APPRENTISSAGE DU LANGAGE C#

Serge Tahé - ISTIA - Université d'Angers
Mai 2002

Introduction

C# est un langage récent. Il a été disponible en versions beta depuis l'année 2000 avant d'être officiellement disponible en février 2002 en même temps que la plate-forme .NET de Microsoft à laquelle il est lié. C# ne peut fonctionner qu'avec cet environnement d'exécution, environnement disponible pour le moment que sur les machines Windows NT, 2000 et XP.

Avec la plate-forme .NET, trois nouveaux langages sont apparus : C#, VB.VET, JSCRIPT.NET. C# est largement une « copie » de Java. VB.NET et JSCRIPT.NET sont des extensions de Visual basic et Jscript pour la plate-forme .NET. Celle-ci rend disponible aux programmes qui s'exécutent en son sein un ensemble très important de classes, classes très proches de celles que l'on trouve au sein des machines virtuelles Java. 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 importantes :

- la plate-forme .NET ne s'exécute que sur les machines Windows alors que Java s'exécute sur différents OS (windows, unix, macintosh).
- 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.

De la même façon que Java ne peut être ignoré, la plate-forme .NET ne peut l'être, à la fois à cause du parc très important de machines windows installées et de l'effort fait par Microsoft pour la promouvoir et l'imposer. Il semble que C# soit un bon choix pour démarrer avec .NET, notamment pour les programmeurs Java, tellement ces deux langages sont proches. Ensuite on pourra passer aisément de C# à VB.NET ou à un autre langage .NET. La syntaxe changera mais les classes .NET resteront les mêmes. Contrairement aux apparences, le passage de VB à VB.NET est difficile. VB n'est pas un langage orienté objets alors que VB.NET l'est complètement. Le programmeur VB va donc être confronté à des concepts qu'il ne maîtrise pas. Il paraît plus simple d'affronter ceux-ci avec un langage entièrement nouveau tel que C# plutôt qu'avec VB.NET où le programmeur VB aura toujours tendance à vouloir revenir à ses habitudes VB.

Ce document n'est pas un cours exhaustif. Il est destiné à des gens connaissant déjà la programmation et qui veulent découvrir C#. Afin de faciliter la comparaison avec Java, il reprend la structure du document "Introduction au langage Java" du même auteur.

Deux livres m'ont aidé :

- Professional C# programming, Editions Wrox
- C# et .NET, Gérard Leblanc, Editions Eyrolles

Ce sont deux excellents ouvrages dont je conseille la lecture.

Serge Tahé, avril 2002

1. LES BASES DU LANGAGE C# 7

1.1 INTRODUCTION 7
1.2 LES DONNEES DE C# 7
1.2.1 LES TYPES DE DONNEES PREDEFINIS 7
1.2.2 CONVERSION ENTRE TYPES SIMPLES ET TYPES OBJETS 8
1.2.3 NOTATION DES DONNEES LITTERALES 8
1.2.4 DECLARATION DES DONNEES 8
1.2.5 LES CONVERSIONS ENTRE NOMBRES ET CHAINES DE CARACTERES 9
1.2.6 LES TABLEAUX DE DONNEES 10
1.3 LES INSTRUCTIONS ELEMENTAIRES DE C# 12
1.3.1 ECRITURE SUR ECRAN 12
1.3.2 LECTURE DE DONNEES TAPEES AU CLAVIER 13
1.3.3 EXEMPLE D'ENTREES-SORTIES 13
1.3.4 REDIRECTION DES E/S 13
1.3.5 AFFECTATION DE LA VALEUR D'UNE EXPRESSION A UNE VARIABLE 14
1.4 LES INSTRUCTIONS DE CONTROLE DU DEROULEMENT DU PROGRAMME 20
1.4.1 ARRET 20
1.4.2 STRUCTURE DE CHOIX SIMPLE 20
1.4.3 STRUCTURE DE CAS 21
1.4.4 STRUCTURE DE REPETITION 21
1.5 LA STRUCTURE D'UN PROGRAMME C# 24
1.6 COMPILATION ET EXECUTION D'UN PROGRAMME C# 24
1.7 L'EXEMPLE IMPOTS 24
1.8 ARGUMENTS DU PROGRAMME PRINCIPAL 26
1.9 LES ENUMERATIONS 27
1.10 LA GESTION DES EXCEPTIONS 28
1.11 PASSAGE DE PARAMETRES A UNE FONCTION 31
1.11.1 PASSAGE PAR VALEUR 31
1.11.2 PASSAGE PAR REFERENCE 31
1.11.3 PASSAGE PAR REFERENCE AVEC LE MOT CLE OUT 32

2. CLASSES, STUCTURES, INTERFACES 33

2.1 L' OBJET PAR L'EXEMPLE 33
2.1.1 GENERALITES 33
2.1.2 DEFINITION DE LA CLASSE PERSONNE 33
2.1.3 LA METHODE INITIALISE 34
2.1.4 L'OPERATEUR NEW 34
2.1.5 LE MOT CLE THIS 35
2.1.6 UN PROGRAMME DE TEST 35
2.1.7 UTILISER UN FICHIER DE CLASSES COMPILEES (ASSEMBLY) 36
2.1.8 UNE AUTRE METHODE INITIALISE 37
2.1.9 CONSTRUCTEURS DE LA CLASSE PERSONNE 37
2.1.10 LES REFERENCES D'OBJETS 38
2.1.11 LES OBJETS TEMPORAIRES 39
2.1.12 METHODES DE LECTURE ET D'ECRITURE DES ATTRIBUTS PRIVES 40
2.1.13 LES PROPRIETES 41
2.1.14 LES METHODES ET ATTRIBUTS DE CLASSE 42
2.1.15 PASSAGE D'UN OBJET A UNE FONCTION 43
2.1.16 UN TABLEAU DE PERSONNES 44
2.2 L'HERITAGE PAR L'EXEMPLE 45
2.2.1 GENERALITES 45
2.2.2 CONSTRUCTION D'UN OBJET ENSEIGNANT 46
2.2.3 SURCHARGE D'UNE METHODE OU D'UNE PROPRIETE 47
2.2.4 LE POLYMORPHISME 49
2.2.5 SURCHARGE ET POLYMORPHISME 49
2.3 REDEFIR LA SIGNIFICATION D'UN OPERATEUR POUR UNE CLASSE 52
2.3.1 INTRODUCTION 52
2.3.2 UN EXEMPLE 52
2.4 DEFINIR UN INDEXEUR POUR UNE CLASSE 53

2.5 LES STRUCTURES **55**
2.6 LES INTERFACES **58**
2.7 LES ESPACES DE NOMS **61**
2.8 L'EXEMPLE IMPOTS **62**

3. CLASSES .NET D'USAGE COURANT **66**

3.1 CHERCHER DE L'AIDE AVEC SDK.NET **66**
3.1.1 WINCV 66
3.2 CHERCHER DE L'AIDE SUR LES CLASSES AVEC VS.NET **69**
3.2.1 HELP/CONTENTS 69
3.2.2 HELP/INDEX 72
3.3 LA CLASSE STRING **73**
3.4 LA CLASSE ARRAY **75**
3.5 LA CLASSE ARRAYLIST **77**
3.6 LA CLASSE HASHTABLE **79**
3.7 LA CLASSE STREAMREADER **81**
3.8 LA CLASSE STREAMWRITER **82**
3.9 LA CLASSE REGEX **83**
3.9.1 VERIFIER QU'UNE CHAINE CORRESPOND A UN MODELE DONNE 85
3.9.2 TROUVER TOUS LES ELEMENTS D'UNE CHAINE CORRESPONDANT A UN MODELE 86
3.9.3 RECUPERER DES PARTIES D'UN MODELE 87
3.9.4 UN PROGRAMME D'APPRENTISSAGE 88
3.9.5 LA METHODE SPLIT 89
3.10 LES CLASSES BINARYREADER ET BINARYWRITER **90**

4. INTERFACES GRAPHIQUES AVEC C# ET VS.NET **93**

4.1 LES BASES DES INTERFACES GRAPHIQUES **93**
4.1.1 UNE FENETRE SIMPLE 93
4.1.2 UN FORMULAIRE AVEC BOUTON 94
4.2 CONSTRUIRE UNE INTERFACE GRAPHIQUE AVEC VISUAL STUDIO.NET **97**
4.2.1 CREATION INITIALE DU PROJET 97
4.2.2 LES FENETRE DE L'INTERFACE DE VS.NET 98
4.2.3 EXECUTION D'UN PROJET 100
4.2.4 LE CODE GENERE PAR VS.NET 100
4.2.5 CONCLUSION 102
4.3 FENETRE AVEC CHAMP DE SAISIE, BOUTON ET LIBELLE **102**
4.3.1 LE CODE LIE A LA GESTION DES EVENEMENTS 107
4.3.2 CONCLUSION 108
4.4 QUELQUES COMPOSANTS UTILES **108**
4.4.1 FORMULAIRE FORM 108
4.4.2 ETIQUETTES LABEL ET BOITES DE SAISIE TEXTBOX 109
4.4.3 LISTES DEROULANTES COMBOBOX 110
4.4.4 COMPOSANT LISTBOX 112
4.4.5 CASES A COCHER CHECKBOX, BOUTONS RADIO BUTTONRADIO 114
4.4.6 VARIATEURS SCROLLBAR 115
4.5 ÉVENEMENTS SOURIS **117**
4.6 CREER UNE FENETRE AVEC MENU **119**
4.7 COMPOSANTS NON VISUELS **124**
4.7.1 BOITES DE DIALOGUE OPENFILEDIALOG ET SAVEFILEDIALOG 124
4.7.2 BOITES DE DIALOGUE FONTCOLOR ET COLORDIALOG 129
4.7.3 TIMER 131
4.8 L'EXEMPLE IMPOTS **133**

5. GESTION D'EVENEMENTS **136**

5.1 OBJETS DELEGATE **136**
5.2 GESTION D'EVENEMENTS **137**

6. ACCES AUX BASES DE DONNEES **142**

6.1 GENERALITES **142**
6.2 LES DEUX MODES D'EXPLOITATION D'UNE SOURCE DE DONNEES **143**
6.3 ACCES AUX DONNEES EN MODE CONNECTE **144**
6.3.1 LES BASES DE DONNEES DE L'EXEMPLE 144
6.3.2 UTILISATION D'UN PILOTE ODBC 148
6.3.3 UTILISATION D'UN PILOTE OLE DB 152
6.3.4 EXEMPLE 1 : MISE A JOUR D'UNE TABLE 153
6.3.5 EXEMPLE 2 : IMPOTS 157
6.4 ACCES AUX DONNEES EN MODE DECONNECTE **160**

7. LES THREADS D'EXECUTION **161**

7.1 INTRODUCTION **161**
7.2 CREATION DE THREADS D'EXECUTION **162**
7.3 INTERET DES THREADS **164**
7.4 ACCES A DES RESSOURCES PARTAGEES **165**
7.5 ACCES EXCLUSIF A UNE RESSOURCE PARTAGEE **166**
7.6 SYNCHRONISATION PAR EVENEMENTS **169**

8. PROGRAMMATION TCP-IP **172**

8.1 GENERALITES **172**
8.1.1 LES PROTOCOLES DE L'INTERNET 172
8.1.2 LE MODELE OSI 172
8.1.3 LE MODELE TCP/IP 173
8.1.4 FONCTIONNEMENT DES PROTOCOLES DE L'INTERNET 175
8.1.5 LES PROBLEMES D'ADRESSAGE DANS L'INTERNET 176
8.1.6 LA COUCHE RESEAU DITE COUCHE IP DE L'INTERNET 179
8.1.7 LA COUCHE TRANSPORT : LES PROTOCOLES UDP ET TCP 180
8.1.8 LA COUCHE APPLICATIONS 181
8.1.9 CONCLUSION 182
8.2 GESTION DES ADRESSES RESEAU **182**
8.3 PROGRAMMATION TCP-IP **185**
8.3.1 GENERALITES 185
8.3.2 LES CARACTERISTIQUES DU PROTOCOLE TCP 185
8.3.3 LA RELATION CLIENT-SERVEUR 186
8.3.4 ARCHITECTURE D'UN CLIENT 186
8.3.5 ARCHITECTURE D'UN SERVEUR 186
8.3.6 LA CLASSE TCPCLIENT 186
8.3.7 LA CLASSE NETWORKSTREAM 187
8.3.8 ARCHITECTURE DE BASE D'UN CLIENT INTERNET 188
8.3.9 LA CLASSE TCPLISTENER 188
8.3.10 ARCHITECTURE DE BASE D'UN SERVEUR INTERNET 189
8.4 EXEMPLES **190**
8.4.1 SERVEUR D'ECHO 190
8.4.2 UN CLIENT POUR LE SERVEUR D'ECHO 191
8.4.3 UN CLIENT TCP GENERIQUE 193
8.4.4 UN SERVEUR TCP GENERIQUE 198
8.4.5 UN CLIENT WEB 201
8.4.6 CLIENT WEB GERANT LES REDIRECTIONS 203
8.4.7 SERVEUR DE CALCUL D'IMPOTS 205

9. SERVICES WEB **210**

9.1 INTRODUCTION **210**
9.2 UN PREMIER SERVICE WEB **210**
9.3 UN CLIENT HTTP-GET **216**
9.4 UN CLIENT HTTP-POST **222**
9.5 UN CLIENT SOAP **226**
9.6 ENCAPSULATION DES ECHANGES CLIENT-SERVEUR **230**
9.6.1 LA CLASSE D'ENCAPSULATION 230
9.6.2 UN CLIENT CONSOLE 233

9.6.3 UN CLIENT GRAPHIQUE WINDOWS 235
9.7 UN CLIENT PROXY 238
9.8 CONFIGURER UN SERVICE WEB 243
9.9 LE SERVICE WEB IMPOTS 245
9.9.1 LE SERVICE WEB 245
9.9.2 GENERER LE PROXY DU SERVICE IMPOTS 250
9.9.3 UTILISER LE PROXY AVEC UN CLIENT 250

10. A SUIVRE… 253

1. Les bases du langage C#

1.1 Introduction

Nous traitons C# d'abord comme un langage de programmation classique. Nous aborderons les objets ultérieurement. Dans un programme on trouve deux choses

- des données
- les instructions qui les manipulent

On s'efforce généralement de séparer les données des instructions :

```
+--------------------+
¦      DONNEES       ¦
+--------------------¦
¦                    ¦
¦      INSTRUCTIONS  ¦
¦                    ¦
+--------------------+
```

1.2 Les données de C#

C# utilise les types de données suivants:

1. les nombres entiers
2. les nombres réels
3. les nombres décimaux
4. les caractères et chaînes de caractères
5. les booléens
6. les objets

1.2.1 Les types de données prédéfinis

Type	Codage	Domaine
char	2 octets	caractère Unicode
int	4 octets	[-2^{31}, 2^{31}-1] [–2147483648, 2147483647]
uint	4 octets	[0, 2^{32}-1] [0, 4294967295]
long	8 octets	[-2^{63}, 2^{63} -1] [–9223372036854775808, 9223372036854775807]
ulong	8 octets	[0, 2^{64} -1] [0, 18446744073709551615]
sbyte	1 octet	[-2^{7} , 2^{7} -1] [-128,+127]
byte	1 octet	[0 , 2^{8} -1] [0,255]
short	2 octets	[-2^{15}, 2^{15}-1] [-32768, 32767]
ushort	2 octets	[0, 2^{16}-1] [0,65535]
float	4 octets	[1.5 10^{-45}, 3.4 10^{+38}] en valeur absolue
double	8 octets	[5.0 × 10^{-324}, 1.7 10^{+308}] en valeur absolue
decimal	16 octets	[1.0 10^{-28},7.9 10^{+28}] en valeur absolue avec 28 chiffres significatifs
bool	1 bit	true, false
Char	référence d'objet	char
String	référence d'objet	chaîne de caractères
DateTime	référence d'objet	date et heure
Int32	référence d'objet	int
Int64	référence d'objet	long
Byte	référence d'objet	byte
Float	référence d'objet	float
Double	référence d'objet	double
Decimal	référence d'objet	decimal
Boolean	référence d'objet	boolean

1.2.2 Conversion entre types simples et types objets

Dans le tableau ci-dessus, on découvre qu'il y a deux types possibles pour un entier sur 32 bits : *int* et *Int32*. Le type int est un type simple dont on ne manipule que la valeur. *Int32* est une classe. Un objet de ce type est complexe et possède des attributs et méthodes. C# est amené à faire des conversion implicites entre ces deux types. Ainsi si une fonction attend comme paramètre un objet de type *Int32*, on pourra lui passer une donnée de type *int*. Le compilateur fera implicitement la conversion *int* -->*Int32*. On appelle cela le "*boxing*" c.a.d. littéralement la mise en boîte d'une valeur dans un objet. L'inverse est également vrai. Là où une fonction attend une valeur de type *int*, on pourra lui passer une donnée de type *Int32*. La conversion se fera là encore automatiquement et s'appelle le "*unboxing*". Les opérations implicites de boxing/unboxing se font sur les types suivants :

int	Int32
long	Int64
decimal	Decimal
bool	Boolean
char	Char
byte	Byte
float	Float
double	Double
enum	Enum

1.2.3 Notation des données littérales

entier int (32 bits)	145, -7, 0xFF (hexadécimal)
entier long (64 bits)	100000L
réel double	134.789, -45E-18 (-45 10^{-18})
réel float	134.789**F**, -45E-18**F** (-45 10^{-18})
réel decimal	100000M
caractère char	'A', 'b'
chaîne de caractères string	"aujourd'hui" "c:\\chap1\\paragraph3" @"c:\chap1\paragraph3"
booléen bool	true, false
date	new DateTime(1954,10,13) (an, mois, jour) pour le 13/10/1954

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 remplacée par un seul \ non interprété. 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**.

1.2.4 Déclaration des données

1.2.4.1 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.

1.2.4.2 Déclaration des constantes

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

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

ex : const *float PI=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 :

   ex : **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 :

   final *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.

1.2.4.3 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,variable2,...,variablen;**

où *identificateur_de_type* est un type prédéfini ou bien un type défini par le programmeur.

1.2.5 Les conversions entre nombres et chaînes de caractères

nombre -> chaîne	*"" + nombre*
chaine -> int	*int.Parse(chaine) ou Int32.Parse*
chaîne -> long	*long.Parse(chaine) pu Int64.Parse*
chaîne -> double	*double.Parse(chaîne) ou Double.Parse(chaîne)*
chaîne -> float	*float.Parse(chaîne) ou Float.Parse(chaîne)*

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** en C#. Cette erreur peut être gérée par la clause *try/catch* suivante :

```
try{
 appel de la fonction susceptible de générer l'exception
} catch (Exception e){
 traiter l'exception e
}
instruction suivante
```

Si la fonction ne génère pas d'exception, on passe alors à **instruction suivante**, sinon on passe dans le corps de la clause *catch* puis à **instruction suivante**. Nous reviendrons ultérieurement sur la gestion des exceptions. Voici un programme présentant les principales techniques de conversion entre nombres et chaînes de caractères. Dans cet exemple la fonction affiche écrit à l'écran la valeur de son paramètre. Ainsi *affiche(S)* écrit la valeur de S à l'écran.

```
// espaces de noms importés
using System;

// la classe de test
public class conv1{

  public static void Main(){

    String S;
    const int i=10;
    const long l=100000;
    const float f=45.78F;
    double d=-14.98;

    // nombre --> chaîne
    S=""+i;
    affiche(S);
    S=""+l;
    affiche(S);
    S=""+f;
    affiche(S);
```

```
    S=""+d;
    affiche(S);

    //boolean --> chaîne
    const bool b=false;
    S=""+b;
    affiche(S);

    // chaîne --> int
    int i1;
    i1=int.Parse("10");
    affiche(""+i1);
    try{
      i1=int.Parse("10.67");
      affiche(""+i1);
    } catch (Exception e){
      affiche("Erreur "+e.Message);
    }

    // chaîne --> long
    long l1;
    l1=long.Parse("100");
    affiche(""+l1);
    try{
      l1=long.Parse("10.675");
      affiche(""+l1);
    } catch (Exception e){
      affiche("Erreur "+e.Message);
    }

    // chaîne --> double
    double d1;
    d1=double.Parse("100,87");
    affiche(""+d1);
    try{
      d1=double.Parse("abcd");
      affiche(""+d1);
    } catch (Exception e){
      affiche("Erreur "+e.Message);
    }

    // chaîne --> float
    float f1;
    f1=float.Parse("100,87");
    affiche(""+f1);
    try{
      d1=float.Parse("abcd");
      affiche(""+f1);
    } catch (Exception e){
      affiche("Erreur "+e.Message);
    }

}// fin main

  public static void affiche(String S){
    Console.Out.WriteLine("S="+S);
  }
}// fin classe
```

Les résultats obtenus sont les suivants :

```
S=10
S=100000
S=45.78
S=-14.98
S=False
S=10
S=Erreur The input string was not in a correct format.
S=100
S=Erreur The input string was not in a correct format.
S=100.87
S=Erreur The input string was not in a correct format.
S=100.87
S=Erreur The input string was not in a correct format.
```

On remarquera que les nombres réels sous forme de chaîne de caractères doivent utiliser la virgule et non le point décimal. Ainsi on écrira

`double d1=10.7;` mais `double d2=int.Parse("10,7");`

1.2.6 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};
```

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}};
```

Le nombre d'éléments dans chacune des dimensions peut être obtenue par la méthode **GetLenth(i)** où i=0 représente la dimension correspondant au 1er indice, i=1 la dimension correspondant au 2ième indice, …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 tableaux ne sont pas créés lors de la déclaration ci-dessus. L'exemple ci-dessous illustre la création d'un tableau de tableaux :

```
    // un tableau de tableaux
    string[][] noms=new string[3][];
    for (int i=0;i<noms.Length;i++){
      noms[i]=new string[i+1];
    }//for
    // initialisation
    for (int i=0;i<noms.Length;i++){
      for(int j=0;j<noms[i].Length;j++){
        noms[i][j]="nom"+i+j;
      }//for j
    }//for i
```

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 :

```
// tableaux

using System;

// classe de test
public class test{
  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["+i+"]="+entiers[i]);
    }//for

    // 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["+i+","+j+"]="+réels[i,j]);
      }//for j
    }//for i

    // un tableau de tableaux
    string[][] noms=new string[3][];
    for (int i=0;i<noms.Length;i++){
      noms[i]=new string[i+1];
    }//for
    // initialisation
    for (int i=0;i<noms.Length;i++){
      for(int j=0;j<noms[i].Length;j++){
        noms[i][j]="nom"+i+j;
      }//for j
    }//for i
    // affichage
    for (int i=0;i<noms.Length;i++){
      for(int j=0;j<noms[i].Length;j++){
```

```
        Console.Out.WriteLine("noms["+i+"]["+j+"]="+noms[i][j]);
      }//for j
    }//for i


  }//Main
}//class
```

A l'exécution, nous obtenons les résultats suivants :

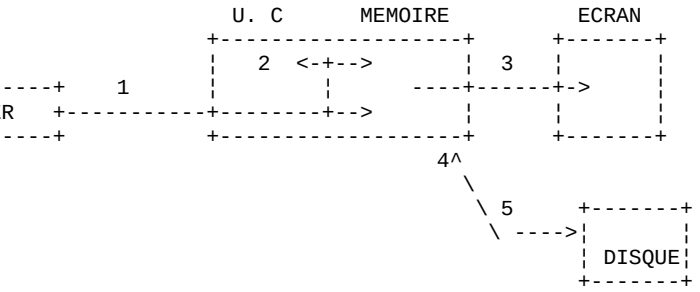
```
entiers[0]=0
entiers[1]=10
entiers[2]=20
entiers[3]=30
réels[0,0]=0.5
réels[0,1]=1.7
réels[1,0]=8.4
réels[1,1]=-6
noms[0][0]=nom00
noms[1][0]=nom10
noms[1][1]=nom11
noms[2][0]=nom20
noms[2][1]=nom21
noms[2][2]=nom22
```

1.3 Les instructions élémentaires de C#

On distingue

1 les instructions élémentaires exécutées par l'ordinateur.
2 les instructions de contrôle du déroulement du programme.

Les instructions élémentaires apparaissent clairement lorsqu'on considère la structure d'un micro-ordinateur et de ses périphériques.



1. lecture d'informations provenant du clavier
2. traitement d'informations
3. écriture d'informations à l'écran
4. lecture d'informations provenant d'un fichier disque
5. écriture d'informations dans un fichier disque

1.3.1 Ecriture sur écran

Il existe différentes instructions d'écriture à l'écran :

Console.Out.WriteLine(expression)
Console.WriteLine(expression)
Console.Error.WriteLine (expression)

où *expression* est tout type de donnée qui puisse être converti en chaîne de caractères pour être affiché à l'écran. Dans les exemples vus jusqu'ici, nous n'avons utilisé que l'instruction *Console.Out.WriteLine(expression)*.

La classe *System.Console* donne accès aux opérations d'écriture écran (**Write**, **WriteLine**). La classe *Console* a deux propriétés **Out** et **Error** qui sont des **flux d'écriture** de type *StreamWriter* :

- *Console.WriteLine()* est équivalent à *Console.Out.WriteLine()* et écrit sur le flux **Out** associé habituellement à l'écran.

- *Console.Error.WriteLine()* écrit sur le flux **Error**, habituellement associé lui aussi l'écran.

Les flux *Out* et *Error* sont associés par défaut l'écran. Mais ils peuvent être redirigés vers des fichiers texte au moment de l'exécution du programme comme nous le verrons prochainement.

1.3.2 Lecture de données tapées au clavier

Le flux de données provenant du clavier est désigné par l'objet *Console.In* de type *StreamReader*. Ce type d'objets permet de lire une ligne de texte avec la méthode *ReadLine* :

```
  String ligne=Console.In.readLine();
```

La ligne tapée au clavier est rangée dans la variable *ligne* et peut ensuite être exploitée par le programme.Le flux **In** peut être redirigé vers un fichier comme les flux **Out** et **Error**.

1.3.3 Exemple d'entrées-sorties

Voici un court programme d'illustration des opérations d'entrées-sorties clavier/écran :

```
using System;

public class io1{

    public static void Main (){

    // écriture sur le flux Out
    object obj=new object();
    Console.Out.WriteLine(""+obj);

    // écriture sur le flux Error
    int i=10;
    Console.Error.WriteLine("i="+i);

    // lecture d'une ligne saisie au clavier
    Console.Out.Write("Tapez une ligne : ");
    string ligne=Console.In.ReadLine();
    Console.Out.WriteLine("ligne="+ligne);
  }//fin main
}//fin classe
```

et les résultats de l'exécution :

```
System.Object
i=10
Tapez une ligne : je suis là
ligne=je suis là
```

Les instructions

```
    object obj=new object();
    Console.Out.WriteLine(""+obj);
```

ont pour but de montrer que n'importe quel objet peut faire l'objet d'un affichage. Nous ne chercherons pas ici à expliquer la signification de ce qui est affiché.

1.3.4 Redirection des E/S

Il existe sous DOS et UNIX trois périphériques stadard appelés :

1. périphérique d'entrée standard - désigne par défaut le clavier et porte le n° 0
2. périphérique de sortie standard - désigne par défaut l'écran et porte le n° 1
3. périphérique d'erreur standard - désigne par défaut l'écran et porte le n° 2

En C#, le flux d'écriture *Console.Out* écrit sur le périphérique 1, le flux d'écriture *Console.Error* écrit sur le périphérique 2 et le flux de lecture *Console.In* lit les données provenant du périphérique 0.

Lorsqu'on lance un programme sous Dos ou Unix, on peut fixer quels seront les périphériques 0, 1 et 2 pour le programme exécuté. Considérons la ligne de commande suivante :

```
pg arg1 arg2 .. argn
```

Derrière les arguments *argi* du programme *pg*, on peut rediriger les périphériques d'E/S standard vers des fichiers:

0<in.txt	le flux d'entrée standard n° 0 est redirigé vers le fichier *in.txt*. Dans le programme le flux *Console.In* prendra donc ses données dans le fichier *in.txt*.
1>out.txt	redirige la sortie n° 1 vers le fichier *out.txt*. Cela entraîne que dans le programme le flux *Console.Out* écrira ses données dans le fichier *out.txt*
1>>out.txt	idem, mais les données écrites sont ajoutées au contenu actuel du fichier *out.txt*.
2>error.txt	redirige la sortie n° 2 vers le fichier *error.txt*. Cela entraîne que dans le programme le flux *Console.Error* écrira ses données dans le fichier *error.txt*
2>>error.txt	idem, mais les données écrites sont ajoutées au contenu actuel du fichier *error.txt*.
1>out.txt 2>error.txt	Les périphériques 1 et 2 sont tous les deux redirigés vers des fichiers

On notera que pour rediriger les flux d'E/S du programme *pg* vers des fichiers, le programme *pg* n'a pas besoin d'être modifié. C'est l'OS qui fixe la nature des périphériques 0,1 et 2. Considérons le programme suivant :

```
// imports
using System;

// redirections
public class console2{
  public static void Main(String[] args){
    // lecture flux In
    string data=Console.In.ReadLine();
    // écriture flux Out
    Console.Out.WriteLine("écriture dans flux Out : " + data);
    // écriture flux Error
    Console.Error.WriteLine("écriture dans flux Error : " + data);
  }//Main
}//classe
```

Faisons une première exécution de ce programme :

```
E:\data\serge\MSNET\c#\bases\1>console2
test
écriture dans flux Out : test
écriture dans flux Error : test
```

L'exécution précédente ne redirige aucun des flux d'E/S standard **In**, **Out**, **Error**. Nos allons maintenant rediriger les trois flux. Le flux *In* sera redirigé vers un fichier *in.txt*, le flux *Out* vers le fichier *out.txt*, le flux *Error* vers le fichier *error.txt*. Cette redirection a lieu sur la ligne de commande sous la forme

```
E:\data\serge\MSNET\c#\bases\1>console2 0<in.txt 1>out.txt 2>error.txt
```

L'exécution donne les résultats suivants :

```
E:\data\serge\MSNET\c#\bases\1>more in.txt
test

E:\data\serge\MSNET\c#\bases\1>console2 0<in.txt 1>out.txt 2>error.txt

E:\data\serge\MSNET\c#\bases\1>more out.txt
écriture dans flux Out : test

E:\data\serge\MSNET\c#\bases\1>more error.txt
écriture dans flux Error : test
```

On voit clairement que les flux *Out* et *In* n'écrivent pas sur les mêmes périphériques.

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

On s'intéresse ici à l'opération *variable=expression;*

L'expression peut être de type : arithmétique, relationnelle, booléenne, caractères

1.3.5.1 Interprétation de l'opération d'affectation

L'opération *variable=expression;*

est elle-même une **expression** dont l'évaluation se déroule de la façon suivante :

- La partie droite de l'affectation est évaluée : le résultat est une valeur V.
- la valeur V est affectée à la variable
- la valeur V est aussi la valeur de l'affectation vue cette fois en tant qu'expression.

C'est ainsi que l'opération

V1=V2=expression

est légale. A cause de la priorité, c'est l'opérateur = le plus à droite qui va être évalué. On a donc
V1=(V2=expression)

L'expression *V2=expression* est évaluée et a pour valeur *V*. L'évaluation de cette expression a provoqué l'affectation de V à V2. L'opérateur = suivant est alors évalué sous la forme :

V1=V

La valeur de cette expression est encore V. Son évaluation provoque l'affectation de V à V1.

Ainsi donc, l'opération *V1=V2=expression*

est une expression dont l'évaluation

1 provoque l'affectation de la valeur de *expression* aux variables V1 et V2
2 rend comme résultat la valeur de *expression*.

On peut généraliser à une expression du type :

V1=V2=....=Vn=expression

1.3.5.2 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**.

Il existe diverses fonctions mathématiques. En voici quelques-unes :

double Sqrt(double x)	racine carrée
double Cos(double x)	Cosinus
double Sin(double x)	Sinus
double Tan(double x)	Tangente
double Pow(double x,double y)	x à la puissance y (x>0)
double Exp(double x)	Exponentielle
double Log(double x)	Logarithme népérien
double Abs(double x)	valeur absolue

etc...

