels[0,0]=0,5
réels[0,1]=1,7
réels[1,0]=8,4
```

```
réels[1,1]=-6
noms[0][0]=nom00
noms[1][0]=nom10
noms[1][1]=nom11
noms[2][0]=nom20
noms[2][1]=nom21
noms[2][2]=nom22
```

3.2 Les instructions élémentaires de C#

3.2.1 Affectation de la valeur d'une expression à une variable

En C#, on affecte une valeur à une variable au moyen de l'opérateur « = » :

```
variable = expression ;
```

Lors de l'exécution, la valeur de *expression* est évaluée et son résultat est stocké dans *variable*. Cela n'est naturellement possible que si le résultat produit par *expression* est de même type que *variable*. Dans certains cas, une conversion implicite est possible (par exemple lorsque *expression* est un entier et *variable* est réel), mais bien souvent il faudra avoir recours à une conversion de type explicite (voir 3.2.11)

Les chapitres qui suivent présentent les diverses règles qui régissent l'évaluation d'une expression.

3.2.2 Expression arithmétique

Les opérateurs des expressions arithmétiques sont les suivants :

- + addition
- soustraction
- * multiplication
- / division : le résultat est le quotient exact si l'un au moins des opérandes est réel. Si les deux opérandes sont entiers le résultat est **le quotient entier**. Ainsi 5/2 -> 2 et 5.0/2 -> 2.5.
- % division : le résultat est le reste quelque soit la nature des opérandes, le quotient étant lui entier. C'est donc l'opération **modulo**.

De plus, il existe diverses fonctions mathématiques qui retournent des valeurs numériques. sont définies dans une classe C# appelée **Math**. Lorsqu'on les utilise, il faut les préfixer avec le nom de la classe où elles sont définies. Ainsi on écrira :

```
double x, y=4;
x=Math.Sqrt(y) ;
```

La définition complète de la classe *Math* est la suivante :

Méthode	Description
Abs	Retourne la valeur absolue d'un nombre spécifié.
Acos	Retourne l'angle dont le cosinus est le nombre spécifié.
Asin	Retourne l'angle dont le sinus est le nombre spécifié.
Atan	Retourne l'angle dont la tangente est le nombre spécifié.
Atan2	Retourne l'angle dont la tangente est le quotient de deux nombres spécifiés.
BigMul	Génère le produit intégral de deux nombres 32 bits.
Ceiling	Retourne le plus petit entier supérieur ou égal au nombre spécifié.
Cos	Retourne le cosinus de l'angle spécifié.
Cosh	Retourne le cosinus hyperbolique de l'angle spécifié.
DivRem	Retourne le quotient de deux nombres, ainsi que le reste (paramètre de sortie).
Exp	Retourne e élevé à la puissance spécifiée.
Floor	Retourne le plus grand entier inférieur ou égal au nombre spécifié.
IEEERemainder	Retourne le reste de la division d'un nombre spécifié par un autre.
Log	Retourne le logarithme d'un nombre spécifié.
Log10	Retourne le logarithme de base 10 d'un nombre spécifié.
Max	Retourne le plus grand de deux nombres spécifiés.
Min	Retourne le plus petit de deux nombres.
Pow	Retourne un nombre spécifié élevé à la puissance spécifiée.
Round	Arrondit une valeur à l'entier le plus proche ou au nombre spécifié de décimales.
Sign	Retourne une valeur indiquant le signe d'un nombre.
Sin	Retourne le sinus de l'angle spécifié.
Sinh	Retourne le sinus hyperbolique de l'angle spécifié.
Sqrt	Retourne la racine carrée d'un nombre spécifié.
Tan	Retourne la tangente de l'angle spécifié.

Tableau 4 Classe Math

3.2.3 Priorités dans l'évaluation des expressions arithmétiques

La priorité des opérateurs lors de l'évaluation d'une expression arithmétique est la suivante (du plus prioritaire au moins prioritaire) :

[fonctions], [()], [*, /, %], [+, -]

Les opérateurs d'un même bloc [] ont même priorité.

3.2.4 Expressions relationnelles

Les opérateurs sont les suivants :

<, <=, ==, !=, >, >=

Leur priorités :

1. >, >=, <, <=

2. ==, !=

Le résultat d'une expression relationnelle est le booléen *false* si l'expression est fautive, *true* sinon.

```
bool fin;
int x=...;
fin=x>4 ;
```

Comparaison de deux caractères

Soient deux caractères C1 et C2. Il est possible de les comparer avec les opérateurs

<, <=, ==, !=, >, >=

Ce sont alors leurs codes Unicode, qui sont des nombres, qui sont comparés. Selon l'ordre Unicode on a les relations suivantes :

espace < .. < '0' < '1' < .. < '9' < .. < 'A' < 'B' < .. < 'Z' < .. < 'a' < 'b' < .. < 'z'

Comparaison de deux chaînes de caractères

Elles sont comparées caractère par caractère. La première inégalité rencontrée entre deux caractères induit une inégalité de même sens sur les chaînes.

Exemples :

Soit à comparer les chaînes "Chat" et "Chien"

"Chat" "Chien"

'C' = 'C'

'h' = 'h'

'a' < 'i'

Cette dernière inégalité permet de dire que "Chat" < "Chien".

Soit à comparer les chaînes "Chat" et "Chaton". Il y a égalité tout le temps jusqu'à épuisement de la chaîne "Chat". Dans ce cas, la chaîne épuisée est déclarée la plus "petite". On a donc la relation

"Chat" < "Chaton".

Fonctions de comparaison de deux chaînes

On peut utiliser les opérateurs relationnels == et != pour tester l'égalité ou non de deux chaînes, ou bien la méthode *Equals* de la classe *System.String*. Pour les relations < <= > >=, il faut utiliser la méthode *CompareTo* de la classe *System.String* :

```
using System;
namespace Chap1 {
    class P05 {
        static void Main(string[] args) {
            string chaine1="chat", chaine2="chien";
            int n = chaine1.CompareTo(chaine2);
            bool egal = chaine1.Equals(chaine2);
            Console.WriteLine("i={0}, egal={1}", n, egal);
            Console.WriteLine("chien==chaine1:{0},
                             chien!=chaine2:{1}",
                             "chien"==chaine1,
                             "chien" !=chaine2);
        }
    }
}
```

- Ligne 6, la variable *n* aura la valeur :
 - 0 si les deux chaînes sont égales
 - 1 si chaîne n°1 > chaîne n°2
 - 1 si chaîne n°1 < chaîne n°2
- Ligne 8, la variable *egal* aura la valeur *true* si les deux chaînes sont égales, *false* sinon.
- Ligne 10, on utilise les opérateurs == et != pour vérifier l'égalité ou non de deux chaînes.

Les résultats de l'exécution :

```
i=-1, egal=False
chien==chaine1:False, chien!=chaine2:False
```

3.2.5 Expressions booléennes

Les opérateurs utilisables sont le ET logique (&&), le OU logique (||) et le NON logique (!). Le résultat d'une expression booléenne est un booléen.

Priorités des opérateurs :

1. !
2. &&
3. ||

```
double x = 3.5;
bool valide = x > 2 && x < 4;
```

Il n'est pas nécessaire d'avoir recours à des parenthèses car les opérateurs relationnels **ont priorité** sur les opérateurs && et ||.

3.2.6 Traitement de bits

Soient *i* et *j* deux entiers.

<i>i</i> << <i>n</i>	décale <i>i</i> de <i>n</i> bits sur la gauche. Les bits entrants sont des zéros
<i>i</i> >> <i>n</i>	décale <i>i</i> de <i>n</i> bits sur la droite. Si <i>i</i> est un entier signé (signed char, int, long) le bit de signe est préservé
<i>i</i> & <i>j</i>	fait le ET logique de <i>i</i> et <i>j</i> bit à bit
<i>i</i> <i>j</i>	fait le OU logique de <i>i</i> et <i>j</i> bit à bit
~ <i>i</i>	complémente <i>i</i> à 1
<i>i</i> ^ <i>j</i>	fait le OU EXCLUSIF de <i>i</i> et <i>j</i>

Soit le code suivant :

```
short i = 100, j = -13;
ushort k = 0xF123;
Console.WriteLine("i=0x{0:x4}, j=0x{1:x4}, k=0x{2:x4}", i, j, k);
Console.WriteLine("i<<4=0x{0:x4}, "+
    "i>>4=0x{1:x4}, "+
    "k>>4=0x{2:x4}, "+
    "i&j=0x{3:x4}, "+
    "i|j=0x{4:x4}, " +
    "~i=0x{5:x4}, " +
    "j<<2=0x{6:x4}, " +
    "j>>2=0x{7:x4}",
    i << 4,
    i >> 4,
    k >> 4,
    (short)(i & j),
    (short)(i | j),
    (short)(~i),
    (short)(j << 2),
    (short)(j >> 2));
```

- Le format {0:x4} affiche le paramètre n° 0 au format hexadécimal (x) avec 4 caractères (4).

Les résultats de l'exécution sont les suivants :

```
i=0x0064, j=0xfff3, k=0xf123
i<<4=0x0640,
i>>4=0x0006, k>>4=0x0f12, i&j=0x0060, i|j=0xfff7, ~i=0xff9b, j<<2=0xffcc, j>>2=0xfffc
```

3.2.7 Combinaison d'opérateurs

$a=a+b$ peut s'écrire $a+=b$

$a=a-b$ peut s'écrire $a-=b$

Il en est de même avec les opérateurs $/$, $\%$, $*$, $<<$, $>>$, $\&$, $|$, \wedge . Ainsi $a=a/2$; peut s'écrire $a/=2$;

3.2.8 Opérateurs d'incrément et de décrémentation

La notation *variable++* signifie *variable=variable+1* ou encore *variable+=1* La notation *variable--* signifie *variable=variable-1* ou encore *variable-=1*.

La notation est également possible mais peut donner un résultat différent selon les cas

Exemple : le code

```
int i = 1;
Console.WriteLine("Première fois: {0}", i++);
Console.WriteLine("Deuxième fois: {0}", ++i);
Console.WriteLine("Troisième fois: {0}", ++i);
Console.WriteLine("Quatrième fois: {0}", i++);
```

Donne le résultat :

```
Première fois: 1
Deuxième fois: 3
Troisième fois: 4
Quatrième fois: 4
```

Lors de la première écriture, on affiche la valeur (1) puis on incrémente i. Lors de la deuxième, on incrémente d'abord et on affiche ensuite (donc 3).

3.2.9 L'opérateur ternaire « ? »

L'expression

```
expr_cond ? expr1:expr2
```

est évaluée de la façon suivante :

1. l'expression *expr_cond* est évaluée. C'est une expression conditionnelle à valeur *vrai* ou *faux*
2. Si elle est vraie, la valeur de l'expression est celle de *expr1* et *expr2* n'est pas évaluée.

3. Si elle est fausse, c'est l'inverse qui se produit : la valeur de l'expression est celle de *expr2* et *expr1* n'est pas évaluée.

L'opération *i=(j>4 ? j+1:j-1);* affectera à la variable *i* : *j+1* si *j>4*, *j-1* sinon. C'est la même chose que d'écrire *if(j>4) i=j+1; else i=j-1;* mais c'est plus concis.

3.2.10 Priorité générale des opérateurs

() [] fonction	gd
! ~ ++ --	dg
new (type) opérateurs cast */%	dg
+ -	gd
<< >>	gd
< <= > >= instanceof	gd
== !=	gd
&	gd
^	gd
	gd
&&	gd
	gd
?:	dg
= += -= etc. .	dg

gd indique qu'a priorité égale, c'est la priorité gauche-droite qui est observée. Cela signifie que lorsque dans une expression, l'on a des opérateurs de même priorité, c'est l'opérateur le plus à gauche dans l'expression qui est évalué en premier. **dg** indique une priorité droite-gauche.

3.2.11 Les conversions de type

Il est possible, dans une expression, de changer momentanément le codage d'une valeur. On appelle cela changer le type d'une donnée ou en anglais **type casting**. La syntaxe du changement du type d'une valeur dans une expression est la suivante:

(type) valeur

La valeur prend alors le type indiqué. Cela entraîne un changement de codage de la valeur.

```
using System;
namespace Chap1 {
    class P06 {
        static void Main(string[] args) {
            int i = 3, j = 4;
            float f1=i/j;
            float f2=(float)i/j;
            Console.WriteLine("f1={0}, f2={1}", f1, f2);
        }
    }
}
```

- ligne7, *f1* aura la valeur 0.0. La division $\frac{3}{4}$ est une division entière puisque les deux opérandes sont de type *int*.
- ligne8, *(float)i* est la valeur de *i* transformée en *float*. Maintenant, on a une division entre un réel de type *float* et un entier de type *int*. C'est la division entre nombres réels qui est alors faite. La valeur de *j* sera elle également transformée en type *float*, puis la division des deux réels sera faite. *f2* aura alors la valeur 0,75. Voici les résultats de l'exécution :

```
f1=0, f2=0,75
```

Dans l'opération *(float)i* :

- *i* est une valeur codée de façon exacte sur 2 octets
- *(float) i* est la même valeur codée de façon approchée en réel sur 4 octets

Il y a donc transcodage de la valeur de *i*. Ce transcodage n'a lieu que le temps d'un calcul, la variable *i* conservant toujours son type *int*.

3.2.12 Mise en Forme de Types Numériques

Lorsqu'il s'agit d'afficher une valeur numérique (popup, champ texte, écriture dans un fichier), il faut procéder à sa mise en forme. Cela se fait au moyen d'une **chaîne de format numérique standard**, qui se présente sous la forme Axx, où A est un caractère alphabétique appelé spécificateur de format et xx un entier facultatif appelé spécificateur de précision. Le spécificateur de précision est compris entre 0 et 99 ; il affecte le nombre de chiffres dans le résultat. Toute chaîne de format numérique contenant plusieurs caractères alphabétiques, y compris un espace blanc, est interprétée comme une chaîne de format numérique personnalisée.

Voir annexe pour les détails sur les spécificateurs standards.

Les chaînes de format numérique standard sont prises en charge par nombre de méthodes :

- ToString
- Write et WriteLine des classes StreamWriter et Console,
- String.Format
- StringBuilder.AppendFormat

Leur utilisation est toujours la même : on donne tout d'abord une chaîne de caractère qui contient des champs délimités par les caractères « { } ». Le champ contient un numéro d'ordre, qui peut être suivi de « : » et de la chaîne de format. Suivent les valeurs numériques à mettre dans les champs, séparées par des virgules.

Exemple :

Le code

```
Console.WriteLine("La somme de {0} et {1} est égale à {2}", 3, 7, 3+7);
Console.WriteLine("{0} divisé par {1} est égal à {2:F4}", 3, 7, 3.0/7.0);
Console.WriteLine("{2} est plus grand que {1}, qui est plus grand que {0}", 1, 2, 3);
int Nombre = 7654;
Console.WriteLine("Décimal: {0}, Hexadécimal: {0:X}", Nombre);
```

Produit le résultat :

```
La somme de 3 et 7 est égale à 10
3 divisé par 7 est égal à 0.4286
3 est plus grand que 2, qui est plus grand que 1
Décimal: 7654, Hexadécimal: 1DE6
```

3.2.13 Mise en Forme de Dates

Sur le même modèle, on va pouvoir mettre en forme les données de type DateTime, mais au moyen de spécificateurs de date.

Exemple :

Le code

```
DateTime Doom = new DateTime(2012, 12, 21, 13, 20, 0);
Console.WriteLine("L'\apocalypse est pour le {0:d}", Doom);
Console.WriteLine("L'\apocalypse est pour le {0:D}", Doom);
Console.WriteLine("L'\apocalypse est pour le {0:f}", Doom);
Console.WriteLine("L'\apocalypse est pour le {0:dd MMM yy}", Doom);
```

Produit le résultat :

```
L'\apocalypse est pour le 21.12.2012
L'\apocalypse est pour le vendredi 21 décembre 2012
L'\apocalypse est pour le vendredi 21 décembre 2012 13:20
L'\apocalypse est pour le 21 déc. 12
```

3.3 Les instructions de contrôle du déroulement du programme

3.3.1 Arrêt

La méthode *Exit* définie dans la classe *Environment* permet d'arrêter l'exécution d'un programme.

Exit provoque la fin du processus en cours et rend la main au processus appelant. La valeur de *status* peut être utilisée par celui-ci. Sous DOS, cette variable *status* est rendue dans la variable système **ERRORLEVEL** dont la valeur peut être testée dans un fichier batch. Sous Unix, avec l'interpréteur de commandes Shell Bourne, c'est la variable **\$?** qui récupère la valeur de *status*.

```
Environment.Exit(0);
```

arrêtera l'exécution du programme avec une valeur d'état à 0.

3.3.2 Structure de choix simple (if)

Syntaxe :

```
if (condition)
    {actions_condition_vraie;}
else
    {actions_condition_fausse;}
```

notes:

- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- les accolades ne sont pas terminées par point-virgule.
- les accolades ne sont nécessaires que s'il y a plus d'une action.
- la clause *else* peut être absente.
- il n'y a pas de clause *then*.

Exemples

```
if (x>0)
{
    nx=nx+1;
    sx=sx+x;
}
else
    dx=dx-x;

if (dx<0) Environment.Exit(0);

if (sx>MAX_SX)
{
    Console.WriteLine("On est au max");
    Environment.Exit(0);
}
```

On peut imbriquer les structures de choix. Syntaxe :

```
if (condition1)
    if (condition2)
        {...}
    else
        {...}
else
    {...}
```

Il faut toutefois être vigilant lorsqu'on imbrique des structures ; le problème suivant se pose parfois :

```
using System;
namespace Chap1 {
    class P07 {
        static void Main(string[] args) {
            int n = 5;
            if (n > 1)
                if (n > 6) Console.Out.WriteLine(">6");
            else
                Console.Out.WriteLine("<=1");
        }
    }
}
```

Dans cet exemple, le résultat ne sera pas celui dont on devine l'intention par la mise en page. La règle est qu'un *else* se rapporte toujours au *if* le plus proche, donc ici, il se rapporte au *if(n>6)* et non pas au *if(n>1)*.

Nous voulions mettre un *else* au *if(n>1)* et pas de *else* au *if(n>6)*. A cause de la remarque précédente, nous sommes obligés de mettre des accolades au *if(n>1) {...} else ...*

```
if (n > 1)
{
    if (n > 6) Console.WriteLine(">6");
}
else
    Console.WriteLine("<=1");
```

3.3.3 Structure de cas (switch)

Syntaxe :

```
switch(expression)
{
    case v1:
        actions_1;
        break;
    case v2:
    case v3 :
        actions_2ou3;
        break;
    default:
        actions_sinon;
        break;
}
```

Notes :

- La valeur de l'expression de contrôle du *switch* peut être un entier, un caractère, une chaîne de caractères
- L'expression de contrôle est entourée de parenthèses.
- La clause *default* peut être absente.
- Les valeurs v_i sont des valeurs possibles de l'expression. Si l'expression a pour valeur v_i , les actions derrière la clause *case* v_i sont exécutées.
- L'instruction *break* fait sortir de la structure de cas.
- Chaque bloc d'instructions lié à une valeur v_i doit se terminer par une instruction de branchement (*break*, *return*, ...) sinon le compilateur signale une erreur.

Exemple

```
ConsoleKeyInfo K = Console.ReadKey();
switch (K.KeyChar)
{
    case 'a':
        Console.WriteLine("Choix a");
        break;
    case 'b':
    case 'B':
        Console.WriteLine("Choix b");
        break;
    case 'c':
        Console.WriteLine("Choix c");
        break;
    default:
        Console.WriteLine("Choix incorrect");
        break;
}
```

3.3.4 Boucle While (tant que)

Syntaxe :

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

On exécute les instructions entre {} tant que la condition est vérifiée. La boucle peut ne jamais être exécutée.

Notes:

- La condition est entourée de parenthèses.
- Chaque instruction est terminée par point-virgule.
- L'accolade n'est nécessaire que s'il y a plus d'une instruction.
- L'accolade n'est pas suivie de point-virgule.

Exemple :

```
StreamReader Fichier = new StreamReader(@"C:\Log.txt");
while (!Fichier.EndOfStream)
    Console.WriteLine (Fichier.ReadLine());
```

3.3.5 Boucle Do While (jusqu'à ce que)

Syntaxe :

```
Do { instructions; } while (condition);
```

On exécute les instructions entre {} jusqu'à ce que la condition devienne fausse. La boucle est exécutée au moins une fois.

Notes:

- La condition est entourée de parenthèses.
- Chaque instruction est terminée par point-virgule.
- L'accolade n'est nécessaire que s'il y a plus d'une instruction.
- L'accolade n'est pas suivie de point-virgule.

Exemple :

```
char ch ;
int x;
do
{
    Console.WriteLine("Tapez une lettre entre a et z :");
    x = Console.Read();
    ch = Convert.ToChar(x);
}
while ((ch < 'a') || ((ch > 'z')) ;
```

3.3.6 Boucle For (pour)**Syntaxe :**

```
for (instructions_départ; condition; instructions_fin_boucle)
{
    instructions ;
}
```

On boucle tant que la condition est vraie (évaluée avant chaque tour de boucle). *Instructions_départ* sont effectuées avant d'entrer dans la boucle pour la première fois. *Instructions_fin_boucle* sont exécutées après chaque tour de boucle.

Notes :

- Il est possible de mettre plusieurs instructions de départ et de fin de boucle. Dans ce cas, les différentes instructions sont séparées par des virgules.

Exemple :

Les fragments de code suivants calculent tous la somme des 10 premiers nombres entiers.