Toutes ces fonctions 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 :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public sealed class Math :
    object
{

    // Fields
    public static const double E;
    public static const double PI;

    // Constructors

    // Methods
    public static long Abs(long value);
    public static int Abs(int value);
    public static short Abs(short value);
    public static SByte Abs(SByte value);
    public static double Abs(double value);
    public static Decimal Abs(Decimal value);
    public static float Abs(float value);
    public static double Acos(double d);
    public static double Asin(double d);
    public static double Atan(double d);
    public static double Atan2(double y, double x);
    public static double Ceiling(double a);
    public static double Cos(double d);
    public static double Cosh(double value);
    public virtual bool Equals(object obj);
    public static double Exp(double d);
    public static double Floor(double d);
    public virtual int GetHashCode();
    public Type GetType();
    public static double IEEERemainder(double x, double y);
    public static double Log(double a, double newBase);
    public static double Log(double d);
    public static double Log10(double d);
    public static Decimal Max(Decimal val1, Decimal val2);
    public static byte Max(byte val1, byte val2);
    public static short Max(short val1, short val2);
    public static UInt32 Max(UInt32 val1, UInt32 val2);
    public static UInt64 Max(UInt64 val1, UInt64 val2);
    public static long Max(long val1, long val2);
    public static int Max(int val1, int val2);
    public static double Max(double val1, double val2);
    public static float Max(float val1, float val2);
    public static UInt16 Max(UInt16 val1, UInt16 val2);
    public static SByte Max(SByte val1, SByte val2);
    public static int Min(int val1, int val2);
    public static UInt32 Min(UInt32 val1, UInt32 val2);
    public static short Min(short val1, short val2);
    public static UInt16 Min(UInt16 val1, UInt16 val2);
    public static long Min(long val1, long val2);
    public static double Min(double val1, double val2);
    public static Decimal Min(Decimal val1, Decimal val2);
    public static UInt64 Min(UInt64 val1, UInt64 val2);
    public static float Min(float val1, float val2);
    public static byte Min(byte val1, byte val2);
    public static SByte Min(SByte val1, SByte val2);
    public static double Pow(double x, double y);
    public static double Round(double a);
    public static Decimal Round(Decimal d);
    public static Decimal Round(Decimal d, int decimals);
    public static double Round(double value, int digits);
    public static int Sign(SByte value);
    public static int Sign(short value);
    public static int Sign(int value);
    public static int Sign(long value);
    public static int Sign(Decimal value);
    public static int Sign(double value);
    public static int Sign(float value);
    public static double Sin(double a);
    public static double Sinh(double value);
    public static double Sqrt(double d);
    public static double Tan(double a);
    public static double Tanh(double value);
    public virtual string ToString();
} // end of System.Math
```

1.3.5.3 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.

1.3.5.4 Expressions relationnelles

Les opérateurs sont les suivants :

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

priorités des opérateurs

1. >, >=, <, <=
2. ==, !=

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

```
boolean 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 ASCII, qui sont des nombres, qui sont alors comparés. On rappelle que selon l'ordre ASCII 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 comparaisons de deux chaînes

On peut utiliser ici les opérateurs relationnels <, <=, ==, !=, >, >= ou des méthodes de la classe *String* :

```
String chaine1, chaine2;
chaine1=…;
chaine2=…;
int i=chaine1.CompareTo(chaine2);
boolean egal=chaine1.Equals(chaine2)
```

Ci-dessus, la variable *i* 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

La variable *egal* aura la valeur *true* si les deux chaînes sont égales.

1.3.5.5 Expressions booléennes

Les opérateurs utilisables sont AND (&&) OR(||) NOT (!). Le résultat d'une expression booléenne est un booléen.

priorités des opérateurs :

1. !
2. &&
3. ||

```
int fin;
    int x;
    fin= x>2 && x<4;
```

Les opérateurs relationnels **ont priorité** sur les opérateurs && et ||.

1.3.5.6 Traitement de bits

Les opérateurs

Soient i et j deux entiers.

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

Soit

```
int i=0x123F, k=0xF123;
uint j=0xF123;
```

opération	valeur
i<<4	0x23F0
i>>4	0x0123 le bit de signe est préservé.
k>>4	0xFF12 le bit de signe est préservé.
i&j	0x1023
i|j	0xF33F
~i	0xEDC0

1.3.5.7 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 /, %,* ,<<, >>, &, |, ^. Ainsi *a=a+2;* peut s'écrire *a+=2;*

1.3.5.8 Opérateurs d'incrémentation 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*.

1.3.5.9 L'opérateur ?

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*. *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*. *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.

1.3.5.10 Priorité générale des opérateurs

```
() []  fonction                    gd
! ~ ++ --                          dg
new (type) opérateurs cast         dg
*  /  %                            gd
+  -                               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.

1.3.5.11 Les changements 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.

```
int i, j;
float isurj;
isurj= (float)i/j;   // priorité de () sur /
```

Ici il est nécessaire de changer le type de *i* ou *j* en réel sinon la division donnera le quotient entier et non réel.

- *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*.

1.4 Les instructions de contrôle du déroulement du programme

1.4.1 Arrêt

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

syntaxe *void Exit(int status)*
action arrête le processus en cours et rend la valeur *status* au processus père

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 à DOS dans la variable système **ERRORLEVEL** dont la valeur peut être testée dans un fichier batch. Sous Unix, 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.

1.4.2 Structure de choix simple

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*.

L'équivalent algorithmique de cette structure est la structure *si .. alors … sinon* :

```
si condition
    alors actions_condition_vraie
    sinon actions_condition_fausse
finsi
```

exemple

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

On peut imbriquer les structures de choix :

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

Se pose parfois le problème suivant :

```
public static void Main(){
    int n=5;

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

Dans l'exemple précédent, le *else* se rapporte à quel *if* ? La règle est qu'un *else* se rapporte toujours au *if* le plus proche : *if(n>6)* dans l'exemple. Considérons un autre exemple :

```
public static void Main()
{  int n=0;

   if(n>1)
     if(n>6)   Console.Out.WriteLine (">6");
     else;     // else du if(n>6) : rien à faire
   else Console.Out.WriteLine ("<=1"); // else du if(n>1)
}
```

Ici 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 un else au *if(n>6)*, dans lequel il n'y a aucune instruction.

1.4.3 Structure de cas

La syntaxe est la suivante :

```
switch(expression) {
  case v1:
      actions1;
      break;
  case v2:
      actions2;
      break;
     . .. .. .. .. ..
  default:
      actions_sinon;
      break;
}
```

notes

- La valeur de l'expression de contrôle, ne peut être qu'un entier.
- 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. Si elle est absente à la fin du bloc d'instructions de la valeur v_i, le compilateur signale une erreur.

exemple

En algorithmique

```
selon la valeur de choix
    cas 0
         arrêt
    cas 1
         exécuter module M1
    cas 2
        exécuter module M2
    sinon
         erreur<--vrai
findescas
```

En C#

```
     int choix=0; bool erreur=false;
     switch(choix){
       case 0: Environment.Exit(0);
       case 1: M1();break;
       case 2: M2();break;
       default: erreur=true;break;
     }
```

1.4.4 Structure de répétition

1.4.4.1 Nombre de répétitions connu

Structure for

La syntaxe est la suivante :

```
  for (i=id;i<=if;i=i+ip){
    actions;
    }
```

Notes

- les 3 arguments du *for* sont à l'intérieur d'une parenthèse et séparés par des points-virgules.
- chaque action du *for* est terminée par un point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

L'équivalent algorithmique est la structure *pour* :

```
pour i variant de id à if avec un pas de ip
    actions
finpour
```

qu'on peut traduire par une structure *tantque* :

```
i ← id
tantque i<=if
      actions
      i← i+ip
fintantque
```

Structure foreach

La syntaxe est la suivante :

```
foreach (type variable in expression)
  instructions;
}
```

Notes

- *expression* est une collection d'objets. La collection d'objets 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, toutes les valeurs de la collection.

Ainsi le code suivant :

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

afficherait :

```
paul
hélène
jacques
sylvie
```

1.4.4.2 Nombre de répétitions inconnu

Il existe de nombreuses structures en C# pour ce cas.

Structure tantque (while)

```
  while(condition){
```

```
        actions;
    }
```

On boucle tant que la condition est vérifiée. La boucle peut ne jamais être exécutée.

notes:

- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

La structure algorithmique correspondante est la structure tantque :

```
tantque condition
    actions
fintantque
```

Structure répéter jusqu'à (do while)

La syntaxe est la suivante :

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

On boucle jusqu'à ce que la condition devienne fausse ou tant que la condition est vraie. Ici la boucle est faite au moins une fois.

notes

- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

La structure algorithmique correspondante est la structure *répéter … jusqu'à* :

```
répéter
    actions
jusqu'à condition
```

Structure pour générale (for)

La syntaxe est la suivante :

```
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

- les différentes instructions dans *instructions_depart* et *instructions_fin_boucle* sont séparées par des virgules.

La structure algorithmique correspondante est la suivante :

```
instructions_départ
tantque condition
    actions
    instructions_fin_boucle
fintantque
```

Exemples

Les programmes suivants calculent tous la somme des n premiers nombres entiers.

```
1  for(i=1, somme=0;i<=n;i=i+1)
     somme=somme+a[i];

2   for (i=1, somme=0;i<=n;somme=somme+a[i], i=i+1);

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

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

1.4.4.3 Instructions de gestion de boucle

break fait sortir de la boucle for, while, do ... while.
continue fait passer à l'itération suivante des boucles for, while, do ... while

1.5 La structure d'un programme C#

Un programme C# n'utilisant pas de classe définie par l'utilisateur ni de fonctions autres que la fonction principale *Main* pourra avoir la structure suivante :

```
public class test1{
  public static void Main(){
    … code du programme
  }// main
}// class
```

Si on utilise des fonctions susceptibles de générer des exceptions qu'on ne souhaite pas gérer finement, on pourra encadrer le code du programme par une clause **try / catch** :

```
public class test1{
  public static void Main(){
    try{
    … code du programme
    } catch (Exception e){
      // gestion de l'erreur
    }// try
  }// main
}// class
```

1.6 Compilation et exécution d'un programme C#

DOS>csc test1.cs

produit un fichier *test1.exe* interprétable par la plate-forme .NET. Le programme csc.exe est le compilateur C# et se trouve dans l'arborescence du SDK, par exemple *C:\WINNT\Microsoft.NET\Framework\v1.0.2914\csc.exe*. Si l'installation de .NET s'est faite correctement, l'exécutable *csc.exe* doit être dans le chemin des exécutables du DOS.

DOS> test1

exécute le fichier *test1.exe*.

1.7 L'exemple IMPOTS

On se propose d'écrire un programme permettant de calculer l'impôt d'un contribuable. On se place dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

- on calcule le nombre de parts du salarié **nbParts=nbEnfants/2 +1** s'il n'est pas marié, **nbEnfants/2+2** s'il est marié, où *nbEnfants* est son nombre d'enfants.
- s'il a au moins trois enfants, il a une demi-part de plus
- on calcule son revenu imposable **R=0.72*S** où S est son salaire annuel
- on calcule son coefficient familial **QF=R/nbParts**
- on calcule son impôt **I**. Considérons le tableau suivant :

12620.0	0	0
13190	0.05	631
15640	0.1	1290.5
24740	0.15	2072.5
31810	0.2	3309.5
39970	0.25	4900
48360	0.3	6898.5
55790	0.35	9316.5
92970	0.4	12106
127860	0.45	16754.5
151250	0.50	23147.5
172040	0.55	30710
195000	0.60	39312
0	0.65	49062

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=23000 on trouvera la ligne

24740 0.15 2072.5

L'impôt I est alors égal à **0.15*R - 2072.5*nbParts**. Si QF est tel que la relation *QF<=champ1* n'est jamais vérifiée, alors ce sont les coefficients de la dernière ligne qui sont utilisés. Ici :

0 0.65 49062

ce qui donne l'impôt I=0.65*R - 49062*nbParts.

Le programme C# correspondant est le suivant :

```
using System;

public class impots{

  // ------------ main
  public static void Main(){

    // tableaux de données nécessaires au calcul de l'impôt
    decimal[] Limites=new decimal[]
{12620M,13190M,15640M,24740M,31810M,39970M,48360M,55790M,92970M,127860M,151250M,172040M,195000M,0M};
    decimal[] CoeffN=new decimal[]
{0M,631M,1290.5M,2072.5M,3309.5M,4900M,6898.5M,9316.5M,12106M,16754.5M,23147.5M,30710M,39312M,49062M};

    // on récupère le statut marital
    bool OK=false;
    string reponse=null;
    while(! OK){
      Console.Out.Write("Etes-vous marié(e) (O/N) ? ");
      reponse=Console.In.ReadLine().Trim().ToLower();
      if (reponse!="o" && reponse!="n")
          Console.Error.WriteLine("Réponse incorrecte. Recommencez");
        else OK=true;
    }//while
    bool Marie = reponse=="o";

    // nombre d'enfants
    OK=false;
    int NbEnfants=0;
    while(! OK){
      Console.Out.Write("Nombre d'enfants : ");
      reponse=Console.In.ReadLine();
      try{
        NbEnfants=int.Parse(reponse);
        if(NbEnfants>=0) OK=true;
        else Console.Error.WriteLine("Réponse incorrecte. Recommencez");
      } catch(Exception){
        Console.Error.WriteLine("Réponse incorrecte. Recommencez");
      }// try
    }// while

    // salaire
    OK=false;
    int Salaire=0;
    while(! OK){
      Console.Out.Write("Salaire annuel : ");
      reponse=Console.In.ReadLine();
      try{
        Salaire=int.Parse(reponse);
        if(Salaire>=0) OK=true;
        else Console.Error.WriteLine("Réponse incorrecte. Recommencez");
```

```
      } catch(Exception){
        Console.Error.WriteLine("Réponse incorrecte. Recommencez");
      }// try
    }// while

      // calcul du nombre de parts
    decimal NbParts;
    if(Marie) NbParts=(decimal)NbEnfants/2+2;
      else NbParts=(decimal)NbEnfants/2+1;
    if (NbEnfants>=3) NbParts+=0.5M;

      // revenu imposable
    decimal Revenu;
    Revenu=0.72M*Salaire;

      // quotient familial
    decimal QF;
    QF=Revenu/NbParts;

      // recherche de la tranche d'impots correspondant à QF
    int i;
    int NbTranches=Limites.Length;
    Limites[NbTranches-1]=QF;
    i=0;
    while(QF>Limites[i]) i++;
      // l'impôt
    int impots=(int)(i*0.05M*Revenu-CoeffN[i]*NbParts);

    // on affiche le résultat
    Console.Out.WriteLine("Impôt à payer : " + impots);
  }// main
}// classe
```

Le programme est compilé dans une fenêtre Dos par :

```
E:\data\serge\MSNET\c#\impots\4>C:\WINNT\Microsoft.NET\Framework\v1.0.2914\csc.exe impots.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.
```

La compilation produit un exécutable *impots.exe* :

```
E:\data\serge\MSNET\c#\impots\4>dir
30/04/2002  16:16                2 274 impots.cs
30/04/2002  16:16                5 120 impots.exe
```

Il faut noter que impots.exe n'est pas directement exécutable par le processeur mais uniquement. Il contient en réalité du code intermédiaire qui n'est exécutable que sur une plate-forme .NET. Les résultats obtenus sont les suivants :

```
E:\data\serge\MSNET\c#\impots\4>impots
Etes-vous marié(e) (O/N) ? o
Nombre d'enfants : 3
Salaire annuel : 200000
Impôt à payer : 16400
```

```
E:\data\serge\MSNET\c#\impots\4>impots
Etes-vous marié(e) (O/N) ? n
Nombre d'enfants : 2
Salaire annuel : 200000
Impôt à payer : 33388
```

```
E:\data\serge\MSNET\c#\impots\4>impots
Etes-vous marié(e) (O/N) ? w
Réponse incorrecte. Recommencez
Etes-vous marié(e) (O/N) ? q
Réponse incorrecte. Recommencez
Etes-vous marié(e) (O/N) ? o
Nombre d'enfants : q
Réponse incorrecte. Recommencez
Nombre d'enfants : 2
Salaire annuel : q
Réponse incorrecte. Recommencez
Salaire annuel : 1
Impôt à payer : 0
```

1.8 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 P avec la commande :

P arg0 arg1 … argn

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

public static void main(String[] arg);

on aura arg[0]="arg0", arg[1]="arg1" … Voici un exemple :

```
// imports
using System;

public class arg1{
  public static void Main(String[] args){
    // on liste les paramètres
    Console.Out.WriteLine("Il y a  " + args.Length + " arguments");
    for (int i=0;i<args.Length;i++){
      Console.Out.WriteLine("arguments["+i+"]="+args[i]);
    }
  }//Main
}//classe
```

L'exécution donne les résultats suivants :

```
E:\data\serge\MSNET\c#\bases\0>arg1
Il y a  0 arguments

E:\data\serge\MSNET\c#\bases\0>arg1 a b c
Il y a  3 arguments
arguments[0]=a
arguments[1]=b
arguments[2]=c
```

Dans la déclaration de la méthode statique *Main*

```
  public static void Main(String[] args){
```

le paramètre **args** est un tableau de chaînes de caractères qui reçoit les arguments passés sur la ligne de commande lors de l'appel du programme.

- *args.Length* est le nombre d'élements du tableau *args*
- *args[i]* est l'élément i du tableau

1.9 Les énumérations

Une énumération est un type de données dont le domaine de valeurs est un ensemble de cosntantes entières. Considérons un programme qui a 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 mention {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
    mention maMention=mention.Passable;
```

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

```
    if(maMention==mention.Passable){
      Console.Out.WriteLine("Peut mieux faire");
    }//if
```

On peut obtenir toutes les valeurs de l'énumération :

```
    // liste des mentions
    foreach(mention m in Enum.GetValues(maMention.GetType())){
```

```
      Console.Out.WriteLine(m);
    }//foreach
```

De la même façon que le type simple *int* est équivalent à la classe *Int32*, le type simple *enum* est équivalent à la classe *Enum*. Cette classe a une méthode statique *GetValues* qui permet d'obtenir toutes les valeurs d'un type énuméré que l'on passe en paramètre. Celui-ci doit être un objet de type **Type** qui est une classe d'information sur le type d'une donnée. Le type d'une variable *v* est obtenu par *v.GetType()*. Donc ici *maMention.GetType()* donne l'objet *Type* de l'énumération *mentions* et *Enum.GetValues(maMention.GetType())* la liste des valeurs de l'énumération *mentions*.

Si on écrit maintenant

```
    foreach(int m in Enum.GetValues(maMention.GetType())){
      Console.Out.WriteLine(m);
    }//foreach
```

on obtiendra la liste des valeurs de l'énumération sous forme d'entiers. C'est ce que montre le programme suivant :

```
// énumération

using System;

public class intro{
  // une énumération
  enum mention {Passable,AssezBien,Bien,TrèsBien, Excellent};
  public static void Main(){
    // une variable qui prend ses valeurs dans l'énumération mentions
    mention maMention=mention.Passable;
    // affichage valeur variable
    Console.Out.WriteLine("mention="+maMention);
    // test avec valeur de l'énumération
    if(maMention==mention.Passable){
      Console.Out.WriteLine("Peut mieux faire");
    }//if
    // liste des mentions
    foreach(mention m in Enum.GetValues(maMention.GetType())){
      Console.Out.WriteLine(m);
    }//foreach
    foreach(int m in Enum.GetValues(maMention.GetType())){
      Console.Out.WriteLine(m);
    }//foreach
  }//Main
}//classe
```

Les résultats d'exécution sont les suivants :

```
mention=Passable
Peut mieux faire
Passable
AssezBien
Bien
TrèsBien
0
1
2
3
```

1.10 La gestion des exceptions

De nombreuses fonctions C# sont susceptibles de générer des exceptions, c'est à dire des erreurs. Lorsqu'une fonction est susceptible de générer une exception, le programmeur devrait la gérer dans le but d'obtenir des programmes plus résistants aux erreurs : il faut toujours éviter le "plantage" sauvage d'une application.

La gestion d'une exception se fait selon le schéma suivant :

try{
appel de la fonction susceptible de générer l'exception
} catch (Exception e){
traiter l'exception e
}
instruction suivante

Si la fonction ne génère pas d'exception, on passe alors à *instruction suivante*, sinon on passe dans le corps de la clause *catch* puis à *instruction suivante*. Notons les points suivants :

- **e** est un objet dérivé du type *Exception*. On peut être plus précis en utilisant des types tels que *IOException*, *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{
 appel de la fonction susceptible de générer l'exception
} catch (IOException e){
 traiter l'exception e
}
} catch (SystemException e){
 traiter l'exception e
}
instruction suivante
```

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

```
try{
   appel de la fonction susceptible de générer l'exception
 } catch (Exception e){
   traiter l'exception e
 }
 finally{
   code exécuté après try ou catch
 }
 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 *catch {...}*.
- La classe *Exception* a une propriété **Message** qui est un message détaillant l'erreur qui s'est produite. Ainsi si on veut afficher celui-ci, on écrira :

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

- 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.Error.WriteLine("L'erreur suivante s'est produite : "+ex.ToString());
  ...
}//catch
```

On peut écrire aussi :

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

Nous avons ici une opération *string + Exception* qui va être automatiquement transformée en *string + Exception.ToString()* par le compilateur afin de faire la concaténation de deux chaînes de caractères.

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

```
// tableaux

// imports
using System;

public class tab1{
  public static void Main(String[] args){
    // déclaration & initialisation d'un tableau
```

```
    int[] tab=new int[] {0,1,2,3};
    int i;
    // affichage tableau avec un for
    for (i=0; i<tab.Length; i++)
      Console.Out.WriteLine("tab[" + i + "]=" + tab[i]);
    // affichage tableau avec un for each
    foreach (int élmt in tab)
       Console.Out.WriteLine(élmt);
    // génération d'une exception
    try{
      tab[100]=6;
    }catch (Exception e){
      Console.Error.WriteLine("L'erreur suivante s'est produite : " + e);
    }//try-catch
    finally{
     String attente=Console.ReadLine();
    }
  }//Main
}//classe
```

L'exécution du programme donne les résultats suivants :

```
E:\data\serge\MSNET\c#\bases\2>tab1
tab[0]=0
tab[1]=1
tab[2]=2
tab[3]=3
0
1
2
3
L'erreur suivante s'est produite : System.IndexOutOfRangeException: Exception of
 type System.IndexOutOfRangeException was thrown.
   at tab1.Main(String[] args)
```

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

```
// imports
using System;

public class console1{
  public static void Main(String[] args){
    // On demande le nom
    System.Console.Write("Nom : ");
    // lecture réponse
    String nom=System.Console.ReadLine();
    // on demande l'âge
    int age=0;
    Boolean ageOK=false;
    while ( ! ageOK){
      // question
      Console.Out.Write("âge : ");
      // lecture-vérification réponse
      try{
        age=int.Parse(System.Console.ReadLine());
        ageOK=true;
      }catch {
        Console.Error.WriteLine("Age incorrect, recommencez...");
      }//try-catch
    }//while
    // affichage final
    Console.Out.WriteLine("Vous vous appelez " + nom + " et vous avez " + age + " ans");
    Console.ReadLine();
  }//Main
}//classe
```

Quelques résultats d'exécution :

```
E:\data\serge\MSNET\c#\bases\1>console1
Nom : dupont
âge : 23
Vous vous appelez dupont et vous avez 23 ans
```

```
E:\data\serge\MSNET\c#\bases\1>console1
Nom : dupont
âge : xx
Age incorrect, recommencez...
âge : 12
Vous vous appelez dupont et vous avez 12 ans
```

1.11 Passage de paramètres à une fonction

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

```
  private static void changeInt(int a){
    a=30;
    Console.Out.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 utlisation de cette fonction :

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

Ici dans l'instruction `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.

1.11.1 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 recopiée 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.

```
// passage de paramètres par valeur à une fonction

using System;

public class param2{
  public static void Main(){
    int age=20;
    changeInt(age);
    Console.Out.WriteLine("Paramètre effectif age="+age);
  }
  private static void changeInt(int a){
    a=30;
    Console.Out.WriteLine("Paramètre formel a="+a);
  }
}
```

Les résultats obtenus sont les suivants :

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

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

1.11.2 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 :

```
// passage de paramètres par valeur à une fonction

using System;

public class param2{
  public static void Main(){
    int age=20;
    changeInt(ref age);
    Console.Out.WriteLine("Paramètre effectif age="+age);
  }
  private static void changeInt(ref int a){
    a=30;
    Console.Out.WriteLine("Paramètre formel a="+a);
  }
}
```

et les résultats d'exécution :

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

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

1.11.3 Passage par référence avec le mot clé out

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

```
// passage de paramètres par valeur à une fonction

using System;

public class param2{
  public static void Main(){
    int age;
    changeInt(ref age);
    Console.Out.WriteLine("Paramètre effectif age="+age);
  }
  private static void changeInt(ref int a){
    a=30;
    Console.Out.WriteLine("Paramètre formel a="+a);
  }
}
```

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

```
   Use of unassigned local variable 'age'
```

On peut contourner l'obstacle en affectant une valeur initiale à *age*. 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 :

```
// passage de paramètres par valeur à une fonction

using System;

public class param2{
  public static void Main(){
    int age=20;
    changeInt(out age);
    Console.Out.WriteLine("Paramètre effectif age="+age);
  }
  private static void changeInt(out int a){
    a=30;
    Console.Out.WriteLine("Paramètre formel a="+a);
  }
}
```

2. Classes, stuctures, interfaces

2.1 L' objet par l'exemple

2.1.1 Généralités

Nous abordons maintenant, par l'exemple, la programmation objet. Un objet est une entité qui contient des données qui définissent son état (on les appelle des propriétés) et des fonctions (on les appelle des méthodes). Un objet est créé selon un modèle qu'on appelle une classe :

```
public class C1{
  type1 p1;    // propriété p1
  type2 p2;    // propriété p2
  …
  type3 m3(…){   // méthode m3
    …
  }
  type4 m4(…){   // méthode m4
    …
  }
  …
}
```

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

```
  int i,j;
```

crée deux objets (le terme est incorrect ici) de type (classe) *int*. Leur seule propriété est leur valeur.

Si *O1* est un objet de type *C1*, *O1.p1* désigne la propriété *p1* de *O1* et *O1.m1* la méthode *m1* de *O1*.

Considérons un premier modèle d'objet : la classe *personne*.

2.1.2 Définition de la classe personne

La définition de la classe *personne* sera la suivante :

```
public class personne{
  // attributs
  private string prenom;
  private string nom;
  private int age;

  // méthode
  public void initialise(string P, string N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
  public void identifie(){
    Console.Out.WriteLine(prenom+","+nom+","+age);
  }
}
```

Nous avons ici la définition d'une classe, donc d'un type de données. Lorsqu'on va créer des variables de ce type, on les appellera des objets ou des instances de classes. Une classe est donc un moule à partir duquel sont construits des objets.

Les membres ou champs d'une classe peuvent être des données (attributs), des méthodes (fonctions), des propriétés. Les propriétés sont des méthodes particulières servant à connaître ou fixer la valeur d'attributs de l'objet. Ces champs peuvent être accompagnés de l'un des trois mots clés suivants :

privé Un champ privé (private) n'est accessible que par les seules méthodes internes de la classe
public Un champ public (public) est accessible par toute fonction définie ou non au sein de la classe
protégé Un champ protégé (protected) n'est accessible que par les seules méthodes internes de la classe ou d'un objet dérivé (voir ultérieurement le concept d'héritage).

En général, les données d'une classe sont déclarées privées alors que ses méthodes et propriétés sont déclarées publiques. Cela signifie que l'utilisateur d'un objet (le programmeur)

- n'aura pas accès directement aux données privées de l'objet
- pourra faire appel aux méthodes publiques de l'objet et notamment à celles qui donneront accès à ses données privées.

La syntaxe de déclaration d'un objet est la suivante :

```
public class objet{
  private  donnée ou méthode ou propriété privée
  public  donnée ou méthode ou propriété publique
  protected  donnée ou méthode ou propriété protégée
}
```

L'ordre de déclaration des attributs *private, protected et public* est quelconque.

2.1.3 La méthode initialise

Revenons à notre classe personne déclarée comme :

```
public class personne{
  // attributs
  private string prenom;
  private string nom;
  private int age;

  // méthode
  public void initialise(string P, string N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
  public void identifie(){
    Console.Out.WriteLine(prenom+","+nom+","+age);
  }
}
```

Quel est le rôle de la méthode *initialise* ? Parce que *nom*, *prenom* et *age* sont des données privées de la classe *personne*, les instructions :

```
personne p1;
p1.prenom="Jean";
p1.nom="Dupont";
p1.age=30;
```

sont illégales. Il nous faut initialiser un objet de type *personne* via une méthode publique. C'est le rôle de la méthode *initialise*. On écrira :

```
personne p1;
p1.initialise("Jean","Dupont",30);
```

L'écriture *p1.initialise* est légale car *initialise* est d'accès public.

2.1.4 L'opérateur new

La séquence d'instructions

```
personne p1;
p1.initialise("Jean","Dupont",30);
```

est incorrecte. L'instruction

```
  personne p1;
```

déclare *p1* comme une référence à un objet de type *personne*. Cet objet n'existe pas encore et donc *p1* n'est pas initialisé. C'est comme si on écrivait :

```
personne p1=null;
```

où on indique explicitement avec le mot clé *null* que la variable *p1* ne référence encore aucun objet. Lorsqu'on écrit ensuite

```
p1.initialise("Jean","Dupont",30);
```

on fait appel à la méthode *initialise* de l'objet référencé par *p1*. Or cet objet n'existe pas encore et le compilateur signalera l'erreur. Pour que *p1* référence un objet, il faut écrire :

```
personne p1=new personne();
```

Cela a pour effet de créer un objet de type *personne* non encore initialisé : les attributs *nom* et *prenom* qui sont des références d'objets de type *String* auront la valeur *null*, et *age* la valeur *0*. Il y a donc une initialisation par défaut. Maintenant que *p1* référence un objet, l'instruction d'initialisation de cet objet

```
p1.initialise("Jean","Dupont",30);
```

est valide.

2.1.5 Le mot clé this

Regardons le code de la méthode *initialise* :

```
public void initialise(string P, string N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }
```

L'instruction *this.prenom=P* signifie que l'attribut *prenom* de l'objet courant (*this*) reçoit la valeur *P*. Le mot clé *this* désigne l'objet courant : celui dans lequel se trouve la méthode exécutée. Comment le connaît-on ? Regardons comment se fait l'initialisation de l'objet référencé par p1 dans le programme appelant :

```
p1.initialise("Jean","Dupont",30);
```

C'est la méthode *initialise* de l'objet *p1* qui est appelée. Lorsque dans cette méthode, on référence l'objet *this*, on référence en fait l'objet *p1*. La méthode *initialise* aurait aussi pu être écrite comme suit :

```
public void initialise(string P, string N, int age){
   prenom=P;
   nom=N;
   this.age=age;
  }
```

Lorsqu'une méthode d'un objet référence un attribut *A* de cet objet, l'écriture *this.A* est implicite. On doit l'utiliser explicitement lorsqu'il y a conflit d'identificateurs. C'est le cas de l'instruction :

```
this.age=age;
```

où *age* désigne un attribut de l'objet courant ainsi que le paramètre *age* reçu par la méthode. Il faut alors lever l'ambiguïté en désignant l'attribut *age* par *this.age*.

2.1.6 Un programme de test

Voici un court programme de test :

```
using System;

public class personne{
  // attributs
  private string prenom;
  private string nom;
  private int age;

  // méthode
  public void initialise(string P, string N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
```

```
  public void identifie(){
    Console.Out.WriteLine(prenom+","+nom+","+age);
  }
}

public class test1{
  public static void Main(){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
    p1.identifie();
  }
}
```

et les résultats obtenus :

```
E:\data\serge\MSNET\c#\objetsPoly\1>C:\WINNT\Microsoft.NET\Framework\v1.0.2914\csc.exe personne1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.
```

```
E:\data\serge\MSNET\c#\objetsPoly\1>personne1
Jean,Dupont,30
```

2.1.7 Utiliser un fichier de classes compilées (assembly)

On notera que dans l'exemple précédent il y a deux classes dans notre programme de test : les classes *personne* et *test1*. Il y a une autre façon de procéder :

- on compile la classe personne dans un fichier particulier appelé un assemblage (assembly). Ce fichier a une extension .dll
- on compile la classe test1 en référençant l'assemblage qui contient la classe personne.

Les deux fichiers source deviennent les suivants :

test1.cs

```
public class test1{
  public static void Main(){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
    p1.identifie();
  }
}//classe test1
```

personne.cs

```
using System;

public class personne{
  // attributs
  private string prenom;
  private string nom;
  private int age;

  // méthode
  public void initialise(string P, string N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
  public void identifie(){
    Console.Out.WriteLine(prenom+","+nom+","+age);
  }
}// classe personne
```

La classe *personne* est compilée par l'instruction suivante :

```
E:>csc.exe /t:library personne.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.
```

```
E:\data\serge\MSNET\c#\objetsPoly\2>dir
26/04/2002  08:24                 520 personne.cs
26/04/2002  08:26                 169 test1.cs
26/04/2002  08:26               3 584 personne.dll
```

La compilation a produit un fichier *personne.dll*. C'est l'option de compilation */t:library* qui indique de produire un fichier "assembly". Maintenant compilons le fichier *test1.cs* :

```
E:\data\serge\MSNET\c#\objetsPoly\2>csc /r:personne.dll test1.cs
```

```
E:\data\serge\MSNET\c#\objetsPoly\2>dir
26/04/2002  08:24                 520 personne.cs
26/04/2002  08:26                 169 test1.cs
26/04/2002  08:26               3 584 personne.dll
26/04/2002  08:53               3 072 test1.exe
```

L'option de compilation */r:personne.dll* indique au compilateur qu'il trouvera certaines classes dans le fichier *personne.dll*. Lorsque dans le fichier source *test1.cs*, il trouvera une référence à la classe *personne* classe non déclarée dans le source *test1.cs*, il cherchera la classe *personne* dans les fichiers .dll référencés par l'option */r*. Il trouvera ici la classe *personne* dans l'assemblage *personne.dll*. On aurait pu mettre dans cet assemblage d'autres classes. Pour utiliser lors de la compilation plusieurs fichiers de classes compilées, on écrira :

```
csc /r:fic1.dll /r:fic2.dll ... fichierSource.cs
```

L'exécution du programme *test1.exe* donne les résultats suivants :

```
E:\data\serge\MSNET\c#\objetsPoly\2>test1
Jean,Dupont,30
```

2.1.8 Une autre méthode initialise

Considérons toujours la classe *personne* et rajoutons-lui la méthode suivante :

```
public void initialise(personne P){
    prenom=P.prenom;
    nom=P.nom;
    this.age=P.age;
  }
```

On a maintenant deux méthodes portant le nom *initialise* : c'est légal tant qu'elles admettent des paramètres différents. C'est le cas ici. Le paramètre est maintenant une référence *P* à une personne. Les attributs de la personne *P* sont alors affectés à l'objet courant (*this*). On remarquera que la méthode *initialise* a un accès direct aux attributs de l'objet *P* bien que ceux-ci soient de type *private*. C'est toujours vrai : un objet *O1* d'une classe *C* a toujours accès aux attributs des objets de la même classe *C*.

Voici un test de la nouvelle classe *personne*, celle-ci ayant été compilée dans *personne.dll* comme il a été expliqué précédemment :

```
using System;

public class test1{
  public static void Main(){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
    Console.Out.Write("p1=");
    p1.identifie();
    personne p2=new personne();
    p2.initialise(p1);
    Console.Out.Write("p2=");
    p2.identifie();
  }
}
```

et ses résultats :

```
p1=Jean,Dupont,30
p2=Jean,Dupont,30
```

2.1.9 Constructeurs de la classe personne

Un constructeur est une méthode qui porte le nom de la classe et qui est appelée lors de la création de l'objet. On s'en sert généralement pour l'initialiser. C'est une méthode qui peut accepter des arguments mais qui ne rend aucun résultat. Son prototype ou sa définition ne sont précédés d'aucun type (même pas *void*).

Si une classe a un constructeur acceptant n arguments *argi*, la déclaration et l'initialisation d'un objet de cette classe pourra se faire sous la forme :

classe objet =new classe(*arg1*,*arg2*, ... *argn*);

ou

classe objet;
…
objet=new classe(*arg1*,*arg2*, ... *argn*);

Lorsqu'une classe a un ou plusieurs constructeurs, l'un de ces constructeurs doit être obligatoirement utilisé pour créer un objet de cette classe. Si une classe *C* n'a aucun constructeur, elle en a un par défaut qui est le constructeur sans paramètres : *public C()*. Les attributs de l'objet sont alors initialisés avec des valeurs par défaut. C'est ce qui s'est passé lorsque dans les programmes précédents, où on avait écrit :

```
  personne p1;
  p1=new personne();
```

Créons deux constructeurs à notre classe *personne* :

```
using System;

public class personne{
  // attributs
  private string prenom;
  private string nom;
  private int age;

  // constructeurs
  public personne(String P, String N, int age){
    initialise(P,N,age);
  }
  public personne(personne P){
    initialise(P);
  }

  // méthodes d'initialisation de l'objet
  public void initialise(string P, string N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }
  public void initialise(personne P){
    prenom=P.prenom;
    nom=P.nom;
    this.age=P.age;
  }

  // méthode
  public void identifie(){
    Console.Out.WriteLine(prenom+","+nom+","+age);
  }
}
```

Nos deux constructeurs se contentent de faire appel aux méthodes *initialise* correspondantes. On rappelle que lorsque dans un constructeur, on trouve la notation *initialise(P)* par exemple, le compilateur traduit par *this.initialise(P)*. Dans le constructeur, la méthode *initialise* est donc appelée pour travailler sur l'objet référencé par *this*, c'est à dire l'objet courant, celui qui est en cours de construction.

Voici un court programme de test :

```
using System;

public class test1{
  public static void Main(){
    personne p1=new personne("Jean","Dupont",30);
    Console.Out.Write("p1=");
    p1.identifie();
    personne p2=new personne(p1);
    Console.Out.Write("p2=");
    p2.identifie();
  }
}
```

et les résultats obtenus :

```
p1=Jean,Dupont,30
p2=Jean,Dupont,30
```

2.1.10 Les références d'objets

Nous utilisons toujours la même classe *personne*. Le programme de test devient le suivant :

```
using System;
```

```
public class test1{
  public static void Main(){
    // p1
    personne p1=new personne("Jean","Dupont",30);
    Console.Out.Write("p1=");  p1.identifie();
    // p2 référence le même objet que p1
    personne p2=p1;
    Console.Out.Write("p2="); p2.identifie();
    // p3 référence un objet qui sera une copie de l'objet référencé par p1
    personne p3=new personne(p1);
    Console.Out.Write("p3=");  p3.identifie();
    // on change l'état de l'objet référencé par p1
    p1.initialise("Micheline","Benoît",67);
    Console.Out.Write("p1=");  p1.identifie();
    // comme p2=p1, l'objet référencé par p2 a du changer d'état
    Console.Out.Write("p2=");  p2.identifie();
    // comme p3 ne référence pas le même objet que p1, l'objet référencé par p3 n'a pas du changer
    Console.Out.Write("p3=");  p3.identifie();
  }
}
```

Les résultats obtenus sont les suivants :

```
p1=Jean,Dupont,30
p2=Jean,Dupont,30
p3=Jean,Dupont,30
p1=Micheline,Benoît,67
p2=Micheline,Benoît,67
p3=Jean,Dupont,30
```

Lorsqu'on déclare la variable *p1* par

```
personne p1=new personne("Jean","Dupont",30);
```

p1 référence l'objet *personne("Jean","Dupont",30)* mais n'est pas l'objet lui-même. En C, on dirait que c'est un pointeur, c.a.d. l'adresse de l'objet créé. Si on écrit ensuite :

```
  p1=null
```

Ce n'est pas l'objet *personne("Jean","Dupont",30)* qui est modifié, c'est la référence *p1* qui change de valeur. L'objet *personne("Jean","Dupont",30)* sera "perdu" s'il n'est référencé par aucune autre variable.

Lorsqu'on écrit :

```
personne p2=p1;
```

on initialise le pointeur *p2* : il "pointe" sur le même objet (il désigne le même objet) que le pointeur *p1*. Ainsi si on modifie l'objet "pointé" (ou référencé) par *p1*, on modifie celui référencé par *p2*.

Lorsqu'on écrit :

```
personne p3=new personne(p1);
```

il y a création d'un nouvel objet, copie de l'objet référencé par *p1*. Ce nouvel objet sera référencé par *p3*. Si on modifie l'objet "pointé" (ou référencé) par *p1*, on ne modifie en rien celui référencé par *p3*. C'est ce que montrent les résultats obtenus.

2.1.11 Les objets temporaires

Dans une expression, on peut faire appel explicitement au constructeur d'un objet : celui-ci est construit, mais nous n'y avons pas accès (pour le modifier par exemple). Cet objet temporaire est construit pour les besoins d'évaluation de l'expression puis abandonné. L'espace mémoire qu'il occupait sera automatiquement récupéré ultérieurement par un programme appelé "ramasse-miettes" dont le rôle est de récupérer l'espace mémoire occupé par des objets qui ne sont plus référencés par des données du programme.

Considérons le nouveau programme de test suivant :

```
using System;

public class test1{
  public static void Main(){
    new personne(new personne("Jean","Dupont",30)).identifie();
  }
}
```

et modifions les constructeurs de la classe *personne* afin qu'ils affichent un message :

```
// constructeurs
  public personne(String P, String N, int age){
    Console.Out.WriteLine("Constructeur personne(String, String, int)");
    initialise(P,N,age);
  }
  public personne(personne P){
    Console.Out.WriteLine("Constructeur personne(personne)");
    initialise(P);
  }
```

Nous obtenons les résultats suivants :

```
Constructeur personne(String, String, int)
Constructeur personne(personne)
Jean,Dupont,30
```

montrant la construction successive des deux objets temporaires.

2.1.12 Méthodes de lecture et d'écriture des attributs privés

Nous rajoutons à la classe *personne* les méthodes nécessaires pour lire ou modifier l'état des attributs des objets :

```
using System;

public class personne{

  // attributs
  private String prenom;
  private String nom;
  private int age;

  // constructeurs
  public personne(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  public personne(personne P){
    this.prenom=P.prenom;
    this.nom=P.nom;
    this.age=P.age;
  }

  // identifie
  public void identifie(){
    Console.Out.WriteLine(prenom+","+nom+","+age);
  }

  // accesseurs
  public String getPrenom(){
    return prenom;
  }
  public String getNom(){
    return nom;
  }
  public int getAge(){
    return age;
  }

  //modifieurs
  public void setPrenom(String P){
    this.prenom=P;
  }
  public void setNom(String N){
    this.nom=N;
  }
  public void setAge(int age){
    this.age=age;
  }
}//classe
```

Nous testons la nouvelle classe avec le programme suivant :

```
using System;

public class test1{
  public static void Main(){
    personne P=new personne("Jean","Michelin",34);
```

```
    Console.Out.WriteLine("P=("+P.getPrenom()+","+P.getNom()+","+P.getAge()+")");
    P.setAge(56);
    Console.Out.WriteLine("P=("+P.getPrenom()+","+P.getNom()+","+P.getAge()+")");
  }
}
```

et nous obtenons les résultats suivants :

```
P=(Jean,Michelin,34)
P=(Jean,Michelin,56)
```

2.1.13 Les propriétés

Il existe une autre façon d'avoir accès aux attributs d'une classe c'est de créer des propriétés. Celles-ci nous permettent de manipuler des attributs privés comme s'ils étaient publics.

Considérons la classe *personne* suivante où les accesseurs et modifieurs précédents ont été remplacés par des **propriétés** en lecture et écriture :

```
using System;

public class personne{

  // attributs
  private String _prenom;
  private String _nom;
  private int _age;

  // constructeurs
  public personne(String P, String N, int age){
    this._prenom=P;
    this._nom=N;
    this._age=age;
  }

  public personne(personne P){
    this._prenom=P._prenom;
    this._nom=P._nom;
    this._age=P._age;
  }

  // identifie
  public void identifie(){
    Console.Out.WriteLine(_prenom+","+_nom+","+_age);
  }

  // propriétés
  public string prenom{
    get { return _prenom; }
    set { _prenom=value; }
  }//prenom

  public string nom{
    get { return _nom; }
    set { _nom=value; }
  }//nom

  public int age{
    get { return _age; }
    set {
      // age valide ?
      if(value>=0){
        _age=value;
      } else
        throw new Exception("âge ("+value+") invalide");
    }//if
  }//age
}//classe
```

Une propriété permet de lire (**get**) ou de fixer (**set**) la valeur d'un attribut. Dans notre exemple, nous avons préfixé les noms des attributs du signe _ afin que les propriétés portent le nom des attributs primitifs. En effet, une propriété ne peut porter le même nom que l'attribut qu'elle gère car alors il y a un conflit de noms dans la classe. Nous avons donc appelé nos attributs *_prenom*, *_nom*, *_age* et modifié les constructeurs et méthodes en conséquence. Nous avons ensuite créé trois propriétés *nom*, *prenom* et *age*. Une propriété est déclarée comme suit :

public type propriété{
  get {...}
  set {...}
}

où *type* doit être le type de l'attribut géré par la propriété. Elle peut avoir deux méthodes appelées **get** et **set**. La méthode **get** est habituellement chargée de rendre la valeur de l'attribut qu'elle gère (elle pourrait rendre autre chose, rien ne l'empêche). La méthode **set** reçoit un paramètre appelé **value** qu'elle affecte normalement à l'attribut qu'elle gère. Elle peut en profiter pour faire des vérifications sur la validité de la valeur reçue et éventuellement lancer un exception si la valeur se révèle invalide. C'est ce qui est fait ici pour l'âge.

Comment ces méthodes **get** et **set** sont-elles appelées ? Considérons le programme de test suivant :

```
using System;

public class test1{
  public static void Main(){
    personne P=new personne("Jean","Michelin",34);
    Console.Out.WriteLine("P=("+P.prenom+","+P.nom+","+P.age+")");
    P.age=56;
    Console.Out.WriteLine("P=("+P.prenom+","+P.nom+","+P.age+")");
    try{
      P.age=-4;
    } catch (Exception ex){
      Console.Error.WriteLine(ex.Message);
    }//try-catch
  }//Main
}//classe
```

Dans l'instruction

```
    Console.Out.WriteLine("P=("+P.prenom+","+P.nom+","+P.age+")");
```

on cherche à avoir les valeurs des propriétés *prenom*, *nom* et *age* de la personne P. C'est la méthode **get** de ces propriétés qui est alors appelée et qui rend la valeur de l'attribut qu'elles gèrent.

Dans l'instruction

```
    P.age=56;
```

on veut fixer la valeur de la propriété *age*. C'est alors la méthode set de cette propriété qui est alors appelée. Elle recevra 56 dans son paramètre **value**.

Une propriété **P** d'une classe **C** qui ne définirait que la méthode **get** est dite en lecture seule. Si c est un objet de classe C, l'opération *c.P=valeur* sera alors refusée par le compilateur.

L'exécution du programme de test précédent donne les résultats suivants :

```
P=(Jean,Michelin,34)
P=(Jean,Michelin,56)
âge (-4) invalide
```

Les propriétés nous permettent donc de manipuler des attributs privés comme s'ils étaient publics.

2.1.14 Les méthodes et attributs de classe

Supposons qu'on veuille compter le nombre d'objets personnes créées dans une application. On peut soi-même gérer un compteur mais on risque d'oublier les objets temporaires qui sont créés ici ou là. Il semblerait plus sûr d'inclure dans les constructeurs de la classe personne, une instruction incrémentant un compteur. Le problème est de passer une référence de ce compteur afin que le constructeur puisse l'incrémenter : il faut leur passer un nouveau paramètre. On peut aussi inclure le compteur dans la définition de la classe. Comme c'est un attribut de la classe elle-même et non d'un objet particulier de cette classe, on le déclare différemment avec le mot clé *static* :

```
  private static long _nbPersonnes;   // nombre de personnes créées
```

Pour le référencer, on écrit *personne._nbPersonnes* pour montrer que c'est un attribut de la classe *personne* elle-même. Ici, nous avons créé un attribut privé auquel on n'aura pas accès directement en-dehors de la classe. On crée donc une propriété publique pour donner accès à l'attribut de classe *nbPersonnes*. Pour rendre la valeur de *nbPersonnes* la méthode *get* de cette propriété n'a pas besoin d'un objet *personne* particulier : en effet *_nbPersonnes* n'est pas l'attribut d'un objet particulier, il est l'attribut de toute une classe. Aussi a-t-on besoin d'une propriété déclarée elle-aussi *static* :

```
// propriété de classe
  public static long nbPersonnes{
    get{ return _nbPersonnes;}
  }//nbPersonnes
```

qui de l'extérieur sera appelée avec la syntaxe *personne.nbPersonnes*. Voici un exemple.

La classe *personne* devient la suivante :

```
using System;

public class personne{

  // attributs de classe
  private static long _nbPersonnes=0;

  // attributs d'instance
  private String _prenom;
  private String _nom;
  private int _age;

  // constructeurs
  public personne(String P, String N, int age){
    // une personne de plus
    _nbPersonnes++;
    this._prenom=P;
    this._nom=N;
    this._age=age;
  }

  public personne(personne P){
    // une personne de plus
    _nbPersonnes++;
    this._prenom=P._prenom;
    this._nom=P._nom;
    this._age=P._age;
  }

  // identifie
  public void identifie(){
    Console.Out.WriteLine(_prenom+","+_nom+","+_age);
  }

  // propriété de classe
  public static long nbPersonnes{
    get{ return _nbPersonnes;}
  }//nbPersonnes

  // propriétés d'instance
  public string prenom{
    get { return _prenom; }
    set { _prenom=value; }
  }//prenom

  public string nom{
    get { return _nom; }
    set { _nom=value; }
  }//nom

  public int age{
    get { return _age; }
    set {
      // age valide ?
      if(value>=0){
        _age=value;
      } else
        throw new Exception("âge ("+value+") invalide");
    }//if
  }//age
}//classe
```

Avec le programme suivant :

```
using System;

public class test1{
  public static void Main(){
    personne p1=new personne("Jean","Dupont",30);
    personne p2=new personne(p1);
    new personne(p1);
    Console.Out.WriteLine("Nombre de personnes créées : "+personne.nbPersonnes);
  }// main
}//test1
```

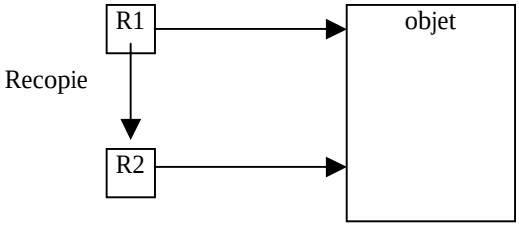
on obtient les résultats suivants :

```
    Nombre de personnes créées : 3
```

2.1.15 Passage d'un objet à une fonction

Nous avons déjà dit que par défaut C# passait les paramètres effectifs d'une fonction par valeur : les valeurs des paramètres effectifs sont recopiées dans les paramètres formels. Dans le cas d'un objet, il ne faut pas se laisser tromper par l'abus de langage qui est fait systématiquement en parlant d'objet au lieu de référence d'objet. Un objet n'est manipulé que via une référence (un pointeur) sur lui. Ce qui est donc transmis à une fonction, n'est pas l'objet lui-même mais une référence sur cet objet. C'est donc la valeur de la référence et non la valeur de l'objet lui-même qui est dupliquée dans le paramètre formel : il n'y a pas construction d'un nouvel objet.

Si une référence d'objet R1 est transmise à une fonction, elle sera recopiée dans le paramètre formel correspondant R2. Aussi les références R2 et R1 désignent-elles le même objet. Si la fonction modifie l'objet pointé par R2, elle modifie évidemment celui référencé par R1 puisque c'est le même.



C'est ce que montre l'exemple suivant :

```
using System;

public class test1{
  public static void Main(){
    // une personne p1
    personne p1=new personne("Jean","Dupont",30);
    // affichage p1
    Console.Out.Write("Paramètre effectif avant modification : ");
    p1.identifie();
    // modification p1
    modifie(p1);
    // affichage p1
    Console.Out.Write("Paramètre effectif après modification : ");
    p1.identifie();
  }// main

  private static void modifie(personne P){
    // affichage personne P
    Console.Out.Write("Paramètre formel avant modification : ");
    P.identifie();
    // modification P
    P.prenom="Sylvie";
    P.nom="Vartan";
    P.age=52;
    // affichage P
    Console.Out.Write("Paramètre formel après modification : ");
    P.identifie();
  }// modifie

}// class
```

La méthode *modifie* est déclarée *static* parce que c'est une méthode de classe : on n'a pas à la préfixer par un objet pour l'appeler.

Les résultats obtenus sont les suivants :

```
Construction personne(string, string, int)
Paramètre effectif avant modification : Jean,Dupont,30
Paramètre formel avant modification : Jean,Dupont,30
Paramètre formel après modification : Sylvie,Vartan,52
Paramètre effectif après modification : Sylvie,Vartan,52
```

On voit qu'il n'y a construction que d'un objet : celui de la personne *p1* de la fonction M*ain* et que l'objet a bien été modifié par la fonction *modifie*.

2.1.16 Un tableau de personnes

Un objet est une donnée comme une autre et à ce titre plusieurs objets peuvent être rassemblés dans un tableau :

```
import personne;

using System;

public class test1{
```

```
  public static void Main(){
    // un tableau de personnes
    personne[] amis=new personne[3];
    amis[0]=new personne("Jean","Dupont",30);
    amis[1]=new personne("Sylvie","Vartan",52);
    amis[2]=new personne("Neil","Armstrong",66);
    // affichage
    Console.Out.WriteLine("----------------");  int i;
    for(i=0;i<amis.Length;i++)
      amis[i].identifie();
  }
}
```

L'instruction *personne[] amis=new personne[3];* crée un tableau de 3 éléments de type *personne*. Ces 3 éléments sont initialisés ici avec la valeur *null*, c.a.d. qu'ils ne référencent aucun objet. De nouveau, par abus de langage, on parle de tableau d'objets alors que ce n'est qu'un tableau de références d'objets. La création du tableau d'objets, qui est un objet lui-même (présence de *new*) ne crée aucun objet du type de ses éléments : il faut le faire ensuite.

On obtient les résultats suivants :

```
Construction personne(string, string, int)
Construction personne(string, string, int)
Construction personne(string, string, int)
----------------
Jean,Dupont,30
Sylvie,Vartan,52
Neil,Armstrong,66
```

2.2 L'héritage par l'exemple

2.2.1 Généralités

Nous abordons ici la notion d'héritage. Le but de l'héritage est de "personnaliser" une classe existante pour qu'elle satisfasse à nos besoins. Supposons qu'on veuille créer une classe *enseignant* : un enseignant est une personne particulière. Il a des attributs qu'une autre personne n'aura pas : la matière qu'il enseigne par exemple. Mais il a aussi les attributs de toute personne : prénom, nom et âge. Un enseignant fait donc pleinement partie de la classe *personne* mais a des attributs supplémentaires. Plutôt que d'écrire une classe *enseignant* à partir de rien, on préfèrerait reprendre l'acquis de la classe *personne* qu'on adapterait au caractère particulier des enseignants. C'est le concept d'**héritage** qui nous permet cela.

Pour exprimer que la classe *enseignant* hérite des propriétés de la classe *personne*, on écrira :

```
  public class enseignant extends personne
```

personne est appelée la classe parent (ou mère) et *enseignant* la classe dérivée (ou fille). Un objet *enseignant* a toutes les qualités d'un objet *personne* : il a les mêmes attributs et les mêmes méthodes. Ces attributs et méthodes de la classe parent ne sont pas répétées dans la définition de la classe fille : on se contente d'indiquer les attributs et méthodes rajoutés par la classe fille :

Nous supposons que la classe *personne* est définie comme suit :

```
using System;

public class personne{

  // attributs de classe
  private static long _nbPersonnes=0;

  // attributs d'instance
  private String _prenom;
  private String _nom;
  private int _age;

  // constructeurs
  public personne(String P, String N, int age){
    // une personne de plus
    _nbPersonnes++;
    // construction
    this._prenom=P;
    this._nom=N;
    this._age=age;
    // suivi
    Console.Out.WriteLine("Construction personne(string, string, int)");
  }

  public personne(personne P){
    // une personne de plus
```

```
    _nbPersonnes++;
    // construction
    this._prenom=P._prenom;
    this._nom=P._nom;
    this._age=P._age;
    // suivi
    Console.Out.WriteLine("Construction personne(string, string, int)");
  }

  // propriété de classe
  public static long nbPersonnes{
    get{ return _nbPersonnes;}
  }//nbPersonnes

  // propriétés d'instance
  public string prenom{
    get { return _prenom; }
    set { _prenom=value; }
  }//prenom

  public string nom{
    get { return _nom; }
    set { _nom=value; }
  }//nom

  public int age{
    get { return _age; }
    set {
      // age valide ?
      if(value>=0){
        _age=value;
      } else
        throw new Exception("âge ("+value+") invalide");
    }//if
  }//age

  public string identite{
    get { return "personne("+_prenom+","+_nom+","+age+")";}
  }

}//classe
```

La méthode *identifie* a été remplacée par la propriété *identité* en lecture seule et qui identifie la personne. Nous créons une classe *enseignant* héritant de la classe *personne* :

```
using System;

public class enseignant : personne {
// attributs
  private int _section;

// constructeur
  public enseignant(string P, string N, int age,int section) : base(P,N,age) {
    this._section=section;
    // suivi
    Console.Out.WriteLine("Construction enseignant(string,string,int,int)");
  }//constructeur

  // propriété section
  public int section{
    get { return _section; }
    set { _section=value; }
  }// section
}//classe
```

La classe *enseignant* rajoute aux méthodes et attributs de la classe *personne* :

- un attribut *section* qui est le n° de section auquel appartient l'enseignant dans le corps des enseignants (une section par discipline en gros)
- un nouveau constructeur permettant d'initialiser tous les attributs d'un enseignant

La déclaration

```
  public class enseignant : personne {
```

indique que la classe *enseignant* dérive de la classe *personne*.

2.2.2 Construction d'un objet enseignant

Le constructeur de la classe *enseignant* est le suivant :

```
// constructeur
  public enseignant(String P, String N, int age,int section) : base(P,N,age) {
```

```
    this._section=section;
  }//constructeur
```

La déclaration

```
  public enseignant(String P, String N, int age,int section) : base(P,N,age) {
```

déclare que le constructeur reçoit quatre paramètres *P*, *N*, *age*, *section* et en passe trois *(P,N,age)* à sa classe de base, ici la classe *personne*. On sait que cette classe a un constructeur *personne(string, string, int)* qui va permettre de construire une personne avec les paramètres passsés *(P,N,age)*. Une fois la construction de la classe de base terminée, la construction de l'objet *enseignant* se poursuit par l'exécution du corps du constructeur :

```
    this.s_ection=section;
```

En résumé, le constructeur d'une classe dérivée :

- passe à sa classe de base les paramètres dont elle a besoin pour se construire
- utilise les autres paramètres pour initialiser les attributs qui lui sont propres

On aurait pu préférer écrire :

```
// constructeur
  public enseignant(String P, String N, int age,int section){
    this._prenom=P;
    this._nom=N
    this._age=age
    this._section=section;
  }
```

C'est impossible. La classe *personne* a déclaré privés (*private*) ses trois champs *_prenom*, *_nom* et *_age*. Seuls des objets de la même classe ont un accès direct à ces champs. Tous les autres objets, y compris des objets fils comme ici, doivent passer par des méthodes publiques pour y avoir accès. Cela aurait été différent si la classe *personne* avait déclaré protégés (*protected*) les trois champs : elle autorisait alors des classes dérivées à avoir un accès direct aux trois champs. Dans notre exemple, utiliser le constructeur de la classe parent était donc la bonne solution et c'est la méthode habituelle : lors de la construction d'un objet fils, on appelle d'abord le constructeur de l'objet parent puis on complète les initialisations propres cette fois à l'objet fils (*section* dans notre exemple).

Compilons les classes *personne* et *enseignant* dans des assemblages :

```
E:\data\serge\MSNET\c#\objetsPoly\12>csc /t:library personne.cs
E:\data\serge\MSNET\c#\objetsPoly\12>csc /r:personne.dll /t:library enseignant.cs
E:\data\serge\MSNET\c#\objetsPoly\12>dir
26/04/2002  16:15                1 341 personne.cs
26/04/2002  16:30                4 096 personne.dll
26/04/2002  16:32                  345 enseignant.cs
26/04/2002  16:32                3 072 enseignant.dll
```

On remarquera que pour compiler la classe fille *enseignant*, il a fallu référencer le fichier *personne.dll* qui contient la classe *personne*.

Tentons un premier programme de test :

```
using System;

public class test1{
  public static void Main(){
    Console.Out.WriteLine(new enseignant("Jean","Dupont",30,27).identite);
  }
}
```

Ce programme ce contente de créer un objet *enseignant* (new) et de l'identifier. La classe *enseignant* n'a pas de méthode *identité* mais sa classe parent en a une qui de plus est publique : elle devient par héritage une méthode publique de la classe *enseignant*. Les résultats obtenus sont les suivants :

```
Construction personne(string, string, int)
Construction enseignant(string,string,int,int)
personne(Jean,Dupont,30)
```

On voit que :

- un objet *personne* a été construit avant l'objet *enseignant*
- l'identité obtenue est celle de l'objet *personne*

2.2.3 Surcharge d'une méthode ou d'une propriété

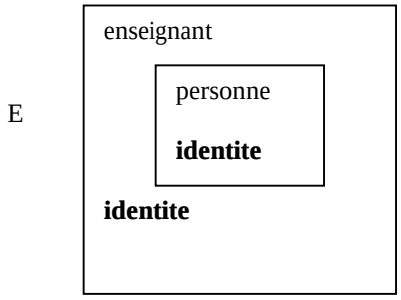
Dans l'exemple précédent, nous avons eu l'identité de la partie *personne* de l'enseignant mais il manque certaines informations propres à la classe *enseignant* (la section). On est donc amené à écrire une propriété permettant d'identifier l'enseignant :

```
using System;

public class enseignant : personne {
  // attributs
  private int _section;

  // constructeur
  public enseignant(string P, string N, int age,int section) : base(P,N,age) {
    this._section=section;
    // suivi
    Console.Out.WriteLine("Construction enseignant(string,string,int,int)");
  }//constructeur

  // propriété section
  public int section{
    get { return _section; }
    set { _section=value; }
  }// section

  // surcharge propriété identité
  public new string identite{
    get { return "enseignant("+base.identite+","+_section+")"; }
  }//propriété identité
}//classe
```

La méthode *identite* de la classe *enseignant* s'appuie sur la méthode *identite* de sa classe mère (*base.identite*) pour afficher sa partie "*personne*" puis complète avec le champ *_section* qui est propre à la classe *enseignant*. Notons la déclaration de la propriété *identite* :

```
  public new string identite{
```

Soit un objet *enseignant* E. Cet objet contient en son sein un objet *personne* :



La propriété identité est définie à la fois dans la classe *enseignant* et sa classe mère *personne*. Dans la classe fille *enseignant*, la propriété *identite* doit être précédée du mot clé **new** pour indiquer qu'on redéfinit une nouvelle propriété *identite* pour la classe *enseignant*.