```
int i, somme, n=10;
for(i=1,somme=0;i<=n;i=i+1)
    somme = somme + i;

for(i=1,somme=0;i<=n;somme=somme+i,i=i+1);

i=1;somme=0;
while(i<=n){somme+=i;i++;}

i = 1; somme = 0;
do somme += i++;
while (i <= n);
```

3.3.7 Boucle Foreach (pour chaque)

Lorsque l'on veut utiliser une structure de boucle pour parcourir une collection, la syntaxe *foreach* est plus compacte que *for*.

Syntaxe :

```
foreach (Type variable in collection)
{
    instructions;
}
```

Notes :

- *Collection* est une collection d'objets énumérable. La collection d'objets énumérable que nous connaissons déjà est le tableau
- *Type* est le type des objets de la collection. Pour un tableau, ce serait le type des éléments du tableau
- *Variable* est une variable locale à la boucle qui va prendre successivement pour valeur chaque élément de la collection.

Exemple :

```
string[] amis = { "paul", "hélène", "jacques", "sylvie" };
foreach (string nom in amis)
{
    Console.WriteLine(nom);
}
```

Ce code donnera le résultat :

```
paul  
hélène  
jacques  
sylvie
```

3.3.8 La Gestion des Exceptions

De nombreuses fonctions C# sont susceptibles de générer des exceptions, c'est à dire des erreurs. C'est alors au programmeur de les gérer dans le but d'obtenir des programmes plus résistants aux erreurs : il faut toujours éviter le "plantage" d'une application.

La gestion d'une exception se au moyen de la structure « try/catch ».

Syntaxe :

```
try  
{  
    instructions ; // code susceptible de générer une exception  
}  
catch (Exception e)  
{  
    instructions ; // traiter l'exception e  
}  
instruction suivante
```

Si les instructions du bloc *try* ne génèrent pas d'exception, on passe alors à *instruction suivante*, sinon on passe dans le corps de la clause *catch* puis à *instruction suivante*.

Notes :

- **e** est un objet de type `Exception` ou dérivé. On peut être plus précis en utilisant des types tels que `IndexOutOfRangeException`, `FormatException`, `SystemException`, etc... : il existe plusieurs types d'exceptions. En écrivant `catch (Exception e)`, on indique qu'on veut gérer toutes les types d'exceptions. Si le code de la clause *try* est susceptible de générer plusieurs types d'exceptions, on peut vouloir être plus précis en gérant l'exception avec plusieurs clauses *catch* :

```
try
{
    instructions ; // code susceptible de générer les exceptions
}
catch ( IndexOutOfRangeException e1)
{
    instructions ; // traiter l'exception e1
}
catch ( FormatException e2)
{
    instructions ; // traiter l'exception e2
}
instruction suivante
```

- On peut ajouter aux clauses try/catch, une clause **finally** :

```
try
{
    instructions ; // code susceptible de générer les exceptions
}
catch ( IndexOutOfRangeException e1)
{
    instructions ; // traiter l'exception e1
}
finally
{
    instructions ;
}
instruction suivante
```

Qu'il y ait exception ou pas, le code de la clause *finally* sera toujours exécuté.

- Dans la clause *catch*, on peut ne pas vouloir utiliser l'objet *Exception* disponible. Au lieu d'écrire *catch (Exception e){..}*, on écrit alors *catch(Exception){...}* ou plus simplement *catch {...}*.
- La classe *Exception* a une propriété **Message** qui est un texte détaillant l'erreur qui s'est produite. Ainsi si on veut afficher celui-ci, on écrira :

```
catch (Exception ex)
{
    Console.WriteLine("L'erreur suivante s'est produite : {0}",ex.Message);
}
```

- La classe *Exception* a une méthode **ToString** qui rend une chaîne de caractères indiquant le type de l'exception ainsi que la valeur de la propriété *Message*. On pourra ainsi écrire :

```
catch (Exception ex)
{
    Console.WriteLine("L'erreur suivante s'est produite : {0}", ex.ToString());
}
```

ou encore :

```
catch (Exception ex)
{
    Console.WriteLine("L'erreur suivante s'est produite : {0}", ex);
}
```

Le compilateur va automatiquement attribuer au paramètre {0}, la valeur *ex.ToString()*.

Exemples :

Le premier exemple montre une exception générée par l'utilisation d'un élément de tableau inexistant :

```
using System;
namespace Chap1 {
    class P08 {
        static void Main(string[] args) {
            // déclaration & initialisation d'un tableau
            int[] tab={0,1,2,3};
            int i;

            // affichage tableau avec un for
            for (i = 0; i < tab.Length; i++)
                Console.WriteLine("tab[{0}]=1", i, tab[i]);

            // affichage tableau avec un foreach
            foreach (int élmnt in tab)
                Console.WriteLine(élmnt);
        }
    }
}
```

```
// génération d'une exception
try
{
    tab[100] = 6;
}
catch (Exception e)
{
    Console.Error.WriteLine("L'erreur suivante s'est produite : " + e);
    return;
} // try-catch
finally
{
    Console.WriteLine("finally ...");
}
}
}
```

La ligne `tab[100] = 6` va générer une exception parce que le tableau `tab` n'a pas d'élément n° 100.
L'exécution du programme donne les résultats suivants :

```
tab[0]=0
tab[1]=1
tab[2]=2
tab[3]=3
0
1
2
3
L'erreur suivante s'est produite : System.IndexOutOfRangeException: L'index se
trouve en dehors des limites du tableau.
à Chap1.P08.Main(String[] args) dans Program.cs:ligne x
finally...
```

- La clause *finally* du code a été exécutée, bien qu'il y ait une instruction *return* dans le bloc *catch*. On retiendra donc que la clause *finally* est toujours exécutée.

Voici un autre exemple où on gère l'exception provoquée par l'affectation d'une chaîne de caractères à une variable de type entier lorsque la chaîne ne représente pas un nombre entier :

```

using System;
namespace Chap1 {
    class P08 {
        static void Main(string[] args) {
            // exemple 2
            // On demande le nom
            Console.Write("Nom : ");
            // lecture réponse
            string nom = Console.ReadLine();
            // on demande l'âge
            int age=0;
            bool ageOK = false;
            while (!ageOK)
            {
                // question
                Console.Write("âge : ");
                // lecture-vérification réponse
                try
                {
                    age = int.Parse(Console.ReadLine());
                    ageOK = age>=1;
                }
                catch
                {} //try-catch
                if (!ageOK)
                {
                    Console.WriteLine("Age incorrect, recommencez...");
                }
            } //while
            // affichage final
            Console.WriteLine("Vous vous appelez {0} et vous avez {1} an(s)", nom, age);
        }
    }
}

```

Notes :

- La ligne tapée au clavier est transformée en nombre entier par la méthode *int.Parse*. Cette méthode lance une exception si la conversion n'est pas possible. C'est pourquoi, l'opération a été placée dans un try / catch.
- Si une exception est lancée, on va dans le catch où rien n'est fait. Ainsi, le booléen *ageOK* positionné à *false*, va-t-il rester à *false*.
- La ligne *ageOK = age>=1* vérifie que l'entier obtenu est bien supérieur ou égal à 1. Si on arrive à cette ligne, c'est que la conversion string -> int a réussi.

Quelques résultats d'exécution :

```
Nom : dupont
âge : 23
Vous vous appelez dupont et vous avez 23 an(s)
Nom : durand
âge:x
Age incorrect, recommencez...
âge : -4
Age incorrect, recommencez...
âge : 12
Vous vous appelez durand et vous avez 12 an(s)
```

3.4 Arguments du programme principal

La fonction principale *Main* peut admettre comme paramètre un tableau de chaînes : *String[]* (ou *string[]*). Ce tableau contient les arguments de la ligne de commande utilisée pour lancer l'application. Ainsi si on lance le programme Prog avec la commande (Dos) suivante :

Prog arg0 arg1 ... argn

et si la fonction *Main* est déclarée comme suit :

```
public static void Main(string[] args)
```

on aura *args[0]="arg0", args[1]="arg1" ...* Voici un exemple :

```
using System;
namespace Chap1 {
    class P10 {
        static void Main(string[] args) {
            // on liste les paramètres reçus
            for (int i = 0; i < args.Length; i++) MessageBox.Show(args[i]);
        }
    }
}
```

Pour passer des arguments au code exécuté en mode debug, on procèdera comme suit :

- Menu Projet / Propriétés
- Onglet Déboguer
- Mettre les arguments dans la boîte « Arguments de la ligne de commande »

Pour passer des arguments à l'application finale (release), on les spécifiera dans les propriétés du raccourci utilisé pour lancer votre application.

Si vous associez un type de fichier (extension) à votre application, vous pouvez démarrer votre application en double-cliquant un fichier ayant l'extension choisie. Le nom du fichier sera automatiquement passé en paramètre.

On notera que la signature

```
public static void Main()
```

est valide si la fonction Main n'attend pas de paramètres.

3.5 Les énumérations

Une énumération est un type de données dont le domaine de valeurs est un ensemble de constantes entières. Considérons un programme qui sert à gérer des mentions à un examen. Il y en aurait cinq : *Passable, AssezBien, Bien, TrèsBien, Excellent*.

On pourrait alors définir une énumération pour ces cinq constantes :

```
enum Mentions { Passable, AssezBien, Bien, TrèsBien, Excellent };
```

De façon interne, ces cinq constantes sont codées par des entiers consécutifs commençant par 0 pour la première constante, 1 pour la suivante, etc... Une variable peut être déclarée comme prenant ces valeurs dans l'énumération :

```
// une variable qui prend ses valeurs dans l'énumération Mentions 2.  
Mentions maMention = Mentions.Passable;
```

On peut comparer une variable aux différentes valeurs possibles de l'énumération :

```
Switch (maMention)  
{  
    case Mentions.Excellent: Console.WriteLine("Impressionnant !"); break ;  
    case Mentions.TrèsBien : Console.WriteLine("Bravo"); break ;  
    default: Console.WriteLine("Allez, un petit effort..."); break ;  
}
```

On peut obtenir toutes les valeurs de l'énumération :

```
// liste des mentions sous forme de chaînes  
foreach (Mentions m in Enum.GetValues(maMention.GetType())) {  
    Console.WriteLine(m);  
}
```

Et si l'on veut afficher le nom de la constante :

```
Console.WriteLine(maMention.ToString());
```

3.6 Les Collections

L'espace de nom (ou librairie) `System.Collections.Generic` offre les moyens de gérer des collections d'objets.

Une collection ressemble à un tableau à une dimension, mais avec plusieurs méthodes qui implémentent des opérations communément utilisées sur de ensembles de données : ajout, suppression, recherche, tri, fusion, ...

Pour utiliser une collection, il faut commencer par déclarer notre intention de faire usage de ce namespace:

```
using System.Collections.Generic;
```

Ensuite, il faut déclarer et initialiser la collection en précisant le type de données qu'elle va contenir. Il peut s'agir d'un type scalaire (entier, réel, caractère, chaîne) ou de type complexe (classe). La syntaxe est :

```
List<type> ma_collection ;  
ma_collection = new List<type>() ;
```

La première ligne ne fait que déclarer un objet *ma_collection*. La seconde l'initialise ; à la suite de cette instruction elle est toujours vide, mais un emplacement mémoire lui a été attribué.

Les attributs principaux d'une collection sont:

Méthode	Description
Count	Le nombre d'objets qu'il y a dans la collection

Les méthodes principales applicables à la collection sont les suivantes :

Méthode	Description
Add(<i>objet</i>)	Ajoute un objet à la fin de la collection
AddRange(<i>collection</i>)	Ajoute plusieurs objets (sous la forme d'une collection) à la fin de la collection
Clear()	Supprime tous les éléments de la collection
Contains(<i>objet</i>)	Retourne vrai si la collection contient l'objet donné. Attention : ce qui est recherché dans la collection est une référence sur le même objet que celui passé, pas un objet ayant les mêmes attributs que celui-ci
CopyTo(<i>Tableau</i> , <i>Index</i>)	Copie le contenu de la collection dans un simple tableau. Le premier élément copié de la collection se retrouvera à l'index donné.
Insert(<i>pos</i> , <i>objet</i>)	Insère l'objet donné à la position pos
RemoveAt(<i>pos</i>)	Supprime l'objet situé à la position pos
Sort(<i>critère</i>)	Trie la collection en fonction d'un critère donné

Prenons comme exemple un jeu de carte. Nous définissons une classe « carte » comme suit :

```
enum CouleurCarte { Coeur, Carreau, Trèfle, Pique } ;

class Carte
{
    public string ValeurTexte; // La valeur de la carte sous forme de texte

    public Carte(int Val, CouleurCarte CouL) // Constructeur
    {
        switch (Val)
        {
            case 1: ValeurTexte = "As"; break;
            case 2: ValeurTexte = "Deux"; break;
            case 3: ValeurTexte = "Trois"; break;
            case 4: ValeurTexte = "Quatre"; break;
            case 5: ValeurTexte = "Cinq"; break;
            case 6: ValeurTexte = "Six"; break;
            case 7: ValeurTexte = "Sept"; break;
            case 8: ValeurTexte = "Huit"; break;
            case 9: ValeurTexte = "Neuf"; break;
            case 10: ValeurTexte = "Dix"; break;
            case 11: ValeurTexte = "Valet"; break;
            case 12: ValeurTexte = "Dame"; break;
            case 13: ValeurTexte = "Roi"; break;
            default: ValeurTexte = "???"; break;
        }
        ValeurTexte += (" de " + CouL.ToString());
    }
}
```

Nous pouvons maintenant déclarer deux collections :

```
List<Carte> Paquet;
List<Carte> Main;
```

La première représente un paquet de carte, la seconde la main d'un joueur. Ensuite nous les initialisons :

```
Paquet = new List<Carte>();
Main = new List<Carte>();
```

Pour remplir le paquet, on a recours à des boucles:

```
for (int v = 1; v <= 13; v++)
{
    Carte UneCarte = new Carte(v, CouleurCarte.Coeur);
    Paquet.Add(UneCarte);
}
for (int v = 1; v <= 13; v++)
{
    Carte UneCarte = new Carte(v, CouleurCarte.Trèfle);
    Paquet.Add(UneCarte);
}
for (int v = 1; v <= 13; v++)
{
    Carte UneCarte = new Carte(v, CouleurCarte.Carreau);
    Paquet.Add(UneCarte);
}
for (int v = 1; v <= 13; v++)
{
    Carte UneCarte = new Carte(v, CouleurCarte.Pique);
    Paquet.Add(UneCarte);
}
```

Ensuite, on peut retirer par exemple neuf cartes choisie au hasard dans le paquet et les ajouter à la main du joueur:

```
Random H = new Random(); // Objet générateur de nombre aléatoire
for (int i = 0; i < 9; i++)
{
    int C = H.Next(0, Paquet.Count - 1); // Choisir une carte du paquet
    Main.Add(Paquet[C]); // L'ajouter à la main du joueur
    Paquet.RemoveAt(C); // La retirer du paquet
}
```

3.7 Passage de paramètres à une fonction

Nous nous intéressons ici au mode de passage des paramètres d'une fonction. Considérons la fonction statique suivante :

```
private static void ChangeInt(int a)
{
    a = 30;
    Console.WriteLine("Paramètre formel a=" + a);
}
```

Dans la définition de la fonction, *a* est appelé un paramètre formel. Il n'est là que pour les besoins de la définition de la fonction *ChangeInt*. Il aurait tout aussi bien pu s'appeler *b*. Considérons maintenant une utilisation de cette fonction :

```
public static void Main()
{
    int age = 20;
    ChangeInt(age);
    Console.WriteLine("Paramètre effectif age=" + age);
}
```

Ici dans l'instruction de la ligne *ChangeInt(age)*, *age* est le paramètre effectif qui va transmettre sa valeur au paramètre formel *a*. Nous nous intéressons à la façon dont un paramètre formel récupère la valeur d'un paramètre effectif.

3.7.1 Passage par valeur

L'exemple suivant nous montre que les paramètres d'une fonction sont par défaut passés par valeur, c'est à dire que la valeur du paramètre effectif est copiée dans le paramètre formel correspondant. On a deux entités distinctes. Si la fonction modifie le paramètre formel, le paramètre effectif n'est lui en rien modifié.

L'exécution du code *Main()* ci-dessus produit donc le résultat :

```
Paramètre formel a=30
Paramètre effectif age=20
```

La valeur 20 du paramètre effectif *age* a été copiée dans le paramètre formel *a*. Celui-ci a été ensuite modifié dans la fonction, mais le paramètre effectif est lui resté inchangé. Ce mode de passage convient aux paramètres d'entrée d'une fonction.

3.7.2 Passage par référence

Dans un passage par référence, le paramètre effectif et le paramètre formel sont une seule et même entité. Si la fonction modifie le paramètre formel, le paramètre effectif est lui aussi modifié. En C#, ils doivent être tous deux précédés du mot clé **ref** :

Voici un exemple :

```
using System;
namespace Chap1 {
    class P12 {
        public static void Main() {
            // exemple 2
            int age2 = 20;
            ChangeInt2(ref age2);
            Console.WriteLine("Paramètre effectif age2=" + age2);
        }

        private static void ChangeInt2(ref int a2) {
            a2 = 30;
            Console.WriteLine("Paramètre formel a2=" + a2);
        }
    }
}
```

on obtient :

```
Paramètre formel a2=30
Paramètre effectif age2=30
```

Le paramètre effectif a suivi la modification du paramètre formel. Ce mode de passage convient aux paramètres de sortie d'une fonction.

3.7.3 Passage par référence avec le mot clé out

Considérons l'exemple précédent dans lequel la variable *age2* ne serait pas initialisée avant l'appel à la fonction *changeInt* :

```
using System;
namespace Chap1 {
    class P12 {
        public static void Main() {
            // exemple 2
            int age2;
            ChangeInt2(ref age2);
            Console.WriteLine("Paramètre effectif age2=" + age2);
        }

        private static void ChangeInt2(ref int a2) {
            a2 = 30;
            Console.WriteLine("Paramètre formel a2=" + a2);
        }
    }
}
```

Lorsqu'on compile ce programme, on a une erreur :

Use of unassigned local variable 'age2'

On peut contourner l'obstacle en affectant une valeur initiale à *age2*. On peut aussi remplacer le mot clé **ref** par le mot clé **out**. On exprime alors que la paramètre est uniquement un paramètre de sortie et n'a donc pas besoin de valeur initiale :

```
using System;
namespace Chap1 {
    class P12 {
        public static void Main() {
            // exemple 3
            int age3;
            ChangeInt3(out age3);
            Console.WriteLine("Paramètre effectif age3=" + age3);
        }

        private static void ChangeInt3(out int a3) {
            a3 = 30;
            Console.WriteLine("Paramètre formel a3=" + a3);
        }
    }
}
```

Les résultats de l'exécution sont les suivants :

```
Paramètre formel a3=30
Paramètre effectif age3=30
```

3.8 Membres Statiques

Le modificateur **static** déclare un membre d'une classe comme statique, c'est-à-dire qui appartient au type lui-même plutôt qu'à un objet spécifique.

Illustrons au moyen de l'exemple suivant. Soit la classe :

```
public class Personne
{
    public string nom;
    public string prenom;

    public Personne(string n, string p) // Constructeur
    {
        nom = n;
        prenom = p;
    }

    public string GetNom() // Retourne le nom complet
    {
        return prenom + " " + nom;
    }
}
```

L'utilisation de cette classe au moyen du code

```
Personne Perso1 = new Personne("Smith", "John");
Personne Perso2 = new Personne("Dupont", "Gérard");
Console.WriteLine("Perso1: " + Perso1.GetNom());
Console.WriteLine("Perso2: " + Perso2.GetNom());
```

Produit le résultat

```
Perso1: John Smith
Perso2: Gérard Dupont
```

Lors de la création des deux objets, des zones mémoire distinctes ont été allouées pour les noms et prénoms des deux personnes.

Modifions maintenant ce code :

```
public class Personne
{
    public static string nom; // <= Statique !!!
    public string prenom;

    public Personne(string n, string p) // Constructeur
    {
        nom = n;
        prenom = p;
    }

    public string GetNom() // Retourne le nom complet
    {
        return prenom + " " + nom;
    }
}
```

Le même code produit maintenant le résultat

```
Perso1: John Dupont
Perso2: Gérard Dupont
```

En effet, lors du second objet (Perso2) la zone mémoire allouée pour le nom est la même que pour Perso1 ! Ainsi, lorsque le deuxième objet est initialisé, le nom de Smith est remplacé par Dupont.

4 Classes .NET d'usage courant

Nous présentons ici quelques classes de la plate-forme .NET fréquemment utilisées

4.1 String

La classe **System.String** est assimilable au type simple **string**. Mais en plus, elle offre de nombreuses propriétés et méthodes.

En voici quelques-unes

Membre	Description
<code>public int Length</code>	nombre de caractères de la chaîne
<code>public bool EndsWith(string value)</code>	rend vrai si la chaîne se termine par <i>value</i>
<code>public bool StartsWith(string value)</code>	rend vrai si la chaîne commence par <i>value</i>
<code>public bool Contains(string value)</code>	rend vrai si la chaîne contient <i>value</i>
<code>public virtual bool Equals(object obj)</code>	rend vrai si la chaînes est égale à <i>obj</i> - équivalent <code>chaîne==obj</code>
<code>public int IndexOf(string value, int startIndex)</code>	rend la première position dans la chaîne de la chaîne <i>value</i> - la recherche commence à partir du caractère n° <i>startIndex</i>
<code>public int IndexOf(char value, int startIndex)</code>	idem mais pour le caractère <i>value</i>
<code>public string Insert(int startIndex, string value)</code>	insère la chaîne <i>value</i> dans chaîne en position <i>startIndex</i>
<code>public static string Join(string separator, string[] value)</code>	méthode de classe - rend une chaîne de caractères, résultat de la concaténation des valeurs du tableau <i>value</i> avec le séparateur <i>separator</i>
<code>public int LastIndexOf(char value, int startIndex, int count)</code> <code>public int LastIndexOf(string value, int startIndex, int count)</code>	idem <i>indexOf</i> mais rend la dernière position au lieu de la première
<code>public string Replace(char oldChar, char newChar)</code>	rend une chaîne copie de la chaîne courante où le caractère <i>oldChar</i> a été remplacé par le caractère <i>newChar</i>
<code>public string[] Split(char[] separator)</code>	la chaîne est vue comme une suite de champs séparés par les caractères présents dans le tableau <i>separator</i> Le résultat est le tableau de ces champs

Membre	Description
<code>public string Substring(int startIndex, int length)</code>	sous-chaîne de la chaîne courante commençant à la position <i>startIndex</i> et ayant <i>length</i> caractères
<code>public string ToLower()</code>	rend la chaîne courante en minuscules
<code>public string ToUpper()</code>	rend la chaîne courante en majuscules
<code>public string Trim()</code>	rend la chaîne courante débarrassée de ses espaces de début et fin

On notera un point important : lorsqu'une méthode rend une chaîne de caractères, celle-ci est une **chaîne différente** de la chaîne sur laquelle a été appliquée la méthode

Ainsi *S1.Trim()* rend une chaîne *S2*, et *S1* et *S2* sont deux chaînes différentes.

Une chaîne *C* peut être considérée comme un tableau de caractères. Ainsi

- **C[i]** est le caractère *i* de *C*
- **C.Length** est le nombre de caractères de *C*

Considérons l'exemple suivant :

```

using System;
namespace Chap3 {
    class Program {
        void affiche (string s) {
            Console.WriteLine(s);
        }

        static void Main(string[] args) {
            string uneChaine = "l'oiseau vole au-dessus des nuages";
            affiche("uneChaine=" + uneChaine);
            affiche("uneChaine.Length=" + uneChaine.Length);
            affiche("chaine[10]=" + uneChaine[10]);
            affiche("uneChaine.IndexOf(\"vole\")=" + uneChaine.IndexOf("vole"));
            affiche("uneChaine.IndexOf(\"x\")=" + uneChaine.IndexOf("x"));
            affiche("uneChaine.LastIndexOf('a')=" + uneChaine.LastIndexOf('a'));
            affiche("uneChaine.LastIndexOf('x')=" + uneChaine.LastIndexOf('x'));
            affiche("uneChaine.Substring(4,7)=" + uneChaine.Substring(4, 7));
            affiche("uneChaine.ToUpper()=" + uneChaine.ToUpper());
            affiche("uneChaine.ToLower()=" + uneChaine.ToLower());
            affiche("uneChaine.Replace('a','A')=" + uneChaine.Replace('a', 'A'));
            string[] champs = uneChaine.Split(null);
            for (int i = 0; i < champs.Length; i++)
                affiche("champs[" + i + "]=[" + champs[i] + "]");
            affiche("Join(\":\",champs)=" + System.String.Join(":", champs));
            affiche("(\"  abc  \").Trim()=[" + "  abc  ".Trim() + "]");
        } //Main
    } //classe
} //namespace

```

L'exécution donne les résultats suivants :

```

uneChaine=l'oiseau vole au-dessus des nuages
uneChaine.Length=34
chaine[10]=o
uneChaine.IndexOf("vole")=9
uneChaine.IndexOf("x")=-1
uneChaine.LastIndexOf('a')=30
uneChaine.LastIndexOf('x')=-1
uneChaine.Substring(4,7)=seau vo
uneChaine.ToUpper()=L'OISEAU VOLE AU-DESSUS DES NUAGES
uneChaine.ToLower()=l'oiseau vole au-dessus des nuages
uneChaine.Replace('a','A')=l'oiseAu vole Au-dessus des nuAges
champs[0]=[l'oiseau]
champs[1]=[vole]
champs[2]=[au-dessus]
champs[3]=[des]
champs[4]=[nuages]
Join(":",champs)=l'oiseau:vole:au-dessus:des:nuages
("  abc  ").Trim()=[abc]

```

Considérons un nouvel exemple :

```
using System; 2

namespace Chap3 {
    class Program {
        static void Main(string[] args) {
            // la ligne à analyser
            string ligne = "un:deux::trois:";
            // les séparateurs de champs
            char[] séparateurs = new char[] { ':' };
            // split
            string[] champs = ligne.Split(séparateurs);
            for (int i = 0; i < champs.Length; i++)
                Console.WriteLine("Champs[" + i + "]= " + champs[i]);
            // join
            Console.WriteLine("join=[" + System.String.Join(":", champs) + "]");
        }
    }
}
```

et les résultats d'exécution :

```
Champs[0]=un
Champs[1]=deux
Champs[2]=
Champs[3]=trois
Champs[4]=
join=[un:deux::trois:]
```

La méthode *Split* de la classe *String* permet de mettre dans un tableau des éléments d'une chaîne de caractères.

La définition de la méthode *Split* utilisée ici est la suivante :

```
public string[] Split(char[] separator);
```

separator est un tableau de caractères. Ces caractères représentent les caractères utilisés pour séparer les champs de la chaîne. On peut donc décomposer une chaîne autour de plusieurs séparateurs. Si le séparateur est une suite d'espaces on utilisera *separator=null*.

Exemple :

```
char[] sep = { ',', '-' };
string data = "Demain, c'est la Saint-Jean";
string[] champs = data
Split(sep);
foreach (string champ in champs)
    Console.WriteLine("<" + champ + ">");
```

donne :

```
<Demain>
< c'est la Saint>
<Jean>
```

La méthode *Join* effectue l'opération inverse, insérant un caractère entre chaque champ

```
public static string Join(string separator, string[] value);
```

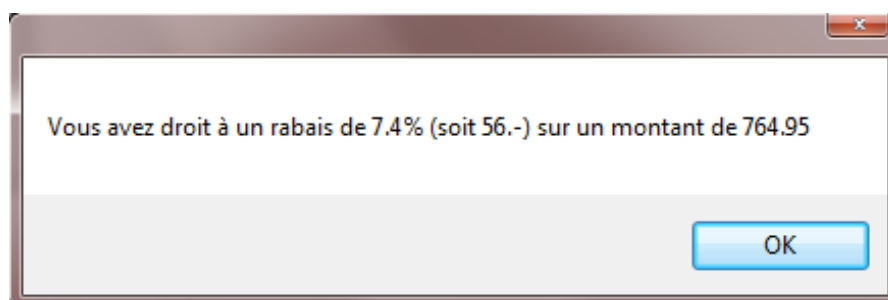
4.1.1 String.Format

La classe *String* peut également être invoquée directement (sans passer par une instance d'objet) pour effectuer certaines opérations.

Le cas le plus courant de cette pratique est celui du formatage de données en chaîne de caractères. Exemple-type : on a calculé deux valeurs dans deux variables et on veut présenter ces résultats sous forme de texte à l'utilisateur.