```
  public new string identite{
```

La classe *enseignant* dispose maintenant de deux propriétés *identite* :

- celle héritée de la classe parent *personne*
- la sienne propre

Si E est un ojet *enseignant*, *E.identite* désigne la méthode *identite* de la classe *enseignant*. On dit que la propriété *identite* de la classe mère est "surchargée" par la propriété *identite* de la classe fille. De façon générale, si *O* est un objet et *M* une méthode, pour exécuter la méthode *O.M*, le système cherche une méthode *M* dans l'ordre suivant :

- dans la classe de l'objet O
- dans sa classe mère s'il en a une
- dans la classe mère de sa classe mère si elle existe
- etc…

L'héritage permet donc de surcharger dans la classe fille des méthodes/propriétés de même nom dans la classe mère. C'est ce qui permet d'adapter la classe fille à ses propres besoins. Associée au polymorphisme que nous allons voir un peu plus loin, la surcharge de méthodes/propriétés est le principal intérêt de l'héritage.

Considérons le même exemple que précédemment :

```
using System;

public class test1{
  public static void Main(){
    Console.Out.WriteLine(new enseignant("Jean","Dupont",30,27).identite);
  }
}
```

Les résultats obtenus sont cette fois les suivants :

```
Construction personne(string, string, int)
Construction enseignant(string,string,int,int)
enseignant(personne(Jean,Dupont,30),27)
```

2.2.4 Le polymorphisme

Considérons une lignée de classes : $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$

où $C_i \rightarrow C_j$ indique que la classe C_j est dérivée de la classe C_i. Cela entraîne que la classe C_j a toutes les caractéristiques de la classe C_i plus d'autres. Soient des objets O_i de type C_i. Il est légal d'écrire :

$O_i=O_j$ avec j>i

En effet, par héritage, la classe C_j a toutes les caractéristiques de la classe C_i plus d'autres. Donc un objet O_j de type C_j contient en lui un objet de type C_i. L'opération

$O_i=O_j$

fait que O_i est une référence à l'objet de type C_i contenu dans l'objet O_j.

Le fait qu'une variable *Oi* de classe *Ci* puisse en fait référencer non seulement un objet de la classe *Ci* mais en fait tout objet dérivé de la classe *Ci* est appelé **polymorphisme** : la faculté pour une variable de référencer différents types d'objets.

Prenons un exemple et considérons la fonction suivante indépendante de toute classe (*static*):

```
public static void affiche(personne p){
  ….
}
```

On pourra aussi bien écrire

```
personne p;
...
affiche(p);
```

que

```
enseignant e;
...
affiche(e);
```

Dans ce dernier cas, le paramètre formel de type *personne* de la fonction *affiche* va recevoir une valeur de type *enseignant*. Comme le type *enseignant* dérive du type *personne*, c'est légal.

2.2.5 Surcharge et polymorphisme

Complétons notre fonction *affiche* :

```
public static void affiche(personne p){
  Console.Out.WriteLine(p.identite);
}
```

La méthode *p.identite* rend une chaîne de caractères identifiant l'objet *personne*. Que se passe-t-il dans le cas de notre exemple précédent dans le cas d'un objet *enseignant* :

```
enseignant e=new enseignant(...);
affiche(e);
```

Regardons l'exemple suivant :

```
using System;

public class test1{
  public static void Main(){
    // un enseignant
    enseignant e=new enseignant("Lucile","Dumas",56,61);
    affiche(e);
    // une personne
    personne p=new personne("Jean","Dupont",30);
    affiche(p);
```

```
  }
  // affiche
  public static void affiche(personne p){
    // affiche identité de p
    Console.Out.WriteLine(p.identite);
  }//affiche
}
```

Les résultats obtenus sont les suivants :

```
Construction personne(string, string, int)
Construction enseignant(string,string,int,int)
personne(Lucile,Dumas,56)
Construction personne(string, string, int)
personne(Jean,Dupont,30)
```

L'exécution montre que l'instruction *p.identite* a exécuté à chaque fois la propriété *identite* d'une *personne*, la personne contenue dans l'*enseignant* e, puis la *personne* p elle-même. Elle ne s'est pas adaptée à l'objet réellement passé en paramètre à *affiche*. On aurait préféré avoir l'identité complète de l'*enseignant* e. Il aurait fallu pour cela que la notation *p.identite* référence la propriété *identite* de l'objet réellement pointé par *p* plutôt que la propriété *identite* de partie "*personne*" de l'objet réellement pointé par *p*.

Il est possible d'obtenir ce résultat en déclarant *identite* comme une propriété **virtuelle** (**virtual**) dans la classe de base *personne* :

```
  public virtual string identite{
    get { return "personne("+_prenom+","+_nom+","+age+")";}
  }
```

Le mot clé **virtual** fait de *identite* une propriété virtuelle. Ce mot clé peut s'appliquer également aux méthodes. Les classes filles qui redéfinissent une propriété ou méthode virtuelle doivent utiliser alors le mot clé **override** au lieu de **new** pour qualifier leur propriété/méthode redéfinie. Ainsi dans la classe *enseignant*, la propriété *identite* est définie comme suit :

```
// surcharge propriété identité
  public override string identite{
    get { return "enseignant("+base.identite+","+_section+")"; }
  }//propriété identité
```

Le programme de test :

```
using System;

public class test1{
  public static void Main(){
    // un enseignant
    enseignant e=new enseignant("Lucile","Dumas",56,61);
    affiche(e);
    // une personne
    personne p=new personne("Jean","Dupont",30);
    affiche(p);
  }
  // affiche
  public static void affiche(personne p){
    // affiche identité de p
    Console.Out.WriteLine(p.identite);
  }//affiche
}
```

produit alors les résultats suivants :

```
Construction personne(string, string, int)
Construction enseignant(string,string,int,int)
enseignant(personne(Lucile,Dumas,56),61)
Construction personne(string, string, int)
personne(Jean,Dupont,30)
```

Cette fois-ci, on a bien eu l'identité complète de l'enseignant. Surchargeons maintenant une méthode plutôt qu'une propriété. La classe *object* est la classe "mère" de toutes les classes C#. Ainsi lorsqu'on écrit :

```
  public class personne
```

on écrit implicitement :

```
  public class personne : object
```

La classe *object* définit une méthode virtuelle **ToString** :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public class object
```

```
{
    // Constructors
    public Object();

    // Methods
    public virtual bool Equals(object obj);
    public static bool Equals(object objA, object objB);
    public virtual int GetHashCode();
    public Type GetType();
    public static bool ReferenceEquals(object objA, object objB);
    public virtual string ToString();
} // end of System.Object
```

La méthode *ToString* rend le nom de la classe à laquelle appartient l'objet comme le montre l'exemple suivant :

```
using System;

public class test1{
  public static void Main(){
    // un enseignant
    Console.Out.WriteLine(new enseignant("Lucile","Dumas",56,61).ToString());
    // une personne
    Console.Out.WriteLine(new personne("Jean","Dupont",30).ToString());
  }
}
```

Les résultats produits sont les suivants :

```
Construction personne(string, string, int)
Construction enseignant(string,string,int,int)
enseignant
Construction personne(string, string, int)
personne
```

On remarquera que bien que nous n'ayons pas redéfini la méthode *ToString* dans les classes *personne* et *enseignant*, on peut cependant constater que la méthode *ToString* de la classe *object* est quand même capable d'afficher le nom réel de la classe de l'objet.

Redéfinissons la méthode *ToString* dans les classes *personne* et *enseignant* :

```
  // ToString
  public override string ToString(){
    // on rend la propriété identite
    return identite;
  }//ToString
```

La définition est la même dans les deux classes. Considérons le programme de test suivant :

```
using System;

public class test1{
  public static void Main(){
    // un enseignant
    enseignant e=new enseignant("Lucile","Dumas",56,61);
    affiche(e);
    // une personne
    personne p=new personne("Jean","Dupont",30);
    affiche(p);
  }
  // affiche
  public static void affiche(personne p){
    // affiche identité de p
    Console.Out.WriteLine(""+p);
  }//affiche
}
```

Attardons-nous sur la méthode *affiche* qui admet pour paramètre une personne p. Que signifie l'expression *""+p* ? Le compilateur va ici chercher à transformet l'objet *p* en *string* et cherche toujours pour cela l'existence d'une méthode appelée *ToSTring*. Donc *""+p* devient *""+p.Tostring()*. La méthode *ToString* étant virtuelle, le compilateur va exécuter la méthode *ToString* de l'objet réellement pointé par *p*. C'est ce que montrent les résultats d'exécution :

```
Construction personne(string, string, int)
Construction enseignant(string,string,int,int)
enseignant(personne(Lucile,Dumas,56),61)
Construction personne(string, string, int)
personne(Jean,Dupont,30)
```

2.3 Redéfir la signification d'un opérateur pour une classe

2.3.1 Introduction

Considérons l'instruction

op1 + op2

où *op1* et *op2* sont deux opérandes. Il est possible de redéfinir le sens de l'opérateur lorsque l'opérande *op1* est un objet de classe *C1*. Il suffit pour cela de définir une méthode statique dans la classe C1 dont la signature est la suivante :

```
public static [type] operator +(C1 opérande1, type2 opérande2);
```

Lorsque le compilateur rencontre l'instruction

op1 + op2

il la traduit alors par *C1.operator+(op1,op2)*. Le type rendu par la méthode operator est important. En effet, considérons l'opération *op1+op2+op3*. Elle est traduite par le compilateur par *(op1+op2)+op3*. Soit *res12* le résultat de *op1+op2*. L'opération qui est faite ensuite est *res12+op3*. Si *res12* est de type *C1*, elle sera traduite elle aussi par *C1.operator+(res12,op3)*. Cela permet d'enchaîner les opérations.

On peut redéfinir également les opérateurs unaires n'ayant qu'un seul opérande. Ainsi si *op1* est un objet de type *C1*, l'opération *op1++* peut être redéfinie par une méthode statique de la classe *C1* :

```
public static [type] operator ++(C1 opérande1);
```

Ce qui a été dit ici est vrai pour la plupart des opérateurs avec cependant quelques exceptions :

- les opérateurs == et != doivent être redéfinis en même temps
- les opérateurs && , | | , [], (), +=, -=, ... ne peuvent être redéfinis

2.3.2 Un exemple

On crée une classe *listeDePersonnes* dérivée de la classe *ArrayList*. Cette classe implémente un tableau dynamique et est présentée dans le chapitre qui suit. De cette classe, nous n'utilisons que les éléments suivants :

- la méthode T.*Add(Object o)* permettant d'ajouter au tableau T un objet o. Ici l'objet o sera un objet *personne*.
- la propriété T.*Count* qui donne le nombre d'éléments du tableau T
- la notation *T[i]* qui donne l'élément i du tableau T

La classe *listeDePersonnes* va hériter de tous les attributs, méthodes et propriétés de la classe *ArrayList*.

Nous redéfinissons la méthode *ToString* afin d'afficher une liste de personnes sous la forme *(personne1, personne2, ..)* où *personnei* est lui-même le résultat de la méthode *ToString* de la classe *personne*.

Enfin nous redéfinissons l'opérateur + afin de pouvoir écrire l'opération *l+p* où l est un objet *listeDePersonnes* et p un objet *personne*. L'opération *l+p* ajoute la personne *p* à la liste de personnes *l*. Cette opération rend une valeur de type *listeDePersonnes*, ce qui permet d'enchaîner les opérateurs + comme il a été expliqué plus haut. Ainsi l'opération *l+p1+p2* est interprétée (priorité des opérateurs) comme *(l+p1)+p2*. L'opération *l+p1* rend une nouvelle liste liste de personnes *l1*. L'opération *(l+p1)+p2* devient alors *l1+p2* qui ajoute la personne *p2* à la liste de personnes *l1*.

Le code est le suivant ;

```
using System;
using System.Collections;

// classe personne
public class listeDePersonnes : ArrayList{

  // redéfinition opérateur +
  // pour ajouter une personne à la liste
  public static listeDePersonnes operator +(listeDePersonnes l, personne p){
    l.Add(p);
    return l;
  }// operator +

  // toString
  public override string ToString(){
    // rend (él1, él2, ..., éln)
    string liste="(";
    int i;
    // on parcourt le tableau dynamique
    for (i=0;i<Count-1;i++){
      liste+="["+base[i]+"]"+",";
```

```
    }//for
    // dernier élémenr
    if(Count!=0) liste+="["+base[i]+"]";
    liste+=")";
    return liste;
  }//ToString

  public static void Main(){
    // une liste de personnes
    listeDePersonnes l=new listeDePersonnes();
    // ajout de personnes
    l=l+new personne("jean",10)+new personne("pauline",12);
    // affichage
    Console.Out.WriteLine("l="+l);
    l=l+new personne ("tintin",27);
    Console.Out.WriteLine("l="+l);
  }//Main
}//class
```

Les résultats :

```
E:\data\serge\MSNET\c#\objets\8>C:\WINNT\Microsoft.NET\Framework\v1.0.2914\csc.exe /r:personne.dll
lstpersonnes.cs

E:\data\serge\MSNET\c#\objets\8>lstpersonnes
l=([jean,10],[pauline,12])
l=([jean,10],[pauline,12],[tintin,27])
```

2.4 Définir un indexeur pour une classe

Nous continuons ici à utiliser la classe *listeDePersonnes*. Si l est un objet *listeDePersonnes*, nous souhaitons pouvoir utiliser la notation *l[i]* pour désigner la personne n° i de la liste l aussi bien en lecture (*personne p=l[i]*) qu'en écriture (*l[i]=new personne(...)*).

Pour pouvoir écrire *l[i]* où *l[i]* désigne un objet *personne*, il nous faut définir une méthode

```
  public new personne this[int i]{
    get { ... }
    set { ... }
  }
```

On appelle cette méthode, un indexeur car elle donne un sens à l'expression *obj[i]* qui rappelle la notation des tableaux alors que *obj* n'est pas un tableau mais un objet. La méthode *get* de l'objet *obj* est appelée lorsqu'on écrit *variable=obj[i]* et la méthode *set* lorsqu'on écrit *obj[i]=valeur*.

La classe *listeDePersonnes* dérive de la classe *ArrayList* qui a elle-même un indexeur :

```
    public object this[ int index ] { virtual get; virtual set; }
```

Pour indiquer que la méthode `public personne this[int i]` de la classe *listeDePersonnes* ne redéfinit pas (*override*) la méthode `public object this[ int index ]` de la classe *ArrayList* on est obligé d'ajouter le mot clé *new* à la déclaration de l'indexeur de *listeDePersonnes*. On écrira donc :

```
  public new personne this[int i]{
    get { ... }
    set { ... }
  }
```

Complétons cette méthode. La méthode *get* est appelée lorsqu'on écrit *variable=l[i]* par exemple où l est une *listeDePersonnes*. On doit alors retourner la personne n° i de la liste l. Ceci se fait en retournant l'objet n° i de la classe *ArrayList* sous-jacente à la classe *listeDePersonnes* avec la notation *base[i]*. L'objet retourné étant de type *Object*, un transtypage vers la classe *personne* est nécessaire.

La méthode *set* est appelée lorsqu'on écrit *l[i]=p* où *p* est une *personne*. Il s'agit alors d'affecter la personne p à l'élément i de la liste l. Ici, la personne *p* représentée par le mot clé *value* est affectée à l'élément *i* de la classe de base *ArrayList*.

L'indexeur de la classe *listeDePersonnes* sera donc le suivant :

```
  public new personne this[int i]{
    get { return (personne) base[i]; }
    set { base[i]=value; }
  }//indexeur
```

Maintenant, on veut pouvoir écrire également *personne p=l["nom"]*, c.a.d indexer la liste l non plus par un n° d'élément mais par un nom de personne. Pour cela on définit un nouvel indexeur :

```
  // un indexeur
  public int this[string N]{
    get {
      // on recherche la personne de nom N
      int i;
      for (i=0;i<Count;i++){
        if (((personne) base[i]).nom==N) return i;
      }//for
      return -1;
    }//get
  }//indexeur[nom]
```

La première ligne

```
public int this[string N]
```

indique qu'on indexe la classe *listeDePersonnes* par une chaîne de caractères N et que le résultat de *l[N]* est un entier. Cet entier sera la position dans la liste de la personne portant le nom N ou -1 si cette personne n'est pas dans la liste. On ne définit que la propriété *get* interdisant ainsi l'écriture *l["nom"]=valeur* qui aurait nécessité la définition de la propriété *set*. Le mot clé *new* n'est pas nécessaire dans la déclaration de l'indexeur car la classe de base *ArrayList* ne définit pas d'indexeur *this[string]*.

Dans le corps du *get*, on parcourt la liste des personnes à la recherche du nom N passé en paramètre. Si on le trouve en position i, on renvoie i sinon on renvoie -1.

Le code complet de la classe est le suivant :

```
using System;
using System.Collections;

// classe personne
public class listeDePersonnes : ArrayList{

  // redéfinition opérateur +
  // pour ajouter une personne à la liste
  public static listeDePersonnes operator +(listeDePersonnes l, personne p){
    l.Add(p);
    return l;
  }// operator +

  // un indexeur
  public new personne this[int i]{
    get { return (personne) base[i]; }
    set { base[i]=value; }
  }//indexeur

  // un indexeur
  public int this[string N]{
    get {
      // on recherche la personne de nom N
      int i;
      for (i=0;i<Count;i++){
        if (((personne) base[i]).nom==N) return i;
      }//for
      return -1;
    }//get
  }//indexeur[nom]

  // toString
  public override string ToString(){
    // rend (él1, él2, ..., éln)
    string liste="(";
    int i;
    // on parcourt le tableau dynamique
    for (i=0;i<Count-1;i++){
      liste+="["+base[i]+"]"+",";
    }//for
    // dernier élémenr
    if(Count!=0) liste+="["+base[i]+"]";
    liste+=")";
    return liste;
  }//ToString

  public static void Main(){
    // une liste de personnes
    listeDePersonnes l=new listeDePersonnes();
    // ajout de personnes
    l=l+new personne("jean",10)+new personne("pauline",12);
    // affichage
    Console.Out.WriteLine("l="+l);
    l=l+new personne ("tintin",27);
    Console.Out.WriteLine("l="+l);
    // changement élément 1
    l[1]=new personne("milou",5);
    // affichage élément 1
    Console.Out.WriteLine("l[1]="+l[1]);
    // affichage liste l
```

```
    Console.Out.WriteLine("l="+l);
    // recherche de personnes
    string[] noms= new String[] {"milou","haddock"};
    for (int i=0;i<noms.Length;i++){
      int inom=l[noms[i]];
      if(inom!=-1) Console.Out.WriteLine("personne("+noms[i]+")="+l[inom]);
      else Console.Out.WriteLine("personne("+noms[i]+") n'existe pas");
    }//for
  }//Main
}//class
```

L'exécution de la fonction *Main* donne les résultats suivants :

```
E:\data\serge\MSNET\c#\objets\9>C:\WINNT\Microsoft.NET\Framework\v1.0.2914\csc.exe /r:personne.dll
lstpersonnes.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.


E:\data\serge\MSNET\c#\objets\9>lstpersonnes
l=([jean,10],[pauline,12])
l=([jean,10],[pauline,12],[tintin,27])
l[1]=milou,5
l=([jean,10],[milou,5],[tintin,27])
personne(milou)=milou,5
personne(haddock) n'existe pas
```

2.5 Les structures

La structure C# est directement issue de la structure du langage C et est très proche de la classe. Une structure est définie comme suit :

```
struct nomStructure{
// attributs
  ...
// propriétés
...
// constructeurs
...
// méthodes
...
}
```

Il y a malgré une similitude de déclaration des différences importantes entre **classe** et **structure**. La notion d'héritage n'existe par exemple pas avec les structures. Si on écrit une classe qui ne doit pas être dérivée quelles sont les différences entre structure et classe qui vont nous aider à choisir entre les deux ? Aidons-nous de l'exemple suivant pour le découvrir :

```
  // structure personne
  struct personne{
    public string nom;
    public int age;
  }

  // classe PERSONNE
  class PERSONNE{
    public string nom;
    public int age;
  }

  // une classe de test
  public class test
  {
    static void Main() {
      // une personne p1
      personne p1;
      p1.nom="paul";
      p1.age=10;
      Console.Out.WriteLine("p1=personne("+p1.nom+","+p1.age+")");
      // une personne p2
      personne p2=p1;
      Console.Out.WriteLine("p2=personne("+p2.nom+","+p2.age+")");
      // p2 est modifié
      p2.nom="nicole";
      p2.age=30;
      // vérification p1 et p2
      Console.Out.WriteLine("p1=personne("+p1.nom+","+p1.age+")");
      Console.Out.WriteLine("p2=personne("+p2.nom+","+p2.age+")");

      // une PERSONNE P1
      PERSONNE P1=new PERSONNE();
      P1.nom="paul";
      P1.age=10;
```

```
      Console.Out.WriteLine("P1=PERSONNE("+P1.nom+","+P1.age+")");
      // une PERSONNE P2
      PERSONNE P2=P1;
      Console.Out.WriteLine("P2=PERSONNE("+P2.nom+","+P2.age+")");
      // P2 est modifié
      P2.nom="nicole";
      P2.age=30;
      // vérification P1 et P2
      Console.Out.WriteLine("P1=PERSONNE("+P1.nom+","+P1.age+")");
      Console.Out.WriteLine("P2=PERSONNE("+P2.nom+","+P2.age+")");
    }//Main
  }//classe
```

Si on exécute ce programme, on obtient les résultats suivants :

```
p1=personne(paul,10)
p2=personne(paul,10)
p1=personne(paul,10)
p2=personne(nicole,30)
P1=PERSONNE(paul,10)
P2=PERSONNE(paul,10)
P1=PERSONNE(nicole,30)
P2=PERSONNE(nicole,30)
```

Là où dans les pages précédentes de ce chapitre on utilisait une classe *personne*, nous utilisons maintenant une structure *personne* :

```
struct personne{
    public string nom;
    public int age;
  }
```

La déclaration `personne p1;` crée une structure (nom,age) et la valeur de p1 est cette structure elle-même. Rappelons que dans le cas d'une classe l'opération équivalente était :

```
personne p1 = new personne(...);
```

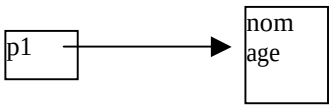
qui créait un objet personne (grosso modo l'équivalent de notre structure) et *p1* était alors l'adresse (la référence) de cet objet.

Résumons

- dans le cas de la structure, la **valeur** de p1 est la structure elle-même
- dans le cas de la classe, la **valeur** de p1 est l'**adresse** de l'objet créé

<u>Structure p1</u>

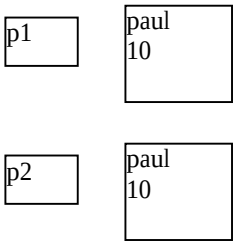
<u>Objet p1</u>



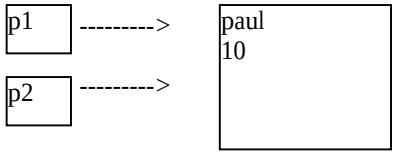
Lorsque dans le programme on écrit

```
personne p2=p1;
```

une nouvelle structure (nom,age) est créée et initialisée avec la valeur de *p1* donc la structure elle-même.



La structure de p1 est donc **dupliquée** dans p2. C'est une recopie de valeur. Dans le cas des classes, la valeur de p1 est recopiée dans p2, mais comme cette valeur est en fait l'adresse de l'objet, celui-ci n'est pas dupliqué. Il a simplement deux références sur lui :



Dans le cas de la structure, si on modifie la valeur de *p2* on ne modifie pas la valeur de *p1*, ce que montre le programme. Dans le cas de l'objet, si on modifie l'objet pointé par *p2*, celui pointé par *p1* est modifié puisque c'est le même. C'est ce que montrent également les résultats du programme.

On retiendra donc de ces explications que :

- la valeur d'une variable de type structure est la structure elle-même
- la valeur d'une variable de type objet est l'adresse de l'objet pointé

Une fois cette différence fondamentale comprise, la structure se montre très proche de la classe comme le montre le nouvel exemple suivant :

```
using System;

  // structure personne
struct personne{
  // attributs
  private string _nom;
  private int _age;
  // propriétés
  public string nom{
    get {return _nom;}
    set {_nom=value;}
  }//nom

  public int age{
    get {return _age;}
    set {_age=value;}
  }//nom
  // Constructeur

  public personne(string NOM, int AGE){
    _nom=NOM;
    _age=AGE;
  }//constructeur

  // TOSTRING
  public override string ToString(){
    return "personne("+nom+","+age+")";
  }//ToString
}

  // une classe de test
public class test {
  static void Main() {
    // une personne p1
    personne p1=new personne("paul",10);
    Console.Out.WriteLine("p1="+p1);
    // une personne p2
    personne p2=p1;
    Console.Out.WriteLine("p2="+p2);
    // p2 est modifié
    p2.nom="nicole";
    p2.age=30;
    // vérification p1 et p2
    Console.Out.WriteLine("p1="+p1);
    Console.Out.WriteLine("p2="+p2);
  }//Main
}//classe
```

On obtient les résultats d'exécution suivants :

```
p1=personne(paul,10)
p2=personne(paul,10)
p1=personne(paul,10)
p2=personne(nicole,30)
```

La seule notable différence ici entre structure et classe, c'est qu'avec une classe les objets p1 et p2 auraient eu la même valeur à la fin du programme, celle de p2.

2.6 Les interfaces

Une interface est un ensemble de prototypes de méthodes ou de propriétés qui forme un contrat. Une classe qui décide d'implémenter une interface s'engage à fournir une implémentation de toutes les méthodes définies dans l'interface. C'est le compilateur qui vérifie cette implémentation.

Voici par exemple la définition de l'interface *System.Collections.Ienumerator* :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public interface System.Collections.IEnumerator
{

    // Properties
    object Current { get; }

    // Methods
    bool MoveNext();
    void Reset();
} // end of System.Collections.IEnumerator
```

Toute classe implémentant cette interface sera déclarée comme

public class C : IEnumerator{
  ...
  object Current{ get {...}}
  bool MoveNext{...}
  void Reset(){...}
}

La propriété *Current* et les méthodes *MoveNext* et *Reset* devront être définies dans la classe C.

Considérons le code suivant :

```
using System;

public struct élève{
  public string nom;
  public double note;
  // constructeur
  public élève(string NOM, double NOTE){
    nom=NOM;
    note=NOTE;
  }//constructeur
}//élève

public class notes{
  // attribut
  protected string matière;
  protected élève[] élèves;

  // constructeur
  public notes (string MATIERE, élève[] ELEVES){
    // mémorisation élèves & matière
    matière=MATIERE;
    élèves=ELEVES;
  }//notes

  // ToString
  public override string ToString(){
    string valeur="matière="+matière +", notes=(";
    int i;
    // on concatène toutes les notes
    for (i=0;i<élèves.Length-1;i++){
      valeur+="["+élèves[i].nom+","+élèves[i].note+"],";
    };
    //dernière note
    if(élèves.Length!=0){ valeur+="["+élèves[i].nom+","+élèves[i].note+"]";}
    valeur+=")";
    // fin
    return valeur;
  }//ToString
}//classe
```

La classe *notes* rassemble les notes d'une classe dans une matière :

```
public class notes{
  // attribut
  protected string matière;
  protected élève[] élèves;
```

Les attributs sont déclarés *protected* pour être accessibles d'une classe dérivée. Le type *élève* est une structure mémorisant le nom de l'élève et sa note dans la matière :

```
public struct élève{
  public string nom;
  public double note;
```

Nous décidons de dériver cette classe *notes* dans une classe *notesStats* qui aurait deux attributs supplémentaires, la moyenne et l'écart-type des notes :

```
public class notesStats : notes, Istats {
  // attribut
  private double _moyenne;
  private double _écartType;
```

La classe *notesStats* implémente l'interface *Istats* suivante :

```
public interface Istats{
  double moyenne();
  double écartType();
}//
```

Cela signifie que la classe *notesStats* doit avoir deux méthodes appelées *moyenne* et *écartType* avec la signature indiquée dans l'interface *Istats*. La classe *notesStats* est la suivante :

```
public class notesStats : notes, Istats {
  // attribut
  private double _moyenne;
  private double _écartType;

  // constructeur
  public notesStats (string MATIERE, élève[] ELEVES): base(MATIERE, ELEVES){
    // calcul moyenne des notes
    double somme=0;
    for (int i=0;i<élèves.Length;i++){
      somme+=élèves[i].note;
    }
    if(élèves.Length!=0) _moyenne=somme/élèves.Length;
      else _moyenne=-1;
    // écart-type
    double carrés=0;
    for (int i=0;i<élèves.Length;i++){
      carrés+=Math.Pow((élèves[i].note-_moyenne),2);
    }//for
    if(élèves.Length!=0) _écartType=Math.Sqrt(carrés/élèves.Length);
      else _écartType=-1;
  }//constructeur


  // ToString
  public override string ToString(){
    return base.ToString()+",moyenne="+_moyenne+",écart-type="+_écartType;
  }//ToString

  // méthodes de l'interface Istats
  public double moyenne(){
    // rend la moyenne des notes
    return _moyenne;
  }//moyenne
  public double écartType(){
    // rend l'écart-type
    return _écartType;
  }//écartType
}//classe
```

La moyenne *_moyenne* et l'écart-type *_ecartType* sont calculés dès la construction de l'objet. Aussi les méthodes *moyenne* et *écartType* n'ont-elles qu'à rendre la valeur des attributs *_moyenne* et *_ecartType*. Les deux méthodes rendent -1 si le tableau des élèves est vide.

La classe de test suivante :

```
// classe de test
public class test{
  public static void Main(){
    // qqs élèves & notes
    élève[] ELEVES=new élève[] { new élève("paul",14),new élève("nicole",16), new élève("jacques",18)};
    // qu'on enregistre dans un objet notes
    notes anglais=new notes("anglais",ELEVES);
    // et qu'on affiche
    Console.Out.WriteLine(""+anglais);
    // idem avec moyenne et écart-type
    anglais=new notesStats("anglais",ELEVES);
    Console.Out.WriteLine(""+anglais);
  }
```

donne les résultats :

```
matière=anglais, notes=([paul,14],[nicole,16],[jacques,18])
matière=anglais, notes=([paul,14],[nicole,16],[jacques,18]),moyenne=16,écart-type=1,63299316185545
```

La classe *notesStats* aurait très bien pu implémenter les méthodes *moyenne* et *écartType* pour elle-même sans indiquer qu'elle implémentait l'interface *Istats*. Quel est donc l'intérêt des interfaces ? C'est le suivant : une fonction peut admettre pour paramètre une donnée ayant le type d'une interface I. Tout objet d'une classe C implémentant l'interface I pourra alors être paramètre de cette fonction. Considérons l'exemple suivant :

```
using System;

// une interface Iexemple
public interface Iexemple{
  int ajouter(int i,int j);
  int soustraire(int i,int j);
}

public class classe1: Iexemple{
  public int ajouter(int a, int b){
    return a+b+10;
  }
  public int soustraire(int a, int b){
    return a-b+20;
  }
}//classe

public class classe2: Iexemple{
  public int ajouter(int a, int b){
    return a+b+100;
  }
  public int soustraire(int a, int b){
    return a-b+200;
  }
}//classe
```

L'interface *Iexemple* définit deux méthodes *ajouter* et *soustraire*. Les classes *classe1* et *classe2* implémentent cette interface. On remarquera que ces classes ne font rien d'autre ceci par souci de simplification de l'exemple. Maintenant considérons l'exemple suivant :

```
// classe de test
public class test{
  // une fonction statique
  private static void calculer(int i, int j, Iexemple inter){
    Console.Out.WriteLine(inter.ajouter(i,j));
    Console.Out.WriteLine(inter.soustraire(i,j));
  }//calculer

  // la fonction Main
  public static void Main(){
    // création de deux objets classe1 et classe2
    classe1 c1=new classe1();
    classe2 c2=new classe2();
    // appels de la fonction statique calculer
    calculer(4,3,c1);
    calculer(14,13,c2);
  }//Main
}//classe test
```

La fonction statique *calculer* admet pour paramètre un élément de type *Iexemple*. Elle pourra donc recevoir pour ce paramètre aussi bien un objet de type *classe1* que de type *classe2*. C'est ce qui est fait dans la fonction *Main* avec les résultats suivants :

```
17
21
127
201
```

On voit donc qu'on a là une propriété proche du polymorphisme vu pour les classes. Si donc un ensemble de classes **Ci** non liées entre-elles par héritage (donc on ne peut utiliser le polymorphisme de l'héritage) présentent un ensemble de méthodes de même signature, il peut être intéressant de regrouper ces méthodes dans une interface **I** dont hériteraient toutes les classes concernées. Des instances de ces classes **Ci** peuvent alors être utilisées comme paramètres de fonctions admettant un paramètre de type **I**, c.a.d. des fonctions n'utilisant que les méthodes des objets **Ci** définies dans l'interface **I** et non les attributs et méthodes particuliers des différentes classes **Ci**.

Notons enfin que l'héritage d'interfaces peut être multiple, c.a.d. qu'on peut écrire

public class classeDérivée:classeDeBase,i_1*,*i_2*,..,*i_n*{*
...

}

où les i_j sont des interfaces.

2.7 Les espaces de noms

Pour écrire une ligne à l'écran, nous utilisons l'instruction

```
Console.Out.WriteLine(...)
```

Si nous regardons la définition de la classe *Console*

```
Namespace: System
Assembly: Mscorlib (in Mscorlib.dll
```

on découvre qu'elle fait partie de l'espace de noms *System*. Cela signifie que la classe *Console* devrait être désignée par *System.Console* et on devrait en fait écrire :

```
System.Console.Out.WriteLine(...)
```

On évite cela en utilisant une clause *using* :

```
using System;
...
Console.Out.WriteLine(...)
```

On dit qu'on **importe** l'espace de noms *System* avec la clause *using*. Lorsque le compilateur va rencontrer le nom d'une classe (ici *Console*) il va chercher à la trouver dans les différents espaces de noms importés par les clauses *using*. Ici il trouvera la classe *Console* dans l'espace de noms *System*. Notons maintenant la seconde information attachée à la classe *Console* :

```
Assembly: Mscorlib (in Mscorlib.dll)
```

Cette ligne indique dans quelle "assemblage" se trouve la définition de la classe *Console*. Lorsqu'on compile en-dehors de Visual Studio.NET et qu'on doit donner les références des différentes *dll* contenant les classes que l'on doit utiliser cette information peut s'avérer utile. On rappelle que pour référencer les *dll* nécessaires à la compilation d'une classe, on écrit :

csc /r:fic1.dll /r:fic2.dll ... prog.cs

Lorsqu'on crée une classe, on peut la créer à l'intérieur d'un espace de noms. Le but de ces espaces de noms est d'éviter les conflits de noms entre classes lorsque celles-ci sont vendues par exemple. Considérons deux entreprises E1 et E2 distribuant des classes empaquetées respectivement dans les *dll*, *E1.dll* et *E2.dll*. Soit un client C qui achète ces deux ensembles de classes dans lesquelles les deux entreprises ont défini toutes deux une classe *personne*. Le client C compile un programme de la façon suivante :

csc /r:E1.dll /r:E2.dll prog.cs

Si le source *prog.cs* utilise la classe *personne*, le compilateur ne saura pas s'il doit prendre la classe *personne* de *E1.dll* ou celle de *E2.dll*. Il signalera une erreur. Si l'entreprise E1 prend soin de créer ses classes dans un espace de noms appelé E1 et l'entreprise E2 dans un espace de noms appelé E2, les deux classes *personne* s'appelleront alors *E1.personne* et *E2.personne*. Le client devra employer dans ses classes soit *E1.personne*, soit *E2.personne* mais pas *personne*. L'espace de noms permet de lever l'ambiguïté.

Pour créer une classe dans un espace de noms, on écrit :

```
namespace espaceDeNoms{
  // définition de la classe
}
```

Pour l'exemple, créons dans un espace de noms notre classe *personne* étudiée précédemment. Nous choisirons *istia.st* comme espace de noms. La classe *personne* devient :

```
// espaces de noms importés
using System;

// création de l'espace de nom istia.st
namespace istia.st {
  public class personne{
    // attributs
    private string prenom;
    private string nom;
    private int age;
```

```
    // méthode
    public void initialise(string P, string N, int age){
      this.prenom=P;
      this.nom=N;
      this.age=age;
    }

    // méthode
    public void identifie(){
      Console.Out.WriteLine(prenom+","+nom+","+age);
    }
  }// classe personne
}//espace de noms
```

Cette classe est compilée dans *personne.dll* :

```
csc /t:library personne.cs
```

Maintenant utilisons la classe *personne* dans une classe de test :

```
// espaces de noms importés
using istia.st;

public class test1{
  public static void Main(){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
    p1.identifie();
  }
}//classe test1
```

Pour éviter d'écrire

```
    istia.st.personne p1=new istia.st.personne();
```

nous avons importé l'espace de noms *istia.st* avec une clause *using* :

```
using istia.st;
```

Maintenant compilons le programme de test :

```
csc /r:personne.dll test1.cs
```

Cela produit un fichier *test1.exe* qui exécuté donne les résultats suivants :

```
Jean,Dupont,30
```

2.8 L'exemple IMPOTS

On reprend le calcul de l'impôt déjà étudié dans le chapitre précédent et on le traite en utilisant une classe. Rappelons le problème :

On se place dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

- on calcule le nombre de parts du salarié **nbParts=nbEnfants/2 +1** s'il n'est pas marié, **nbEnfants/2+2** s'il est marié, où *nbEnfants* est son nombre d'enfants.
- s'il a au moins trois enfants, il a une demie part de plus
- on calcule son revenu imposable **R=0.72*S** où S est son salaire annuel
- on calcule son coefficient familial **QF=R/nbParts**
- on calcule son impôt **I**. Considérons le tableau suivant :

12620.0	0	0
13190	0.05	631
15640	0.1	1290.5
24740	0.15	2072.5
31810	0.2	3309.5
39970	0.25	4900
48360	0.3	6898.5
55790	0.35	9316.5
92970	0.4	12106
127860	0.45	16754.5
151250	0.50	23147.5
172040	0.55	30710
195000	0.60	39312
0	0.65	49062

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=23000 on trouvera la ligne

24740 0.15 2072.5

L'impôt I est alors égal à **0.15*R - 2072.5*nbParts**. Si QF est tel que la relation *QF<=champ1* n'est jamais vérifiée, alors ce sont les coefficients de la dernière ligne qui sont utilisés. Ici :

0 0.65 49062

ce qui donne l'impôt I=0.65*R - 49062*nbParts.

La classe **impôt** sera définie comme suit :

```
// création d'une classe impôt

using System;

public class impôt{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private decimal[] limites, coeffR, coeffN;

  // constructeur
  public impôt(decimal[] LIMITES, decimal[] COEFFR, decimal[] COEFFN){
    // on vérifie que les 3 tablaeux ont la même taille
    bool OK=LIMITES.Length==COEFFR.Length && LIMITES.Length==COEFFN.Length;
    if (! OK) throw new Exception ("Les 3 tableaux fournis n'ont pas la même taille("+
                LIMITES.Length+","+COEFFR.Length+","+COEFFN.Length+")");
    // c'est bon
    this.limites=LIMITES;
    this.coeffR=COEFFR;
    this.coeffN=COEFFN;
  }//constructeur

  // calcul de l'impôt
  public long calculer(bool marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
    decimal nbParts;
    if (marié) nbParts=(decimal)nbEnfants/2+2;
    else nbParts=(decimal)nbEnfants/2+1;
    if (nbEnfants>=3) nbParts+=0.5M;
    // calcul revenu imposable & Quotient familial
    decimal revenu=0.72M*salaire;
    decimal QF=revenu/nbParts;
    // calcul de l'impôt
    limites[limites.Length-1]=QF+1;
    int i=0;
    while(QF>limites[i]) i++;
    // retour résultat
    return (long)(revenu*coeffR[i]-nbParts*coeffN[i]);
  }//calculer
}//classe
```

Un objet **impôt** est créé avec les données permettant le calcul de l'impôt d'un contribuable. C'est la partie stable de l'objet. Une fois cet objet créé, on peut appeler de façon répétée sa méthode **calculer** qui calcule l'impôt du contribuable à partir de son statut marital (marié ou non), son nombre d'enfants et son salaire annuel.

Un programme de test pourait être le suivant :

```
using System;

  class test
  {
    public static void Main()
    {
      // programme interactif de calcul d'impôt
      // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
      // le programme affiche alors l'impôt à payer
```

```
      const string syntaxe="syntaxe : marié nbEnfants salaire\n"
             +"marié : o pour marié, n pour non marié\n"
             +"nbEnfants : nombre d'enfants\n"
             +"salaire : salaire annuel en F";

      // tableaux de données nécessaires au calcul de l'impôt
      decimal[] limites=new decimal[]
{12620M,13190M,15640M,24740M,31810M,39970M,48360M,55790M,92970M,127860M,151250M,172040M,195000M,0M};
      decimal[] coeffR=new decimal[]
{0M,0.05M,0.1M,0.15M,0.2M,0.25M,0.3M,0.35M,0.4M,0.45M,0.5M,0.55M,0.6M,0.65M};
      decimal[] coeffN=new decimal[]
{0M,631M,1290.5M,2072.5M,3309.5M,4900M,6898.5M,9316.5M,12106M,16754.5M,23147.5M,30710M,39312M,49062M};

      // création d'un objet impôt
      impôt objImpôt=null;
      try{
        objImpôt=new impôt(limites,coeffR,coeffN);
      }catch (Exception ex){
        Console.Error.WriteLine("L'erreur suivante s'est produite : " + ex.Message);
        Environment.Exit(1);
      }//try-catch

      // boucle infinie
      while(true){
        // on demande les paramètres du calcul de l'impôt
        Console.Out.Write("Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour
arrêter :");
        string paramètres=Console.In.ReadLine().Trim();
        // qq chose à faire ?
        if(paramètres==null || paramètres==="") break;
        // vérification du nombre d'arguments dans la ligne saisie
        string[] args=paramètres.Split(null);
        int nbParamètres=args.Length;
        if (nbParamètres!=3){
          Console.Error.WriteLine(syntaxe);
          continue;
        }//if
        // vérification de la validité des paramètres
        // marié
        string marié=args[0].ToLower();
        if (marié!="o" && marié !="n"){
          Console.Error.WriteLine(syntaxe+"\nArgument marié incorrect : tapez o ou n");
          continue;
        }//if
        // nbEnfants
        int nbEnfants=0;
        try{
          nbEnfants=int.Parse(args[1]);
          if(nbEnfants<0) throw new Exception();
        }catch (Exception){
          Console.Error.WriteLine(syntaxe+"\nArgument nbEnfants incorrect : tapez un entier positif ou
nul");
          continue;
        }//if
        // salaire
        int salaire=0;
        try{
          salaire=int.Parse(args[2]);
          if(salaire<0) throw new Exception();
        }catch (Exception){
          Console.Error.WriteLine(syntaxe+"\nArgument salaire incorrect : tapez un entier positif ou nul");
          continue;
        }//if
        // les paramètres sont corrects - on calcule l'impôt
        Console.Out.WriteLine("impôt="+objImpôt.calculer(marié=="o",nbEnfants,salaire)+" F");
        // contribuable suivant
      }//while
    }//Main
  }//classe
```

Voici un exemple d'exécution du programme précédent :

```
E:\data\serge\MSNET\c#\impots\3>test

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :q s d
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F
Argument marié incorrect : tapez o ou n

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o s d
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F
Argument nbEnfants incorrect : tapez un entier positif ou nul
```

```
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 d
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F
Argument salaire incorrect : tapez un entier positif ou nul

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :q s d f
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 200000
impôt=22504 F

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :
```

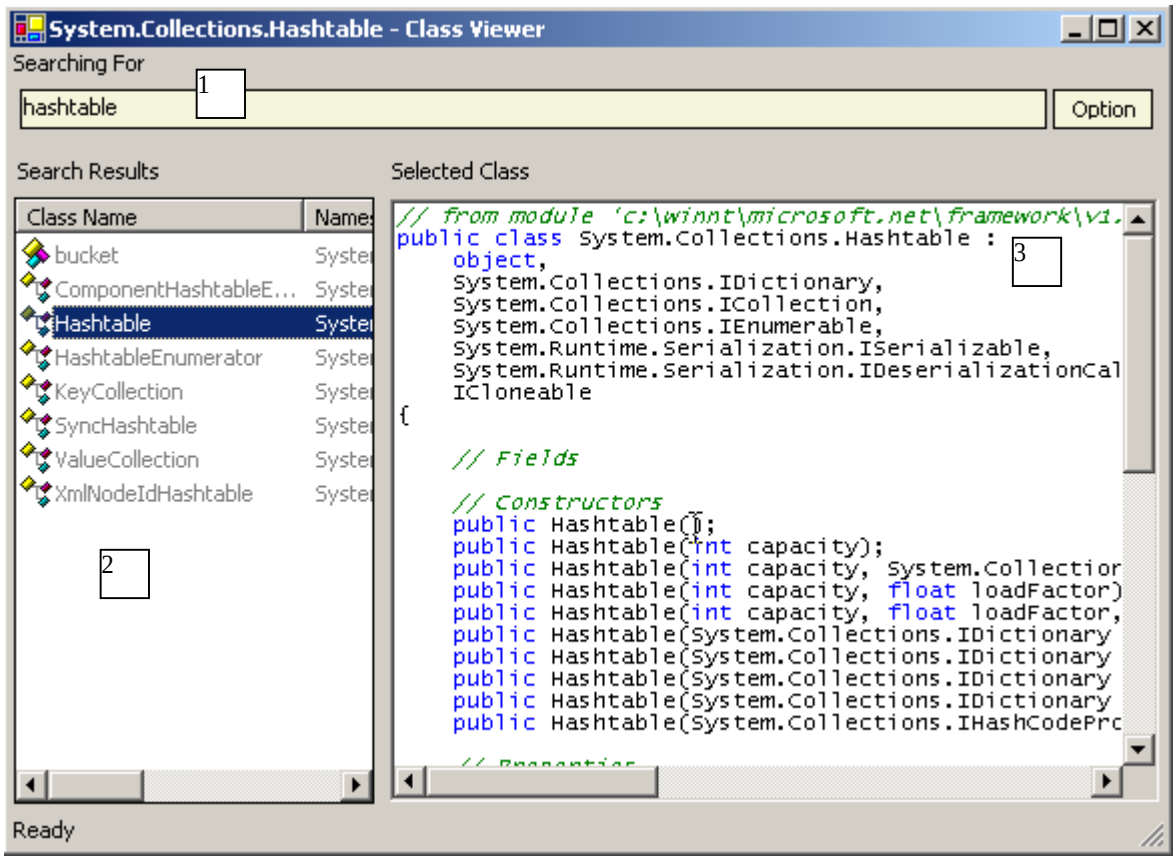
3. Classes .NET d'usage courant

Nous présentons ici quelques classes de la plate-forme .NET présentant un intérêt même pour un débutant. Nous montrons tout d'abord comment obtenir des renseignements sur les centaines de classes disponibles.

3.1 Chercher de l'aide avec SDK.NET

3.1.1 wincv

Si on a installé seulement le SDK et pas Visual Studio.NET on pourra utiliser le programme *wincv.exe* situé normalement dans l'arborescence du sdk, par exemple *C:\Program Files\Microsoft.Net\FrameworkSDK\Bin*. Lorsqu'on lance cet utilitaire, on a l'interface suivante :



On tape en (1) le nom de la classe désirée. Cela ramène en (2) divers thèmes possibles. On choisit celui qui convient et on a le résultat en (3), ici la classe *HashTable*. Cette méthode convient si on connaît le nom de la classe que l'on cherche. Si on veut explorer la liste des possibiltés offertes par la plate-forme .NET ou pourra utiliser le fichier HTML *C:\Program Files\Microsoft.Net\FrameworkSDK\StartHere.htm* qui sert de point de départ pour l'aide sur le SDK :

Microsoft .NET Framework SDK

Welcome to the Beta 2 release of the Microsoft .NET Framework Software Development Kit. The .NET Framework SDK is the essential reference for developers who use .NET Framework technologies.

Getting Started

For those new to the .NET Framework technologies, a great place to start is the "Getting Started with the .NET Framework" section of the .NET Framework SDK documentation.

For known issues and late-breaking information see the Release Notes.

QuickStarts, Tutorials and Samples

The .NET Framework SDK QuickStarts, tutorials and samples are designed to quickly acquaint you with the programming model, architecture and power of writing applications and components based on the .NET Framework technologies.

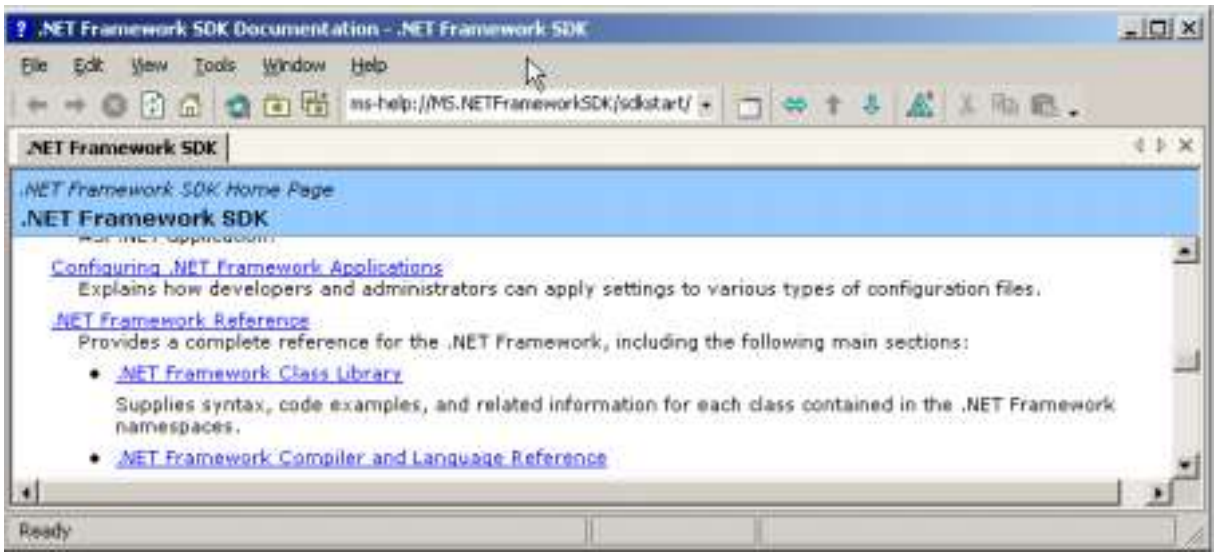
Documentation

The .NET Framework SDK documentation provides a wide range of overviews, programming tasks and class library reference information designed to help you build efficient, powerful, and scalable applications based on .NET Framework technologies.

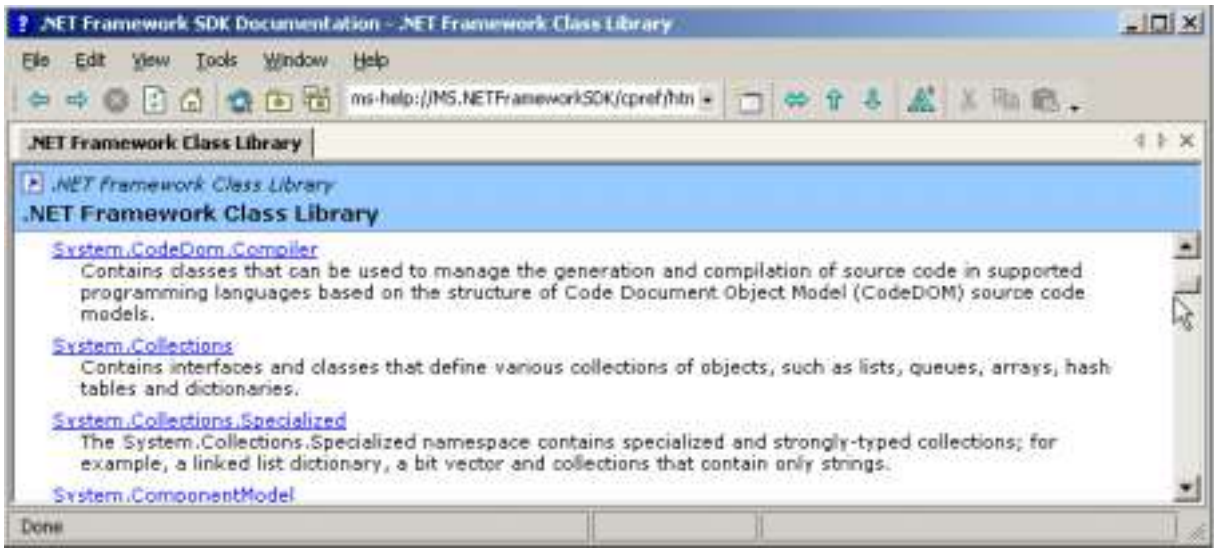
Tools and Debuggers

The .NET Framework SDK tools and debuggers enable you to create, deploy, and manage applications and components that target the .NET Framework.

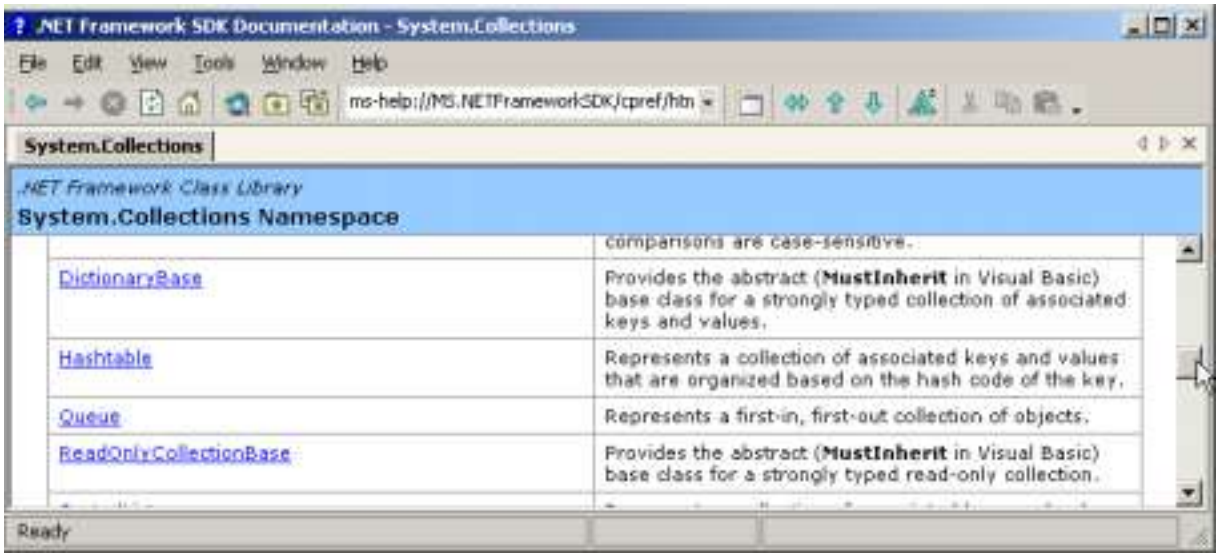
Le lien *.NET Framework SDK Documentation* estcelui qu'il faut suivre pour avoir une vue d'ensemble des classes .NET :



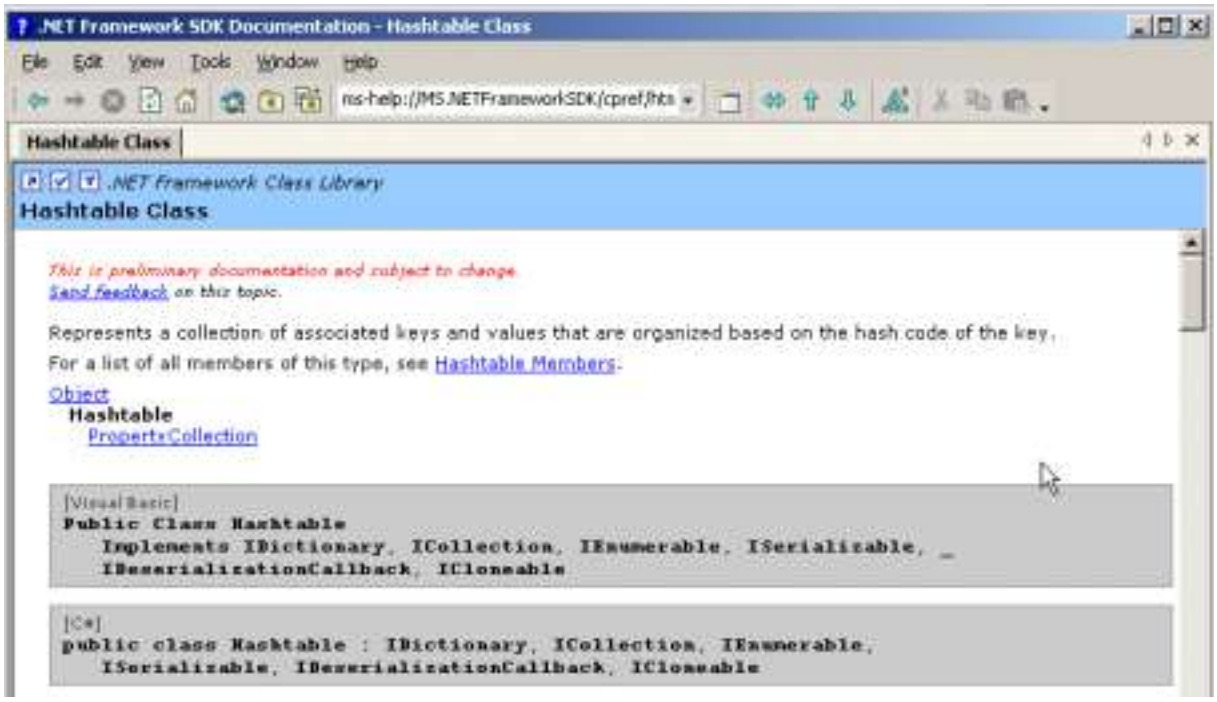
Là, on suivra le lien *.NET Framework Class Library*. On y trouve la liste de toutes les classes de .NET :



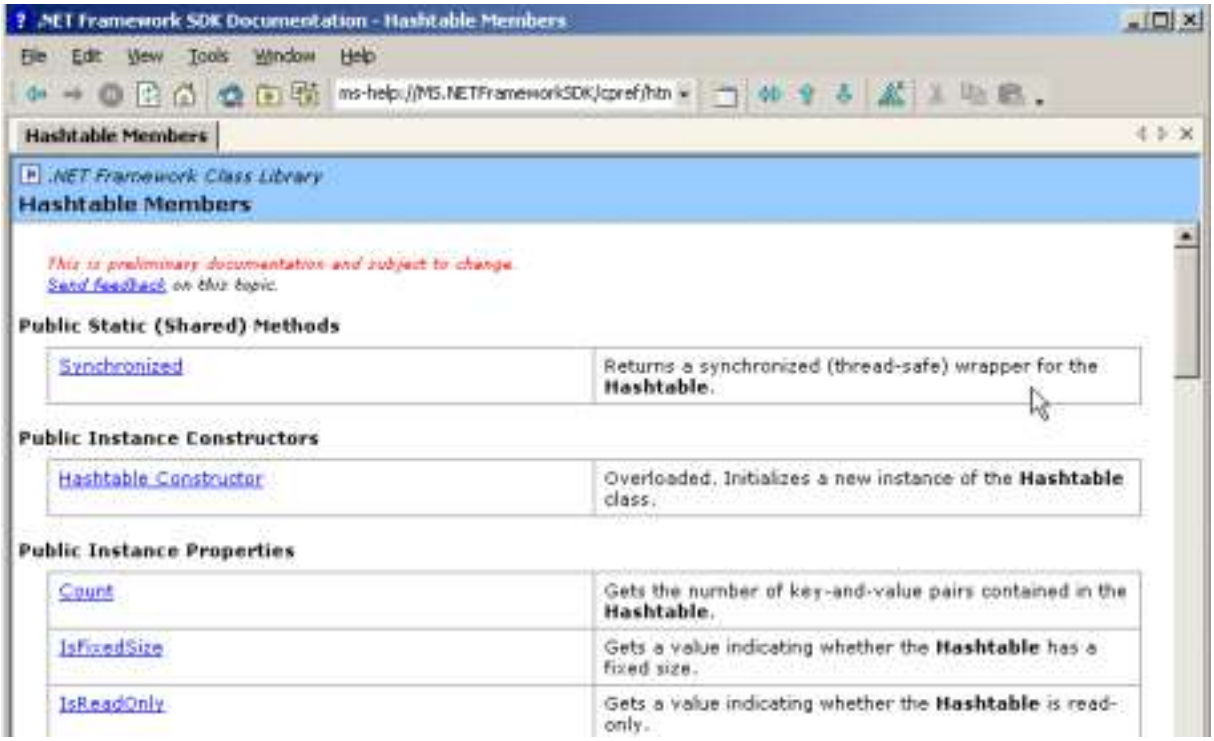
Suivons par exemple le lien *System.Collections*. Cet espace de noms regroupe diverses classes implémentant des collections dont la classe *HashTable* :



Suivons le lien *HashTable* ci-dessus :



On y trouve le prototype de la classe ainsi que des exemples d'utilisation. En suivant ci-dessus le lien *HashTable Members*, on a accès à la description complète de la classe :



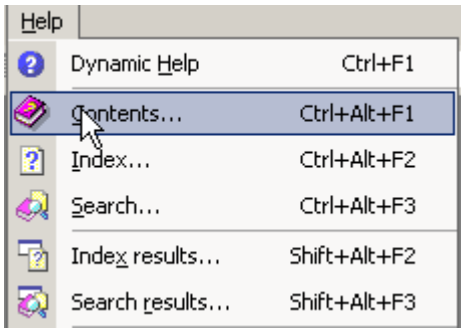
Cette méthode est la meilleure pour découvrir le SDK et ses classes. L'outil WinCV s'avère utile lorsqu'on connaît déjà un peu la classe et qu'on a oublié certains de ses membres. WinCV permet de retrouver rapidement la classe et ses membres.

3.2 Chercher de l'aide sur les classes avec VS.NET

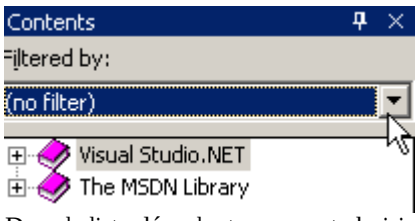
Nous donnons ici quelques indications pour trouver de l'aide avec Visual Studio.NET

3.2.1 Help/Contents

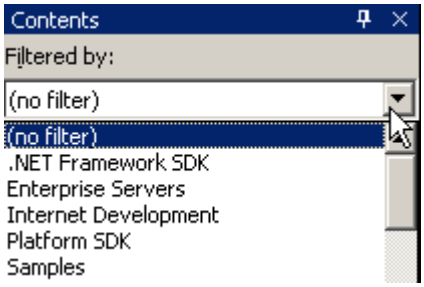
Prenez l'option *Help/Contents* du menu.



On obtient la fenêtre suivante :



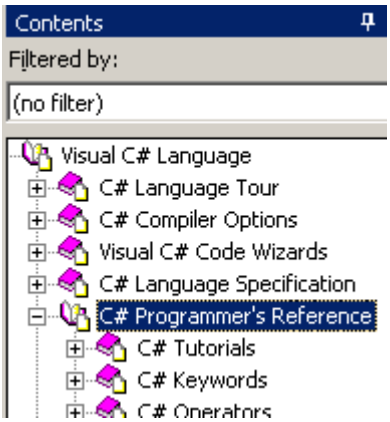
Dans la liste déroulante, on peut choisir un filtre d'aide :



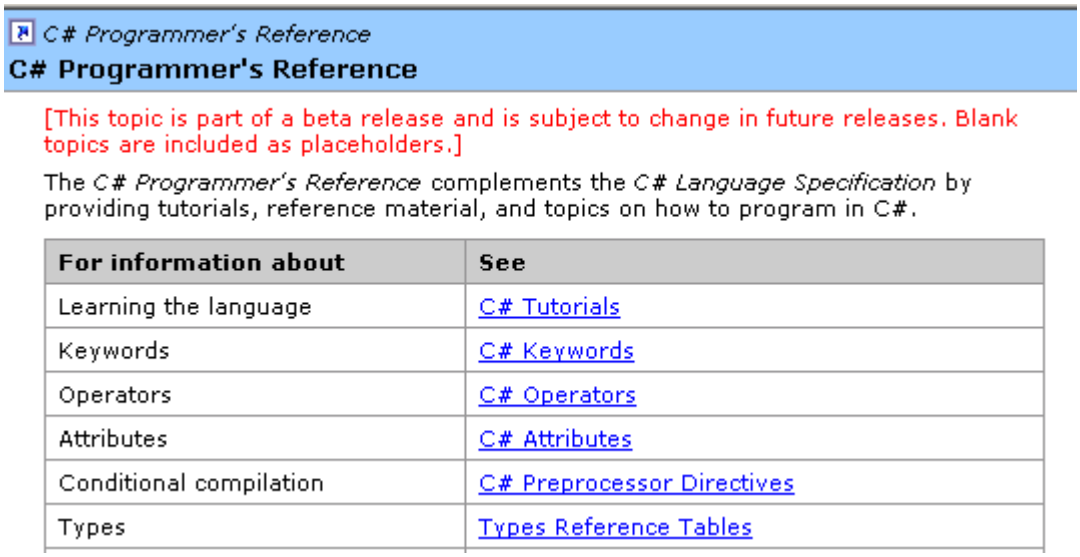
On pourra prendre (no filter) ce qui nous donne accès à toute l'aide. Deux aides sont utiless :

- l'aide sur le langage C# lui-même (syntaxe)
- l'aide sur les classes.NET utilisables par le langage C#

L'aide au langage C# est accessible via *Visual Studio.NET/Visual Basic and Visual C#/Visual C# language/C# Programmer's Reference* :



On obtient la page d'aide suivante :



C# Programmer's Reference

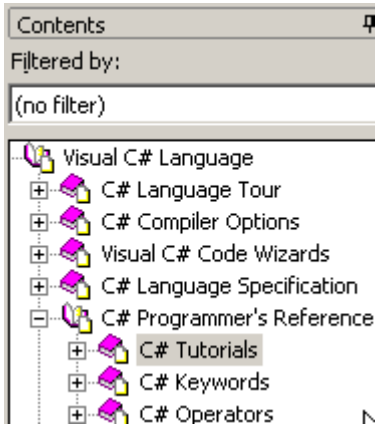
C# Programmer's Reference

[This topic is part of a beta release and is subject to change in future releases. Blank topics are included as placeholders.]

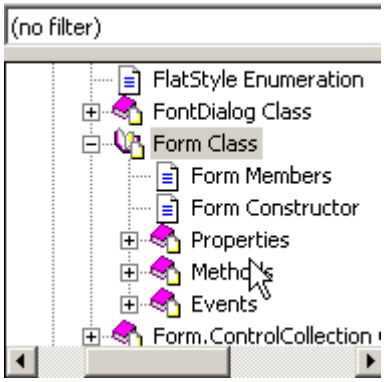
The *C# Programmer's Reference* complements the *C# Language Specification* by providing tutorials, reference material, and topics on how to program in C#.

For information about	See
Learning the language	C# Tutorials
Keywords	C# Keywords
Operators	C# Operators
Attributes	C# Attributes
Conditional compilation	C# Preprocessor Directives
Types	Types Reference Tables

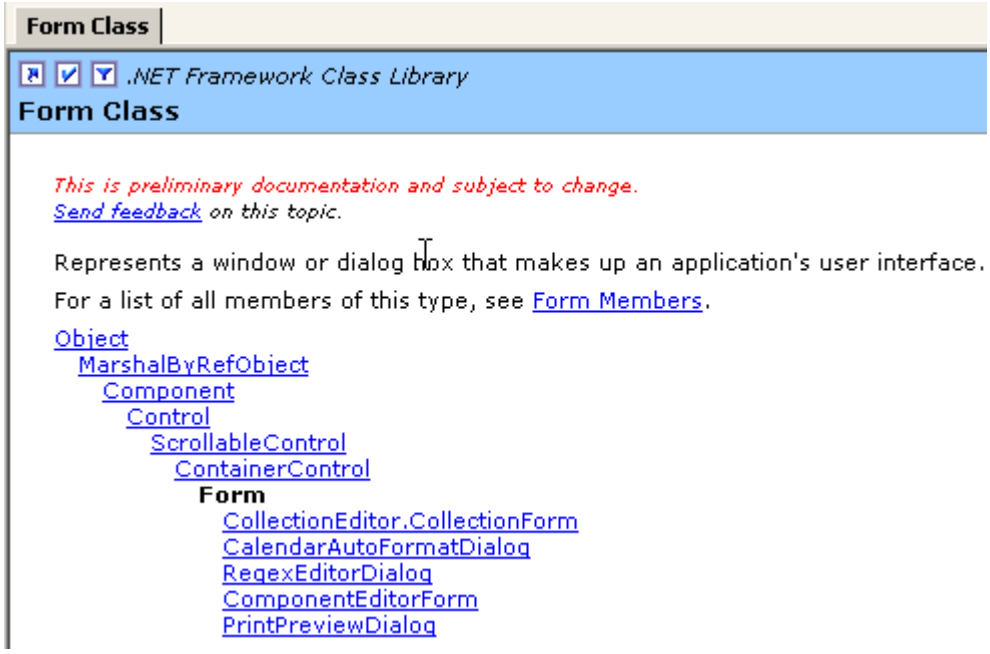
A partir de là, les différentes sous-rubriques nous permettent d'avoir de l'aide sur différents thèmes de C#. On prêtera attention aux tutoriels de C# :



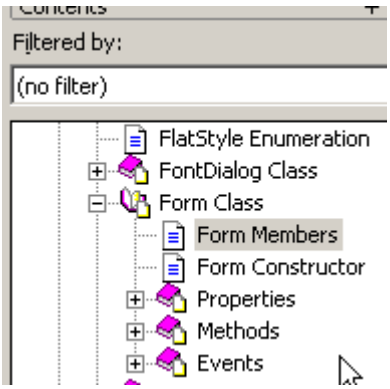
Pour avoir accès aux différentes classes de la plate-forme .NET, on choisira l'aide *Visual Studio.NET/.NET Framework/.NET Framework Reference/.NET Framework Class Library*. On peut alors chercher la classe *System.Windows.Forms.Form* :



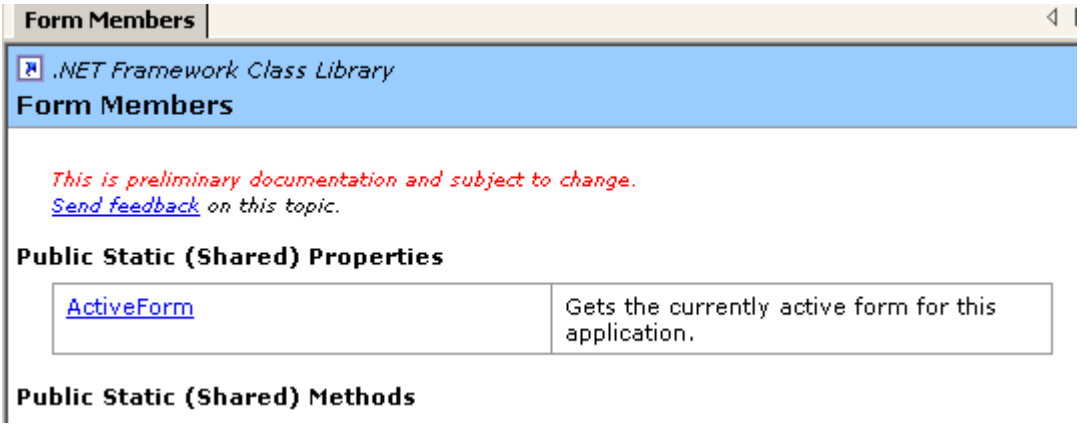
On a alors accès à la page d'aide suivante :



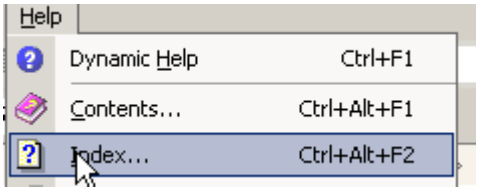
En choisissant l'option *Form members* dans l'aide :



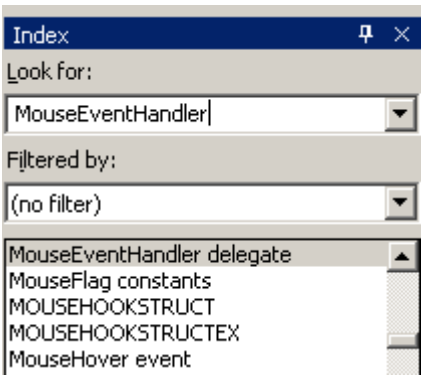
on a la liste de tous les membres de la classe :



3.2.2 Help/Index



L'option *Help/index* permet de chercher une aide plus ciblée que l'aide précédente. Il suffit de taper le mot clé cherché :



MouseEventHandler Delegate

.NET Framework Class Library

MouseEventHandler Delegate

This is preliminary documentation and subject to change.
Send feedback on this topic.

Represents the method that will handle the **MouseDown**, **MouseUp**, or **MouseMove** event of a form, control, or other component.

[Visual Basic]
```
Public Delegate Sub MouseEventHandler( _
    ByVal sender As Object, _
    ByVal e As MouseEventArgs _
)
```

[C#]
```
public delegate void MouseEventHandler(
    object sender,
    MouseEventArgs e
);
```

3.3 La classe String

La classe **String** est identique au type simple **string**. Elle présente de nombreuses propriétés et méthodes. En voici quelques-unes :