```
private void CalculerRabais(double Montant, int Age)
{
    double Pourcent = Math.Round(Age / 10.0 * 2.0, 1);
    int Rabais = (int) (Montant * Pourcent / 100.0);
    string msg = String.Format("Vous avez droit à un rabais de {0:F1}% (soit {1}.-) sur un montant de {2:F2}", Pourcent, Rabais, Montant);
    MessageBox.Show(msg);
}
```

Si on appelle cette fonction avec les paramètres 764.95 et 37, on aura :



Voir sections 3.2.12 et 3.2.13 pour plus de détails, ainsi qu'en annexe pour les spécificateurs standards.

4.2 StringBuilder

Nous avons vu que les méthodes de la classe **String** qui s'appliquaient à une chaîne de caractères *S1* rendait une autre chaîne *S2*. La classe **System.Text.StringBuilder** permet de manipuler *S1* sans avoir à créer une chaîne *S2*. Cela améliore les performances en évitant la multiplication de chaînes à durée de vie très limitée.

La classe admet divers constructeurs :

<code>StringBuilder()</code>	constructeur par défaut
<code>StringBuilder(String value)</code>	construction et initialisation avec <i>value</i>
<code>StringBuilder(String value, int capacité)</code>	construction et initialisation avec <i>value</i> avec une taille de bloc de <i>capacité</i> caractères

Un objet *StringBuilder* travaille avec des blocs de *capacité* caractères pour stocker la chaîne sous-jacente. Par défaut *capacité* vaut 16. Le 3ième constructeur ci-dessus permet de préciser la capacité des blocs. Le nombre de blocs de *capacité* caractères nécessaire pour stocker une chaîne S est ajusté automatiquement par la classe *StringBuilder*. Il existe des constructeurs pour fixer le nombre maximal de caractères dans un objet *StringBuilder*. Par défaut, cette capacité maximale est 2 147 483 647.

Voici un exemple illustrant cette notion de capacité :

```
using System
Text;
using System;
namespace Chap3 {
    class Program {
        static void Main(string[] args) {
            // str
            StringBuilder str = new StringBuilder("test");
            Console.WriteLine("taille={0}, capacité={1}", str.Length, str.Capacity);
            for(int i=0; i<10; i++){
                str.Append("test");
                Console.WriteLine("taille={0}, capacité={1}", str.Length, str.Capacity);
            }
            // str2
            StringBuilder str2 = new StringBuilder("test",10);
            Console.WriteLine("taille={0}, capacité={1}", str2.Length, str2.Capacity);
            for(int i=0; i<10; i++){
                str2.Append("test");
                Console.WriteLine("taille={0}, capacité={1}", str2.Length, str2.Capacity);
            }
        }
    }
}
```

- Création d'un objet *StringBuilder* contenant « test », avec une taille de bloc de 16 caractères

- *str.Length* est le nombre actuel de caractères de la chaîne *str*. *str.Capacity* est le nombre de caractères que peut stocker la chaîne *str* actuelle avant réallocation d'un nouveau bloc.
- *str.Append(String S)* permet de concaténer la chaîne *S* de type *String* à la chaîne *str* de type *StringBuilder*

Résultats :

```
taille=4, capacité=16 2
taille=8, capacité=16 3
taille=12, capacité=16 4
taille=16, capacité=16 5
taille=20, capacité=32 6
taille=24, capacité=32 7
taille=28, capacité=32 8
taille=32, capacité=32 9
taille=36, capacité=64 10
taille=40, capacité=64 11
taille=44, capacité=64 12
taille=4, capacité=10
taille=8, capacité=10
taille=12, capacité=20 15
taille=16, capacité=20 16
taille=20, capacité=20 17
taille=24, capacité=40 18
taille=28, capacité=40 19
taille=32, capacité=40 20
taille=36, capacité=40 21
taille=40, capacité=40 22
taille=44, capacité=80
```

Ces résultats montrent que la classe suit un algorithme qui lui est propre pour allouer de nouveaux blocs lorsque sa capacité est insuffisante :

- lignes 4-5 : augmentation de la capacité de 16 caractères
- lignes 8-9 : augmentation de la capacité de 32 caractères alors que 16 auraient suffi.

Voici quelques-unes des méthodes de la classe :

Méthode	Description
<code>public StringBuilder Append(string value)</code>	ajoute la chaîne <i>value</i> à l'objet <i>StringBuilder</i> . Rend l'objet <i>StringBuilder</i> . Cette méthode est surchargée pour admettre différents types pour <i>value</i> : byte, int, float, double, decimal, ...
<code>public StringBuilder Insert(int index, string value)</code>	insère <i>value</i> à la position <i>index</i> . Cette méthode est surchargée comme la précédente pour accepter différents types pour <i>value</i> .

Méthode	Description
<code>public StringBuilder Remove(int index, int length)</code>	supprime <i>length</i> caractères à partir de la position <i>index</i> .
<code>public StringBuilder Replace(string oldValue, string newValue)</code>	remplace dans <i>StringBuilder</i> , la chaîne <i>oldValue</i> par la chaîne <i>newValue</i> . Il existe une version surchargée (<code>char oldChar, char newChar</code>).
<code>public String ToString()</code>	convertit l'objet <i>StringBuilder</i> en un objet de type <i>String</i> .

Exemple :

```
using System.Text;
using System;
namespace Chap3 {
    class Program {
        static void Main(string[] args) {
            // str3
            StringBuilder str3 = new StringBuilder("test");
            Console.WriteLine(str3.Append("abCD").Insert(2, "xyZT").
                               Remove(0,2).Replace("xy", "XY"));
        }
    }
}
```

Résultat :

XYZTstabCD

4.3 Array

Les tableaux dérivent de la classe **Array**, qui possède diverses méthodes pour trier un tableau, rechercher un élément dans un tableau, redimensionner un tableau, ... Nous présentons certaines propriétés et méthodes de cette classe. Elles sont quasiment toutes surchargées, c.a.d. qu'elles existent en différentes variantes. Tout tableau en hérite.

Propriétés

Propriété	Description
<code>public int Length</code>	nombre total d'éléments du tableau
<code>public int Rank</code>	quelque soit son nombre de dimensions <i>public int</i> nombre total de dimensions du tableau

Méthodes

Méthode	Description
<code>public static int BinarySearch<T>(T[] tableau, T</code>	rend la position de <i>value</i> dans <i>tableau</i> . <i>value</i>)
<code>public static int BinarySearch<T>(T[] tableau, int index, int length, T value)</code>	idem mais cherche dans <i>tableau</i> à partir de la position <i>index</i> et sur <i>length</i> éléments
<code>public static void Clear(Array tableau, int index, int length)</code>	met les <i>length</i> éléments de <i>tableau</i> commençant au n° <i>index</i> à 0 si numériques, false si booléens, null si références
<code>public static void Copy(Array source, Array destination, int length)</code>	copie <i>length</i> éléments de <i>source</i> dans <i>destination</i>
<code>public int GetLength(int i)</code>	nombre d'éléments de la dimension n° i du tableau
<code>public int GetLowerBound(int i)</code>	indice du 1er élément de la dimension n° i
<code>public int GetUpperBound(int i)</code>	indice du dernier élément de la dimension n° i
<code>public static int IndexOf<T>(T[] tableau, T valeur)</code>	rend la position de <i>valeur</i> dans <i>tableau</i> ou -1 si <i>valeur</i> n'est pas trouvée.
<code>public static void Resize<T>(ref T[] tableau, int n)</code>	redimensionne <i>tableau</i> à <i>n</i> éléments. Les éléments déjà présents sont conservés.
<code>public static void Sort<T>(T[] tableau, IComparer<T> comparateur)</code>	trie <i>tableau</i> selon un ordre défini par <i>comparateur</i> .

Le programme suivant illustre l'utilisation de certaines méthodes de la classe *Array* :

```
using System;
namespace Chap3 {
    class Program {
        // type de recherche
        enum TypeRecherche { linéaire, dichotomique };

        // méthode principale
        static void Main(string[] args) {
            // lecture des éléments d'un tableau tapés au clavier
            double[] éléments;
            Saisie(out éléments);
            // affichage tableau non trié
            Affiche("Tableau non trié", éléments);
            // Recherche linéaire dans le tableau non trié
            Recherche(éléments, TypeRecherche.linéaire);
            // tri du tableau
            Array.Sort(éléments);
            // affichage tableau trié
            Affiche("Tableau trié", éléments);
            // Recherche dichotomique dans le tableau trié
            Recherche(éléments, TypeRecherche.dichotomique);
        }

        // Saisie des valeurs du tableau
        static void Saisie(out double[] éléments) {
            bool terminé = false;
            string réponse;
            bool erreur;
            double élément = 0 ;
            int i=0;
            // au départ, le tableau n'existe pas
            éléments = null;
            // boucle de saisie des éléments du tableau
            while (!terminé) {
                // question
                Console.Write("Élément (réel) " + i + " du tableau (rien = terminer) : ");
                // lecture de la réponse
                réponse = Console.ReadLine().Trim();
                // fin de saisie si chaîne vide
                if (réponse.Equals("")) break;
                // vérification saisie
                try
                {
                    élément = Double.Parse(réponse);
                    erreur = false;
                }
                catch
                {

```

```

        Console.Error.WriteLine("Saisie incorrecte, recommencez");
        erreur = true;
    } //try-catch

    if(!erreur)
    {
        // un élément de plus dans le tableau
        i+=1;
        // redimensionnement tableau pour accueillir le nouvel élément
        Array.Resize(ref éléments, i);
        // insertion nouvel élément
        éléments[i - 1] = élément;
    }
} //while
}

// méthode générique pour afficher les éléments d'un tableau
static void Affiche<T>(string texte, T[] éléments) {
    Console.WriteLine(texte.PadRight(50, '-'));
    foreach (T élément in éléments) Console.WriteLine(élément);
}

// recherche d'un élément dans le tableau
// éléments : tableau de réels
// TypeRecherche : dichotomique ou linéaire
static void Recherche(double[] éléments, TypeRecherche type) {
    // Recherche
    bool terminé = false;
    string réponse = null;
    double élément = 0;
    bool erreur = false;
    int i=0;
    while (!terminé) {
        // question
        Console.WriteLine("Elément cherché (rien pour arrêter) : ");
        // lecture-vérification réponse
        réponse = Console.ReadLine().Trim();
        // fini ?
        if (réponse.Equals("")) break;
        // vérification
        try
        {
            élément = Double.Parse(réponse);
            erreur = false;
        }
        catch
        {
            Console.WriteLine("Erreur, recommencez...");
            erreur = true;
        } //try-catch
    }
}

```

```

// si pas d'erreur
if(!erreur) {
    // on cherche l'élément dans le tableau
    if (type == TypeRecherche.dichotomique)
        // recherche dichotomique
        i = Array.BinarySearch(éléments, élément);
    else
        // recherche linéaire
        i = Array.IndexOf(éléments, élément); // Affichage réponse
    if(i>=0)
        Console.WriteLine("Trouvé en position " + i);
    else
        Console.WriteLine("Pas dans le tableau");
} //if
} //while
}
}

```

Notes :

- La méthode *Saisie* saisit les éléments d'un tableau *éléments* tapés au clavier. Comme on ne peut dimensionner le tableau à priori (on ne connaît pas sa taille finale), on est obligés de le redimensionner à chaque nouvel élément.
- Un algorithme plus efficace aurait été d'allouer de la place au tableau par groupe de N éléments. Un tableau n'est cependant pas fait pour être redimensionné . Ce cas là est mieux traité avec une liste (ArrayList, List<T>).
- La méthode *Recherche* permet de rechercher dans le tableau *éléments*, un élément tapé au clavier. Le mode de recherche est différent selon que le tableau est trié ou non. Pour un tableau non trié, on fait une recherche linéaire avec la méthode *IndexOf*. Pour un tableau trié, on fait une recherche dichotomique avec la méthode *BinarySearch*
- Ligne 18 : on trie le tableau *éléments*. On utilise ici, une variante de *Sort* qui n'a qu'un paramètre : le tableau à trier. La relation d'ordre utilisée pour comparer les éléments du tableau est alors celle implicite de ces éléments. Ici, les éléments sont numériques. C'est l'ordre naturel des nombres qui est utilisé.

Résultats :

```

Élément (réel) 0 du tableau (rien pour terminer) : 3,6
Élément (réel) 1 du tableau (rien pour terminer) : 7,4
Élément (réel) 2 du tableau (rien pour terminer) : -1,5
Élément (réel) 3 du tableau (rien pour terminer) : -7
Élément (réel) 4 du tableau (rien pour terminer) :
Tableau non trié-----
3,6
7,4
-1,5
-7
Élément cherché (rien pour arrêter) : 12. 7,4
Trouvé en position 1
Élément cherché (rien pour arrêter) : 15. 0
Pas dans le tableau
Élément cherché (rien pour arrêter) :

Tableau trié-----
-7
-1,5
3,6
7,4
Élément cherché (rien pour arrêter) :
7,4
Trouvé en position 3
Élément cherché (rien pour arrêter) :
0
Pas dans le tableau
Élément cherché (rien pour arrêter) :

```

4.4 StreamReader

La classe *System.IO.StreamReader* permet de lire le contenu d'un fichier texte. Elle est en fait également capable d'exploiter des flux qui ne sont pas des fichiers. Ce chapitre présente quelques-unes de ses propriétés et méthodes.

Constructeurs

<pre>public StreamReader(string path)</pre>	construit un flux de lecture à partir du fichier de chemin <i>path</i> . Le contenu du fichier peut être encodé de diverses façons. Il existe un constructeur qui permet de préciser le codage utilisé. Par défaut, c'est le codage UTF-8 qui est utilisé
---	---

Propriétés

Propriété	Description
<code>public bool EndOfStream</code>	True si le flux a été lu entièrement

Méthodes

Propriété	Description
<code>public void Close()</code>	ferme le flux et libère les ressources allouées pour sa gestion. A faire obligatoirement après
<code>public override int Peek()</code>	rend le caractère suivant du flux sans le consommer. Un Peek supplémentaire rendrait donc le même caractère.
<code>public override int Read()</code>	rend le caractère suivant du flux et avance d'un caractère dans le flux.
<code>public override int Read(char[] buffer, int index, int count)</code>	lit count caractères dans le flux et les met dans buffer à partir de la position index. Rend le nombre de caractères lus - peut être 0.
<code>public override string ReadLine()</code>	rend la ligne suivante du flux ou null si on était à la fin du flux.
<code>public override string ReadToEnd()</code>	rend la fin du flux ou "" si on était à la fin du flux.

Exemple :

```

using System;
using System.IO;
namespace Chap3 {
    class Program {
        static void Main(string[] args) {
            // répertoire d'exécution
            Console.WriteLine("Répertoire d'exécution : "+Environment.CurrentDirectory);
            string ligne = null;
            StreamReader fluxInfos = null;
            // lecture contenu du fichier infos.txt
            try
            {
                // lecture 1
                Console.WriteLine("Lecture 1-----");
                using (fluxInfos = new StreamReader("infos.txt"))
                {
                    do
                    {
                        ligne = fluxInfos.ReadLine();
                        Console.WriteLine(ligne);
                    } while (ligne != null) ;
                }
                // lecture 2
                Console.WriteLine("Lecture 2-----");
                using (fluxInfos = new StreamReader("infos.txt"))
                {
                    Console.WriteLine(fluxInfos.ReadToEnd());
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("L'erreur suivante s'est produite : " + e.Message);
            }
        }
    }
}

```

Notes :

- Ligne 2 : Afin d'avoir accès à la classe StreamReader (entre autres), il est nécessaire de déclarer l'utilisation de l'espace de nom *System.IO*
- Ligne 8 : affiche le nom du répertoire d'exécution
- La structure *using (flux=newStreamReader(...))* est une facilité pour ne pas avoir à fermer explicitement le flux après son exploitation. Cette fermeture est faite automatiquement dès qu'on sort de la portée du *using*.

- Le fichier lu s'appelle *infos.txt*. Comme c'est un nom relatif, il sera cherché dans le répertoire d'exécution affiché à la ligne 8. S'il n'y est pas, une exception sera lancée et gérée par le try / catch.
- Lignes 16-20 : le fichier est lu par lignes successives.
- Ligne 25 : le fichier est lu d'un seul coup

Avec le fichier info.txt placé dans le sous-répertoire bin/Debug du répertoire de votre solution VS et contenant :

```
12620:0:0
13190:0,05:631
15640:0,1:1290,5
```

on obtient le résultat :

```
Répertoire d'exécution :
C:\Data\CoursCSharp\Exercices\Chantier\SandBox\SandBox\bin\Debug
Lecture 1-----
12620:0:0
13190:0,05:631
15640:0,1:1290,5

Lecture 2-----
12620:0:0
13190:0,05:631
15640:0,1:1290,5
```

Si ligne 15, on met le nom de fichier xx.txt on a les résultats suivants :

```
Lecture 1-----
L'erreur suivante s'est produite : Could not find file
'C:\Data\CoursCSharp\Exercices\Chantier\SandBox\SandBox\bin\Debug\'
```

4.5 StreamWriter

La classe `System.IO.StreamWriter` permet d'écrire dans un fichier texte. Comme la classe `StreamReader`, elle est en fait capable d'exploiter des flux qui ne sont pas des fichiers. Voici quelques-unes de ses propriétés et méthodes :

Constructeurs

<pre>public StreamWriter(string path)</pre>	construit un flux d'écriture dans le fichier de chemin <code>path</code> . Le contenu du fichier peut être encodé de diverses façons. Il existe un constructeur qui permet de préciser le codage utilisé. Par défaut, c'est le codage UTF-8 qui est utilisé.
---	--

Propriétés

Propriété	Description
<code>public bool AutoFlush</code>	fixe le mode d'écriture dans le fichier du buffer associé au flux. Si égal à False , l'écriture dans le flux n'est pas immédiate : il y a d'abord écriture dans une mémoire tampon puis dans le fichier lorsque la mémoire tampon est pleine sinon l'écriture dans le fichier est immédiate (pas de tampon intermédiaire). Par défaut c'est le mode tamponné qui est utilisé. Le tampon n'est écrit dans le fichier que lorsqu'il est plein ou bien lorsqu'on le vide explicitement par une opération Flush ou encore lorsqu'on ferme le flux StreamWriter par une opération Close . Le mode AutoFlush=False est le plus efficace lorsqu'on travaille avec des fichiers parce qu'il limite les accès disque. C'est le mode par défaut pour ce type de flux. Le mode AutoFlush=False ne convient pas à tous les flux, notamment les flux réseau. Pour ceux-ci, qui souvent prennent place dans un dialogue entre deux partenaires, ce qui est écrit par l'un des partenaires doit être immédiatement lu par l'autre. Le flux d'écriture doit alors être en mode AutoFlush=True .
<code>public string NewLine</code>	les caractères de fin de ligne. Par défaut "\r\n". Pour un système Unix, il faudrait utiliser "\n".

Méthodes

Propriété	Description
<code>public void Close()</code>	ferme le flux et libère les ressources allouées pour sa gestion. A faire obligatoirement après exploitation du flux.
<code>public override void Flush()</code>	Ecrit dans le fichier, le buffer du flux, sans attendre qu'il soit plein.
<code>public virtual void Write(T value)</code>	Ecrit value dans le fichier associé au flux. Ici T n'est pas un type générique mais symbolise le fait que la méthode Write accepte différents types de paramètres (string, int, object, ...). La méthode <code>value.ToString</code> est utilisée pour produire la chaîne écrite dans le fichier.
<code>public virtual void WriteLine(T value)</code>	même chose que Write mais avec la marque de fin de ligne (NewLine) en plus.

Exemple :

```

using System;
using System.IO;
namespace Chap3 {
    class Program2 {
        static void Main(string[] args) {
            // répertoire d'exécution
            Console.WriteLine("Répertoire courant : " + Environment.CurrentDirectory);
            string ligne = null; // une ligne de texte
            StreamWriter fluxInfos = null; // le fichier texte
            try
            {
                // création du fichier texte
                using (fluxInfos = new StreamWriter("infos2.txt")) {
                    Console.WriteLine("Mode AutoFlush : {0}", fluxInfos.AutoFlush);
                    // lecture ligne tapée au clavier
                    Console.Write("ligne (rien pour arrêter) : ");
                    ligne = Console.ReadLine().Trim();
                    // boucle tant que la ligne saisie est non vide
                    while (ligne != "")
                    {
                        // écriture ligne dans fichier texte
                        fluxInfos.WriteLine(ligne);
                        // lecture nouvelle ligne au clavier
                        Console.Write("ligne (rien pour arrêter) : ");
                        ligne = Console.ReadLine().Trim();
                    } //while
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("L'erreur suivante s'est produite : " + e.Message);
            }
        }
    }
}

```

Notes :

- ligne13: de nouveau, nous utilisons la syntaxe *using(flux)* afin de ne pas avoir à fermer explicitement le flux par une opération *Close*. Cette fermeture est faite automatiquement à la sortie du *using*.
- Pourquoi un *try / catch*, lignes 11 et 27 ? ligne 13, nous pourrions donner un nom de fichier sous la forme */rep1/rep2/ .../fichier* avec un chemin */rep1/rep2/...* qui n'existe pas, rendant ainsi impossible la création de *fichier*. Une exception serait alors lancée. Il existe d'autres cas d'exception possible (disque plein, droits insuffisants, ...)

Résultat :

```
Répertoire courant : C:\data\2007-2008\c# 2008\poly\Chap3\07\bin\Release
Mode AutoFlush : False
ligne (rien pour arrêter) : 1ère ligne
ligne (rien pour arrêter) : 2ième ligne
ligne (rien pour arrêter) :
```

Le fichier infos2.txt a été créé dans le dossier bin/Release du projet :

```
1ère ligne
2ième ligne
```

4.6 Random

Il est fréquent qu'un programme aie besoin de choisir des valeurs au hasard. C'est à la classe Random que l'on a recours pour cela.

Il faut tout d'abord créer (instancier) un objet à partir de la classe Random :

```
Random Hasard = new Random();
```

Ensuite, on va pouvoir demander à cet objet de nous fournir des valeurs aléatoires au moyen de la méthode Next. Cette méthode peut être utilisée de trois manières différentes :

Next() : Retourne un nombre entier aléatoire non négatif

Next(Max) : Retourne un nombre entier inférieur à Max. Et donc Next(10) ne retournera jamais 10 !

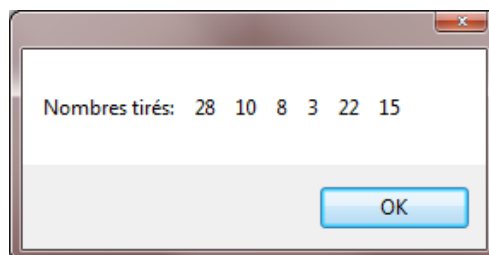
Next(Min,Max) : Retourne un nombre entier supérieur ou égal à Min et inférieur à Max. Et donc Next(10, 100) pourra retourner 10 mais pas 100 !

Attention : Les nombres aléatoires retournés par un objet de la classe Random dépendent du moment où l'objet a été créé. Il faut donc faire attention à ne pas systématiquement créer un nouvel objet à chaque fois que l'on a besoin d'un nombre aléatoire.

Exemple : Simulons un tirage de six numéros de loterie :

```
Random Hasard = new Random();
string Tirage = "Nombres tirés: ";
for (int i = 1; i <= 6; i++)
    Tirage = Tirage + Hasard.Next(1,41).ToString() + " ";
MessageBox.Show(Tirage);
```

On obtient :



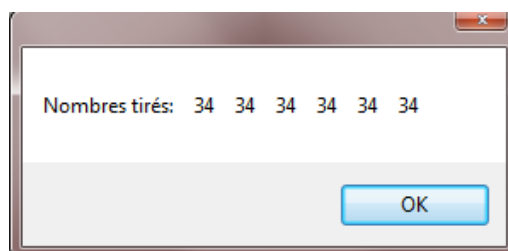
L'objet « Hasard » a été créé une fois et on lui a demandé six numéros.

Exemple **qui ne marche pas** :

Le code suivant est syntaxiquement correct

```
string Tirage = "Nombres tirés: ";  
for (int i = 1; i <= 6; i++)  
{  
    Random Hasard = new Random();  
    Tirage = Tirage + Hasard.Next(1,41).ToString() + " ";  
}  
MessageBox.Show(Tirage);
```

Mais il ne donne pas le résultat auquel on s'attend :



En effet, on recrée un objet Random six fois très rapidement (probablement dans la même milliseconde) et on ne leur demande qu'un seul numéro. Ces six objets différents ayant été créés quasiment au même moment, ils retourneront tous le même nombre – 34 en l'occurrence.

5 Interfaces Graphiques

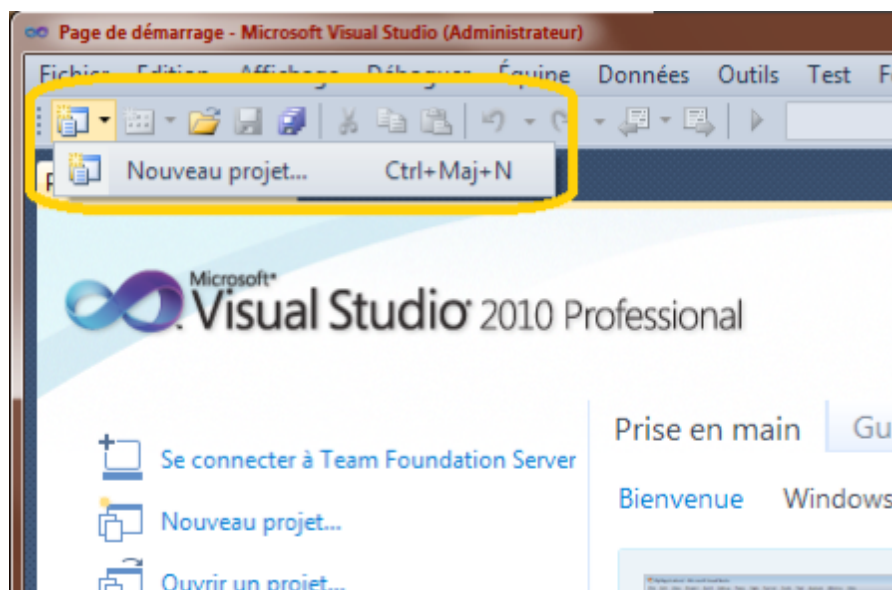
5.1 Les bases des interfaces graphiques

Nous nous proposons ici de donner les premiers éléments pour construire des interfaces graphiques et gérer leurs événements. La démarche et les illustrations que vous allez y rencontrer sont basées sur Visual Studio 2010 Professional.

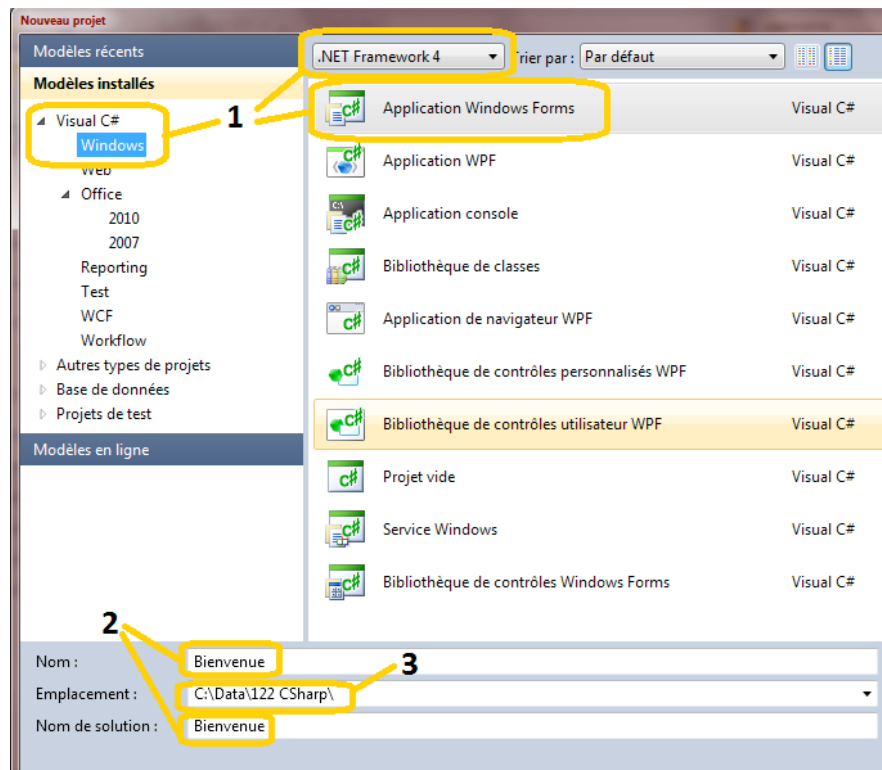
5.1.1 Un premier projet

Nous allons construire un premier projet de type "Application windows", donc un programme à interface graphique. Au passage, nous mettrons en évidence certains éléments de l'environnement Visual Studio.

Démarrez VS et créez un nouveau projet :

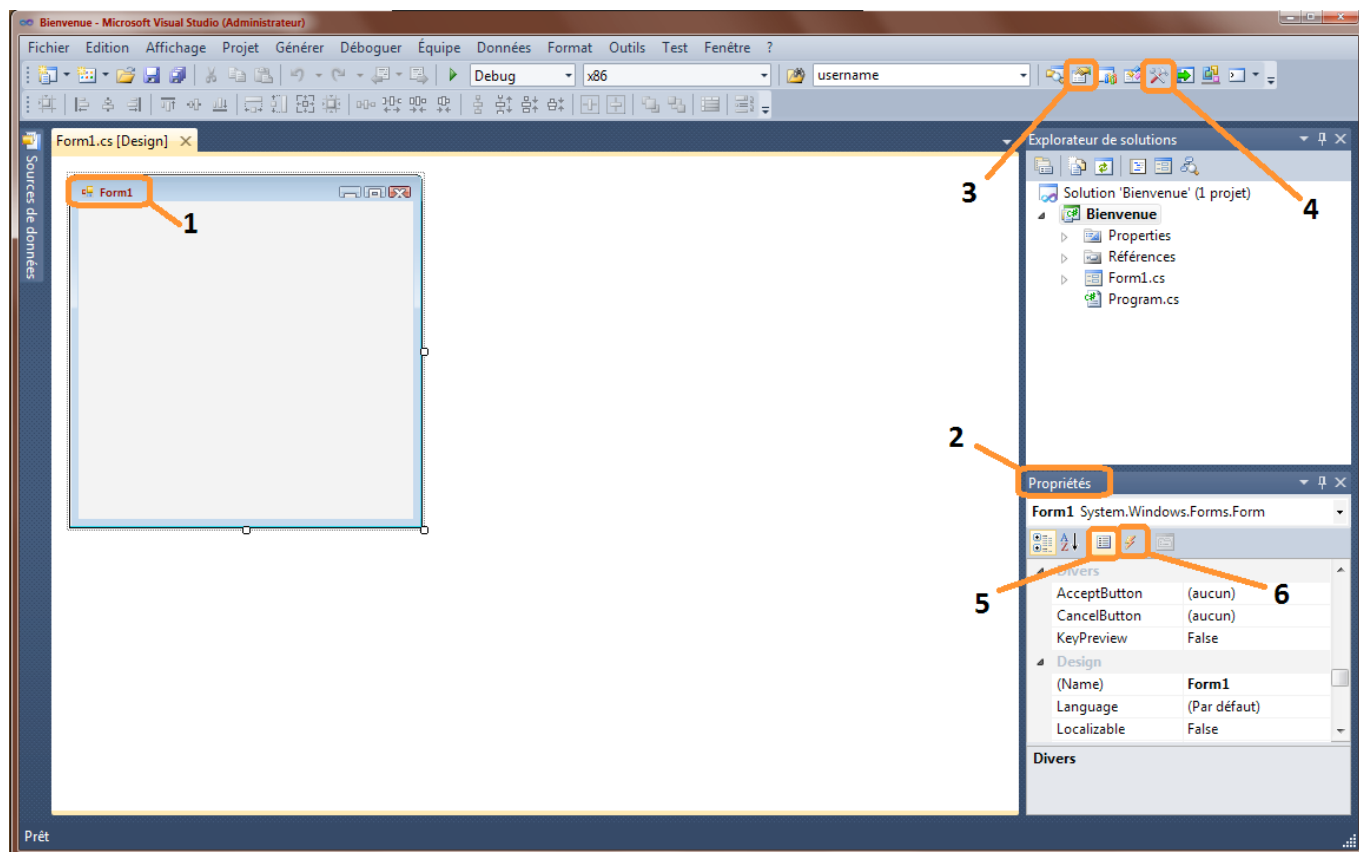


La page de nouveau projet apparaît :



1. Sélectionnez le type d'application : « Visual C#/Windows/Application Windows Forms / .NET Framework 4 »
2. Donnez un nom à votre projet. Par défaut, VS va automatiquement donner ce même nom à la solution.
3. Sélectionnez un répertoire approprié pour le stockage de ce projet ainsi que de tous les exercices liés à ce cours

Lorsque vous cliquez OK, votre projet est créé et l'espace de travail se présente ainsi :

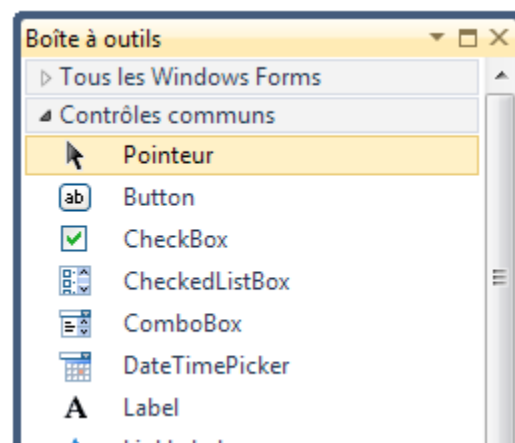


Cet espace est divisé en trois zones principales :

1. Le **formulaire**. C'est le cadre de la fenêtre dans laquelle votre application s'exécutera. Par défaut, ce formulaire est appelé *Form1* et il est naturellement vide.
2. La **feuille de propriétés**. Elle permet d'accéder à toutes les propriétés et événements de l'objet sélectionné dans la partie centrale. Dans l'image ci-dessus, on peut voir que l'objet sélectionné est le formulaire tout entier et que son titre (propriété *Text*) est *Form1*. La feuille de propriété peut être ouverte en tout temps avec le bouton [3]. On bascule entre les propriétés et les événements au moyen des boutons [5] et [6].

En cliquant sur le bouton [4], vous faites apparaître une autre fenêtre essentielle : la boîte à outils.

Cette dernière contient tous les éléments (appelés contrôles) mis à votre disposition pour construire votre application.



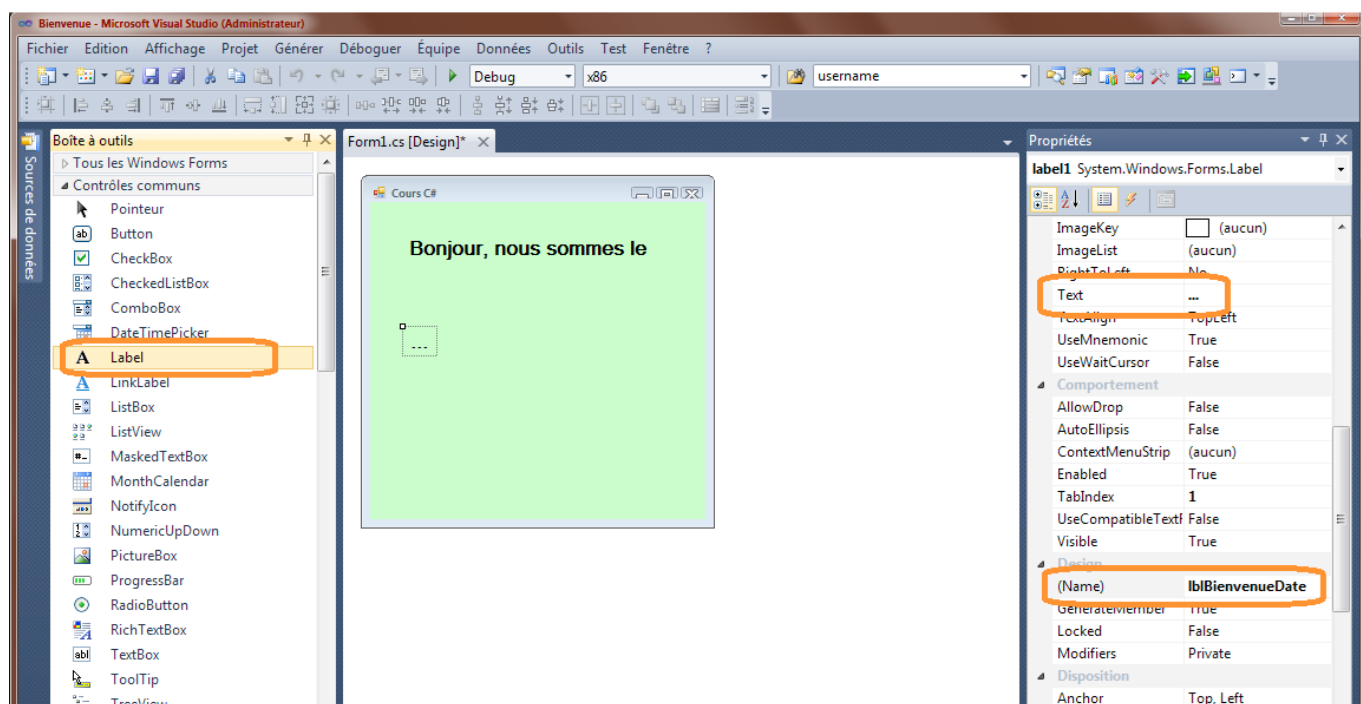
Note : la boîte à outils et la feuille de propriétés peuvent être présentées à votre guise comme Ancrées (comme ci-dessus) ou flottantes.

Nous allons maintenant créer notre application. Commençons par modifier quelques propriétés du formulaire au moyen de la feuille de propriétés :

- Modifiez le nom du formulaire : mettez la propriété (*Name*) du formulaire à *frmBienvenue*.
- Modifiez le titre de la fenêtre : propriété *Text* du formulaire à *Cours C#*.
- Mettez la propriété *StartPosition* à *CenterScreen*.
- Changez la couleur du fond : propriété *BackColor*.

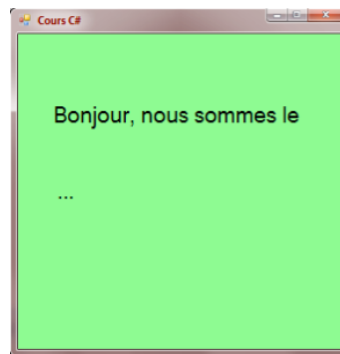
Sélectionnez le type de contrôle *Label* dans la boîte à outils et créez deux contrôles *lblBienvenueTexte* et *lblBienvenueDate*.

Modifiez les propriétés d'apparence de ces labels à votre guise (police, taille, position), mettez la propriété *Text* de *lblBienvenueDate* à « ... ». N'oubliez pas que la fenêtre de propriétés se rapporte à l'objet sélectionné dans votre formulaire.



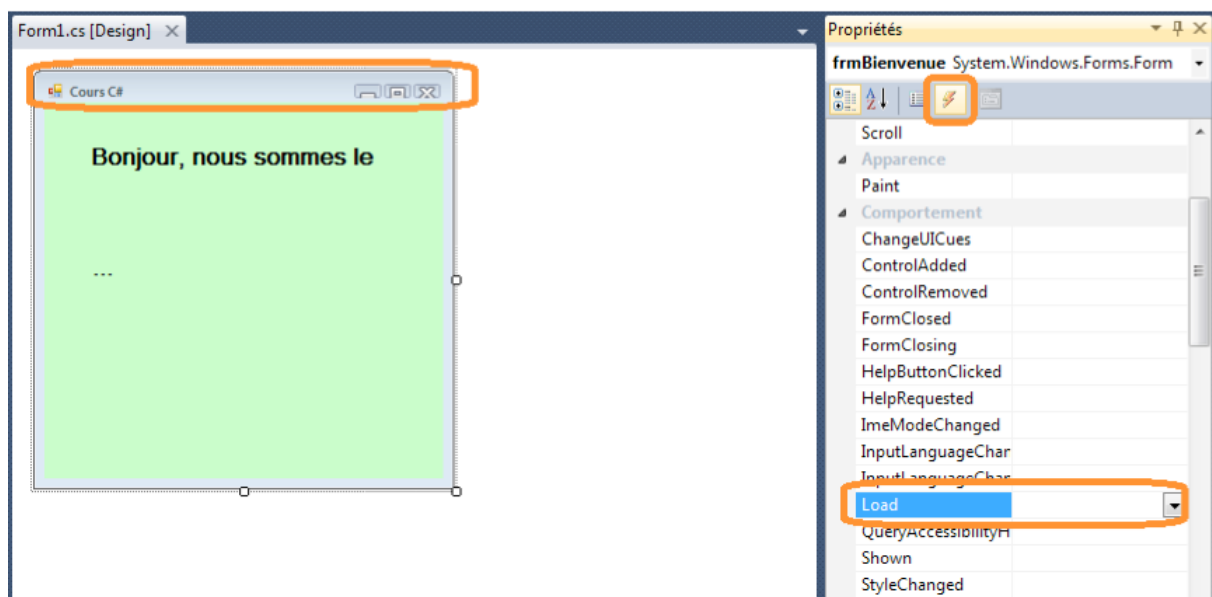
Lancez maintenant l'application (touche [F5] ou bouton )

La fenêtre suivante apparaît au centre de votre écran :



Fermez la fenêtre pour retourner à VS.

Nous allons maintenant ajouter du code à cette application. Sélectionnez le formulaire entier en cliquant sur la barre titre. Dans la feuille de propriétés, affichez la liste des événements. Double-cliquez dans la case (vide) en face de l'événement *Load*.



La page du code liée au formulaire projet Bienvenue apparaît :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

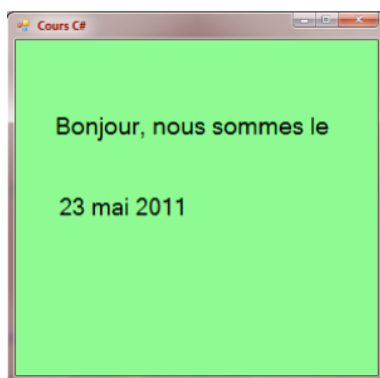
namespace Bienvenue
{
    public partial class frmBienvenue : Form
    {
        public frmBienvenue()
        {
            InitializeComponent();
        }

        private void frmBienvenue_Load(object sender, EventArgs e)
        {
        }
    }
}
```

Dans la méthode *frmBienvenue_Load* (là où se trouve le curseur), insérez le code :

```
lblBienvenueDate.Text = string.Format ("{0:dd MMM yyyy}", DateTime.Now);
```

Exécutez l'application. La fenêtre qui s'affiche contient cette fois la date courante :



Explications :

- Dans un premier temps nous n'avons fait que disposer des éléments graphiques (les contrôles de type *Label*) dans le formulaire de notre application. Nous avons défini des propriétés pour chacun des objets de notre application :
 - *frmBienvenue.BackColor* 128 ; 255 ; 128 → la couleur du fond du formulaire *frmBienvenue* est verte.
 - *lblBienvenueTexte.Font.Size* 14 → la taille de la police du label *lblBienvenueTexte* est de 14 points.
 - Etc...
- Les valeurs de ces propriétés sont définies de manière statique, avant l'exécution du programme.
- Tant que l'application s'exécute, le système détecte les événements causés par l'interaction de l'utilisateur avec l'application : redimensionnement de la fenêtre, déplacement de la fenêtre, click de souris, touche pressée, etc...
- Par défaut, la vaste majorité de ces événements ne cause aucune « réaction » de la part de notre programme. Il ne se passe donc rien de visible jusqu'à ce que l'on quitte l'application.
- Dans un deuxième temps, nous avons décidé d'intercepter l'évènement de chargement (Load) du formulaire. En ajoutant le code de la méthode *frmBienvenue_Load*, nous avons créé un **Gestionnaire d'événement** (EventHandler en anglais).
- Notre gestionnaire d'événement a pour effet de changer le texte statique « ... » que nous avons mis dans *lblBienvenueDate*, pour y mettre la date courante.
- Attention : il est important de rappeler que l'objet *lblBienvenueDate* ne se réduit pas à son simple texte, il est fait d'un grand nombre de propriétés. Notre code ne modifie pas le contrôle dans son ensemble (*lblBienvenueDate*), mais seulement la valeur de son texte (*lblBienvenueDate.Text*). Nous aurions tout aussi bien pu choisir de modifier la couleur de la police (*lblBienvenueDate.Font.Color*) pour choisir une couleur de saison par exemple.

Nous avons découvert à travers cet exemple le mode de programmation dit événementiel : une fois l'interface graphique construite avec Visual studio, le travail du développeur consiste principalement à écrire les gestionnaires des événements qu'il veut gérer pour cette interface graphique. La suite de cette section présente un choix des contrôles les plus courants, ainsi que leurs attributs et événements spécifiques.

5.2 Les composants de base

Nous présentons maintenant diverses applications mettant en jeu les composants les plus courants afin de découvrir les principales méthodes et propriétés de ceux-ci. Pour chaque composant nous présentons les propriétés principales, les méthodes et événements les plus souvent utilisés, le tout suivi d'un exemple d'utilisation

5.2.1 Boîte de Dialogue

Nous commencerons par un élément essentiel permettant l'interaction entre l'application et l'utilisateur : la boîte de dialogue. Il s'agit d'une fenêtre qui permet de :

- Suspendre l'exécution du programme pour afficher un message
- Demander un choix simple à l'utilisateur, comme par exemple une confirmation d'effacement

L'apparition de la boîte de dialogue se fait au moyen de la commande *MessageBox.Show* dont la syntaxe est :

```
MessageBox.Show (Message, Titre, Boutons, Icône) ;
```

Message (obligatoire) est le texte à afficher

Titre (optionnel) est le texte mis dans la barre de la fenêtre.

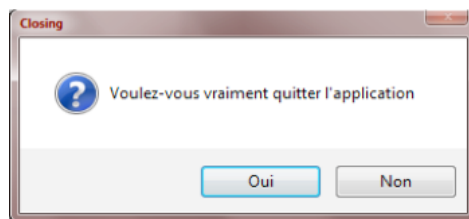
Boutons (optionnel) définit la combinaison de boutons à afficher, à choix parmi les valeurs de l'énumération *MessageBoxButtons*

Icône (optionnel) définit l'icône qui s'affichera dans la boîte de dialogue. A choix parmi les valeurs de l'énumération *MessageBoxIcon*

Le résultat retourné par *MessageBox.Show* est une des valeurs de l'énumération *DialogResult*, et indique le choix fait par l'utilisateur. Ainsi, l'appel :

```
DialogResult Res = MessageBox.Show("Voulez-vous vraiment quitter l'application ?",  
                                   "FormClosing",  
                                   MessageBoxButtons.YesNo,  
                                   MessageBoxIcon.Question);
```

Affichera la boîte de dialogue :



et mettra la valeur *DialogResult.No* dans la variable *Res*.

5.2.2 Propriétés, Méthodes et Evénements communs

Avant de détailler les composants, nous allons nous intéresser à certaines caractéristiques qui sont communes à tous les contrôles décrits par la suite

5.2.2.1 Propriétés

Les propriétés suivantes sont présentes et utiles pour la majorité des composants :

Propriété	Description
Name	Le nom du contrôle. Ce nom n'est pas visible à l'utilisateur, c'est celui qui est utilisé dans le code pour référencer le contrôle. Le nom est habituellement construit en suivant des conventions de codage.
Location	Les coordonnées du point supérieur gauche du contrôle par rapport au coin supérieur gauche de son conteneur (l'écran si le contrôle est un formulaire, le formulaire si le contrôle est un bouton, par exemple)
Size	La taille du contrôle en pixels : Width (Largeur) et Height (Hauteur)
Enabled	Booléen indiquant si le contrôle est actif. Le contrôle est visible, mais on ne peut rien faire avec. Un contrôle inactif est « grisé ».
Visible	Booléen indiquant si le contrôle est visible ou masqué. S'il masqué, il est également inactif : cliquer à l'emplacement d'un bouton invisible n'a pas d'effet.
Text	Le texte associé au contrôle est affiché : le nom du formulaire (dans la barre de fenêtre), le texte dans une boîte de texte, dans un bouton, à côté d'une case à cocher, etc...
Font	La police (nom, taille couleur, ...) utilisée pour afficher le texte dans ce contrôle
ForeColor	La couleur du premier plan du contrôle
BackColor	La couleur du fond du contrôle
BackgroundImage	L'image d'arrière-plan utilisée pour ce contrôle

5.2.2.2 Méthodes

Les méthodes suivantes existent pour la majorité des composants et permettent de les manipuler.

Méthode	Description
SetBounds	Permet de modifier la taille (Size) et/ou la position (Location) du contrôle.
Show	Rend le contrôle visible
Hide	Masque le contrôle
ToString	Retourne une chaîne de caractère qui est le nom du contrôle
Refresh	Force le contrôle à se redessiner

5.2.2.3 Événements

Les événements suivants existent pour la majorité des composants. Il est possible d'écrire un gestionnaire d'événements pour chacun d'entre eux.