```
public int Length { get; }
      // nombre de caractères de la chaîne
public bool EndsWith(string value);
      // rend vrai si la chaîne se termine par value
public bool StartsWith(string value);
      // rend vrai si la chaîne commence par value
public virtual bool Equals(object obj);
      // rend vrai si la chaînes est égale à obj - équivalent chaîne==obj
public int IndexOf(string value, int startIndex);
      // rend la première position dans la chaîne de la chaîne value - la recherche commence à partir du
      caractère n° startIndex
public int IndexOf(char value, int startIndex);
      // idem mais pour le caractère value
public string Insert(int startIndex, string value);
      // insère la chaîne value dans chaîne en position startIndex
public static string Join(string separator, string[] value);
      // méthode de classe - rend une chaîne de caractères, résultat de la concaténation des valeurs du
      tableau value avec le séparateur separator
public int LastIndexOf(char value, int startIndex, int count)
public int LastIndexOf(string value, int startIndex, int count);
      // idem indexOf mais rend la dernière position au lieu de la première
public string Replace(char oldChar, char newChar);
      // rend une chaîne copie de la chaîne courante où le caractère oldChar a été remplacé par le
      caractère newChar
public string[] Split(char[] separator);
      // la chaîne est vue comme une suite de champs séparés par les caractères présents dans le tableau
      separator. Le résultat est le tableau de ces champs
public string Substring(int startIndex, int length);
      // sous-chaîne de la chaîne courante commençant à la position startIndex et ayant length caractères
public string ToLower();
      // rend la chaîne courant en minuscules
public string ToUpper();
      // rend la chaîne courante en majuscules
public string Trim();
      // rend la chaîne courante débarassée de ses espaces de début et fin
```

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;

public class string1{
  // une classe de démonstration
  public static void Main(){
    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]+"]");
    }//for
    affiche("Join(\":\",champs)="+System.String.Join(":",champs));
    affiche("(\"  abc  \").Trim()=["+"  abc  ".Trim()+"]");
  }//Main
  public static void affiche(String msg){
    // affiche msg
    Console.Out.WriteLine(msg);
  }//affiche
}//classe
```

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 :

```
// split, join

using System;

public class splitJoin{
  public 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);
    int i;
    for (i=0;i<champs.Length;i++)
      Console.Out.WriteLine("Champs[" + i + "]=" + champs[i]);
    // join
    Console.Out.WriteLine("join=[" + System.String.Join(":",champs)+"]");
    // attente
    Console.ReadLine();
  }//Main
}//classe
```

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 les champs d'une chaîne de caractères. La définition de la méthode utilisée ici est la suivante :

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

separator tableau de caractères. Ces caractères représentent les caractères utilisés pour séparer les champs de la chaîne de caractères. Ainsi si la chaîne est champ1, champ2, champ3 on pourra utiliser *separator=new char[] {',',}*. Si le séparateur est une suite d'espaces on utilisera *separator=null*.

résultat tableau de chaînes de caractères où chaque élément est un champ de la chaîne.

La méthode **Join** est une méthode statique de la classe *String* :

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

value tableau de chaînes de caractères

separator une chaîne de caractères qui servira de séparateur de champs

résultat une chaîne de caractères formée de la concaténation des éléments du tableau *value* séparés par la chaîne *separator*.

3.4 La classe Array

La classe **Array** implémente un tableau. Nous utiliserons dans notre exemple les propriétés et méthodes suivantes :

```
public int Length { get; }
```

// propriété - nombre d'éléments du tableau

```
public static int BinarySearch(Array array, int index, int length, object value);
```

// méthode de classe - rend la position de *value* dans le tableau trié *array* - cherche à partir de la position *index* et avec *length* éléments

```
public static void Copy(Array sourceArray, Array destinationArray, int length);
```

// méthode de classe - copie *length* éléments de *sourceArray* dans *destinationArray* - *destinationArray* est créé pour l'occasion

```
public static void Sort(Array array);
```

// méthode de classe - trie le tableau *array* - celui doit contenir un type de données ayant une fonction d'ordre par défaut (chaînes, nombres).

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

```
// tableaux

// imports
using System;

public class tab1{
  public static void Main(string[] args){
    // lecture des éléments d'un tableau tapés au clavier
    Boolean terminé=false;
    int i=0;
    double[] éléments1=null;
    double[] éléments2=null;
    double élément=0;
    string réponse=null;
    Boolean erreur=false;

    while (! terminé){
      // question
      Console.Out.Write("Elément (réel) " + i + " du tableau (rien pour 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
      // si pas d'erreur
      if (! erreur){
        // un élémt de plus dans le tableau
        i+=1;
        // nouveau tableau pour accueillir le nouvel élément
        éléments2=new double[i];
        // copie ancien tableau vers nouveau tableau
        if(i!=1) Array.Copy(éléments1,éléments2,i-1);
        // nouveau tableau devient ancien tableau
```

```
        éléments1=éléments2;
        // plus besoin du nouveau tableau
        éléments2=null;
  // insertion nouvel élément
        éléments1[i-1]=élément;
      }//if
    }//while
    // affichage tableau non trié
    System.Console.Out.WriteLine("Tableau non trié");
    for (i=0;i<éléments1.Length;i++)
      Console.Out.WriteLine("éléments[" + i + "]=" + éléments1[i]);
    // tri du tableau
    System.Array.Sort(éléments1);
    // affichage tableau trié
    System.Console.Out.WriteLine("Tableau trié");
    for (i=0;i<éléments1.Length;i++)
      Console.Out.WriteLine("éléments[" + i + "]=" + éléments1[i]);
    // Recherche
    while ( ! terminé){
      // question
      Console.Out.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.Error.WriteLine("Erreur, recommencez...");
        erreur=true;
      }//try-catch
      // si pas d'erreur
      if (! erreur){
        // on cherche l'élément dans le tableau trié
        i=System.Array.BinarySearch(éléments1,0,éléments1.Length,élément);
        // Affichage réponse
        if (i>=0)
         Console.Out.WriteLine("Trouvé en position " + i);
        else Console.Out.WriteLine("Pas dans le tableau");
      }//if
    }//while
  }//Main
}//classe
```

Les résultats écran sont les suivants :

```
Elément (réel) 0 du tableau (rien pour terminer) : 3.6
Saisie incorrecte, recommencez
Elément (réel) 0 du tableau (rien pour terminer) : 3,6
Elément (réel) 1 du tableau (rien pour terminer) : 7,4
Elément (réel) 2 du tableau (rien pour terminer) : -1,5
Elément (réel) 3 du tableau (rien pour terminer) : -7
Elément (réel) 4 du tableau (rien pour terminer) :
Tableau non trié
éléments[0]=3,6
éléments[1]=7,4
éléments[2]=-1,5
éléments[3]=-7
Tableau trié
éléments[0]=-7
éléments[1]=-1,5
éléments[2]=3,6
éléments[3]=7,4
Elément cherché (rien pour arrêter) :
3,6
Trouvé en position 2
Elément cherché (rien pour arrêter) :
4
Pas dans le tableau
Elément cherché (rien pour arrêter) :
```

La première partie du programme construit un tableau à partir de données numériques tapées au clavier. Le tableau ne peut être dimensionné à priori puisqu'on ne connaît pas le nombre d'éléments que va taper l'utilisateur. On travaille alors avec deux tableaux *éléments1* et *éléments2*.

```
        // un élémt de plus dans le tableau
        i+=1;
        // nouveau tableau pour accueillir le nouvel élément
        éléments2=new double[i];
        // copie ancien tableau vers nouveau tableau
        if(i!=1) Array.Copy(éléments1,éléments2,i-1);
        // nouveau tableau devient ancien tableau
        éléments1=éléments2;
```

```
        // plus besoin du nouveau tableau
        éléments2=null;
  // insertion nouvel élément
        éléments1[i-1]=élément;
```

Le tableau *éléments1* contient les valeurs actuellement saisies. Lorsque l'utilisateur ajoute une nouvelle valeur, on construit un tableau *éléments2* avec un élément de plus que *éléments1*, on copie le contenu de *éléments1* dans *éléments2* (Array.Copy), on fait "pointer" *éléments1* sur *éléments2* et enfin on ajoute le nouvel élément à *éléments1*. C'est un processus complexe qui peut être simplifié si au lieu d'utiliser un tableau de taille fixe (*Array*) on utilise un tableau de taille variable (*ArrayList*).

Le tableau est trié avec la méthode **Array.Sort**. Cette méthode peut être appelée avec différents paramètres précisant les règles de tri. Sans paramètres, c'est ici un tri en ordre croissant qui est fait par défault. Enfin, la méthode **Array.BinarySearch** permet de chercher un élément dans un tableau trié.

3.5 La classe ArrayList

La classe *ArrayList* permet d'implémenter des tableaux de taille variable au cours de l'exécution du programme, ce que ne permet pas la classe *Array* précédente. Voici quelques-unes des propriétés et méthodes courantes :

// constructeur

```
ArrayList()
```

// construit un tableau vide

// propriétés

```
int Count
```

// nombre d'éléments du tableau

// méthodes

```
public virtual int Add(object value);
```

ajoute l'objet value à la fin du tableau

```
public virtual void AddRange(System.Collections.ICollection c)
```

ajoute la collection (peut être un tableau) à la fin du tableau

```
public virtual void Clear()
```

efface le tableau

```
public virtual int IndexOf(object value);
```

indice de l'objet value dans le tableau ou -1 s'il n'existe pas

```
public virtual int IndexOf(object value, int startIndex);
```

idem mais en cherchant à partir de l'élément n° startIndex

```
public virtual int LastIndexOf(object value);
public virtual int LastIndexOf(object value, int startIndex);
```

idem mais rend l'indice de la dernière occurrence de value dans le tableau

```
public virtual void Remove(object obj);
```

enève l'objet obj s'il existe dans le tableau

```
public virtual void RemoveAt(int index);
```

enlève l'élément i du tableau

```
public virtual int BinarySearch(object value)
```

rend la position de l'objet value dans le tableau ou -1 s'il ne s'y trouve pas. Le tableau doit être trié

```
public virtual void Sort()
```

trie le tableau. Celui-ci doit contenir des objets ayant une relation d'ordre prédéfinie (chaînes, nombres)

```
public virtual void Sort(System.Collections.IComparer comparer)
```

trie le tableau selon la relation d'ordre établie par la fonction comparer

Reprenons l'exemple traité avec des objets de type *Array* et traitons-le avec des objets de type *ArrayList* :

```
// tableaux dynamiques

// imports
using System;
using System.Collections;

public class tab1{
  public static void Main(string[] args){
    // lecture des éléments d'un tableau tapés au clavier
    Boolean terminé=false;
    int i=0;
    ArrayList éléments=new ArrayList();
    double élément=0;
    string réponse=null;
    Boolean erreur=false;

    while (! terminé){
```

```
      // question
      Console.Out.Write("Elément (réel) " + i + " du tableau (rien pour 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
      // si pas d'erreur
      if (! erreur){
        // un élémt de plus dans le tableau
        éléments.Add(élément);
      }//if
    }//while
    // affichage tableau non trié
    System.Console.Out.WriteLine("Tableau non trié");
    for (i=0;i<éléments.Count;i++)
      Console.Out.WriteLine("éléments[" + i + "]=" + éléments[i]);
    // tri du tableau
    éléments.Sort();
    // affichage tableau trié
    System.Console.Out.WriteLine("Tableau trié");
    for (i=0;i<éléments.Count;i++)
      Console.Out.WriteLine("éléments[" + i + "]=" + éléments[i]);
    // Recherche
    while ( ! terminé){
      // question
      Console.Out.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.Error.WriteLine("Erreur, recommencez...");
        erreur=true;
      }//try-catch
      // si pas d'erreur
      if (! erreur){
        // on cherche l'élément dans le tableau trié
        i=éléments.BinarySearch(élément);
        // Affichage réponse
        if (i>=0)
         Console.Out.WriteLine("Trouvé en position " + i);
        else Console.Out.WriteLine("Pas dans le tableau");
      }//if
    }//while
  }//Main
}//classe
```

Les résultats d'exécution sont les suivants :

```
E:\data\serge\MSNET\c#\bases\3>sort2
Elément (réel) 0 du tableau (rien pour terminer) : 6,7
Elément (réel) 0 du tableau (rien pour terminer) : -4,5
Elément (réel) 0 du tableau (rien pour terminer) : 8,3
Elément (réel) 0 du tableau (rien pour terminer) : 2
Elément (réel) 0 du tableau (rien pour terminer) :
Tableau non trié
éléments[0]=6,7
éléments[1]=-4,5
éléments[2]=8,3
éléments[3]=2
Tableau trié
éléments[0]=-4,5
éléments[1]=2
éléments[2]=6,7
éléments[3]=8,3
Elément cherché (rien pour arrêter) :
2
Trouvé en position 1
Elément cherché (rien pour arrêter) :
1
Pas dans le tableau
Elément cherché (rien pour arrêter) :
```

3.6 La classe Hashtable

La classe *Hashtable* permet d'implémenter un dictionnaire. On peut voir un dictionnaire comme un tableau à deux colonnes :

clé	valeur
clé1	valeur1
clé2	valeur2
..	...

Les clés sont uniques, c.a.d. qu'il ne peut y avoir deux clés indentiques. Les méthodes et propriétés principales de la classe **Hashtable** sont les suivantes :

`public Hashtable();`	crée un dictionnaire vide
`public virtual void Add(object key, object value);`	ajoute une ligne (key,value) dans le dictionnaire où key et value sont des références d'objets.
`public virtual void Remove(object key);`	élimine du dictionnaire la ligne de clé=key
`public virtual void Clear();`	vide le dictionnaire
`public virtual bool ContainsKey(object key);`	rend vrai (true) si la clé key appartient au dictionnaire.
`public virtual bool ContainsValue(object value);`	rend vrai (true) si la valeur value appartient au dictionnaire.
`public int Count { virtual get; }`	propriété : nombre d'éléments du dictionnaire
`public ICollection Keys { virtual get; }`	propriété : collection des clés du dictionnaire
`public ICollection Values { virtual get; }`	propriété : collection des valeurs du dictionnaire
`public object this[ object key ] { virtual get; virtual set; }`	propriété indexée : permet de connaître ou de fixer la valeur associée à une clé key

Considérons le programme exemple suivant :

```
using System;
using System.Collections;

public class HashTable1{
  public static void Main(String[] args){
    String[] liste=new String[] {"jean:20","paul:18","mélanie:10","violette:15"};
    int i;
    String[] champs=null;
    char[] séparateurs=new char[]{':'};
    // remplissage du dictionnaire
    Hashtable dico=new Hashtable();
    for (i=0; i<=liste.Length-1;i++){
      champs=liste[i].Split(séparateurs);
      dico.Add(champs[0],champs[1]);
    }//for
    // nbre d'éléments dans le dictionnaire
    Console.Out.WriteLine("Le dictionnaire a " + dico.Count + " éléments");
    // liste des clés
    Console.Out.WriteLine("[Liste des clés]");
    IEnumerator clés=dico.Keys.GetEnumerator();
    while(clés.MoveNext()){
      Console.Out.WriteLine(clés.Current);
    }
    // liste des valeurs
    Console.Out.WriteLine("[Liste des valeurs]");
    IEnumerator valeurs=dico.Values.GetEnumerator();
    while(valeurs.MoveNext()){
      Console.Out.WriteLine(valeurs.Current);
    }
    // liste des clés & valeurs
    Console.Out.WriteLine("[Liste des clés & valeurs]");
    clés.Reset();
    while(clés.MoveNext()){
      Console.Out.WriteLine("clé="+clés.Current+" valeur="+dico[clés.Current]);
    }
    // on supprime la clé "paul"
    Console.Out.WriteLine("[Suppression d'une clé]");
    dico.Remove("paul");
    // liste des clés & valeurs
    Console.Out.WriteLine("[Liste des clés & valeurs]");
    clés=dico.Keys.GetEnumerator();
    while(clés.MoveNext()){
      Console.Out.WriteLine("clé="+clés.Current+" valeur="+dico[clés.Current]);
    }

    // recherche dans le dictionnaire
    String nomCherché=null;
    Console.Out.Write("Nom recherché (rien pour arrêter) : ");
    nomCherché=Console.ReadLine().Trim();
    Object value=null;
    while (! nomCherché.Equals("")){
```

```
      if (dico.ContainsKey(nomCherché)){
        value=dico[nomCherché];
        Console.Out.WriteLine(nomCherché+","+(String)value);
      }else{
        Console.Out.WriteLine("Nom " + nomCherché + " inconnu");
      }
      // recherche suivante
      Console.Out.Write("Nom recherché (rien pour arrêter) : ");
      nomCherché=Console.ReadLine().Trim();
    }//while
  }//Main
}//Classe
```

Les résultats d'exécution sont les suivants :

```
E:\data\serge\MSNET\c#\bases\7>hashtable1
Le dictionnaire a 4 éléments
[Liste des clés]
mélanie
paul
violette
jean
[Liste des valeurs]
10
18
15
20
[Liste des clés & valeurs]
clé=mélanie valeur=10
clé=paul valeur=18
clé=violette valeur=15
clé=jean valeur=20
[Suppression d'une clé]
[Liste des clés & valeurs]
clé=mélanie valeur=10
clé=violette valeur=15
clé=jean valeur=20
Nom recherché (rien pour arrêter) : paul
Nom paul inconnu
Nom recherché (rien pour arrêter) : jean
jean,20
Nom recherché (rien pour arrêter) :
```

Le programme utilise également un objet **Ienumerator** pour parcourir les collections de clés et de valeurs du dictionnaire de type **ICollection** (cf ci-dessus les propriétés Keys et Values). Une collection est un ensemble d'objets qu'on peut parcourir. L'interface *ICollection* est définie comme suit :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public interface System.Collections.ICollection :System.Collections.IEnumerable
{

    // Properties
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }

    // Methods
    void CopyTo(Array array, int index);
} // end of System.Collections.ICollection
```

La propriété *Count* nous permet de connaître le nombre d'éléments de la collection. L'interface *ICollection* dérive de l'interface **IEnumerable** :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public interface System.Collections.IEnumerable
{

    // Methods
    System.Collections.IEnumerator GetEnumerator();
} // end of System.Collections.IEnumerable
```

Cette interface n'a qu'une méthode **GetEnumerator** qui nous permet d'obtenir un objet de type **IEnumerator** :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public interface System.Collections.IEnumerator
{

    // Properties
    object Current { get; }

    // Methods
    bool MoveNext();
```

```
    void Reset();
} // end of System.Collections.IEnumerator
```

La méthode *GetEnumerator()* d'une collection *ICollection* nous permet de parcourir la collection avec les méthodes suivantes :

MoveNext positionne sur l'élément suivant de la collection. Rend vrai (true) si cet élément existe, faux (false) sinon. Le premier *MoveNext* positionne sur le 1er élément. L'élément "courant" de la collection est alors disponible dans la propriété *Current* de l'énumérateur
Current propriété : élément courant de la collection
Reset repositionne l'énumérateur en début de collection, c.a.d. avant le 1er élément.

La structure d'itération sur les éléments d'une collection (*ICollection*) C est donc la suivante :

```
// définir la collection
ICollection C=...;
// obtenir un énumérateur de cette collection
IEnumerator itérateur=C.GetEnumerator();
// parcourir la collection avec cet énumérateur
while(itérateur.MoveNext()){
  // on a un élément courant
  // exploiter itérateur.Current
}//while
```

3.7 La classe StreamReader

La classe *StreamReader* permet de lire le contenu d'un fichier. Voici quelques-unes de ses propriétés et méthodes :

/ / constructeur

```
public StreamReader(string path)
```

ouvre un flux à partir du fichier path. Une exception est lancée si celui-ci n'existe pas

/ / méthodes

```
public virtual void Close()
```

ferme le flux

```
public virtual string ReadLine()
```

lit une ligne du flux ouvert

```
public virtual string ReadToEnd()
```

lit le reste du flux depuis la position courante

Voici un exemple :

```
// lecture d'un fichier texte

using System;
using System.IO;

public class listes1{
  public static void Main(string[] args){
    string ligne=null;
    StreamReader fluxInfos=null;
    // lecture contenu du fichier
    try{
      fluxInfos=new StreamReader("infos.txt");
      ligne=fluxInfos.ReadLine();
      while (ligne != null){
        System.Console.Out.WriteLine(ligne);
        ligne=fluxInfos.ReadLine();
      }
    }catch (Exception e) {
      System.Console.Error.WriteLine("L'erreur suivante s'est produite : " + e);
    }finally{
      try{
        fluxInfos.Close();
      }catch {}
      // attente
      Console.ReadLine();
    }//try-catch
   }//main
}//classe
```

et ses résultats d'exécution :

```
E:\data\serge\MSNET\c#\bases\4>more infos.txt
12620:0:0
13190:0,05:631
15640:0,1:1290,5
```

```
24740:0,15:2072,5
31810:0,2:3309,5
39970:0,25:4900
48360:0,3:6898,5
55790:0,35:9316,5
92970:0,4:12106
127860:0,45:16754,5
151250:0,5:23147,5
172040:0,55:30710
195000:0,6:39312
0:0,65:49062

E:\data\serge\MSNET\c#\bases\4>file1
12620:0:0
13190:0,05:631
15640:0,1:1290,5
24740:0,15:2072,5
31810:0,2:3309,5
39970:0,25:4900
48360:0,3:6898,5
55790:0,35:9316,5
92970:0,4:12106
127860:0,45:16754,5
151250:0,5:23147,5
172040:0,55:30710
195000:0,6:39312
0:0,65:49062
```

3.8 La classe StreamWriter

La classe *StreamWriter* permet d'écrire dans fichier. Voici quelques-unes de ses propriétés et méthodes :

// constructeur

```
public StreamWriter(string path)
```

ouvre un flux d'écriture dans le fichier *path*. Une exception est lancée si celui-ci ne peut être créé.

// propriétés

```
public bool AutoFlush { virtual get; virtual set; }
```

// si égal à vrai, l'écriture dans le flux ne passe pas par l'intermédiaire d'une mémoire tampon sinon l'écriture dans le flux n'est pas immédiate : il y a d'abord écriture dans une mémoire tampon puis dans le flux lorsque la mémoire tampon est pleine. Par défaut c'est le mode bufferisé qui est utilisé. Il convient bien pour les flux fichier mais généralement pas pour les flux réseau.

```
public string NewLine { virtual get; virtual set; }
```

pour fixer ou connaître la marque de fin de ligne à utiliser par la méthode WriteLine

// méthodes

```
public virtual void Close()
```

ferme le flux

```
public virtual string WriteLine(string value)
```

écrit value dans le flux ouvert suivi d'un saut de ligne

```
public virtual string Write(string value)
```

idem mais sans le saut de ligne

```
public virtual void Flush()
```

écrit la mémoire tampon dans le flux si on travaille en mode bufferisé

Considérons l'exemple suivant :

```
// écriture dans un fichier texte

using System;
using System.IO;

public class writeTextFile{
  public static void Main(string[] args){
    string ligne=null;          // une ligne de texte
    StreamWriter fluxInfos=null;  // le fichier texte
    try{
      // création du fichier texte
       fluxInfos=new StreamWriter("infos.txt");
      // lecture ligne tapée au clavier
       Console.Out.Write("ligne (rien pour arrêter) : ");
      ligne=Console.In.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.Out.Write("ligne (rien pour arrêter) : ");
        ligne=Console.In.ReadLine().Trim();
       }//while
    }catch (Exception e) {
      System.Console.Error.WriteLine("L'erreur suivante s'est produite : " + e);
    }finally{
      // fermeture fichier
      try{
        fluxInfos.Close();
      }catch {}
    }//try-catch
   }//main
}//classe
```

et les résultats d'exécution :

```
E:\data\serge\MSNET\c#\bases\4b>file1
ligne (rien pour arrêter) : ligne1
ligne (rien pour arrêter) : ligne2
ligne (rien pour arrêter) : ligne3
ligne (rien pour arrêter) :

E:\data\serge\MSNET\c#\bases\4b>more infos.txt
ligne1
ligne2
ligne3
```

3.9 La classe Regex

La classe *Regex* permet l'utilisation d'expression régulières. Celles-ci permettent de tester le format d'une chaîne de caractères. Ainsi on peut vérifier qu'une chaîne représentant une date est bien au format jj/mm/aa. On utilise pour cela un modèle et on compare la chaîne à ce modèle. Ainsi dans cet exemple, j m et a doivent être des chiffres. Le modèle d'un format de date valide est alors **"\d\d/\d\d/\d\d"** où le symbole **\d** désigne un chiffre. Les symboles utilisables dans un modèle sont les suivants (documentation Microsoft) :

Caractère	Description
\	Marque le caractère suivant comme caractère spécial ou littéral. Par exemple, "n" correspond au caractère "n". "\n" correspond à un caractère de nouvelle ligne. La séquence "\\" correspond à "\", tandis que "\(" correspond à "(".
^	Correspond au début de la saisie.
$	Correspond à la fin de la saisie.
*****	Correspond au caractère précédent zéro fois ou plusieurs fois. Ainsi, "zo*" correspond à "z" ou à "zoo".
+	Correspond au caractère précédent une ou plusieurs fois. Ainsi, "zo+" correspond à "zoo", mais pas à "z".
?	Correspond au caractère précédent zéro ou une fois. Par exemple, "a?ve?" correspond à "ve" dans "lever".
.	Correspond à tout caractère unique, sauf le caractère de nouvelle ligne.
(modèle**)**	Recherche le *modèle* et mémorise la correspondance. La sous-chaîne correspondante peut être extraite de la collection **Matches** obtenue, à l'aide d'Item **[0]...[n]**. Pour trouver des correspondances avec des caractères entre parenthèses ( ), utilisez "\(" ou "\)".
x|y	Correspond soit à *x* soit à *y*. Par exemple, "z|foot" correspond à "z" ou à "foot". "(z|f)oo" correspond à "zoo" ou à "foo".
{*n*}	*n* est un nombre entier non négatif. Correspond exactement à *n* fois le caractère. Par exemple, "o{2}" ne correspond pas à "o" dans "Bob," mais aux deux premiers "o" dans "foooooot".
{*n*,}	*n* est un entier non négatif. Correspond à au moins *n* fois le caractère. Par exemple, "o{2,}" ne correspond pas à "o" dans "Bob", mais à tous les "o" dans "foooooot". "o{1,}" équivaut à "o+" et "o{0,}" équivaut à "o*".
{*n*,*m***}**	*m* et *n* sont des entiers non négatifs. Correspond à au moins *n* et à au plus *m* fois le caractère. Par exemple, "o{1,3}" correspond aux trois premiers "o" dans "foooooot" et "o{0,1}" équivaut à "o?".
[*xyz***]**	Jeu de caractères. Correspond à l'un des caractères indiqués. Par exemple, "[abc]" correspond à "a" dans "plat".
[^*xyz***]**	Jeu de caractères négatif. Correspond à tout caractère non indiqué. Par exemple, "[^abc]" correspond à "p" dans "plat".
[*a-z***]**	Plage de caractères. Correspond à tout caractère dans la série spécifiée. Par exemple, "[a-z]" correspond à tout caractère alphabétique minuscule compris entre "a" et "z".
[^*m-z***]**	Plage de caractères négative. Correspond à tout caractère ne se trouvant pas dans la série spécifiée. Par exemple, "[^m-z]" correspond à tout caractère ne se trouvant pas entre "m" et "z".
\b	Correspond à une limite représentant un mot, autrement dit, à la position entre un mot et un espace. Par exemple, "er\b" correspond à "er" dans "lever", mais pas à "er" dans "verbe".
\B	Correspond à une limite ne représentant pas un mot. "en*t\B" correspond à "ent" dans "bien entendu".
\d	Correspond à un caractère représentant un chiffre. Équivaut à [0-9].
\D	Correspond à un caractère ne représentant pas un chiffre. Équivaut à [^0-9].
\f	Correspond à un caractère de saut de page.
\n	Correspond à un caractère de nouvelle ligne.
\r	Correspond à un caractère de retour chariot.
\s	Correspond à tout espace blanc, y compris l'espace, la tabulation, le saut de page, etc. Équivaut à "[\f\n\r\t\v]".
\S	Correspond à tout caractère d'espace non blanc. Équivaut à "[^ \f\n\r\t\v]".
\t	Correspond à un caractère de tabulation.
\v	Correspond à un caractère de tabulation verticale.
\w	Correspond à tout caractère représentant un mot et incluant un trait de soulignement. Équivaut à "[A-Za-z0-9_]".
\W	Correspond à tout caractère ne représentant pas un mot. Équivaut à "[^A-Za-z0-9_]".
****num	Correspond à *num*, où *num* est un entier positif. Fait référence aux correspondances mémorisées. Par exemple, "(.)\1" correspond à deux caractères identiques consécutifs.
*****n*	Correspond à *n*, où *n* est une valeur d'échappement octale. Les valeurs d'échappement octales doivent comprendre 1, 2 ou 3 chiffres. Par exemple, "\11" et "\011" correspondent tous les deux à un caractère de tabulation. "\0011" équivaut à "\001" & "1". Les valeurs d'échappement octales ne doivent pas excéder 256. Si c'était le cas, seuls les deux premiers chiffres seraient pris en compte dans l'expression. Permet d'utiliser les codes ASCII dans des expressions régulières.
\x*n*	Correspond à *n*, où *n* est une valeur d'échappement hexadécimale. Les valeurs d'échappement

hexadécimales doivent comprendre deux chiffres obligatoirement. Par exemple, "\x41" correspond à "A". "\x041" équivaut à "\x04" & "1". Permet d'utiliser les codes ASCII dans des expressions régulières.

Un élément dans un modèle peut être présent en 1 ou plusieurs exemplaires. Considérons quelques exemples autour du symbole **\d** qui représente 1 chiffre :

modèle	signification
\d	un chiffre
\d?	0 ou 1 chiffre
\d*	0 ou davantage de chiffres
\d+	1 ou davantage de chiffres
\d{2}	2 chiffres
\d{3,}	au moins 3 chiffres
\d{5,7}	entre 5 et 7 chiffres

Imaginons maintenant le modèle capable de décrire le format attendu pour une chaîne de caractères :

chaîne recherchée	modèle
une date au format jj/mm/aa	\d{2}/\d{2}/\d{2}
une heure au format hh:mm:ss	\d{2}:\d{2}:\d{2}
un nombre entier non signé	\d+
un suite d'espaces éventuellement vide	\s*
un nombre entier non signé qui peut être précédé ou suivi d'espaces	\s*\d+\s*
un nombre entier qui peut être signé et précédé ou suivi d'espaces	\s*[+|-]?\s*\d+\s*
un nombre réel non signé qui peut être précédé ou suivi d'espaces	\s*\d+(.\d*)?\s*
un nombre réel qui peut être signé et précédé ou suivi d'espaces	\s*[+|]?\s*\d+(.\d*)?\s*
une chaîne contenant le mot juste	\bjuste\b

On peut préciser où on recherche le modèle dans la chaîne :

modèle	signification
^modèle	le modèle commence la chaîne
modèle$	le modèle finit la chaîne
^modèle$	le modèle commence et finit la chaîne
modèle	le modèle est cherché partout dans la chaîne en commençant par le début de celle-ci.

chaîne recherchée	modèle
une chaîne se terminant par un point d'exclamation	!$
une chaîne se terminant par un point	\.$
une chaîne commençant par la séquence //	^//
une chaîne ne comportant qu'un mot éventuellement suivi ou précédé d'espaces	^\s*\w+\s*$
une chaîne ne comportant deux mot éventuellement suivis ou précédés d'espaces	^\s*\w+\s*\w+\s*$
une chaîne contenant le mot secret	\bsecret\b

Les sous-ensembles d'un modèle peuvent être "récupérés". Ainsi non seulement, on peut vérifier qu'une chaîne correspond à un modèle particulier mais on peut récupérer dans cette chaîne les éléments correspondant aux sous-ensembles du modèle qui **ont été entourés de parenthèses**. Ainsi si on analyse une chaîne contenant une date jj/mm/aa et si on veut de plus récupérer les éléments jj, mm, aa de cette date on utilisera le modèle (\d\d)/(\d\d)/(\d\d).

3.9.1 Vérifier qu'une chaîne correspond à un modèle donné

Un objet de type Regex se construit de la façon suivante :
// constructeur

```
public Regex(string pattern)
```

construit un objet "expression régulière" à partir d'un modèle passé en paramètre (pattern)

Une fois l'expression régulière modèle construit, on peut la comparer à des chaînes de caractères avec la méthode **IsMatch** :

```
public bool IsMatch(string input)
```

vrai si la chaîne input correspond au modèle de l'expression régulière

Voici un exemple :

```
// expression régulières

using System;
using System.Text.RegularExpressions;

public class regexp1{

  public static void Main(){
    // une expression régulière modèle
    string modèle1=@"^\s*\d+\s*$";
    Regex regex1=new Regex(modèle1);
    // comparer un exemplaire au modèle
    string exemplaire1="  123  ";
    if (regex1.IsMatch(exemplaire1)){
      affiche("["+exemplaire1 + "] correspond au modèle ["+modèle1+"]");
    }else{
      affiche("["+exemplaire1 + "] ne correspond pas au modèle ["+modèle1+"]");
    }//if
    string exemplaire2="  123a  ";
    if (regex1.IsMatch(exemplaire2)){
      affiche("["+exemplaire2 + "] correspond au modèle ["+modèle1+"]");
    }else{
      affiche("["+exemplaire2 + "] ne correspond pas au modèle ["+modèle1+"]");
    }//if
  }//Main

    public static void affiche(string msg){
      Console.Out.WriteLine(msg);
    }//affiche

}//classe
```

et les résultats d'exécution :

```
[  123  ] correspond au modèle [^\s*\d+\s*$]
[  123a  ] ne correspond pas au modèle [^\s*\d+\s*$]
```

3.9.2 Trouver tous les éléments d'une chaîne correspondant à un modèle

La méthode **Matches**

```
public System.Text.RegularExpressions.MatchCollection Matches(string input)
```

rend une collection d'éléments de la chaîne *input* correspondant au modèle comme le montre l'exemple suivant :

```
// expression régulières

using System;
using System.Text.RegularExpressions;

public class regexp1{
  public static void Main(){

    // plusieurs occurrences du modèle dans l'exemplaire
    string modèle2=@"\d+";
    Regex regex2=new Regex(modèle2);
    string exemplaire3="  123  456  789 ";
    MatchCollection résultats=regex2.Matches(exemplaire3);
    affiche("Modèle=["+modèle2+"],exemplaire=["+exemplaire3+"]");
    affiche("Il y a " + résultats.Count + " occurrences du modèle dans l'exemplaire ");
    for (int i=0;i<résultats.Count;i++){
      affiche(résultats[i].Value+" en position " + résultats[i].Index);
    }//for
  }//Main

    public static void affiche(string msg){
      Console.Out.WriteLine(msg);
    }//affiche

}//classe
```

La classe **MatchCollection** a une propriété **Count** qui est le nombre d'éléments de la collection. Si *résultats* est un objet *MatchCollection*, *résultats[i]* est l'élément i de cette collection et est de type **Match**. Dans notre exemple, *résultats* est l'ensemble des éléments de la chaîne *exemplaire3* correspondant au modèle *modèle2* et *résultats[i]* l'un de ces éléments. La classe *Match* a deux propriétés qui nous intéressent ici :

- **Value** : la valeur de l'objet Match donc l'élément correspondant au modèle
- **Index** : la position où l'élément a été trouvé dans la chaîne explorée

Les résultats de l'exécution du programme précédent :

```
Modèle=[\d+],exemplaire=[  123  456  789 ]
Il y a 3 occurrences du modèle dans l'exemplaire
123 en position 2
456 en position 7
789 en position 12
```

3.9.3 Récupérer des parties d'un modèle

Des sous-ensembles d'un modèle peuvent être "récupérés". Ainsi non seulement, on peut vérifier qu'une chaîne correspond à un modèle particulier mais on peut récupérer dans cette chaîne les éléments correspondant aux sous-ensembles du modèle qui **ont été entourés de parenthèses**. Ainsi si on analyse une chaîne contenant une date jj/mm/aa et si on veut de plus récupérer les éléments jj, mm, aa de cette date on utilisera le modèle (\d\d)/(\d\d)/(\d\d).

Examinons l'exemple suivant :

```
// expression régulières
```

```
using System;
using System.Text.RegularExpressions;

public class regexp1{
  public static void Main(){
    // capture d'éléments dans le modèle
    string modèle3=@"(\d\d):(\d\d):(\d\d)";
    Regex regex3=new Regex(modèle3);
    string exemplaire4="Il est 18:05:49";
    // vérification modèle
    Match résultat=regex3.Match(exemplaire4);
    if (résultat.Success){
      // l'exemplaire correspond au modèle
      affiche("L'exemplaire ["+exemplaire4+"] correspond au modèle ["+modèle3+"]");
      // on affiche les groupes
      for (int i=0;i<résultat.Groups.Count;i++){
        affiche("groupes["+i+"]=["+résultat.Groups[i].Value+"] en position "+résultat.Groups[i].Index);
      }//for
    }else{
      // l'exemplaire ne correspond pas au modèle
      affiche("L'exemplaire["+exemplaire4+" ne correspond pas au modèle ["+modèle3+"]");
    }
  }//Main

    public static void affiche(string msg){
      Console.Out.WriteLine(msg);
    }//affiche
}//classe
```

L'exécution de ce programme produit les résultats suivants :

```
L'exemplaire [Il est 18:05:49] correspond au modèle [(\d\d):(\d\d):(\d\d)]
groupes[0]=[18:05:49] en position 7
groupes[1]=[18] en position 7
groupes[2]=[05] en position 10
groupes[3]=[49] en position 13
```

La nouveauté se trouve dans la partie de code suivante :

```
    // vérification modèle
    Match résultat=regex3.Match(exemplaire4);
    if (résultat.Success){
      // l'exemplaire correspond au modèle
      affiche("L'exemplaire ["+exemplaire4+"] correspond au modèle ["+modèle3+"]");
      // on affiche les groupes
      for (int i=0;i<résultat.Groups.Count;i++){
        affiche("groupes["+i+"]=["+résultat.Groups[i].Value+"] en position "+résultat.Groups[i].Index);
      }//for
    }else{
```

La chaîne *exemplaire4* est comparée au modèle *regex3* au travers de la méthode *Match*. Celle-ci rend un objet *Match* déjà présenté. Nous utilisons ici deux nouvelles propriétés de cette classe :

- **Success** : indique s'il y a eu correspondance
- **Groups** : collection où
  - Groups[0] correspond à la partie de la chaîne correspondant au modèle
  - Groups[i] (i>=1) correspond au groupe de parenthèses n° i

Si résultat est de type *Match*, *résultats.Groups* est de type *GroupCollection* et *résultats.Groups[i]* de type *Group*. La classe *Group* a deux propriétés que nous utilisons ici :

- **Value** : la valeur de l'objet Group qui l'élément correspondant au contenu d'une paraenthèse
- **Index** : la position où l'élément a été trouvé dans la chaîne explorée

3.9.4 Un programme d'apprentissage

Trouver l'expression régulière qui nous permet de vérifier qu'une chaîne correspond bien à un certain modèle est parfois un véritable défi. Le programme suivant permet de s'entraîner. Il demande un modèle et une chaîne et indique alors si la chaîne correspond ou non au modèle.

```
// expression régulières

using System;
using System.Text.RegularExpressions;

public class regexp1{
  public static void Main(){
    // une expression régulière modèle
    string modèle,chaine;
    Regex regex=null;
    MatchCollection résultats;
    // on demande à l'utilisateur les modèles et les exemplaires à comparer à celui-ci
    while(true){
      // on demande le modèle
      Console.Out.Write("Tapez le modèle à tester ou fin pour arrêter :");
      modèle=Console.In.ReadLine();
      // fini ?
      if(modèle.Trim().ToLower()=="fin") break;
      // on crée l'exepression régulière
      try{
        regex=new Regex(modèle);
      }catch(Exception ex){
        Console.Error.WriteLine("Erreur : "+ex.Message);
        continue;
      }//tryècatch
      // on demande à l'utilisateur les exemplaires à comparer au modèle
      while(true){
        Console.Out.Write("Tapez la chaîne à comparer au modèle ["+modèle+"] ou fin pour arrêter :");
        chaine=Console.In.ReadLine();
        // fini ?
        if(chaine.Trim().ToLower()=="fin") break;
        // on fait la comparaison
        résultats=regex.Matches(chaine);
        // succès ?
        if(résultats.Count==0){
          Console.Out.WriteLine("Je n'ai pas trouvé de correspondances");
          continue;
        }//if
        // on affiche les éléments correspondant au modèle
        for(int i=0;i<résultats.Count;i++){
          Console.Out.WriteLine("J'ai trouvé la correspondance ["+résultats[i].Value
            +"] en position "+résultats[i].Index);
          // des sous-éléments
          if(résultats[i].Groups.Count!=1){
            for(int j=1;j<résultats[i].Groups.Count;j++){
              Console.Out.WriteLine("\tsous-élément ["+résultats[i].Groups[j].Value+"] en position "+
                résultats[i].Groups[j].Index);
            }//for j
          }//if
        }//for i
        // chaîne suivante
      }//while
      // modèle suivant
    }//while
  }//Main

    public static void affiche(string msg){
      Console.Out.WriteLine(msg);
    }//affiche
}//classe
```

Voici un exemple d'exécution :

```
Tapez le modèle à tester ou fin pour arrêter :\d+
Tapez la chaîne à comparer au modèle [\d+] ou fin pour arrêter :123 456 789
J'ai trouvé la correspondance [123] en position 0
J'ai trouvé la correspondance [456] en position 4
J'ai trouvé la correspondance [789] en position 8
Tapez la chaîne à comparer au modèle [\d+] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :(\d\d):(\d\d)
Tapez la chaîne à comparer au modèle [(\d\d):(\d\d)] ou fin pour arrêter :14:15
abcd 17:18 xyzt
```

```
J'ai trouvé la correspondance [14:15] en position 0
        sous-élément [14] en position 0
        sous-élément [15] en position 3
J'ai trouvé la correspondance [17:18] en position 11
        sous-élément [17] en position 11
        sous-élément [18] en position 14
Tapez la chaîne à comparer au modèle [(\d\d):(\d\d)] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :^\s*\d+\s*$
Tapez la chaîne à comparer au modèle [^\s*\d+\s*$] ou fin pour arrêter :  1456
J'ai trouvé la correspondance [  1456] en position 0
Tapez la chaîne à comparer au modèle [^\s*\d+\s*$] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :^\s*(\d+)\s*$
Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou fin pour arrêter :1456
J'ai trouvé la correspondance [1456] en position 0
        sous-élément [1456] en position 0
Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou fin pour arrêter :abcd 1
456
Je n'ai pas trouvé de correspondances
Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :fin
```

3.9.5 La méthode Split

Nous avons déjà rencontré cette méthode dans la classe *String* :

```
public string[] Split(char[] separator);
      // la chaîne est vue comme une suite de champs séparés par les caractères présents dans le tableau
      separator. Le résultat est le tableau de ces champs
```

Le séparateur de champs de lachaîne est ici un des caractères du tableau *separator*. La méthode *Split* de la classe *Regex* nous permet d'exprimer le séparateur en fonction d'un modèle :

```
public string[] Split(string input)
```

La chaîne input est décomposée en champs, ceux-ci étant séparés par un séparateur correspondant au modèle de l'objet *Regex* courant.

Supposons par exemple que vous ayez dans un fichier texte des lignes de la forme *champ1*, *champ2*, .., *champn*. Les champs sont séparés par une virgule mais celle-ci peut être précédée ou suivie d'espaces. La méthode *Split* de la classe *string* ne convient alors pas. Celle de la méthode *RegEx* apporte la solution. Si ligne est la ligne lue, les champs pourront être obtenus par

```
string[] champs=new Regex("s*,\s*").Split(ligne);
```

comme le montre l'exemple suivant :

```
using System;
using System.Text.RegularExpressions;

public class split1{
  public static void Main(){
    // une ligne
    string ligne="abc  , def  , ghi";
    // un modèle
    Regex modèle=new Regex(@"\s*,\s*");
    // décomposition de ligne en champs
    string[] champs=modèle.Split(ligne);
    // affichage
    for(int i=0;i<champs.Length;i++){
      Console.Out.WriteLine("champs["+i+"]=["+champs[i]+"]");
    }//for
  }//Main
}//classe
```

Les résultats d'exécution :

```
champs[0]=[abc]
champs[1]=[def]
champs[2]=[ghi]
```

3.10 Les classes BinaryReader et BinaryWriter

Les classes **BinaryReader** et **BinaryWriter** servent à lire et écrire des fichiers binaires. Considérons l'application suivante. On veut écrire un programme qui s'appellerait de la façon suivante :

```
// syntaxe pg texte bin
// on lit un fichier texte (texte) et on range son contenu dans un
// fichier binaire
// le fichier texte a des lignes de la forme nom : age
// qu'on rangera dans une structure string, int
```

Le fichier texte a le contenu suivant :

```
paul : 10
helene : 15
jacques : 11
sylvain : 12
```

Le programme est le suivant :

```
// BinaryWriter

using System;
using System.IO;
using System.Text.RegularExpressions;

// syntaxe pg texte bin
// on lit un fichier texte (texte) et on range son contenu dans un
// fichier binaire
// le fichier texte a des lignes de la forme nom : age
// qu'on rangera dans une structure string, int

public class writer1{
  public static void Main(string[] arguments){
    // il faut 2 arguments
    int nbArgs=arguments.Length;
    if(nbArgs!=2){
      Console.Error.WriteLine("syntaxe : pg texte binaire");
      Environment.Exit(1);
    }//if
    // ouverture du fichier texte en lecture
    StreamReader input=null;
    try{
      input=new StreamReader(arguments[0]);
    }catch(Exception){
      Console.Error.WriteLine("Impossible d'ouvrir le fichier ["+arguments[0]+"] en lecture");
      Environment.Exit(2);
    }//try-catch
    // ouverture du fichier binaire en écriture
    BinaryWriter output=null;
    try{
      output=new BinaryWriter(new FileStream(arguments[1],FileMode.Create,FileAccess.Write));
    }catch(Exception){
      Console.Error.WriteLine("Impossible d'ouvrir le fichier ["+arguments[1]+"] en écriture");
      Environment.Exit(3);
    }//try-catch

    // lecture fichier texte - écriture fichier binaire
    // ligne du fichier texte
    string ligne;
    // le séparateur des champs de la ligne
    Regex séparateur=new Regex(@"\s*:\s*");
    // le modèle de l'âge
    Regex modAge=new Regex(@"\s*\d+\s*");
    // n° de ligne
    int numLigne=0;
    while((ligne=input.ReadLine())!=null){
      // ligne vide ?
      if (ligne.Trim()=="") continue;
      // une ligne de plus
      numLigne++;
      // une ligne nom : age
      string[] champs=séparateur.Split(ligne);
      // il nous faut 2 champs
      if (champs.Length!=2){
        Console.Error.WriteLine("La ligne n° " + numLigne + " du fichier "+ arguments[0]
          + " a un nombre de champs incorrect");
        // ligne suivante
        continue;
      }//if
      // le second champ doit être un entier >=0
      if (! modAge.IsMatch(champs[1])){
        Console.Error.WriteLine("La ligne n° " + numLigne + " du fichier "+ arguments[0]
          + " a un âge incorrect");
```

```
        // ligne suivante
        continue;
      }//if
      // on écrit les données dans le fichier binaire
      output.Write(champs[0]);
      output.Write(int.Parse(champs[1]));
      // ligne suivante
    }//while
    // fermeture des fichiers
    input.Close();
    output.Close();
  }//main
}//classe
```

Attardons-nous sur les opérations concernant la classe *BinaryWriter* :

- l'objet *BinaryWriter* est ouvert par l'opération

```
      output=new BinaryWriter(new FileStream(arguments[1],FileMode.Create,FileAccess.Write));
```

  L'argument du constructeur doit être un flux (Stream). Ici c'est un flux construit à partir d'un fichier (FileStream) dont on donne :
  - le nom
  - l'opération à faire, ici *FileMode.Create* pour créer le fichier
  - le type d'accès, ici *FileAccess.Write* pour un accès en écriture au fichier

- l'opération d'écriture

```
      // on écrit les données dans le fichier binaire
      output.Write(champs[0]);
      output.Write(int.Parse(champs[1]));
```

  La classe *BinaryWriter* dispose de différentes méthodes *Write* surchargées pour écrire les différents types de données simples

- l'opération de fermeture du flux

```
      output.Close();
```

Les résultats de l'exécution précédente vont nous être donnés par le programme qui suit. Celui-ci accepte également deux arguments :

```
// syntaxe pg bin texte
// on lit un fichier binaire bin et on range son contenu dans un fichier texte
(texte)
// le fichier binaire a une structure string, int
// le fichier texte a des lignes de la forme nom : age
```

On fait donc l'opération inverse. On lit un fichier binaire pour créer un fichier texte. Si le fichier texte produit est identique au fichier originel on saura que la conversion texte --> binaire --> texte s'est bien passée. Le code est le suivant :

```
// BinaryReader

using System;
using System.IO;

// syntaxe pg bin texte
// on lit un fichier binaire bin et on range son contenu dans un fichier texte (texte)
// le fichier binaire a une structure string, int
// le fichier texte a des lignes de la forme nom : age

public class reader1{
  public static void Main(string[] arguments){
    // il faut 2 arguments
    int nbArgs=arguments.Length;
    if(nbArgs!=2){
      Console.Error.WriteLine("syntaxe : pg binaire texte");
      Environment.Exit(1);
    }//if

    // ouverture du fichier binaire en lecture
    BinaryReader input=null;
    try{
      input=new BinaryReader(new FileStream(arguments[0],FileMode.Open,FileAccess.Read));
    }catch(Exception){
      Console.Error.WriteLine("Impossible d'ouvrir le fichier ["+arguments[0]+"] en lecture");
      Environment.Exit(2);
    }//try-catch

    // ouverture du fichier texte en écriture
    StreamWriter output=null;
    try{
```

```
      output=new StreamWriter(arguments[1]);
    }catch(Exception){
      Console.Error.WriteLine("Impossible d'ouvrir le fichier ["+arguments[1]+"] en écriture");
      Environment.Exit(3);
    }//try-catch

    // lecture fichier binaire - écriture fichier texte
    string nom;  // nom d'une personne
    int age;   // son âge
    // boucle d'exploitation du fichier binaire
    while(true){
      // lecture nom
      try{
        nom=input.ReadString();
      }catch(Exception){
        // fin du fichier
        break;
      }//try-catch
      // lecture age
      try{
        age=input.ReadInt32();
      }catch(Exception){
        Console.Error.WriteLine("Le fichier " + arguments[0] +
          " ne semble pas avoir un format correct");
        break;
      }//try-catch
      // écriture dans fichier texte
      output.WriteLine(nom+":"+age);
      // personne suivante
    }// while
    // on ferme tout
    input.Close();
    output.Close();
  }//Main
}//classe
```

Attardons-nous sur les opérations concernant la classe *BinaryReader* :

- l'objet BinaryReader est ouvert par l'opération

```
      input=new BinaryReader(new FileStream(arguments[0],FileMode.Open,FileAccess.Read));
```

  L'argument du constructeur doit être un flux (Stream). Ici c'est un flux construit à partir d'un fichier (FileStream) dont on donne :
  - le nom
  - l'opération à faire, ici *FileMode.Open* pour ouvrir un fichier existant
  - le type d'accès, ici *FileAccess.Read* pour un accès en lecture au fichier

- l'opération de lecture

```
      nom=input.ReadString();
      age=input.ReadInt32();
```

  La classe *BinaryReader* dispose de différentes méthodes *ReadXX* pour lire les différents types de données simples

- l'opération de fermeture du flux

```
      input.Close();
```

Si on exécute les deux programmes à la chaîne transformant *personnes.txt* en *personnes.bin* puis *personnes.bin* en *personnes.txt2* on a :

```
E:\data\serge\MSNET\c#\fichiers\BINARY~1\1>more personnes.txt
paul : 10
helene : 15
jacques : 11
sylvain : 12

E:\data\serge\MSNET\c#\fichiers\BINARY~1\2>more personnes.txt2
paul:10
helene:15
jacques:11
sylvain:12

E:\data\serge\MSNET\c#\fichiers\BINARY~1\2>dir
29/04/2002  18:19                   54 personnes.txt
29/04/2002  18:19                   44 personnes.bin
29/04/2002  18:20                   44 personnes.txt2
```

4. Interfaces graphiques avec C# et VS.NET

Nous nous proposons ici de montrer comment construire des interfaces graphiques avec C#. Nous voyons tout d'abord quelles sont les classes de base de la plate-forme .NET qui nous permettent de construire une interface graphique. Nous n'utilisons dans un premier temps aucun outil de génération automatique. Puis nous utiliserons Visual Studio.NET (VS.NET), un outil de développement de Microsoft facilitant le développement d'applications avec les langages .NET et notamment la construction d'interfaces graphiques. La version VS.NET utilisée est la version anglaise.

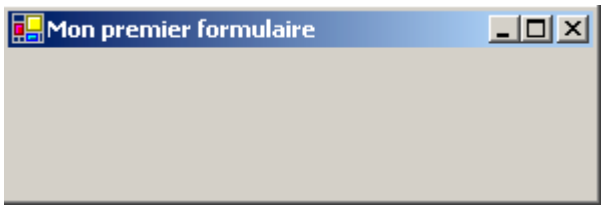
4.1 Les bases des interfaces graphiques

4.1.1 Une fenêtre simple

Considérons le code suivant :

```
using System;
using System.Drawing;
using System.Windows.Forms;

// la classe formulaire
public class Form1 : Form {

  // le constructeur
  public Form1() {
    // titre de la fenêtre
    this.Text = "Mon premier formulaire";
    // dimensions de la fenêtre
    this.Size=new System.Drawing.Size(300,100);
  }//constructeur

  // fonction de test
  public static void Main(string[] args) {
    // on affiche le formulaire
    Application.Run(new Form1());
  }
}//classe
```

L'exécution du code précédent affiche la fenêtre suivante :



Une interface graphique dérive en général de la classe de base *System.Windows.Forms.Form* :

```
public class Form1 : System.Windows.Forms.Form
```

La classe de base *Form* définit une fenêtre de base avec des boutons de fermeture, agrandissement/réduction, une taille ajustable, ... et gère les événements sur ces objets graphiques. Ici nous spécialisons la classe de base en lui fixant un titre et ses largeur (300) et hauteur (100). Ceci est fait dans son constructeur :

```
  public Form1() {
    // titre de la fenêtre
    this.Text = "Mon premier formulaire";
    // dimensions de la fenêtre
    this.Size=new System.Drawing.Size(300,100);
  }//constructeur
```

Le titre de la fenêtre est fixée par la propriété *Text* et les dimensions par la propriété *Size*. *Size* est défini dans l'espace de noms *System.Drawing* et est une structure.

La fonction Main lance l'application graphique de la façon suivante :

```
    Application.Run(new Form1());
```

Un formulaire de type *Form1* est créé et affiché, puis l'application se met à l'écoute des événements qui se produisent sur le formulaire (clics, déplacements de souris, ...) et fait exécuter ceux que le formulaire gère. Ici, notre formulaire ne gère pas d'autre événement que ceux gérés par la classe de base *Form* (clics sur boutons fermeture, agrandissement/réduction, changement de taille de la fenêtre, déplacement de la fenêtre, ...).

4.1.2 Un formulaire avec bouton

Ajoutons maintenant un bouton à notre fenêtre :

```
    using System;
    using System.Drawing;
    using System.Windows.Forms;

    // la classe formulaire
    public class Form1 : Form {
      // attributs
      Button cmdTest;

      // le constructeur
      public Form1() {
        // le titre
        this.Text = "Mon premier formulaire";
        // les dimensions
        this.Size=new System.Drawing.Size(300,100);
        // un bouton
          // création
        this.cmdTest = new Button();
          // position
        cmdTest.Location = new System.Drawing.Point(110, 20);
          // taille
        cmdTest.Size = new System.Drawing.Size(80, 30);
          // libellé
        cmdTest.Text = "Test";
          // gestionnaire d'évt
        cmdTest.Click += new System.EventHandler(cmdTest_Click);
        // ajout bouton au formulaire
        this.Controls.Add(cmdTest);
      }//constructeur

      // gestionnaire d'événement
      private void cmdTest_Click(object sender, EventArgs evt){
        // il y a eu un clic sur le bouton - on le dit
        MessageBox.Show("Clic sur bouton", "Clic sur
bouton",MessageBoxButtons.OK,MessageBoxIcon.Information);
      }//cmdTest_Click

      // fonction de test
      public static void Main(string[] args) {
        // on affiche le formulaire
        Application.Run(new Form1());
      }
    }//classe
```

Nous avons rajouté au formulaire un bouton :

```
          // création
        this.cmdTest = new Button();
          // position
        cmdTest.Location = new System.Drawing.Point(110, 20);
          // taille
        cmdTest.Size = new System.Drawing.Size(80, 30);
          // libellé
        cmdTest.Text = "Test";
          // gestionnaire d'évt
        cmdTest.Click += new System.EventHandler(cmdTest_Click);
        // ajout bouton au formulaire
        this.Controls.Add(cmdTest);
```

La propriété *Location* fixe les coordonnées (110,20) du point supérieur gauche du bouton à l'aide d'une structure *Point*. Les largeur et hauteur du bouton sont fixées à (80,30) à l'aide d'une structure *Size*. La propriété *Text* du bouton permet de fixer le libellé du bouton. La classe bouton possède un événement Click défini comme suit :

```
    public event EventHandler Click;
```

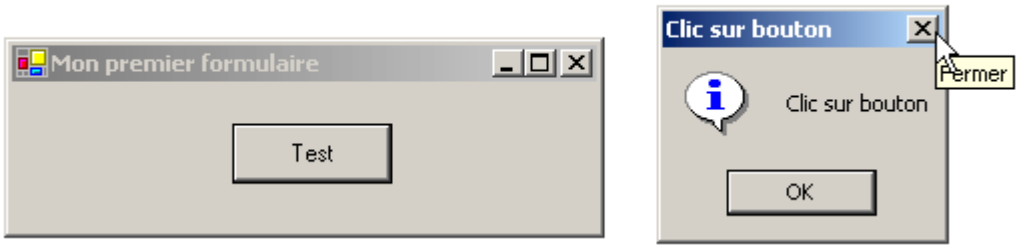
Un objet *EventHandler* est construit avec le nom d'une méthode f ayant la signature suivante :

```
public delegate void EventHandler(
   object sender,
   EventArgs e
);
```

Ici, lors d'un clic sur le bouton *cmdTest*, la méthode *cmdTest_Click* sera appelée. Celle-ci est définie comme suit conformément au modèle *EventHandler* précédent :

```
      // gestionnaire d'événement
      private void cmdTest_Click(object sender, EventArgs evt){
        // il y a eu un clic sur le bouton - on le dit
        MessageBox.Show("Clic sur bouton", "Clic sur bouton", MessageBoxButtons.OK,
MessageBoxIcon.Information);
      }//cmdTest_Click
```

On se contente d'afficher un message :



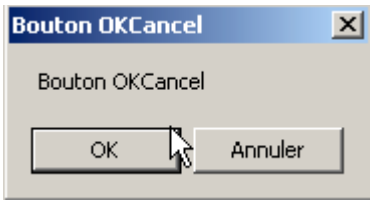
La classe *MessageBox* sert à afficher des messages dans une fenêtre. Nous avons utilisé ici le constructeur

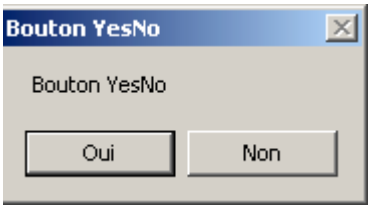
```
    public static System.Windows.Forms.DialogResult Show(string text, string caption,
System.Windows.Forms.MessageBoxButtons buttons, System.Windows.Forms.MessageBoxIcon icon);
```

avec

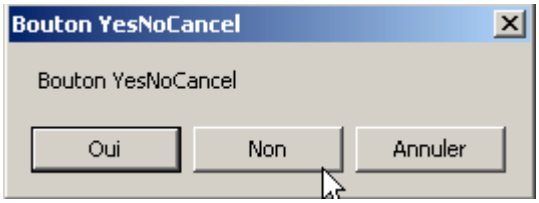
text	le message à afficher
caption	le titre de la fenêtre
buttons	les boutons présents dans la fenêtre
icon	l'icone présente dans la fenêtre

Le paramètre *buttons* peut prendre ses valeurs parmi les constantes suivantes :

constante	boutons
AbortRetryIgnore	
OK	
OKCancel	
RetryCancel	
YesNo	

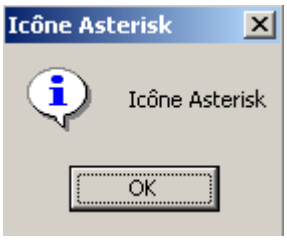


YesNoCancel



Le paramètre *icon* peut prendre ses valeurs parmi les constantes suivantes :

Asterisk



Error idem Stop

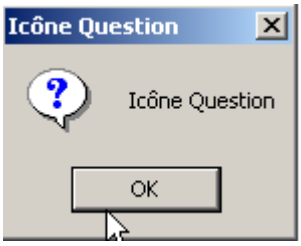
Exclamation idem Warning

Hand

Information idem Asterisk

None

Question



Stop idem Hand

Warning

La méthode *Show* est une méthode statique qui rend un résultat de type `System.Windows.Forms.DialogResult` qui est une énumération :