Événement	Est déclenché
Click	Lorsque l'utilisateur clique sur le composant

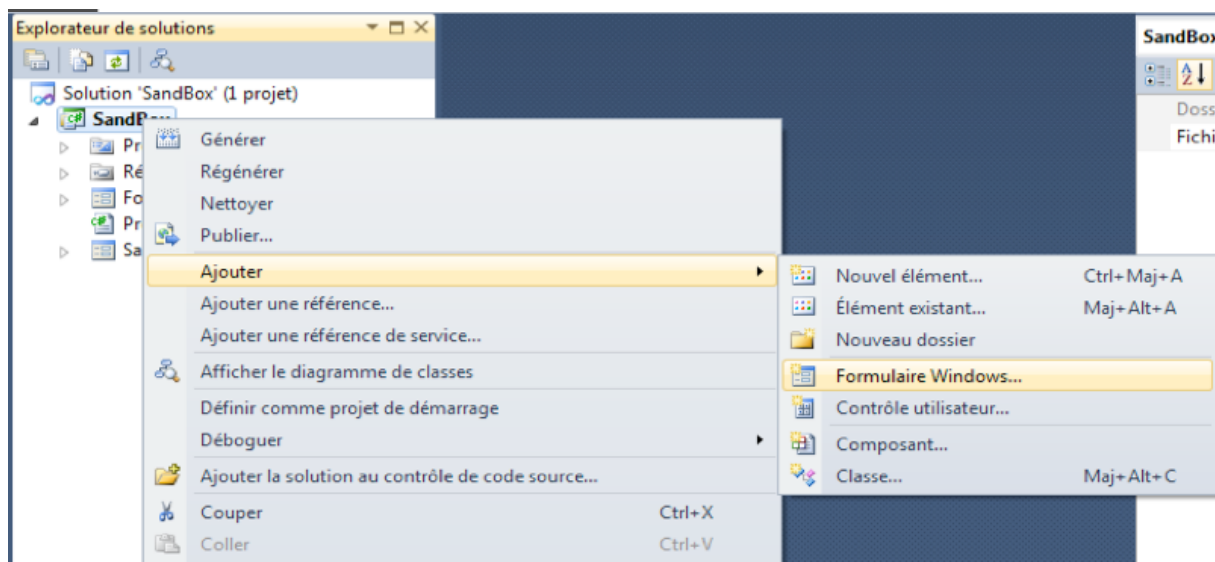
Événement	Est déclenché
DoubleClick	Lorsque l'utilisateur double-clique sur le composant
MouseHover	Lorsque la souris reste en dessus du contrôle
Validated	Après la validation réussie du contrôle
Validating	Lors de la validation du contrôle

5.2.3 Formulaire (« Form »)

Nous commençons par présenter le composant indispensable, le formulaire sur lequel on dépose des composants. Nous nous attardons ici sur quelques événements importants d'un formulaire.

5.2.3.1 Création

Par défaut, toute nouvelle application C# a un formulaire (nommé – par défaut – Form1). Bien souvent cela suffit, mais si vous devez ajouter un formulaire à votre application, il faut aller dans l'explorateur de solution, faire click-droit sur votre projet et sélectionner Ajouter/Formulaire Windows



5.2.3.2 Propriétés Spécifiques

Propriété	Description
MaximizeBox, MinimizeBox	Booléens permettant de définir si la fenêtre du formulaire a les boutons qui maximisent / minimisent la fenêtre
StartPosition	Détermine la position du formulaire lorsqu'il s'ouvre pour la première fois.
Controls	La liste de tous les contrôles déposés sur le formulaire

5.2.3.3 Méthodes Spécifiques

Méthode	Description
BringToFront	Fait passer le formulaire au premier plan
SendToBack	Envoie le formulaire au dernier plan
Close	Ferme le formulaire
CenterToScreen	Centre le formulaire dans l'écran

5.2.3.4 Événements Spécifiques

Attribut	Est déclenché
Load	Lorsque le formulaire est en cours de chargement. Il se produit avant même que le formulaire ne soit affiché.
FormClosing	Lorsque la fermeture du formulaire a été demandée, mais pas encore effectuée. Cet événement permet d'annuler la fermeture si certaines conditions ne sont pas satisfaisantes par exemple
FormClosed	Lorsque le formulaire est fermé
LocationChanged	Lorsque le formulaire est déplacé
Resize	Lorsque la fenêtre est redimensionnée

Exemple :

Nous construisons un formulaire de nom *Form1* sans composant et dont le code de [Form1.cs] est le suivant :

```
using System;
using System.Windows.Forms;
namespace Chap5 {
    public partial class Form1 : Form {

        public Form1() {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e) {
            // chargement initial du formulaire
            MessageBox.Show("Evt Load", "Load");
        }

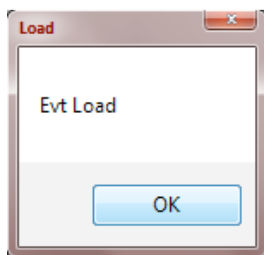
        private void Form1_FormClosing(object sender, FormClosingEventArgs e) {
            // le formulaire est en train de se fermer
            MessageBox.Show("Evt FormClosing", "FormClosing");
            // on demande confirmation
            DialogResult réponse = MessageBox.Show("Voulez-vous vraiment quitter ?",
                                                    "Closing",
                                                    MessageBoxButtons.YesNo,
                                                    MessageBoxIcon.Question);
            if (réponse == DialogResult.No) e.Cancel = true;
        }

        private void Form1_FormClosed(object sender, FormClosedEventArgs e) {
            // le formulaire va être fermé
            MessageBox.Show("Evt FormClosed", "FormClosed");
        }
    }
}
```

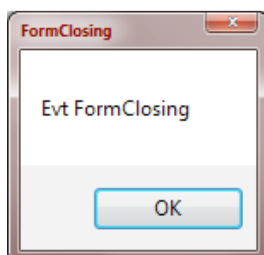
Notes :

- Nous utilisons la fonction *MessageBox* pour être averti des différents événements.

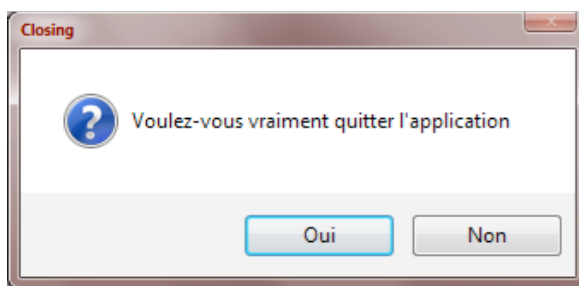
- ligne 10 : L'événement *Load* va se produire au démarrage de l'application avant même que le formulaire ne s'affiche :



- ligne 15 : L'événement *FormClosing* va se produire lorsque l'utilisateur ferme la fenêtre.



- ligne 19 : Nous lui demandons alors s'il veut vraiment quitter l'application :



- ligne 20 : S'il répond Non, nous fixons la propriété *Cancel* de l'événement *CancelEventArgs* que la méthode a reçu en paramètre. Si nous mettons cette propriété à *False*, la fermeture de la fenêtre est abandonnée, sinon elle se poursuit et l'événement *FormClosed* va alors se produire :

5.2.4 Etiquette (« Label »)

L'étiquette *Label* est composant texte qui permet simplement de poser un texte sur le formulaire. Les propriétés *Text* et *Font* vous permettent de donner l'apparence voulue à l'étiquette.

N'étant qu'une indication textuelle, il est très rare que l'on aie à gérer des événements liés aux étiquettes.

5.2.5 Boîte de saisie (« TextBox »)

Le contrôle *TextBox* est un composant champ de saisie de texte. Leur propriété principale est *Text* qui désigne le contenu du champ de saisie. Cette propriété est en lecture/écriture.

5.2.5.1 Propriétés Spécifiques

Propriété	Description
Multiline	Définit si la boîte de saisie peut contenir plusieurs lignes (true) ou une seule (false)
TextAlign	Alignement du texte à gauche (left), au centre (center) ou à droite (right)
Text	Le contenu du champ
WordWrap	Indique si les lignes sont automatiquement renvoyées à la ligne pour les contrôles multilignes
AcceptReturn	Indique si les retours de chariot sont acceptés en tant qu'entrée dans les contrôles multiligne

5.2.5.2 Méthodes Spécifiques

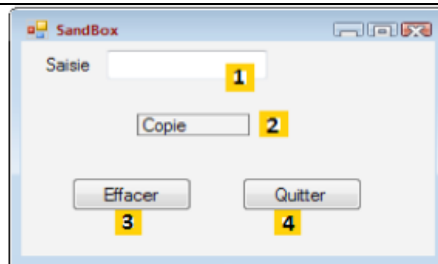
Méthode	Description
AppendText	Ajoute du texte au texte en cours
Clear	Vide le champ
Focus	Met le focus d'entrée sur le champ

5.2.5.3 Événements Spécifiques

Événement	Est déclenché
Enter	Lorsque l'utilisateur entre dans le contrôle
KeyPress	Lorsqu'une touche est pressée alors que le contrôle a le focus
TextChanged	Lorsque le contenu du champ a changé. Note : cela signifie que l'événement est déclenché à chaque frappe de touche
Validated	Lorsque l'on quitte le champ après avoir entièrement introduit la valeur

Exemple

L'événement habituellement utilisé pour *TextBox* est *TextChanged* qui signale que l'utilisateur a modifié le champ de saisie. Voici un exemple qui utilise l'événement *TextChanged* pour suivre les évolutions d'un champ de saisie :



n°	type	nom	rôle
1	TextBox	textBoxSaisie	champ de saisie
2	Label	labelControle	affiche le texte de 1 en temps réel AutoSize=False, Text=(rien)
3	Button	buttonEffacer	pour effacer les champs 1 et 2
4	Button	buttonQuitter	pour quitter l'application

Le code de cette application est le suivant :

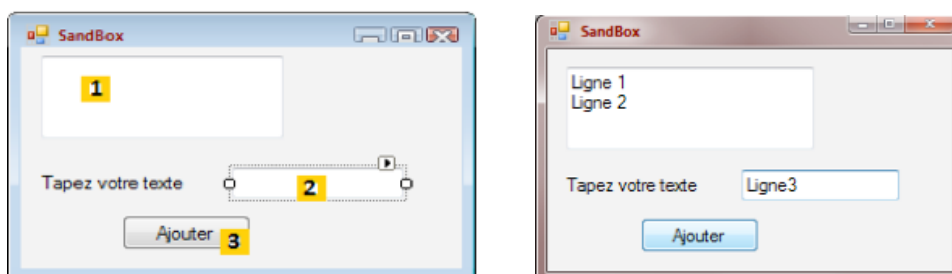
```
using System.Windows.Forms;
namespace Chap5 {
    public partial class Form1 : Form {
        public Form1() {
            InitializeComponent();
        }
        private void textBoxSaisie_TextChanged(object sender, System.EventArgs e) {
            // le contenu du TextBox a changé - on le copie dans le Label labelControle
            labelControle.Text = textBoxSaisie.Text;
        }
        private void buttonEffacer_Click(object sender, System.EventArgs e) {
            // on efface le contenu de la boîte de saisie
            textBoxSaisie.Text = "";
        }
        private void buttonQuitter_Click(object sender, System.EventArgs e) {
            // clic sur bouton Quitter - on quitte l'application
            Application.Exit();
        }
        private void Form1_Shown(object sender, System.EventArgs e) {
            // on met le focus sur le champ de saisie
            textBoxSaisie.Focus();
        }
    }
}
```

- ligne 24 : l'événement *[Form].Shown* a lieu lorsque le formulaire est affiché
- ligne 26 : on met alors le focus (pour une saisie) sur le composant *textBoxSaisie*.

- ligne 9 : l'événement `[TextBox].TextChanged` se produit à chaque fois que le contenu d'un composant `TextBox` change
- ligne 11 : on recopie le contenu du composant `[TextBox]` dans le composant `[Label]`
- ligne 14 : gère le clic sur le bouton `[Effacer]`
- ligne 16 : on met la chaîne vide dans le composant `[TextBox]`
- ligne 19 : gère le clic sur le bouton `[Quitter]`
- ligne 21 : pour arrêter l'application en cours d'exécution. On se rappelle que l'objet `Application` sert à lancer l'application dans la méthode `[Main]` de `[Form1.cs]` :

```
static void Main() {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

L'exemple suivant utilise un `TextBox` multilignes :



La liste des contrôles est la suivante :

n°	type	nom	rôle
1	TextBox	textBoxLignes	champ de saisie multilignes
2	TextBox	textBoxLigne	champ de saisie monoligne
3	Button	buttonAjouter	Ajoute le contenu de 2 à 1

Pour qu'un `TextBox` devienne multilignes on positionne les propriétés suivantes du contrôle :

<code>Multiline=true</code>	pour accepter plusieurs lignes de texte
<code>ScrollBars=(None, Horizontal, Vertical, Both)</code>	pour demander à ce que le contrôle ait des barres de défilement (Horizontal, Vertical, Both) ou non (None)
<code>AcceptReturn=(True, False)</code>	si égal à true, la touche Entrée fera passer à la ligne
<code>AcceptTab=(True, False)</code>	si égal à true, la touche Tab générera une tabulation dans le texte

L'application permet de taper des lignes directement dans [1] ou d'en ajouter via [2] et [3].

Le code de l'application est le suivant :

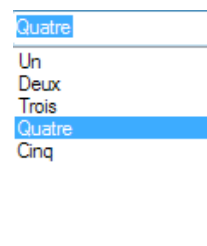
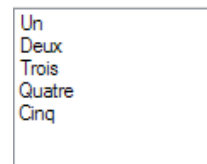
```
using System.Windows.Forms;
using System;
namespace Chap5 {
    public partial class Form1 : Form {
        public Form1() {
            InitializeComponent();
        }
        private void buttonAjouter_Click(object sender, System.EventArgs e) {
            // ajout du contenu de textBoxLigne à celui de textBoxLignes
            textBoxLignes.Text += textBoxLigne.Text+Environment.NewLine;
            textBoxLigne.Text = "";
        }
        private void Form1_Shown(object sender, EventArgs e) {
            // on met le focus sur le champ de saisie
            textBoxLigne.Focus();
        }
    }
}
```

- ligne 18 : lorsque le formulaire est affiché (évt *Shown*), on met le focus sur le champ de saisie *textBoxLigne*
- ligne 10 : gère le clic sur le bouton [Ajouter]
- ligne 12 : le texte du champ de saisie *textBoxLigne* est ajouté au texte du champ de saisie *textBoxLignes* suivi d'un saut de ligne.
- ligne 13 : le champ de saisie *textBoxLigne* est effacé

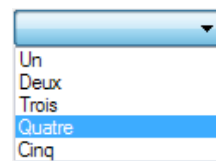
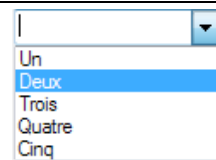
5.2.6 Liste (« ListBox ») et Liste Déroulante (« ComboBox »)

Ces deux types de contrôles permettent de gérer des listes de valeurs sous plusieurs formes.

- Liste non déroulante (ListBox), dans laquelle il est possible de sélectionner une ou plusieurs valeurs :
- Liste non déroulante avec zone d'édition (ComboBox simple), dans laquelle on ne peut saisir qu'une valeur, mais aussi introduire une valeur qui ne se trouve pas dans la liste :



- Liste déroulante avec zone d'édition (ComboBox dropdown), dans laquelle on ne peut saisir qu'une valeur, mais aussi introduire une valeur qui ne se trouve pas dans la liste :
- Liste déroulante sans zone d'édition (l'utilisateur est forcé de choisir une des valeurs proposées ; il ne peut pas en introduire une autre) :



5.2.6.1 Propriétés Applicables Aux Deux Types de Contrôles

Propriété	Description
Items	Les valeurs proposées par le ComboBox. C'est une propriété indexée, <i>Items[i]</i> désignant l'élément <i>i</i> du Combo. L'indexation commence à 0. Cette liste est en lecture seule.
Items.Count	Le nombre de valeurs qui se trouvent dans la liste
SelectedIndex	Index de l'élément sélectionné. Vaut -1 si aucun élément n'est sélectionné.
SelectedItem	L'élément sélectionné. Utiliser « <i>SelectedItem.ToString()</i> » pour obtenir la valeur de l'élément sous forme textuelle. À ne pas confondre avec la propriété « <i>SelectedText</i> » qui contient le texte que l'utilisateur a (peut-être) sélectionné à l'intérieur du champ.

En d'autres termes, l'élément *Item* sélectionné dans la liste *C* est *C.SelectedItem* ou *C.Items[C.SelectedIndex]* où *C.SelectedIndex* est le n° de l'élément sélectionné, ce n° partant de zéro pour le premier élément. Le texte sélectionné peut être obtenu de diverses façons : *C.SelectedItem.Text*, *C.Text*

Note : On peut s'étonner qu'une liste puisse contenir des objets alors que visuellement elle affiche des chaînes de caractères. Si une liste contient un objet *obj*, elle affiche la chaîne *obj.ToString()*. On se rappelle que tout objet a une méthode *ToString* héritée de la classe *object* et qui rend une chaîne de caractères "représentative" de l'objet.

5.2.6.2 Propriétés Applicables Aux ListBox Uniquement

Propriété	Description
SelectionMode	Permet de définir si l'utilisateur peut sélectionner plusieurs éléments de la liste ou non
SelectedIndices	La liste des index des éléments sélectionnés

Propriété	Description
SelectedItems	La liste des éléments sélectionnés

5.2.6.3 Propriétés Applicables Aux ComboBox Uniquement

Propriété	Description
DropDownStyle	Définit le type de ComboBox. Trois valeurs possibles : 1. Simple : liste non déroulante avec zone d'édition 2. DropDown : liste déroulante avec zone d'édition 3. DropDownList : liste déroulante sans zone d'édition Par défaut, le type d'un ComboBox est DropDown

5.2.6.4 Méthodes Applicables Aux Deux Types de Contrôles

Méthode	Description
Items.Add (Item)	Ajoute l'élément Item à la fin de la liste de valeurs
Items.Insert (Item,Pos)	Insère l'élément Item à la position Pos de la liste de valeurs
Items.Remove (Item)	Supprime l'élément Item de la liste de valeur
Items.RemoveAt (Pos)	Supprime l'élément de la liste qui se trouve à l'index Pos
Items.Clear	Supprime toutes les valeurs de la liste
Items.IndexOf (Item)	Retourne la position de Item dans la liste (-1 s'il n'y figure pas)
Items.Contains (Item)	Retourne true si Item existe dans la liste

5.2.6.5 Méthodes Applicables Aux ListBox Uniquement

Méthode	Description
ClearSelected	Désélectionne tous les éléments de la liste

5.2.6.6 Événements Applicables Aux Deux Types de Contrôles

Événement	Est déclenché
SelectedIndexChanged	Lorsque l'élément sélectionné de la liste a changé.
Enter	Lorsque l'utilisateur entre dans le contrôle
KeyPress	Lorsqu'une touche est pressée alors que le contrôle a le focus
TextChanged	Lorsque le contenu du champ a changé. Note : cela signifie que l'événement est déclenché à chaque frappe de touche
Validated	Lorsque l'on quitte le champ après avoir entièrement introduit la valeur au clavier

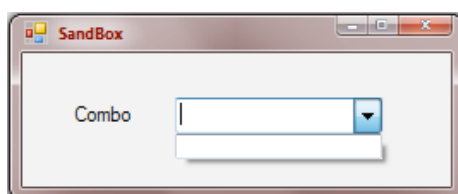
Exemple 1 : ComboBox

Nous créons le formulaire suivant :



n°	type	nom	rôle
1	ComboBox	comboNombres	contient des chaînes de caractères DropDownStyle=DropDownList

Note : à ce stade, nous ne mettons aucune valeur dans la propriété *Items*. Lorsque l'on exécute l'application et que l'on « déroule » la liste, on a :



Lors du choix d'un élément dans la liste déroulante se produit l'événement *SelectedIndexChanged*. Dans l'application, nous utilisons cet événement pour afficher l'élément de la liste qui a été sélectionné.

Le code de l'application est le suivant :

```
using System.Windows.Forms;
namespace Chap5 {
    public partial class Form1 : Form {
        private int previousSelectedIndex=0;

        public Form1() {
            InitializeComponent();
            // remplissage combo
            comboBoxNombres.Items.AddRange(new string[] { "zéro", "un",
                                                            "deux", "trois", "quatre" });

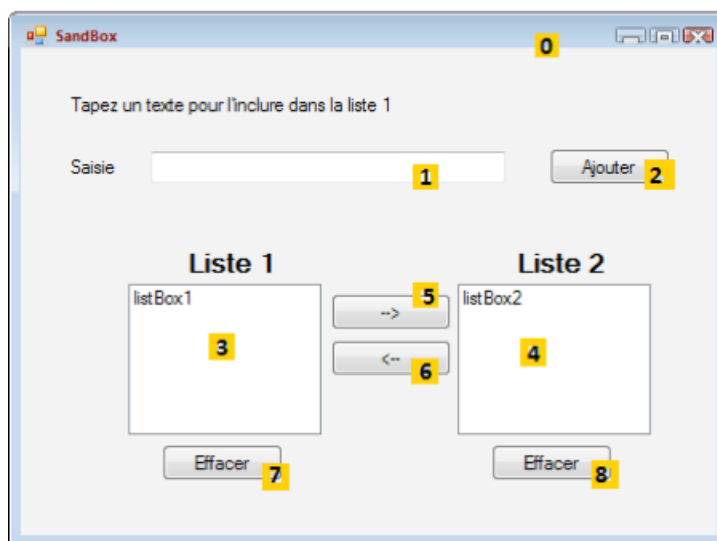
            // sélection élément n° 0
            comboBoxNombres.SelectedIndex = 0;
        }
        private void comboBoxNombres_SelectedIndexChanged(object sender,
                                                         System.EventArgs e) {
            int newSelectedIndex = comboBoxNombres.SelectedIndex;
            if (newSelectedIndex != previousSelectedIndex) {
                // l'élément sélectionné à changé - on l'affiche
                MessageBox.Show(string.Format("Élément sélectionné : ({0},{1})",
                                                comboBoxNombres.Text,
                                                newSelectedIndex),
                                "Combo", MessageBoxButtons.OK,
                                MessageBoxIcon.Information);

                // on note le nouvel index
                previousSelectedIndex = newSelectedIndex;
            }
        }
    }
}
```

- ligne 5 : *previousSelectedIndex* mémorise le dernier index sélectionné dans le combo
- ligne 10 : remplissage du combo avec un tableau de chaînes de caractères
- ligne 12 : le 1er élément est sélectionné
- ligne 15 : la méthode exécutée à chaque fois que l'utilisateur sélectionne un élément du combo. Contrairement à ce que pourrait laisser croire le nom de l'événement, celui-ci a lieu même si l'élément sélectionné est le même que le précédent.
- ligne 16 : on note l'index de l'élément sélectionné
- ligne 17 : s'il est différent du précédent
- ligne 19 : on affiche le n° et le texte de l'élément sélectionné
- ligne 21 : on note le nouvel index

Exemple 2 : ListBox

On se propose de construire l'interface suivante :



Les composants de cette fenêtre sont les suivants :

n°	type	nom	rôle/propriétés
0	Form	Form1	Formulaire FormBorderStyle=FixedSingle (cadre non redimensionnable)
1	TextBox	textBoxSaisie	champ de saisie
2	Button	buttonAjouter	bouton permettant d'ajouter le contenu du champ de saisie [1] dans la liste [3]
3	ListBox	listBox1	liste 1 SelectionMode=MultiExtended
4	ListBox	listBox2	liste 2 SelectionMode=MultiSimple
5	Button	button1vers2	transfère les éléments sélectionnés de liste 1 vers liste 2
6	Button	button2vers1	fait l'inverse
7	Button	buttonEffacer1	vide la liste 1
8	Button	buttonEffacer2	vide la liste 2

Les composants *ListBox* ont un mode de sélection de leurs éléments qui est défini par leur propriété *SelectionMode* :

One	un seul élément peut être sélectionné
MultiExtended	multi-sélection possible : maintenir appuyée la touche SHIFT et cliquer sur un élément étend la sélection de l'élément précédemment sélectionné à

	l'élément courant.
MultiSimple	multi-sélection possible : un élément est sélectionné / désélectionné par un clic de souris ou par appui sur la barre d'espace

Fonctionnement de l'application

- L'utilisateur tape du texte dans le champ 1. Il l'ajoute à la liste 1 avec le bouton *Ajouter* (2). Le champ de saisie (1) est alors vidé et l'utilisateur peut ajouter un nouvel élément.
- Il peut transférer des éléments d'une liste à l'autre en sélectionnant l'élément à transférer dans l'une des listes et en choisissant le bouton de transfert adéquat 5 ou 6. L'élément transféré est ajouté à la fin de la liste de destination et enlevé de la liste source.
- Il peut double-cliquer sur un élément de la liste 1. Cet élément est alors transféré dans la boîte de saisie pour modification et enlevé de la liste 1. Les boutons sont allumés ou éteints selon les règles suivantes :
 - le bouton *Ajouter* n'est allumé que s'il y a un texte non vide dans le champ de saisie
 - le bouton [5] de transfert de la liste 1 vers la liste 2 n'est allumé que s'il y a un élément sélectionné dans la liste 1
 - le bouton [6] de transfert de la liste 2 vers la liste 1 n'est allumé que s'il y a un élément sélectionné dans la liste 2
 - les boutons [7] et [8] d'effacement des listes 1 et 2 ne sont allumés que si la liste à effacer contient des éléments.

Dans les conditions précédentes, tous les boutons doivent être éteints lors du démarrage de l'application. C'est la propriété *Enabled* des boutons qu'il faut alors positionner à *false*. On peut le faire au moment de la conception ce qui aura pour effet de générer le code correspondant dans la méthode *InitializeComponent* ou le faire nous-mêmes dans le constructeur comme ci-dessous :

```
public Form1() {
    InitializeComponent();
    // --- initialisations complémentaires ---
    // on inhibe un certain nombre de boutons
    buttonAjouter.Enabled = false;
    button1vers2.Enabled = false;
    button2vers1.Enabled = false;
    buttonEffacer1.Enabled = false;
    buttonEffacer2.Enabled = false;
}
```

L'état du bouton *Ajouter* est contrôlé par le contenu du champ de saisie. C'est l'événement *TextChanged* qui nous permet de suivre les changements de ce contenu :

```
private void textBoxSaisie_TextChanged(object sender, System.EventArgs e) {  
    // le contenu de textBoxSaisie a changé  
    // le bouton Ajouter n'est allumé que si la saisie est non vide  
    boutonAjouter.Enabled = textBoxSaisie.Text.Trim() != "";  
}
```

L'état des boutons de transfert dépend du fait qu'un élément a été sélectionné ou non dans la liste qu'ils contrôlent :

```
private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e) {  
    // un élément a été sélectionné  
    // on allume le bouton de transfert 1 vers 2  
    bouton1vers2.Enabled = true;  
}  
private void listBox2_SelectedIndexChanged(object sender, System.EventArgs e) {  
    // un élément a été sélectionné  
    // on allume le bouton de transfert 2 vers 1  
    bouton2vers1.Enabled = true;  
}
```

Le code associé au clic sur le bouton *Ajouter* est le suivant :

```
private void boutonAjouter_Click(object sender, System.EventArgs e) {  
    // ajout d'un nouvel élément à la liste 1  
    listBox1.Items.Add(textBoxSaisie.Text.Trim());  
    // raz de la saisie  
    textBoxSaisie.Text = "";  
    // Liste 1 n'est pas vide  
    boutonEffacer1.Enabled = true;  
    // retour du focus sur la boîte de saisie  
    textBoxSaisie.Focus();  
}
```

On notera la méthode *Focus* qui permet de mettre le "focus" sur un contrôle du formulaire.

Le code associé au clic sur les boutons *Effacer* :

```
private void buttonEffacer1_Click(object sender, System.EventArgs e) {
    // on efface la liste 1
    listBox1.Items.Clear();
    // bouton Effacer
    buttonEffacer1.Enabled = false;
}
private void buttonEffacer2_Click(object sender, System.EventArgs e) {
    // on efface la liste 2
    listBox2.Items.Clear();
    // bouton Effacer
    buttonEffacer2.Enabled = false;
}
```

Le code de transfert des éléments sélectionnés d'une liste vers l'autre :

```
private void button1vers2_Click(object sender, System.EventArgs e) {
    // transfert de l'élément sélectionné dans Liste 1 dans Liste 2
    transfert(listBox1, button1vers2, buttonEffacer1, listBox2, button2vers1,
    buttonEffacer2);
}
private void button2vers1_Click(object sender, System.EventArgs e) {
    // transfert de l'élément sélectionné dans Liste 2 dans Liste 1
    transfert(listBox2, button2vers1, buttonEffacer2, listBox1, button1vers2,
    buttonEffacer1);
}
```

Les deux méthodes ci-dessus délèguent le transfert des éléments sélectionnés d'une liste à l'autre à une même méthode privée appelée **transfert** :

```
// transfert
private void transfert(ListBox l1, Button button1vers2, Button buttonEffacer1,
    ListBox l2, Button button2vers1, Button buttonEffacer2) {
    // transfert dans la liste l2 des éléments sélectionnés de la liste l1
    for (int i = l1.SelectedIndices.Count - 1; i >= 0; i--) {
        // index de l'élément sélectionné
        int index = l1.SelectedIndices[i];
        // ajout dans l2
        l2.Items.Add(l1.Items[index]);
        // suppression dans l1
        l1.Items.RemoveAt(index);
    }
    // boutons Effacer
    buttonEffacer2.Enabled = l2.Items.Count != 0;
    buttonEffacer1.Enabled = l1.Items.Count != 0;
    // boutons de transfert
    button1vers2.Enabled = false;
}
```

Notes :

- La méthode **transfert** reçoit six paramètres :
 - une référence sur la liste contenant les éléments sélectionnés appelée ici **l1**. Lors de l'exécution de l'application, l1 est soit *listBox1* soit *listBox2*. On voit des exemples d'appel, lignes 3 et 8 des procédures de transfert *buttonXversY_Click*.
 - une référence sur le bouton de transfert lié à la liste **l1**. Par exemple si **l1** est *listBox2*, ce sera *button2vers1* (cf appel ligne 8)
 - une référence sur le bouton d'effacement de la liste l1. Par exemple si l1 est *listBox1*, ce sera *buttonEffacer1* (cf appel ligne 3)
 - les trois autres références sont analogues mais font référence à la liste **l2**.
- La collection [ListBox].SelectedIndices représente les indices des éléments sélectionnés dans le composant [ListBox]. C'est une collection :
 - [ListBox].SelectedIndices.Count est le nombre d'élément de cette collection
 - [ListBox].SelectedIndices[i] est l'élément n° i de cette collection

On parcourt la collection en sens inverse : on commence par la fin de la collection pour terminer par le début. Nous expliquerons pourquoi.
- entier *index* : indice d'un élément sélectionné de la liste **l1**
- cet élément est ajouté dans la liste **l2**
- ligne j : l'élément est supprimé de la liste **l1**. Parce qu'il est supprimé, il n'est plus sélectionné. La collection **l1.SelectedIndices** va être recalculée. Elle va perdre l'élément qui vient d'être supprimé. Tous les éléments qui sont après celui-ci vont voir leur n° passer de n à n-1.
 - si la boucle de la était croissante et qu'elle venait de traiter l'élément n° 0, elle va ensuite traiter l'élément n° 1. Or l'élément qui portait le n° 1 avant la suppression de l'élément n° 0, va ensuite porter le n° 0. Il sera alors oublié par la boucle.
 - si la boucle est décroissante et qu'elle vient de traiter l'élément n° n, elle va ensuite traiter l'élément n° n-1. Après suppression de l'élément n° n, l'élément n° n-1 ne change pas de n°. Il est donc traité au tour de boucle suivant.
- l'état des boutons [Effacer] dépend de la présence ou non d'éléments dans les listes associées
- si la liste **l2** n'a plus d'éléments sélectionnés : on éteint son bouton de transfert.

5.2.7 Case à Cocher (« CheckBox ») et Bouton Radio (« RadioButton »)

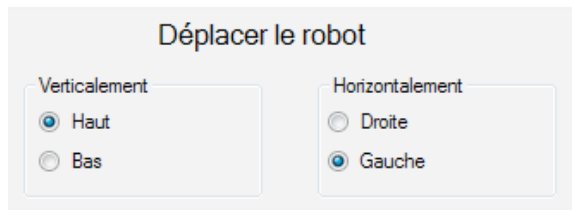
Ces contrôles permettent à l'utilisateur de faire des choix en cochant des cases. La différence avec les listes vues précédemment est qu'ici la liste de possibilités parmi lesquelles l'utilisateur va choisir n'est pas modifiable.

On distingue deux cas de figure :

1. Les cases à cocher où les choix sont indépendants les uns des autres

Vérifier: ☒ Orthographe ☒ Grammaire ☐ Mise en page

2. Les boutons radio où les choix sont mutuellement exclusifs :



Un seul bouton radio d'un groupe peut être sélectionné à la fois. Sélectionner un bouton a pour effet de désélectionner tout autre bouton du groupe. Si l'on a besoin de plusieurs groupes (comme dans l'exemple ci-dessus), on place les boutons à l'intérieur d'un conteneur « GroupBox ». Note : le formulaire fait office de GroupBox ; il n'est donc pas nécessaire (mais malgré tout possible) de placer les boutons radio dans un groupe s'il n'y a qu'un seul groupe.

5.2.7.1 Propriétés Spécifiques

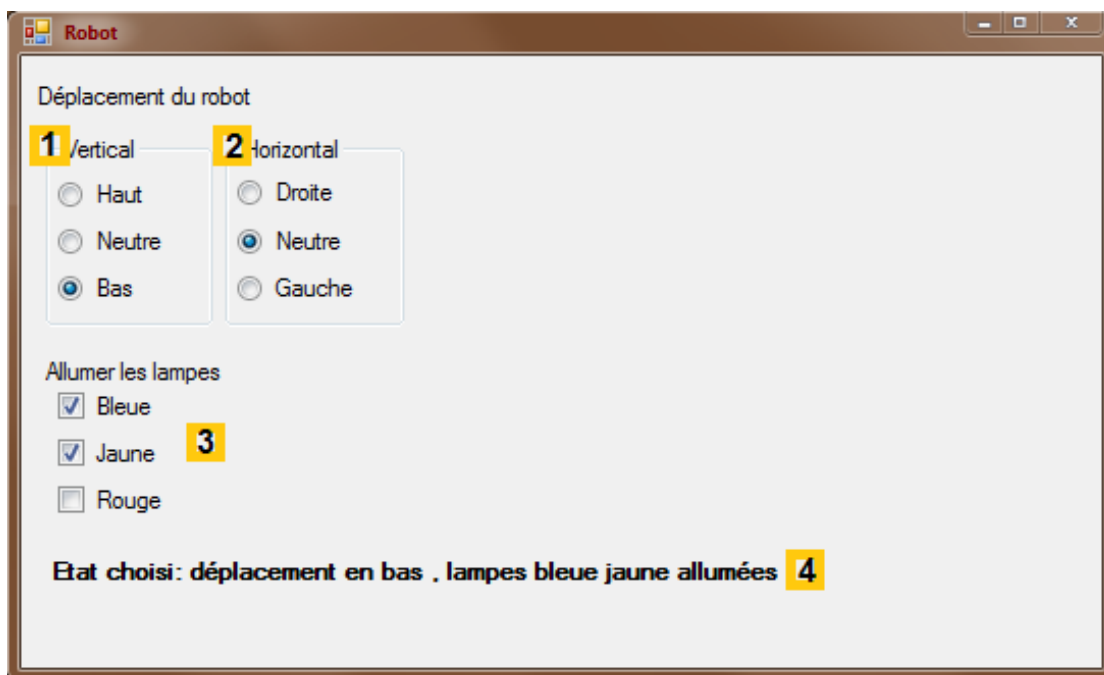
Propriété	Description
Checked	Booléen qui dit si la case ou le bouton est coché

5.2.7.2 Événements Spécifiques

Événement	Est déclenché
CheckedChanged	Lorsque la case ou le bouton change d'état. Note : étant donné le caractère binaire de ces contrôles, l'événement Click peut également être utilisé.

Exemple

Nous nous proposons d'écrire l'application suivante :



Les composants de la fenêtre sont les suivants :

n°	type	nom	rôle/propriétés
1, 2	GroupBox	grpBox1, grpBox2	Deux conteneurs de composants. On peut y déposer d'autres composants.
	RadioButton	rbtVHaut, rbtVNeutre, rbtVBas, rbthDroite, rbtHNeutre, rbtHGauche	6 boutons radio Des boutons radio présents dans un même conteneur, ici le GroupBox, sont exclusifs l'un de l'autre : seul l'un d'entre-eux est allumé.
3	CheckBox	chkBleu, chkJaune, chkRouge	3 cases à cocher, aucune n'est cochée au départ
4	Label	lblComportement	Un label dans lequel on va interpréter les boutons et cases sous forme de texte

L'événement qui nous intéresse pour ces contrôles est l'événement *CheckChanged* indiquant que l'état de la case à cocher ou du bouton radio a changé. Cet état est représenté dans les deux cas par la propriété booléenne *Checked* qui à **vrai** signifie que le contrôle est coché.

Le code de notre application est :

```
private void Afficher()
{
    string res = "Etat choisi: "; // la variable res contient le texte à afficher à
    la fin
    if (rbtVNeutre.Checked && rbtHNeutre.Checked) // aucun mouvement
        res = res + "pas de déplacement ,";
    else
    {
        res = res + "déplacement ";
        if (rbtVHaut.Checked) res = res + "en haut ";
        if (rbtVBas.Checked) res = res + "en bas ";
        if (rbtHDroite.Checked) res = res + "à droite ";
        if (rbtHGauche.Checked) res = res + "à gauche ";
        res = res + ", ";
    }

    string lampes = "lampes ";
    if (chkBleu.Checked) lampes = lampes + "bleue ";
    if (chkJaune.Checked) lampes = lampes + "jaune ";
    if (chkRouge.Checked) lampes = lampes + "rouge ";
    if (lampes == "lampes ") // pas de changement -> pas de lampe allumée
        lampes = "pas de lampes ";
    res = res + lampes + "allumées";
    lblComportement.Text = res;
}

private void rbtVHaut_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

private void rbtVNeutre_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

...
```

Notes :

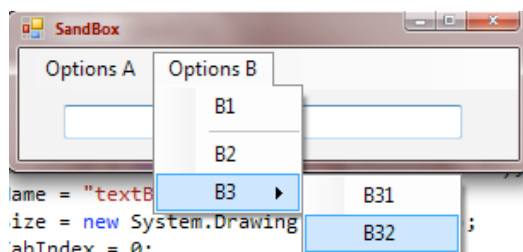
- On remarque l'utilisation d'une méthode privée (=fonction) *Afficher()*. On fait cela car on va réécrire l'entier du status à chaque fois que l'un des boutons/cases change d'état. Ainsi, le

gestionnaire de l'événement « CheckedChange » de chacun des 9 boutons/cases ne fait qu'une chose : appeler Afficher()

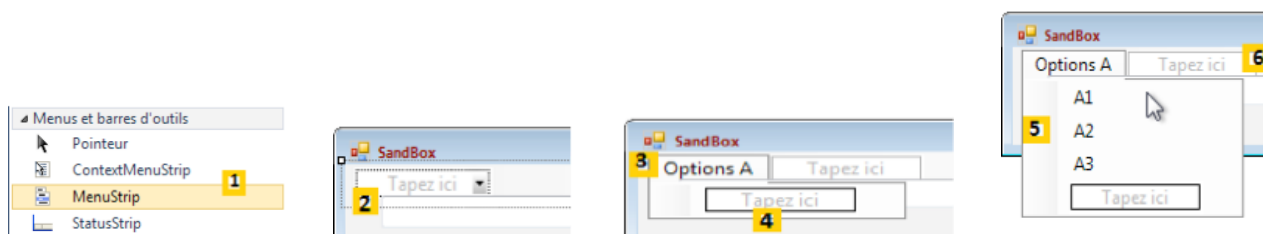
- Les « ... » remplace les gestionnaires d'événements des autres boutons/cases qui sont tous identiques aux deux premiers.

5.2.8 Menus

Voyons maintenant comment créer une fenêtre avec menu. Nous allons créer la fenêtre suivante :



Pour créer un menu, on choisit le composant "MenuStrip" dans la barre "Menus & Tollbars" :

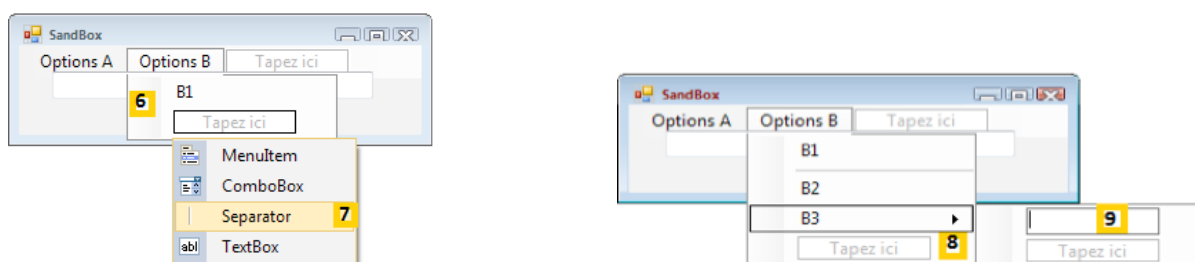


[1] : choix du composant [MenuStrip]

[2] : on a alors un menu qui s'installe sur le formulaire avec des cases vides intitulées "Tapez ici". Il suffit d'y indiquer les différentes options du menu.

[3] : le libellé "Options A" a été tapé. On passe au libellé [4]

[5] : les libellés des options A ont été saisis. On passe au libellé [6]



[6] : les premières options B

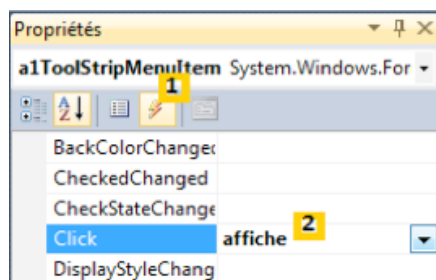
[7] : sous B1, on met un séparateur. Celui-ci est disponible dans un combo associé au texte "Type Here"

[8] : pour faire un sous-menu, utiliser la flèche [8] et taper le sous-menu dans [9]

Il reste à nommer les différents composants du formulaire :

n°	type	nom	rôle/propriétés
1	Label	labelStatut	pour afficher le texte de l'option de menu cliquée
2	toolStripMenuItem	toolStripMenuItemOptionsA toolStripMenuItemA1 toolStripMenuItemA2 toolStripMenuItemA3	options de menu sous l'option principale "Options A"
3	toolStripMenuItem	toolStripMenuItemOptionsB toolStripMenuItemB1 toolStripMenuItemB2 toolStripMenuItemB3	options de menu sous l'option principale "Options B"
4	toolStripMenuItem	toolStripMenuItemB31 toolStripMenuItemB32	options de menu sous l'option principale "B3"

Dans la structure du menu, sélectionnons l'option A1 et cliquons droit pour avoir accès aux propriétés du contrôle :



Dans l'onglet *événements* [1], on associe la méthode *affiche* [2] à l'événement *Click*. Cela signifie que l'on souhaite que le clic sur l'option A1 soit traité par une méthode appelée *affiche*. Visual studio génère automatiquement la méthode *affiche* dans la fenêtre de code :

```
private void affiche(object sender, EventArgs e) {
}
```

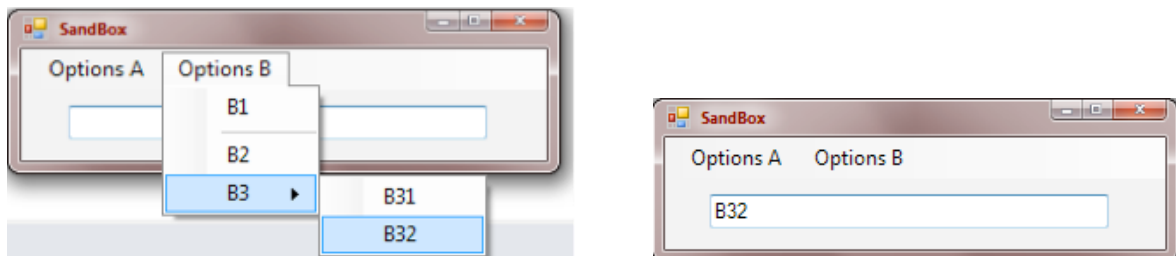
Dans cette méthode, nous nous contenterons d'afficher dans le label *labelStatut* la propriété *Text* de l'option de menu qui a été cliquée :

```
private void affiche(object sender, EventArgs e) {
    // affiche dans le TextBox le nom du sous-menu choisi
    labelStatut.Text = ((ToolStripMenuItem)sender).Text;
}
```

La source de l'événement *sender* est de type *object*. Les options de menu sont elle de type *ToolStripMenuItem*, aussi est-on obligé de faire un transtypage de *object* vers *ToolStripMenuItem*.

Pour toutes les options de menu, on fixe le gestionnaire du clic à la méthode *affiche*.

Exécutons l'application et sélectionnons un élément de menu :

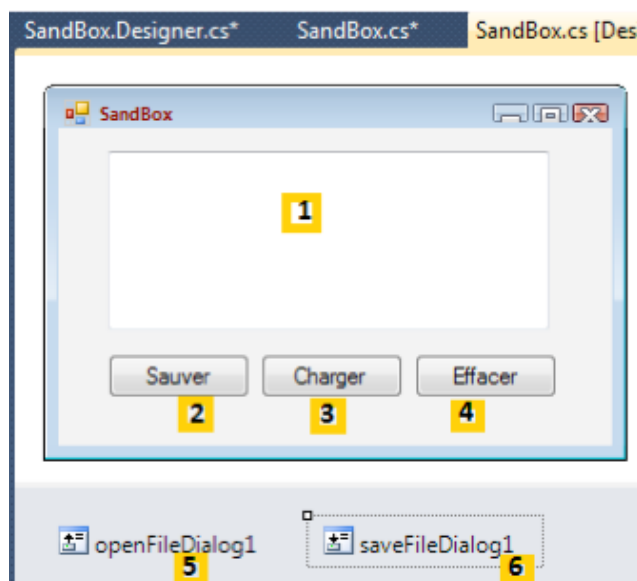


5.3 Composants non visuels

Nous nous intéressons maintenant à un certain nombre de composants non visuels : on les utilise lors de la conception mais on ne les voit pas lors de l'exécution.

5.3.1 Boîtes de dialogue OpenFileDialog et SaveFileDialog

Nous allons construire l'application suivante :



Les contrôles sont les suivants :

n°	type	nom	rôle/propriétés
1	TextBox	TextBoxLignes	texte tapé par l'utilisateur ou chargé à partir d'un fichier MultiLine=True, ScrollBars=Both, AcceptReturn=True, AcceptTab=True
2	Button	buttonSauvegarder	permet de sauvegarder le texte de [1] dans un fichier texte
3	Button	buttonCharger	permet de charger le contenu d'un fichier texte dans [1]
4	Button	buttonEffacer	efface le contenu de [1]
5	SaveFileDialog	saveFileDialog1	composant permettant de choisir le nom et l'emplacement du fichier de sauvegarde de [1]. Ce composant est pris dans la barre d'outils, sous « Boîtes de dialogue », et simplement déposé sur le formulaire. Il est alors enregistré mais il n'occupe pas de place sur le formulaire. C'est un composant non visuel
6	OpenFileDialog	openFileDialog1	composant permettant de choisir le fichier à charger dans [1]

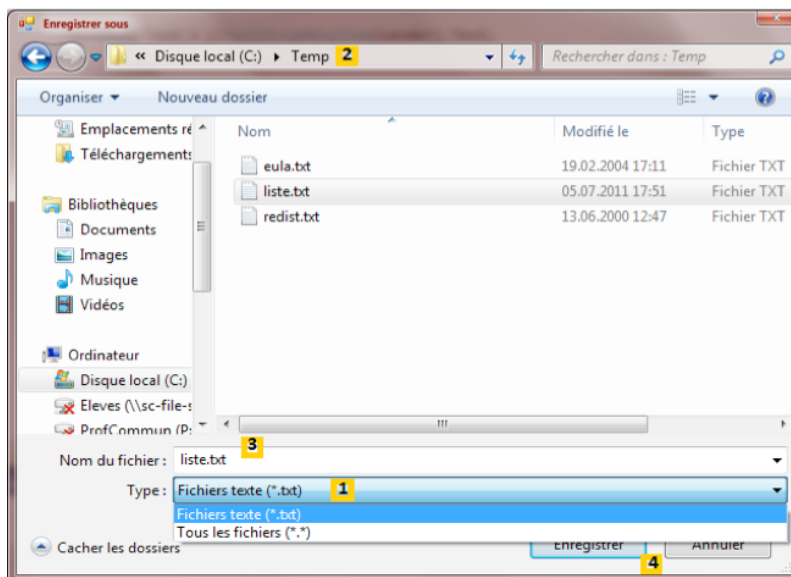
Le code associé au bouton *Effacer* est simple :

```
private void buttonEffacer_Click(object sender, EventArgs e) {
    // on met la chaîne vide dans le TextBox
    textBoxLignes.Text = "";
}
```

Nous utiliserons les propriétés et méthodes suivantes de la classe **SaveFileDialog** :

Champ	Type	Rôle
string Filter	Propriété	Les types de fichiers proposés dans la liste déroulante des types de fichiers de la boîte de dialogue
int FilterIndex	Propriété	le n° du type de fichier proposé par défaut dans la liste ci-dessus. Commence à 0
string InitialDirectory	Propriété	Le dossier présenté initialement pour la sauvegarde du fichier
string FileName	Propriété	le nom du fichier de sauvegarde indiqué par l'utilisateur
DialogResult.ShowDialog()	Méthode	méthode qui affiche la boîte de dialogue de sauvegarde. Rend un résultat de type DialogResult

La méthode *ShowDialog* affiche une boîte de dialogue analogue à la suivante :



1. liste déroulante construite à partir de la propriété **Filter**. Le type de fichier proposé par défaut est fixé par **FilterIndex**
2. dossier courant, fixé par **InitialDirectory** si cette propriété a été renseignée
3. nom du fichier choisi ou tapé directement par l'utilisateur. Sera disponible dans la propriété **FileName**
4. boutons **Enregistrer/Annuler**. Si le bouton *Enregistrer* est utilisé, la fonction *ShowDialog* rend le résultat **DialogResult.OK**

La procédure de sauvegarde peut s'écrire ainsi :

```
private void buttonSauvegarder_Click(object sender, System.EventArgs e) {
    // on sauvegarde la boîte de saisie dans un fichier texte
    // on paramètre la boîte de dialogue saveFileDialog1
    saveFileDialog1.InitialDirectory = Application.ExecutablePath;
    saveFileDialog1.Filter =
        "Fichiers texte (*.txt)|*.txt|Tous les fichiers (*.*)|*.*";
    saveFileDialog1.FilterIndex = 0;
    // on affiche la boîte de dialogue et on récupère son résultat
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        // on récupère le nom du fichier
        string nomFichier = saveFileDialog1.FileName;
        StreamWriter fichier = null;
        try
        {
            // on ouvre le fichier en écriture
            fichier = new StreamWriter(nomFichier);
            // on écrit le texte dedans
            fichier.Write(textBoxLignes.Text);
        }
        catch (Exception ex)
        {
            // problème
            MessageBox.Show("Problème à l'écriture du fichier (" +
                ex.Message + ")", "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
        }
        finally
        {
            // on ferme le fichier
            if (fichier != null) fichier.Dispose();
        }
    }
}
```

Notes

- On fixe le dossier initial (*InitialDirectory*) au dossier (*Application.ExecutablePath*) qui contient l'exécutable de l'application.
- On fixe les types de fichiers à présenter. On notera la syntaxe des filtres:
filtre1|filtre2|..|filtren avec *filtrei= Texte| modèle de fichier*.
Ici l'utilisateur aura le choix entre les fichiers **.txt* et **.**.
- On fixe le type de fichier à présenter en premier à l'utilisateur. Ici l'index 0 désigne les fichiers **.txt*.
- La boîte de dialogue est affichée et son résultat récupéré. Pendant que la boîte de dialogue est affichée, l'utilisateur n'a plus accès au formulaire principal (boîte de dialogue dite modale).

L'utilisateur fixe le nom du fichier à sauvegarder et quitte la boîte soit par le bouton *Enregistrer*, soit par le bouton *Annuler*, soit en fermant la boîte. Le résultat de la méthode *ShowDialog* est *DialogResult.OK* uniquement si l'utilisateur a utilisé le bouton *Enregistrer* pour quitter la boîte de dialogue.

- Ceci fait, le nom du fichier à créer est maintenant dans la propriété *FileName* de l'objet *saveFileDialog1*. On est alors ramené à la création classique d'un fichier texte. On y écrit le contenu du *TextBox* : *textBoxLignes.Text* tout en gérant les exceptions qui peuvent se produire.

La classe **OpenFileDialog** est très proche de la classe *SaveFileDialog*. On utilisera les mêmes méthodes et propriétés que précédemment. La méthode *ShowDialog* affiche une boîte de dialogue analogue à la précédente.

La procédure de chargement du fichier texte peut s'écrire ainsi :

```
private void buttonCharger_Click(object sender, EventArgs e) {
    // on charge un fichier texte dans la boîte de saisie
    // on paramètre la boîte de dialogue openFileDialog1
    openFileDialog1.InitialDirectory = Application.ExecutablePath;
    openFileDialog1.Filter =
        "Fichiers texte (*.txt)|*.txt|Tous les fichiers (*.*)|*.*";
    openFileDialog1.FilterIndex = 0;
    // on affiche la boîte de dialogue et on récupère son résultat
    if (openFileDialog1.ShowDialog() == DialogResult.OK) {
        // on récupère le nom du fichier
        string nomFichier = openFileDialog1.FileName;
        StreamReader fichier = null;
        try
        {
            // on ouvre le fichier en lecture
            fichier = new StreamReader(nomFichier);
            // on lit tout le fichier et on le met dans le TextBox
            textBoxLignes.Text = fichier.ReadToEnd();
        }
        catch (Exception ex)
        {
            // problème
            MessageBox.Show("Problème à la lecture du fichier (" +
                ex.Message + ")", "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
        }
        finally
        {
            // on ferme le fichier
            if (fichier != null) fichier.Dispose();
        }
    }
}
```

Notes

- La boîte de dialogue est affichée et son résultat récupéré. Pendant que la boîte de dialogue est affichée, l'utilisateur n'a plus accès au formulaire principal (boîte de dialogue dite modale). L'utilisateur fixe le nom du fichier à charger et quitte la boîte soit par le bouton *Ouvrir*, soit par le bouton *Annuler*, soit en fermant la boîte. Le résultat de la méthode *ShowDialog* est *DialogResult.OK* uniquement si l'utilisateur a utilisé le bouton *Enregistrer* pour quitter la boîte de dialogue.
- Ceci fait, le nom du fichier à charger est maintenant dans la propriété *FileName* de l'objet *openFileDialog1*. On est alors ramené à la lecture classique d'un fichier texte. On notera, ligne 16, la méthode qui permet de lire la totalité d'un fichier.

5.3.2 Timer

Un composant Timer est un contrôle non visible du formulaire qui déclenche un événement *Tick* à intervalle régulier.

5.3.2.1 Propriétés Spécifiques

Propriété	Description
Enabled	Active / Désactive le timer. Quand le timer est désactivé, l'événement Tick ne se produit plus. Lorsque l'on active le timer, il démarre également.
Interval	Le nombre de millisecondes entre deux événements Tick

5.3.2.2 Méthodes Spécifiques

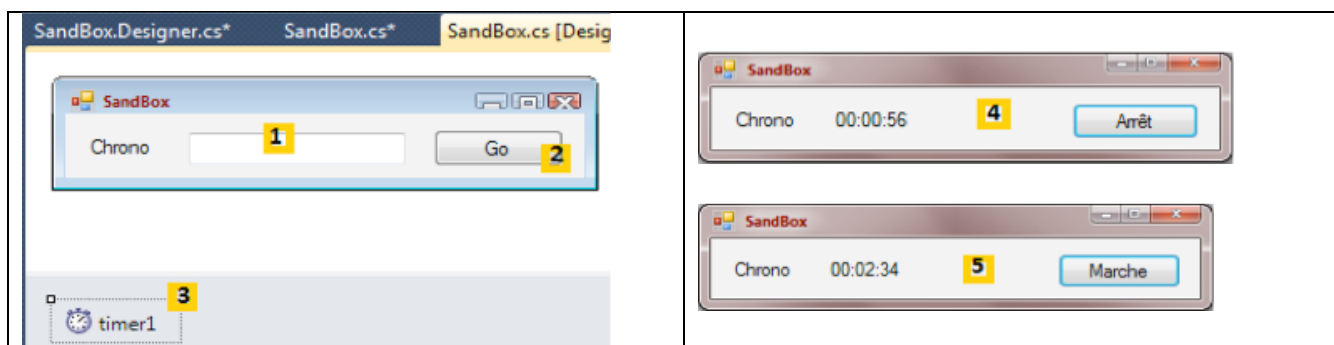
Méthode	Description
Start, Stop	Démarre / Arrête le timer. L'intervalle recommence au début : si l'intervalle est de 10 secondes, qu'on le stoppe et redémarre après 7 secondes, l'événement Tick interviendra après 10 secondes, pas 3.

5.3.2.3 Événements Spécifiques

Événement	Est déclenché
Tick	Tout les interval millisecondes

Exemple

Nous nous proposons ici d'écrire l'application suivante :



n°	type	nom	rôle/propriétés
1	Label	labelChrono	affiche un chronomètre
2	Button	buttonArretMarche	bouton Arrêt/Marche du chronomètre
3	Timer	timer1	composant émettant ici un événement toutes les secondes

En [4], nous voyons le chronomètre en marche, en [5] le chronomètre arrêté.

Pour changer toutes les secondes le contenu du Label *LabelChrono*, il nous faut un composant qui génère un événement toutes les secondes, événement qu'on pourra intercepter pour mettre à jour l'affichage du chronomètre. Ce composant c'est le *Timer* [1] disponible dans la boîte à outils *Composants*.

Dans notre exemple le timer s'appelle *timer1* et *timer1.Interval* est mis à 1000 ms (1s). L'événement *Tick* se produira donc toutes les secondes. Le clic sur le bouton Arrêt/Marche est traité par la procédure *buttonArretMarche_Click* suivante :

```
// variable d'instance
private DateTime début = DateTime.Now;

private void buttonArretMarche_Click(object sender, EventArgs e) {
    // arrêt ou marche ?
    if (buttonArretMarche.Text == "Marche")
    {
        début = DateTime.Now; // on note l'heure de début
        labelChrono.Text = "00:00:00"; // on l'affiche
        timer1.Enabled = true; // on lance le timer
        buttonArretMarche.Text = "Arrêt"; // on change le libellé du bouton
    }
    else
    {
        timer1.Enabled = false; // arrêt du timer
        buttonArretMarche.Text = "Marche"; // on change le libellé du bouton
    }
}
```

Notes

- Le libellé du bouton Arrêt/Marche est soit "Arrêt" soit "Marche". On fait donc un test sur ce libellé pour savoir quoi faire.
- Dans le cas de "Marche", on note l'heure de début dans une variable *début* qui est une variable globale de l'objet formulaire
- Dans tout autre situation, on passe le libellé du bouton à "Marche" et on force l'arrêt du timer.

Il nous reste à traiter l'événement *Tick* sur l'objet *timer1*, événement qui se produit toutes les secondes :

```
private void timer1_Tick(object sender, EventArgs e) {
    // une seconde s'est écoulée
    DateTime maintenant = DateTime.Now;
    TimeSpan durée = maintenant - début;
    // on met à jour le chronomètre
    labelChrono.Text = durée.Hours.ToString("d2") + ":" +
        durée.Minutes.ToString("d2") + ":" +
        durée.Seconds.ToString("d2");
}
```

Notes

- ligne 3 : on note l'heure du moment
- ligne 4 : on calcule le temps écoulé depuis l'heure de lancement du chronomètre. On obtient un objet de type *TimeSpan* qui représente une durée dans le temps.
- ligne 6 : celle-ci doit être affichée dans le chronomètre sous la forme *hh:mm:ss*. Pour cela nous utilisons les propriétés *Hours*, *Minutes*, *Seconds* de l'objet *TimeSpan* qui représentent

respectivement les heures, minutes, secondes de la durée que nous affichons au format `ToString("d2")` pour avoir un affichage sur 2 chiffres.

5.4 Événements souris

Lorsqu'on dessine dans un conteneur, il est important de connaître la position de la souris pour, par exemple, afficher un point lors d'un clic. Les déplacements de la souris provoquent des événements dans le conteneur dans lequel elle se déplace.

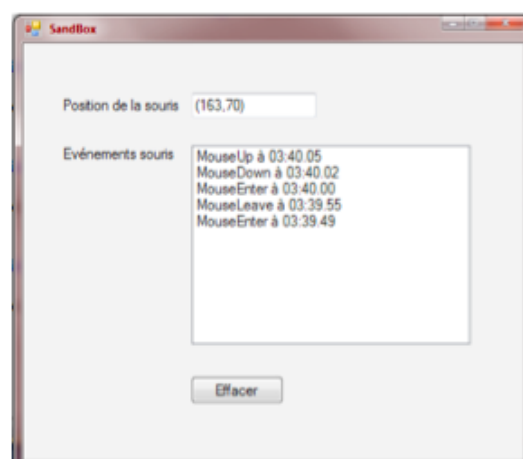
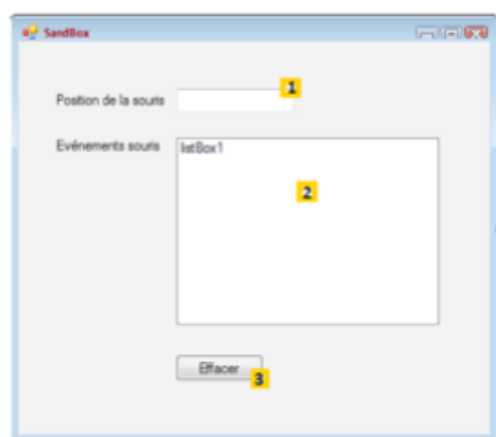


[1] : les événements survenant lors d'un déplacement de la souris sur le formulaire ou sur un contrôle

[2] : les événements survenant lors d'un glisser / lâcher (Drag'nDrop)

Événement	Description
MouseEnter	la souris vient d'entrer dans le domaine du contrôle
MouseLeave	la souris vient de quitter le domaine du contrôle
MouseMove	la souris bouge dans le domaine du contrôle
MouseDown	Pression sur le bouton gauche de la souris
MouseUp	Relâchement du bouton gauche de la souris
DragDrop	l'utilisateur lâche un objet sur le contrôle
DragEnter	l'utilisateur entre dans le domaine du contrôle en tirant un objet
DragLeave	l'utilisateur sort du domaine du contrôle en tirant un objet
DragOver	l'utilisateur passe au-dessus domaine du contrôle en tirant un objet

Voici une application permettant de mieux appréhender à quels moments se produisent les différents événements souris :

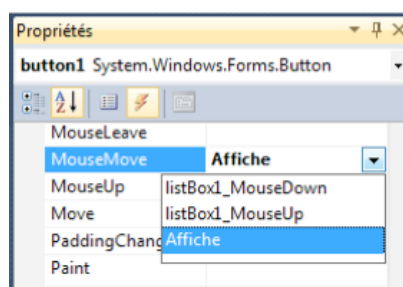


n°	type	nom	rôle/propriétés
1	Label	lblPositionSouris	pour afficher la position de la souris dans le formulaire 1, la liste 2 ou le bouton 3
2	ListBox	listBoxEvts	pour afficher les évts souris autres que MouseMove
3	Button	buttonEffacer	pour effacer le contenu de 2

Pour suivre les déplacements de la souris sur les trois contrôles, on n'écrit qu'un seul gestionnaire, le gestionnaire *affiche* :

```
private void affiche(object sender, MouseEventArgs e) {
    // mvt souris - on affiche les coordonnées (X,Y) de celle-ci
    lblPositionSouris.Text = "(" + e.X + "," + e.Y + ")";
}
```

On désigne ce code comme gestionnaire de l'événement « MouseMove » des trois contrôles que sont le formulaire, la listbox et le bouton.



Note : A chaque fois que la souris entre dans le domaine d'un contrôle son système de coordonnées change. Son origine (0,0) est le coin supérieur gauche du contrôle sur lequel elle se trouve. Ainsi à l'exécution, lorsqu'on passe la souris du formulaire au bouton, on voit clairement le changement de coordonnées.

Pour détecter les entrées et sorties de la souris sur la liste 2, nous traitons les événements *MouseEnter* et *MouseLeave* de cette même liste :

```
private void listBoxEvts_MouseEnter(object sender, System.EventArgs e) {
    // on signale l'évt
    listBoxEvts.Items.Insert(0, string.Format("MouseEnter à {0:hh:mm:ss}",
                                                DateTime.Now));
}

private void listBoxEvts_MouseLeave(object sender, EventArgs e) {
    // on signale l'évt
    listBoxEvts.Items.Insert(0, string.Format("MouseLeave à {0:hh:mm:ss}",
                                                DateTime.Now));
}
```

Pour traiter les clics sur le formulaire, nous traitons les événements *MouseDown* et *MouseUp* :

```
private void listBoxEvts_MouseDown(object sender, MouseEventArgs e) {
    // on signale l'évt
    listBoxEvts.Items.Insert(0, string.Format("MouseDown à {0:hh:mm:ss}",
                                                DateTime.Now));
}

private void listBoxEvts_MouseUp(object sender, MouseEventArgs e) {
    // on signale l'évt
    listBoxEvts.Items.Insert(0, string.Format("MouseUp à {0:hh:mm:ss}", DateTime.Now));
}
```

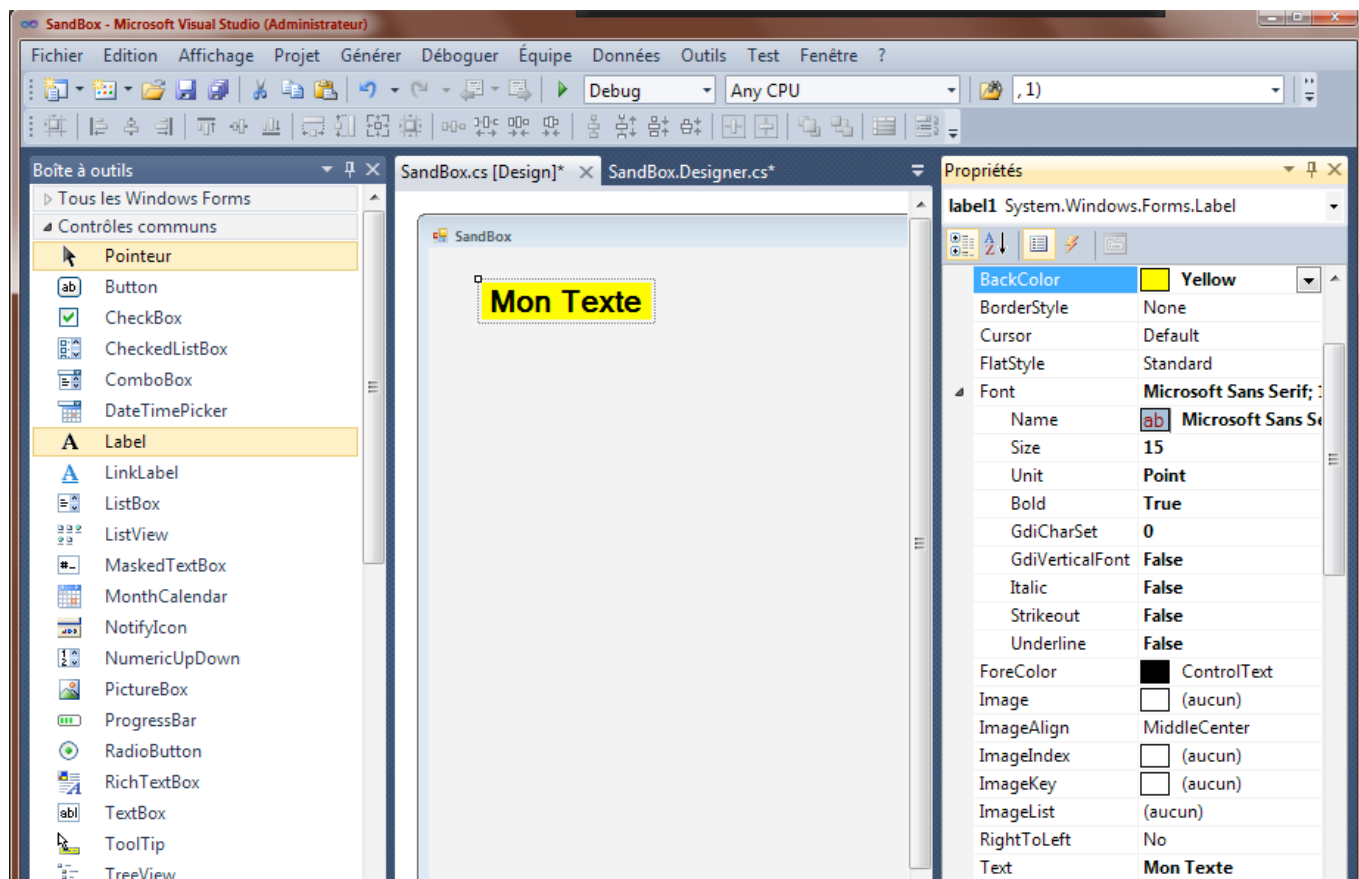
lignes 3 et 8 : les messages sont placés en 1ère position dans le *ListBox* afin que les événements les plus récents soient les premiers dans la liste.

Enfin, le code du gestionnaire de clic sur le bouton *Effacer* :

```
private void buttonEffacer_Click(object sender, EventArgs e) {
    listBoxEvts.Items.Clear();
}
```

5.5 Ajout Dynamique de Composants

Jusqu'ici, tous les composants ont été mis sur le formulaire de manière graphique avec le Concepteur Windows Forms de Visual Studio. En d'autres termes, nous avons sélectionné des composants dans la boîte à outil, nous les avons placés et dimensionnés sur le formulaire au moyen de la souris, puis nous avons modifié certaines caractéristiques dans la feuille de propriétés.

Exemple :

On a « pris » un Label dans la boîte à outil, déposé en haut à gauche du formulaire, puis modifié ses propriétés BackColor (Yellow), Font.Size (15), Font.Bold (True), Text (« Mon Texte »).

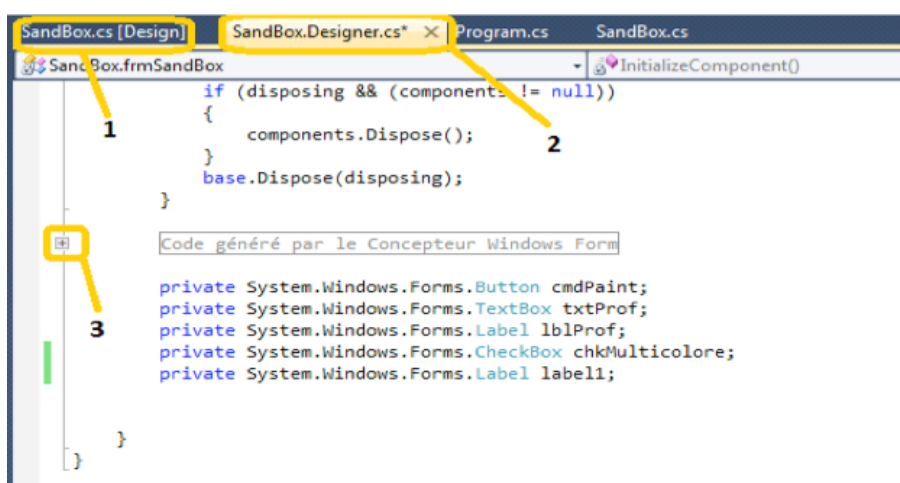
En C#, il est possible de modifier le contenu d'un formulaire durant l'exécution du programme. La marche à suivre pour faire cela consiste à :

1. Créer un objet de la classe correspondant au type de contrôle désiré
2. Modifier les caractéristiques de l'objet
3. Ajouter cet objet à la collection de contrôles du formulaire

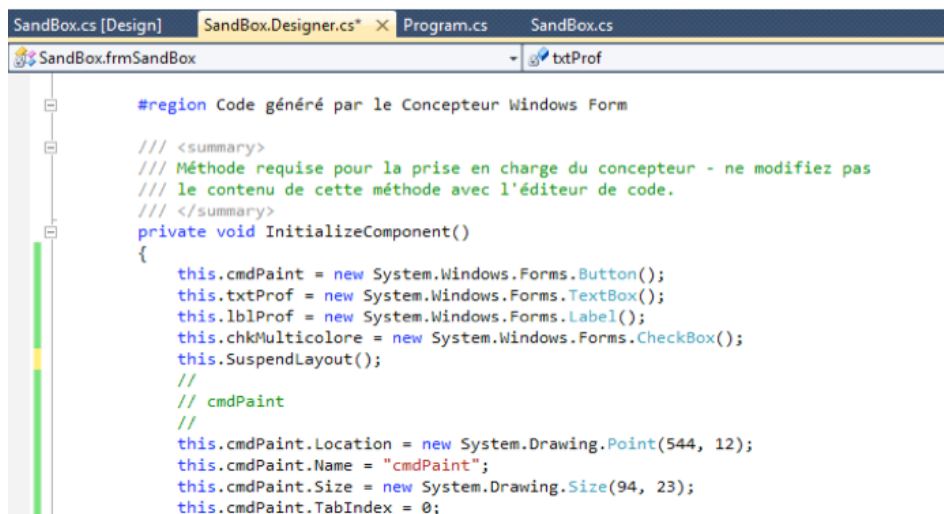
L'exemple ci-dessus peut ainsi être remplacé par le code :

```
Control label1 = new System.Windows.Forms.Label(); // Création d'un contrôle Label
label1.BackColor = System.Drawing.Color.Yellow; // Couleur de fond
label1.Font = new System.Drawing.Font("Microsoft Sans Serif", // Police
    15F,
    System.Drawing.FontStyle.Bold,
    System.Drawing.GraphicsUnit.Point,
    (byte)0));
label1.Location = new System.Drawing.Point(36, 24); // Position
label1.Name = "label1"; // Nom
label1.Size = new System.Drawing.Size(115, 25); // Dimensions
label1.Text = "Mon Texte"; // Texte
Controls.Add(label1); // Ajout au formulaire
```

Note : Un bon moyen pour attaquer la création dynamique de contrôles consiste à créer un contrôle dans le concepteur, puis d'aller voir le code que Visual Studio a automatiquement généré pour ce contrôle .



La conception graphique de votre formulaire se fait dans l'onglet nommé « ...[Design] » (1). L'onglet « ...Designer.cs » (2) contient du code généré par Visual Studio et qui n'a généralement pas besoin d'être modifié. Lorsque vous ouvrez la région « Code généré par le concepteur Windows Form » en cliquant sur (3), vous obtenez :



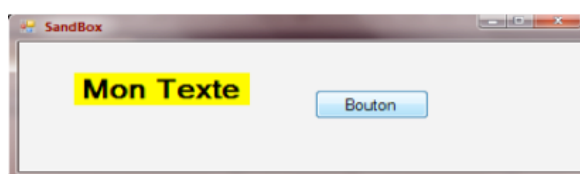
```
#region Code généré par le Concepteur Windows Form
/// <summary>
/// Méthode requise pour la prise en charge du concepteur - ne modifiez pas
/// le contenu de cette méthode avec l'éditeur de code.
/// </summary>
private void InitializeComponent()
{
    this.cmdPaint = new System.Windows.Forms.Button();
    this.txtProf = new System.Windows.Forms.TextBox();
    this.lblProf = new System.Windows.Forms.Label();
    this.chkMulticolore = new System.Windows.Forms.CheckBox();
    this.SuspendLayout();
    //
    // cmdPaint
    //
    this.cmdPaint.Location = new System.Drawing.Point(544, 12);
    this.cmdPaint.Name = "cmdPaint";
    this.cmdPaint.Size = new System.Drawing.Size(94, 23);
    this.cmdPaint.TabIndex = 0;
}
```

Comme le spécifie le commentaire, ne modifiez pas ce code directement, mais inspirez-vous en pour créer vos propres contrôles dynamiquement.

Exemple : Formulaire initial :



On reprend le code créant un label (cité ci-dessus) et on l'associe au bouton (gestionnaire de l'événement « Click » du bouton). Lorsqu'on clique le bouton, on obtient :



Remarque : Ne pas confondre la création du contrôle avec sa propriété « Visible ».

Dans cet exemple, le label n'existe pas dans le formulaire tant qu'on a pas cliqué le bouton !

5.6 Gestionnaire d'Événement Commun

Quand on crée une application Windows Forms en C# au moyen du concepteur graphique uniquement, on se retrouve naturellement avec un gestionnaire d'événement par événement (pris en charge) et par contrôle.

Mais on peut facilement se retrouver dans la situation où plusieurs gestionnaires d'événements doivent accomplir une tâche similaire – voire carrément identique ! (comme par exemple l'application proposée en exemple en section 5.2.7).

La figure ci-dessous illustre cette situation :

```
private void rbtVHaut_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

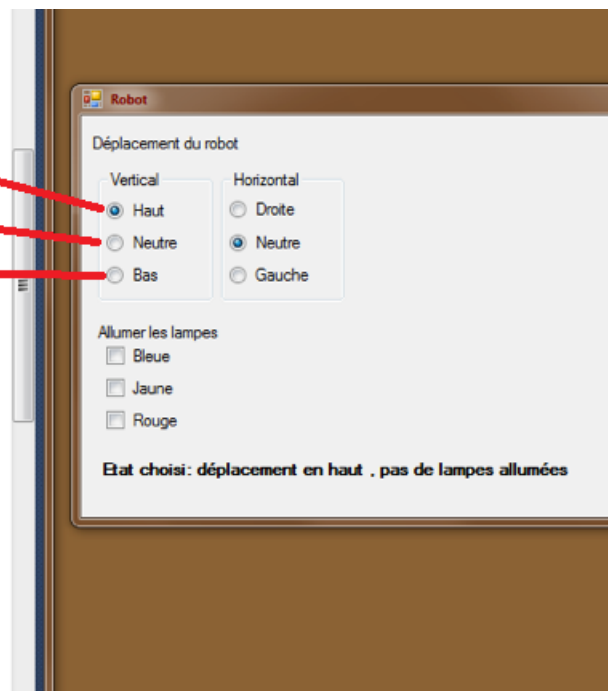
private void rbtVNeutre_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

private void rbtVBas_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

private void rbtHDroite_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

private void rbtHNeutre_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}

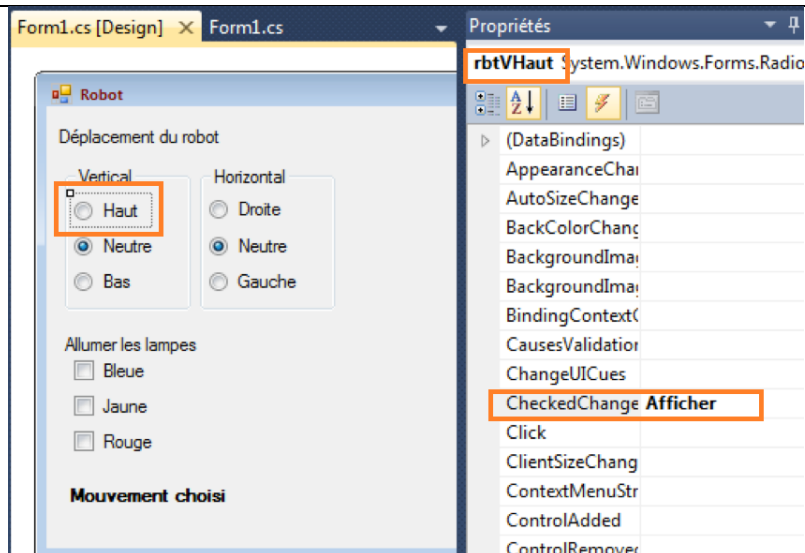
private void rbtHGauche_CheckedChanged(object sender, EventArgs e)
{
    Afficher();
}
```



A gauche l'environnement de développement dans lequel sont codés les gestionnaires d'événements. A droite l'application en cours d'exécution. On constate qu'il y a autant de gestionnaires d'événement que de boutons/cases. Et on constate également que ces gestionnaires d'événement font tous la même chose.

Nous allons voir que l'on peut optimiser la chose.

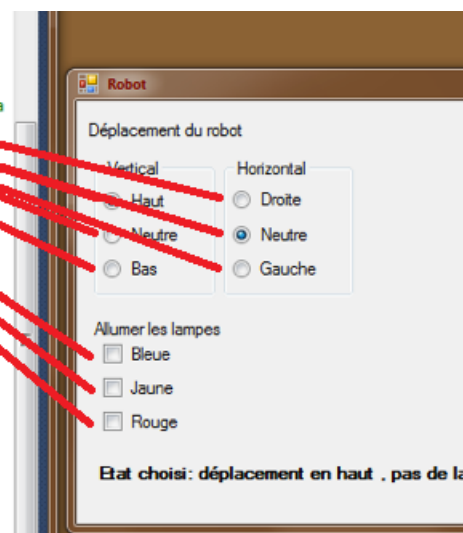
Il suffit pour cela d'aller dans le concepteur graphique de formulaires et de changer pour chaque contrôle la référence à son gestionnaire d'événement « CheckedChanged » :



Une fois que cela est fait, on se retrouve dans une configuration de code nettement plus simple :

```
private void Afficher (object sender, EventArgs e)
{
    string res = "Etat choisi: "; // la variable res contient le texte à afficher à la fin
    if (rbtVNeutre.Checked && rbthHNeutre.Checked) // aucun mouvement
        res = res + "pas de déplacement ";
    else
    {
        res = res + "déplacement ";
        if (rbtVHaut.Checked) res = res + "en haut ";
        if (rbtVBas.Checked) res = res + "en bas ";
        if (rbthDroite.Checked) res = res + "à droite ";
        if (rbthGauche.Checked) res = res + "à gauche ";
        res = res + ", ";
    }

    string lampes = "lampes ";
    if (chkBleu.Checked) lampes = lampes + "bleue ";
    if (chkJaune.Checked) lampes = lampes + "jaune ";
    if (chkRouge.Checked) lampes = lampes + "rouge ";
    if (lampes == "lampes ") // pas de changement -> pas de lampe allumée
        lampes = "pas de lampes ";
}
```



Attention : pour qu'une méthode puisse servir de gestionnaire d'événement, il est nécessaire qu'elle déclare les paramètres « sender » et « e » comme ci-dessus.

Ces paramètres vont d'ailleurs très vite se révéler indispensables. En effet, dans l'exemple ci-dessus, le code exécuté est exactement le même quel que soit le contrôle que l'on a cliqué. Mais cela n'est de loin pas toujours le cas.

On pourrait imaginer que l'on fait effectivement en grande partie le même travail dans tous les cas, mais que l'on veut en plus une confirmation de l'utilisateur qu'il veut bien allumer la lampe rouge.

Pour arriver à faire cela il nous faudra savoir lequel des neuf contrôle est la cause de l'appel à notre code. Ou en d'autres termes : qui est l'expéditeur (« sender ») de l'événement.

On ajoute donc ce code :

```
if (sender is CheckBox) // Le contrôle qui a été coché est de type checkbox
{
    CheckBox cb = (CheckBox)sender; // créer une référence sur LA checkbox cliquée
    if ((cb == chkRouge) && cb.Checked) // Est-ce la rouge ? Est-elle cochée ?
        if (MessageBox.Show("Allumer la lampe rouge ? vraiment ?", "T'es sûr",
            MessageBoxButtons.YesNo, MessageBoxIcon.Question) !=
            System.Windows.Forms.DialogResult.Yes)
        {
            cb.Checked = false; // On la décoche
            return; // et on quitte
        }
}
```

A noter que du moment que l'on a retrouvé dans notre méthode la référence sur l'objet qui nous a appelé, on peut en faire ce que l'on veut (déplacer, redimensionner, changer l'aspect, ...)

5.7 Application Ayant Plusieurs Formulaires

Jusqu'ici, nous n'avons vu que des applications ayant un seul formulaire (fenêtre). Il va très vite s'avérer nécessaire de franchir cette limite.

Lorsque vous créez un formulaire dans Visual Studio, ce dernier génère en fait une classe qui hérite des propriétés d'un formulaire. C'est ce que signifie la ligne

```
public partial class Robot : Form
```

Vous pouvez dès lors dans votre projet créer un nouveau formulaire :

```
Robot Controlleur = new Robot();
```

A partir de là vous pouvez manipuler ce nouveau formulaire en appelant ses méthodes publiques :

```
Controlleur.Show(); // Rendre le formulaire visible  
Controlleur.SetBounds(0, 0, 400, 600); // le mettre dans le coin supérieur gauche  
// tout en le redimensionnant à 400/600
```

Le problème suivant apparaît lorsque l'on veut faire interagir nos formulaires (échanges de données, demandes d'actions). La solution est simple quand c'est le formulaire « père » (celui qui a créé le second formulaire avec new()) : il lui suffit d'appeler une méthode du formulaire fils – que l'on aura codé tout exprès si nécessaire.

Mais que faire si le fils veut invoquer le père ? Il n'a pas de référence sur ce dernier.

Il suffit de faire en sorte que le père « donne ses coordonnées » à son fils lors de la création.

Exemple :

```
public partial class Père : Form  
{  
    Fils Rejeton;  
    int NbFils = 0;  
  
    public Père()  
    {  
        InitializeComponent();  
    }  
  
    public void Dire (string msg)  
    // Cette méthode est appelée par les fils pour poster un message  
    {  
        lstMessages.Items.Add(msg);  
    }  
  
    private void cmdNouveauClient_Click(object sender, EventArgs e)
```

```

// Création d'un nouveau formulaire fils
{
    NbFils++;
    // this est une référence sur soi-même (le formulaire père dans ce cas)
    Rejeton = new Fils(this, "Fils " + NbFils.ToString());
    Rejeton.Show();
}
}

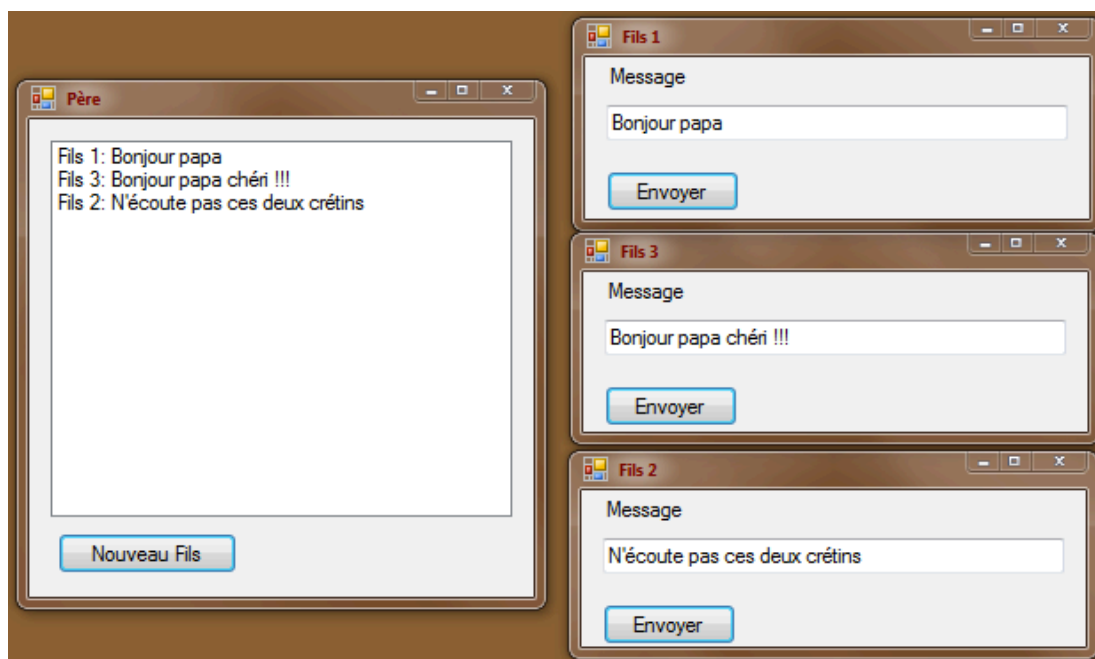
public partial class Fils : Form
{
    Père Papa; // Référence sur le père

    public Fils(Père _papa, string _titre)
    {
        InitializeComponent();
        this.Text = _titre;
        Papa = _papa; // Quand le père nous crée, il dit qui il est (_papa).
                       // On stocke sa référence dans Papa
    }

    private void cmdEnvoyer_Click(object sender, EventArgs e)
    {
        Papa.Dire(this.Text + ": " + txtMsg.Text);
    }
}

```

A l'exécution, ce code produit un résultat de ce style :



Annexe I. Convention de Nommage

Les noms d'objets doivent comporter un préfixe cohérent, permettant d'identifier facilement leur type. Vous trouverez ci-dessous la liste des conventions relatives à certains objets pris en charge par .NET.

Les plus courants :

Type de contrôle	Anglais	Préfixe	Exemple
Bouton de commande	CommandButton	cmd	cmdExit
Cadre	Frame	fra	fraLanguage
Case à cocher	CheckBox	chk	chkReadOnly
Bouton d'option	OptionButton	opt	optTransport
Boîtes de dialogue communes	CommonDialog	dlg	dlgFileOpen
Étiquette	Label	lbl	lblHelpMessage
Feuille	Form	frm	frmEntry
Forme	Shape	shp	shpCircle
Groupe de contrôles	GroupBox	grp	grpOptions
Image	Picture	pic	picVGA
Minuterie	Timer	tmr	tmrAlarm
Zone de texte	TextBox	txt	txtLastName
Zone de liste modifiable, zone de liste déroulante	ComboBox	cbo	cboLanguage
Zone de liste	ListBox	lst	lstPolicyCodes

Mais aussi :

Type de contrôle		Préfixe	Exemple
Barre de défilement horizontale	HScrollBar	hsb	hsbVolume
Barre de défilement verticale	VScrollBar	vsb	vsbRate
Bouton animé	AniPushButton	ani	aniMailBox
Plan	Outline	out	outOrgChart
Contrôle (type spécifique est inconnu)	Control	ctr	ctrCurrent
Contrôle de données	Data	dat	datBiblio
État de la touche	Mhstate	key	keyCaps
Feuille MDI fille	MDI child form	mdi	mdiNote
Graphique	Graph	gra	graRevenue
Grille	Grid	grd	grdPrices
Grille dépendante	DBGrid	dbgrd	dbgrdQueryResult
Image	Image	img	imgIcon
Liste d'images	ImageList	ils	ilsAllIcons
Indicateur	Gauge	gau	gauStatus
Ligne	Line	lin	linVertical
Liste de fichiers	FileListBox	fil	filSource
Liste de dossiers	DirListBox	dir	dirSource
Liste de lecteurs	DriveList Box	drv	drvTarget
Affichage de listes	ListView	lvw	lvwHeadings
Menu	Menu	mnu	mnuFileOpen
Message MAPI	MAPI message	mpm	mpmSentMessage
MS Flex grid	MS Flex grid	msg	msgClients
Onglet	MSTab	mst	mstFirst
OLE	OLE	ole	oleWorksheet
Jeu d'image	PictureClip	clp	clpToolbar
Barre de progression	ProgressBar	prg	prgLoadFile
Éditeur RTF	RichTextBox	rtf	rtfReport
Curseur	Slider	sld	sldScale
Compteur	SpinButton	spn	spnPages
Barre d'état	StatusBar	sta	staDateTime
Barre d'onglets	TabStrip	tab	tabOptions
Barre d'outils	ToolBar	tlb	tlbActions
Affichage de l'arborescence	TreeView	tre	treOrganization
Barre de défilement	UpDown	upd	updDirection
Zone de liste dépendante	DBlist	dblst	dblstJobType

Annexe II. Convention de Codage

Cette annexe définit les règles applicables au code C# écrit dans le cadre du module I-CH 303 au CPNV. Bien que très largement basée sur des pratiques courantes, cette convention ne doit pas être considérée universelle. Elle ne s'applique que dans le cadre du cours mentionné et elle peut tout-à-fait différer sur certains points de conventions similaires dans d'autres contextes.

Commentaires

1. Chaque fichier de code source contient une en-tête avec:
 - Le nom du projet auquel il appartient
 - Le nom du formulaire auquel il est lié
 - Une description générale des fonctionnalités implémentées par le code source de ce fichier
 - Un identifiant de version sous forme "X.Y"
 - Le nom de l'auteur de la version
 - La date de réalisation de la version
2. Chaque variable significative doit faire l'objet d'un commentaire quant à son utilisation. Sont considérées significatives toutes les variables à l'exception des variables de contrôles de boucle et les variables à usage temporaire (lecture d'une touche clavier pour l'exécution d'un menu p.ex.).
3. Des commentaires seront insérés à chaque fois que le code lui-même ne dévoile pas clairement les intentions du développeur.

Indentation et mise en page

1. Les débuts de bloc d'instruction ("{"") sont toujours placés à la ligne, exactement sous l'instruction qu'ils suivent
2. L'indentation d'une ligne doit être augmentée (= décalée vers la droite) si la ligne précédente est:
 - a. Un début de bloc ("{"")
 - b. Une instruction "if", "else", "switch", "case", "for" ou "while"
3. L'indentation d'une ligne doit être réduite (=décalée vers la gauche) si
 - a. C'est une fin de bloc ("}")
 - b. C'est une instruction "case" (exception faite de la première à l'intérieur du "switch")
 - c. C'est une instruction "default" d'un switch
 - d. La ligne précédente est une instruction unique d'un "if", "else", "for" ou "while"

Nommage des méthodes, variables et constantes

1. Les identificateurs de méthodes et de variables suivent la convention de casse dite "Pascal": chaque mot qui compose l'identificateur commence par une majuscule et les mots sont collés les uns aux autres.

Exemples:

```
int NombreDeJoueurs;
public void LancerLeDé()
```

2. Les identificateurs de constantes sont écrits en majuscules.

Exemple:

```
public const int MAX = 100;
```

Annexe III. Chaînes de Format

Numériques

Spécificateur de format	Nom	Résultat	Spécificateur de précision	Exemples
"C" ou "c"	Devise	Une valeur monétaire.	Nombre de chiffres décimaux	123.456 ("C", en-US) -> \$123.46 123.456 ("C", fr-FR) -> 123,46 € 123.456 ("C", ja-JP) -> ¥123 -123.456 ("C3", en-US) -> (\$123.456) -123.456 ("C3", fr-FR) -> -123,456 € -123.456 ("C3", ja-JP) -> -¥123.456
"D" ou "d"	Decimal	Chiffres entiers avec un signe négatif facultatif.	Nombre minimal de chiffres.	1234 ("D") -> 1234 -1234 ("D6") -> -001234
"E" ou "e"	Exponentiel (scientifique)	Notation exponentielle.	Nombre de chiffres décimaux (par défaut :6)	1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1,052033e+003 -1052.0329112756 ("e2", en-US) -> -1.05e+003 -1052.0329112756 ("E2", fr_FR) -> -1,05E+003
"F" ou "f"	Virgule fixe	chiffres intégraux et décimaux avec un signe négatif facultatif	nombre de chiffres décimaux	1234.567 ("F", en-US) -> 1234.57 1234.567 ("F", de-DE) -> 1234,57 1234 ("F1", en-US) -> 1234.0 1234 ("F1", de-DE) -> 1234,0 -1234.56 ("F4", en-US) -> -1234.5600 -1234.56 ("F4", de-DE) -> -1234,5600
"G" ou "g"	Général	format le plus compact (notation à virgule fixe ou scientifique).	nombre de chiffres significatifs	-123.456 ("G", en-US) -> -123.456 123.456 ("G", sv-SE) -> -123,456 123.4546 ("G4", en-US) -> 123.5 123.4546 ("G4", sv-SE) -> 123,5 -1.234567890e-25 ("G", en-US) -> -1.23456789E-25 -1.234567890e-25 ("G", sv-SE) -> -1,23456789E-25
"N" ou "n"	Nombre	chiffres intégraux et décimaux, séparateurs de groupes et séparateur décimal avec un signe négatif facultatif	nombre souhaité de décimales	1234.567 ("N", en-US) -> 1,234.57 1234.567 ("N", ru-RU) -> 1 234,57 1234 ("N", en-US) -> 1,234.0 1234 ("N", ru-RU) -> 1 234,0 -1234.56 ("N", en-US) -> -1,234.560 -1234.56 ("N", ru-RU) -> -1 234,560

Spécificateur de format	Nom	Résultat	Spécificateur de précision	Exemples
"P" ou "p"	Pourcentage	nombre multiplié par 100 et affiché avec un symbole de pourcentage	nombre souhaité de décimales	1 ("P", en-US) -> 100.00 % 1 ("P", fr-FR) -> 100,00 % -0.39678 ("P1", en-US) -> -39.7 % -0.39678 ("P1", fr-FR) -> -39,7 %
"X" ou "x"	Hexadécimal	chaîne hexadécimale	nombre de chiffres dans la chaîne de résultat	255 ("X") -> FF -1 ("x") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 00FF
N'importe quel caractère	Spécificateur inconnu	lève un FormatException au moment de l'exécution		

Dates

Spécific. format	Description	Exemples
"d"	Modèle de date courte	6/15/2009 1:45:30 PM -> 6/15/2009 (en-US) 6/15/2009 1:45:30 PM -> 15/06/2009 (fr-FR) 6/15/2009 1:45:30 PM -> 2009/06/15 (ja-JP)
"D"	Modèle de date longue.	6/15/2009 1:45:30 PM -> Monday, June 15, 2009 (en-US) 6/15/2009 1:45:30 PM -> 15 июня 2009 г. (ru-RU)
"f"	Modèle de date/heure complet (heure courte).	6/15/2009 1:45:30 PM -> Monday, June 15, 2009 1:45 PM (en-US) 6/15/2009 1:45:30 PM -> den 15 juni 2009 13:45 (sv-SE) 6/15/2009 1:45:30 PM -> Δευτέρα, 15 Ιουνίου 2009 1:45 μμ (el-GR)
"F"	Modèle de date/heure complet (heure longue).	6/15/2009 1:45:30 PM -> Monday, June 15, 2009 1:45:30 PM (en-US) 6/15/2009 1:45:30 PM -> den 15 juni 2009 13:45:30 (sv-SE) 6/15/2009 1:45:30 PM -> Δευτέρα, 15 Ιουνίου 2009 1:45:30 μμ (el-GR)
"g"	Modèle de date/heure général (heure courte).	6/15/2009 1:45:30 PM -> 6/15/2009 1:45 PM (en-US) 6/15/2009 1:45:30 PM -> 15/06/2009 13:45 (es-ES) 6/15/2009 1:45:30 PM -> 2009/6/15 13:45 (zh-CN)
"G"	Modèle de date/heure général (heure longue).	6/15/2009 1:45:30 PM -> 6/15/2009 1:45:30 PM (en-US) 6/15/2009 1:45:30 PM -> 15/06/2009 13:45:30 (es-ES) 6/15/2009 1:45:30 PM -> 2009/6/15 13:45:30 (zh-CN)
"M", "m"	Modèle de mois/jour.	6/15/2009 1:45:30 PM -> June 15 (en-US) 6/15/2009 1:45:30 PM -> 15. juni (da-DK) 6/15/2009 1:45:30 PM -> 15 Juni (id-ID)
"s"	Modèle de date/heure pouvant être trié.	6/15/2009 1:45:30 PM -> 2009-06-15T13:45:30
"t"	Modèle d'heure courte.	6/15/2009 1:45:30 PM -> 1:45 PM (en-US) 6/15/2009 1:45:30 PM -> 13:45 (hr-HR)
"T"	Modèle d'heure longue.	6/15/2009 1:45:30 PM -> 1:45:30 PM (en-US) 6/15/2009 1:45:30 PM -> 13:45:30 (hr-HR)
"Y", "y"	Modèle d'année/mois.	6/15/2009 1:45:30 PM -> June, 2009 (en-US) 6/15/2009 1:45:30 PM -> juni 2009 (da-DK) 6/15/2009 1:45:30 PM -> Juni 2009 (id-ID)
Autre	Spécificateur inconnu.	Lève un FormatException runtime.