```
// from module
'c:\winnt\assembly\gac\system.windows.forms\1.0.2411.0__b77a5c561934e089\system.windows.forms.dll'
public enum System.Windows.Forms.DialogResult
```

```
{
    Abort = 0x00000003,
    Cancel = 0x00000002,
    Ignore = 0x00000005,
    No = 0x00000007,
    None = 0x00000000,
    OK = 0x00000001,
    Retry = 0x00000004,
    Yes = 0x00000006,
} // end of System.Windows.Forms.DialogResult
```

Pour savoir sur quel bouton a appuyé l'utilisateur pour fermer la fenêtre de type *MessageBox* on écrira :

DialogResult res=MessageBox.Show(..);
if (res==DialogResult.Yes){ // il a appuyé sur le bouton oui...}

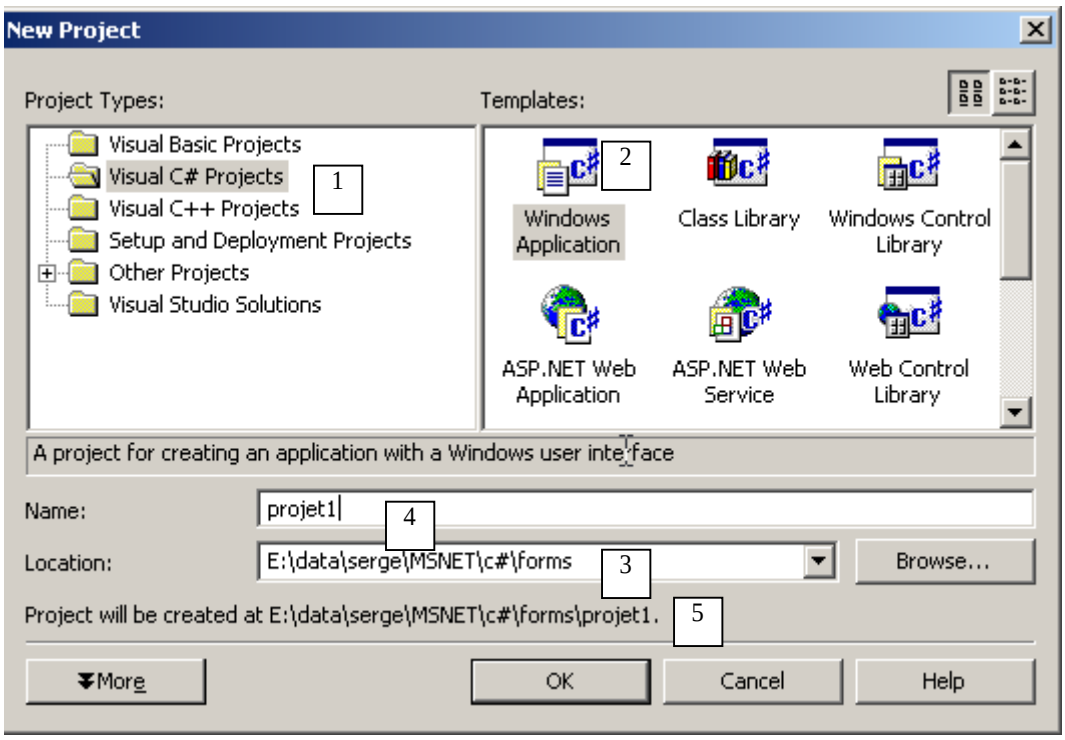
4.2 Construire une interface graphique avec Visual Studio.NET

Nous reprenons certains des exemples vus précédemment en les construisant maintenant avec Visual Studio.NET.

4.2.1 Création initiale du projet

1. Lancez VS.NET et prendre l'option *File/New/Project*

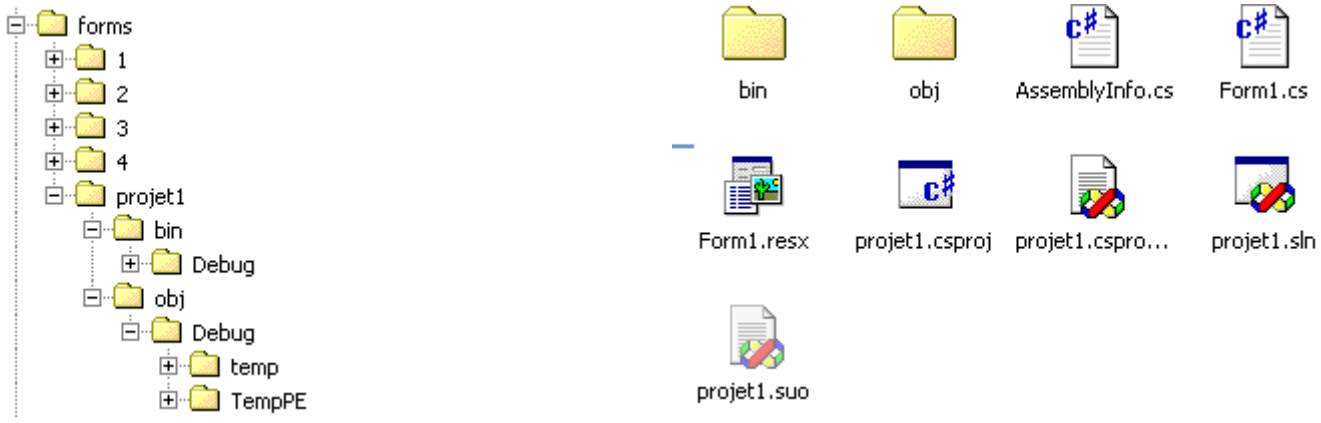
2. donnez les caractéristiques de votre projet



- sélectionnez le type de projet que vous voulez construire, ici un projet C# (1)
- sélectionnez le type d'application que vous voulez construire, ici une application windows (2)
- indiquez dans quel dossier vous voulez placer le sous-dossier du projet (3)
- indiquez le nom du projet (4). Ce sera également le nom du dossier qui contiendra les fichiers du projet
- le nom de ce dossier est rappelé en (5)

3. Un certain nombre de dossiers et de fichiers sont alors créés sous le dossier *projet1* :

sous-dossiers du dossier projet1 **fichiers du dossier projet1**

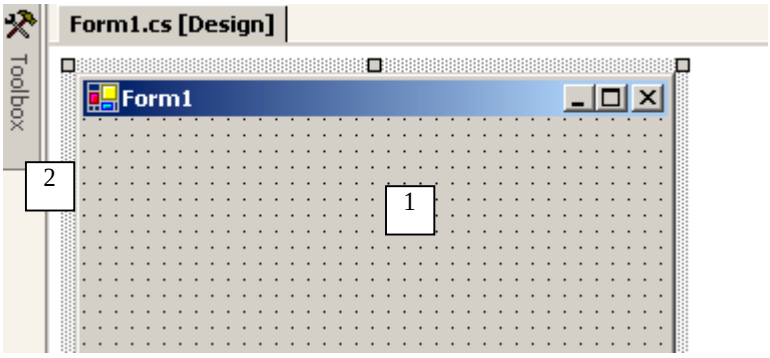


De ces fichiers, seul un est intéressant, le fichier *form1.cs* qui est le fichier source associé au formulaire créé par VS.NET. Nous y reviendrons.

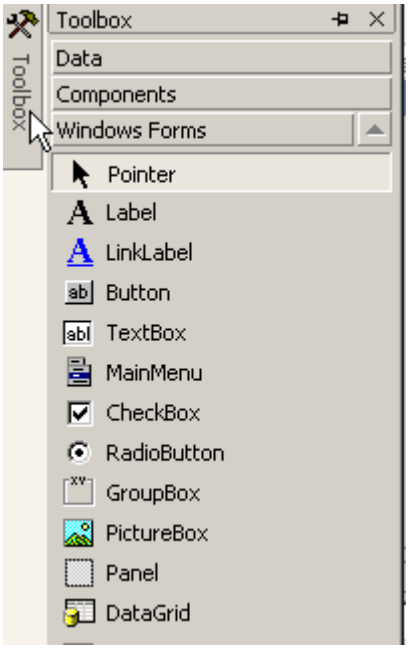
4.2.2 Les fenêtre de l'interface de VS.NET

L'interface de VS.NET laisse maintenant apparaître certains éléments de notre projet *projet1* :

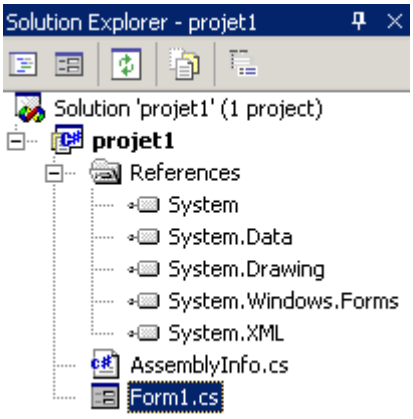
Nous avons une fenêtre de conception de l'interface graphique :



En prenant des contrôles dans la barre d'outils (toolbox 2) et en les déposant sur la surface de la fenêtre (1), nous pouvons construire une interface graphique. Si nos amenons la souris sur la "toolbox" celle-ci s'agrandit et laisse apparaître un certain nombre de contrôles :

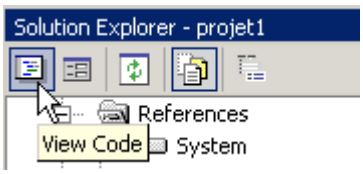


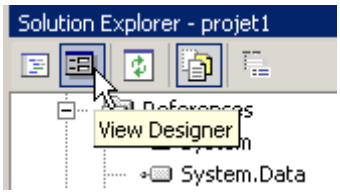
Pour l'instant, nous n'en utilisons aucun. Toujours sur l'écran de VS.NET, nous trouvons la fenêtre de l'explorateur de solutions "*Explorer Solution*" :



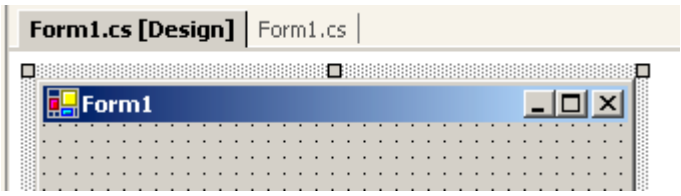
Dans un premier temps, nous ne nous servirons peu de cette fenêtre. Elle montre l'ensemble des fichiers formant le projet. Un seul d'entre-eux nous intéresse : le fichier source de notre programme, ici *Form1.cs*. En cliquant droit sur Form1.cs, on obtient un menu permettant d'accéder soit au code source de notre interface graphique (*View code*) soit à l'interface graphique elle-même (*View Designer*) :

On peut accèder à ces deux entités directement à partir de la fenêtre "*Solution Explorer*" :



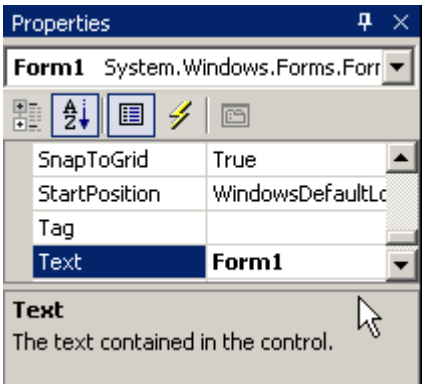


Les fenêtres ouvertes "s'accumulent" dans la fenêtre principale de conception :



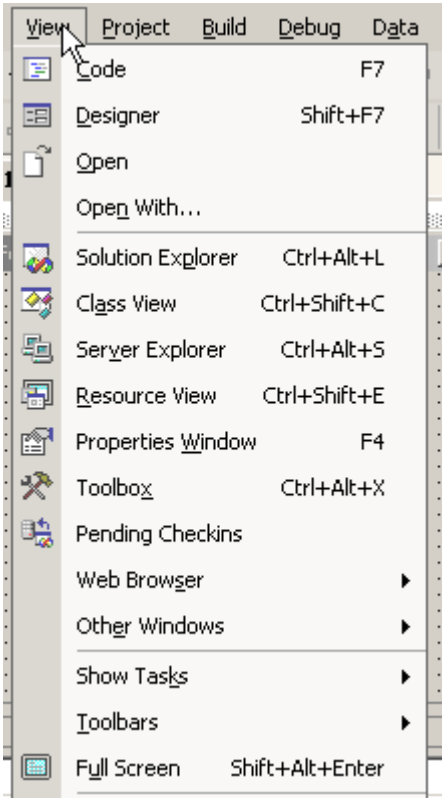
Ici *Form1.cs[Design]* désigne la fenêtre de conception et *Form1.cs* la fenêtre de code. Il suffit de cliquer sur l'un des onglets pour passer d'une fenêtre à l'autre.

Une autre fenêtre importante présente sur l'écran de VS.NET est la fenêtre des propriétés :



Les propriétés exposées dans la fenêtre sont celles du contrôle actuellement sélectionné dans la fenêtre de conception graphique.

On a accès à différentes fenêtres du projet avec le menu View :



On y retrouve les fenêtres principales qui viennent d'être décrites ainsi que leurs raccourcis clavier.

4.2.3 Exécution d'un projet

Alors même que nous n'avons écrit aucun code, nous avons un projet exécutable. Faites *F5* ou *Debug/Start* pour l'exécuter. Nous obtenons la fenêtre suivante :

Cette fenêtre peut être agrandie, mise en icône, redimensionnée et fermée.

4.2.4 Le code généré par VS.NET

Regardons le code (*View/Code*) de notre application :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace projet1
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
```

```
  {
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      // TODO: Add any constructor code after InitializeComponent call
      //
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.Size = new System.Drawing.Size(300,300);
      this.Text = "Form1";
    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

Une interface graphique dérive de la classe de base *System.Windows.Forms.Form* :

```
  public class Form1 : System.Windows.Forms.Form
```

La classe de base *Form* définit une fenêtre de base avec des boutons de fermeture, agrandissement/réduction, une taille ajustable, ... et gère les événements sur ces objets graphiques.

Le constructeur du formulaire utilise une méthode *InitializeComponent* dans laquelle les contrôles du formulaires sont créés et initialisés.

```
    public Form1()
    {
      InitializeComponent();
      // autres initialisations
    }
```

Tout autre travail à faire dans le constructeur peut être fait après l'appel à *InitializeComponent*. La méthode *InitializeComponent*

```
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.Size = new System.Drawing.Size(300,300);
      this.Text = "Form1";
    }
```

fixe le titre de la fenêtre "Form1", sa largeur (300) et sa hauteur (300). Le titre de la fenêtre est fixée par la propriété *Text* et les dimensions par la propriété *Size*. *Size* est défini dans l'espace de noms *System.Drawing* et est une structure.

La fonction *Main* lance l'application graphique de la façon suivante :

```
        Application.Run(new Form1());
```

Un formulaire de type *Form1* est créé et affiché, puis l'application se met à l'écoute des événements qui se produisent sur le formulaire (clics, déplacements de souris, ...) et fait exécuter ceux que le formulaire gère. Ici, notre formulaire ne gère pas d'autre événement que ceux gérés par la classe de base *Form* (clics sur boutons fermeture, agrandissement/réduction, changement de taille de la fenêtre, déplacement de la fenêtre, ...).

Le formulaire utilise un attribut *components* qui n'est utilisé nulle part. La méthode *dispose* ne sert également à rien ici. Il en de même de certains espaces de noms (*Collections*, *ComponentModel*, *Data*) utilisés et de celui défini pour le projet *projet1*. Aussi, dans cet exemple le code peut être simplifié à ce qui suit :

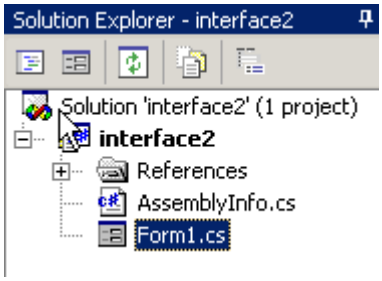
```
using System;
using System.Drawing;
using System.Windows.Forms;

  public class Form1 : System.Windows.Forms.Form
  {

    // constructeur
    public Form1()
    {
      // construction du formulaire avec ses composants
      InitializeComponent();
      // autres initialisations
    }//constructeur


    private void InitializeComponent()
    {
      // taille de la fenêtre
      this.Size = new System.Drawing.Size(300,300);
      // titre de la fenêtre
      this.Text = "Form1";
    }

    static void Main()
    {
      // on lance l'appli
      Application.Run(new Form1());
    }
  }
```

4.2.5 Conclusion

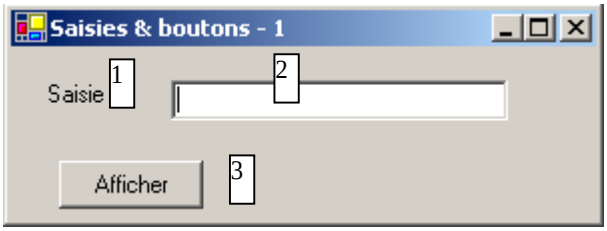
Nous accepterons maintenant tel quel le code généré par VS.NET et nous contenterons d'y ajouter le nôtre notamment pour gérer les événements liés aux différents contrôles du formulaire.

4.3 Fenêtre avec champ de saisie, bouton et libellé

Dans l'exemple précédent, nous n'avions pas mis de composants dans la fenêtre. Nous commençons un nouveau projet appelé *interface2*. Pour cela nous suivons la procédure explicitée précédemment pour créer un projet :



Construisons maintenant une fenêtre avec un bouton, un libellé et un champ de saisie :



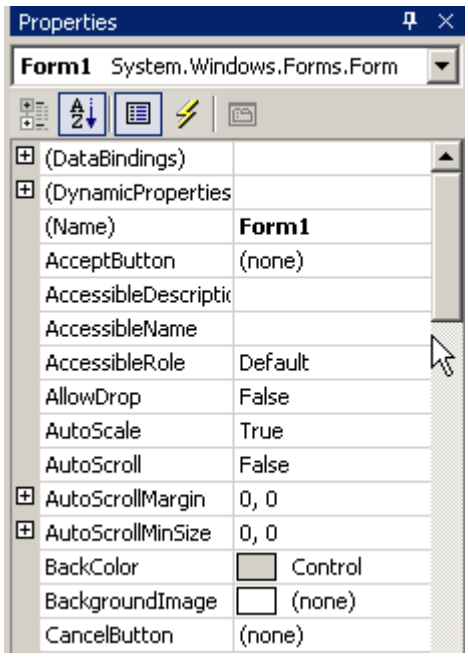
Les champs sont les suivants :

n°	nom	type	rôle
1	lblSaisie	Label	un libellé
2	txtSaisie	TextBox	une zone de saisie
3	cmdAfficher	Button	pour afficher dans une boîte de dialogue le contenu de la zone de saisie txtSaisie

On pourra procéder comme suit pour construire cette fenêtre :

Cliquez droit dans la fenêtre en-dehors de tout composant et choisissez l'option *Properties* pour avoir accès aux propriétés de la fenêtre :

La fenêtre de propriétés apparaît alors sur la droite :



Certaines de ces propriétés sont à noter :

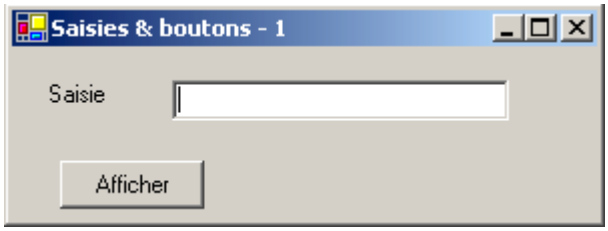
BackColor pour fixer la couleur de fond de la fenêtre
ForeColor pour fixer la couleur des dessins ou du texte sur la fenêtre
Menu pour associer un menu à la fenêtre
Text pour donner un titre à la fenêtre
FormBorderStyle pour fixer le type de fenêtre
Font pour fixer la police de caractères des écritures dans la fenêtre
Name pour fixer le nom de la fenêtre

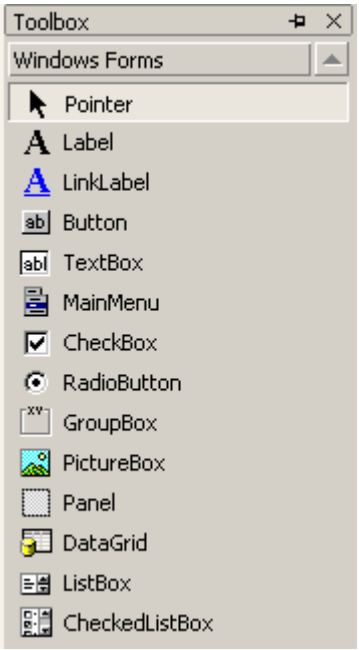
Ici, nous fixons les propriétés *Text* et *Name* :

Text Saisies & boutons - 1
Name frmSaisiesBoutons

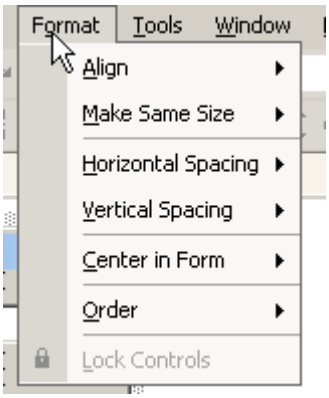
A l'aide de la barre "Tollbox"

- sélectionnez les composants dont vous avez besoin
- déposez-les sur la fenêtre et donnez-leur leurs bonnes dimensions





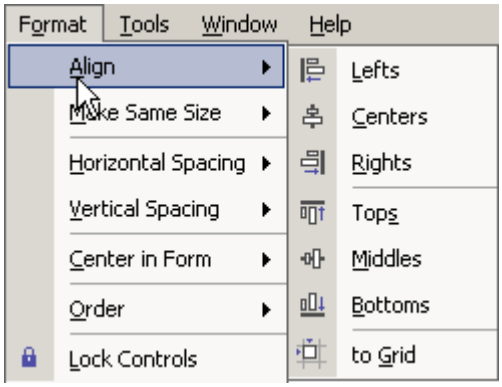
Une fois le composant choisi dans le "toolbox", utilisez la touche "Echap" pour faire disparaître la barre d'outils, puis déposez et dimensionnez le composant. faites-le pour les trois composants nécessaires : *Label*, *TextBox*, *Button*. Pour aligner et dimensionner correctement les composants, utilisez le menu *Format* :



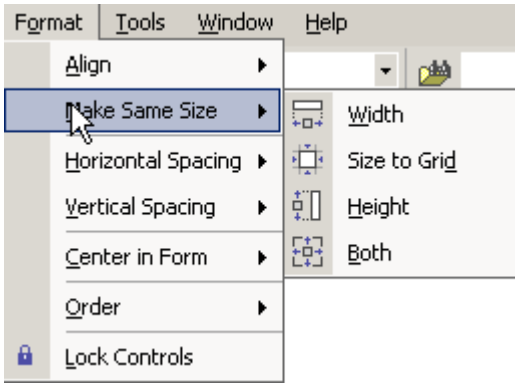
Le principe du formatage est le suivant :

1. sélectionnez les différents composants à formater ensemble (touche Ctrl appuyée)
2. sélectionnez le type de formatage désiré

L'option *Align* vous permet d'aligner des composants

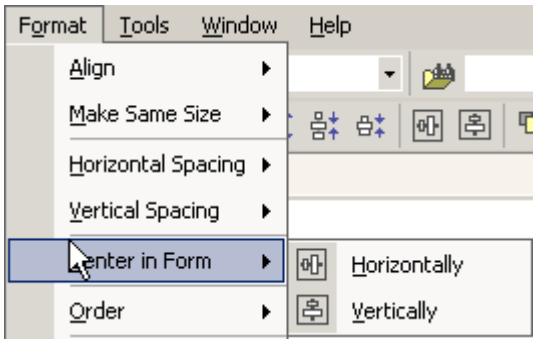


L'option *Make Same Size* permet de faire que des composants aient la même hauteur ou la même largeur :

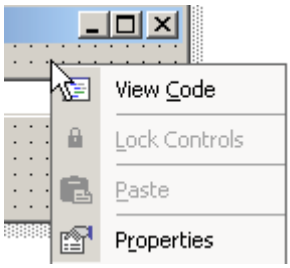


L'option *Horizontal Spacing* permet par exemple d'aligner horizontalement des composants avec des intervalles entre eux de même taille. Idem pour l'option *Vertical Spacing* pour aligner verticalement.

L'option *Center in Form* permet de centrer un composant horizontalement ou verticalement dans la fenêtre :

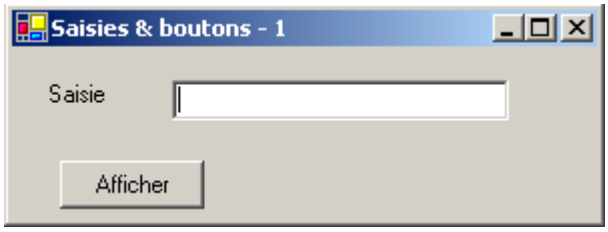


Une fois que les composants sont correctement placés sur la fenêtre, fixez leurs propriétés. Pour cela, cliquez droit sur le composant et prenez l'option *Properties* :

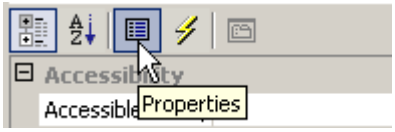


- l'étiquette 1 (Label)
  Sélectionnez le composant pour avoir sa fenêtre de propriétés. Dans celle-ci, modifiez les propriétés suivantes : **name** : lblSaisie, **text** : Saisie
- le champ de saisie 2 (TextBox)
  Sélectionnez le composant pour avoir sa fenêtre de propriétés. Dans celle-ci, modifiez les propriétés suivantes : **name** : txtSaisie, **text** : ne rien mettre
- le bouton 3 (Button) : **name** : cmdAfficher, **text** : Afficher
- la fenêtre elle-même : **name** : frmSaisies&Boutons, **text** : Saisies & boutons - 1

Nous pouvons exécuter (F5) notre projet pour avoir un premier aperçu de la fenêtre en action :

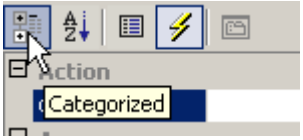


Fermez la fenêtre. Il nous reste à écrire la procédure liée à un clic sur le bouton *Afficher*. Sélectionnez le bouton pour avoir accès à sa fenêtre de propriétés. Celle-ci a plusieurs onglets :



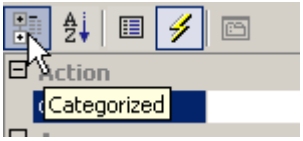
Properties liste des propriétés par ordre alphabétique
Events événements liés au contrôle

Les propriétés et événements d'un contrôle sont accessibles par catégories ou par ordre alphabétique :



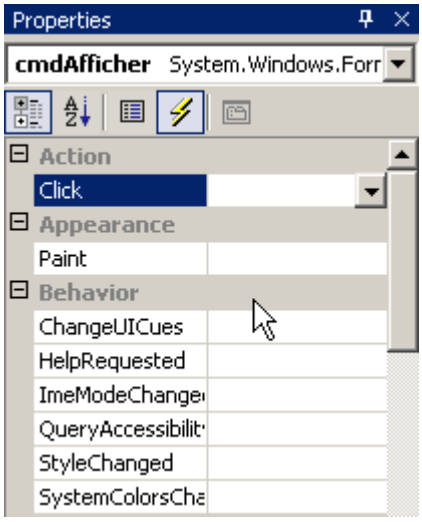
Categorized Propriétés ou événements par catégorie
Alphabetic Propriétés ou événements par ordre alphabétique

L'image suivante montre les événements par catégorie



alors que celle-ci montre les propriétés par ordre alphabétique.

Choisissez l'onglet *Events* pour le bouton *cmdAfficher* :



La colonne de gauche de la fenêtre liste les événements possibles sur le bouton. Un clic sur un bouton correspond à l'événement *Click*. La colonne de droite contient le nom de la procédure appelée lorsque l'événement correspondant se produit. Double-cliquez sur la cellule de l'événement Click. On passe alors automatiquement dans la fenêtre de code pour écrire le gestionnaire de l'événement *Click* sur le bouton *cmdAfficher* :

```
private void cmdAfficher_Click(object sender, System.EventArgs e) {

}
```

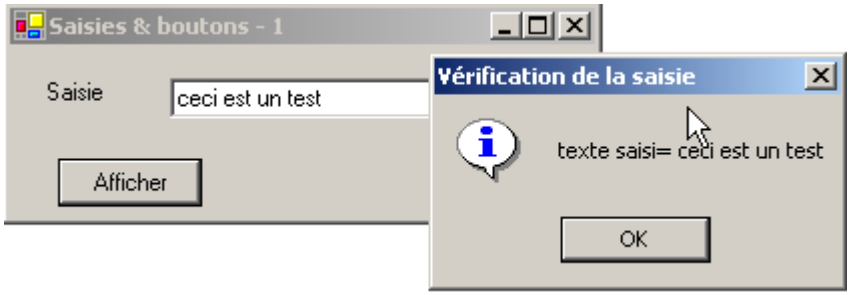
Le gestionnaire d'événement précédent a deux paramètres :

sender l'objet à la source de l'événement (ici le bouton)
e un objet *EventArgs* qui détaille l'événement qui s'est produit

Nous n'utiliserons aucun de ces paramètres ici. Il ne nous reste plus qu'à compléter le code. Ici, nous voulons présenter une boîte de dialogue avec dedans le contenu du champ *txtSaisie* :

```
   private void cmdAfficher_Click(object sender, System.EventArgs e) {
     // on affiche le texte qui a été saisi dans la boîte de saisie TxtSaisie
     MessageBox.Show("texte saisi= "+txtSaisie.Text,"Vérification de la
saisie",MessageBoxButtons.OK,MessageBoxIcon.Information);
   }
```

Si on exécute l'application on obtient la chose suivante :



4.3.1 Le code lié à la gestion des événements

Outre la fonction *cmdAfficher_Click* que nous avons écrite, VS.NET a généré dans la méthode *InitializeComponents* qui crée et initialise les composants du formulaire la ligne suivante :

```
      this.cmdAfficher.Click += new System.EventHandler(this.cmdAfficher_Click);
```

Click est un événement de la classe Button. Il y est déclaré comme suit :

public event EventHandler Click

La syntaxe

```
this.cmdAfficher.Click +=
```

sert à ajouter un gestionnaire pour l'événement *Click* sur le bouton *cmdAfficher*. Ce gestionnaire doit être de type *System.EventHandler*. Le constructeur de cette classe admet un paramètre qui est la référence d'une méthode dont le prototype doit être *void f(object, EventArgs)*. Le premier paramètre est la référence de l'objet à la source de l'événement, ici le bouton. Le second paramètre est un objet de type *EventArgs* ou d'une classe dérivée.

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.2914\mscorlib.dll'
public class EventArgs :
    object
{

    // Fields
    public static readonly EventArgs Empty;

    // Constructors
    public EventArgs();

    // Methods
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
} // end of System.EventArgs
```

Le type *EventArgs* est très général et n'apporte en fait aucune information. Pour un clic sur un bouton, c'est suffisant. Pour un déplacement de souris sur un formulaire, on aurait un événement défini par :

public event MouseEventHandler MouseMove

La classe *MouseEventHandler* est définie comme :

```
public delegate void MouseEventHandler(
   object sender,
   MouseEventArgs e
)
```

C'est une fonction déléguée (delegate) de fonctions de signature *void f (object, MouseEventArgs)*. La classe *MouseEventArgs* est elle définie par :

```
// from module
'c:\winnt\assembly\gac\system.windows.forms\1.0.2411.0__b77a5c561934e089\system.windows.forms.dll'
public class System.Windows.Forms.MouseEventArgs :
    EventArgs
{

    // Fields

    // Constructors
```

```
    public MouseEventArgs(System.Windows.Forms.MouseButtons button, int clicks, int x, int y, int delta);

    // Properties
    public MouseButtons Button { get; }
    public int Clicks { get; }
    public int Delta { get; }
    public int X { get; }
    public int Y { get; }

    // Methods
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
} // end of System.Windows.Forms.MouseEventArgs
```

On voit que la classe *MouseEventArgs* est plus riche que la classe *EventArgs*. On peut par exemple connaître les coordonnées de la souris X et Y au moment où se produit l'événement.

4.3.2 Conclusion

Des deux projets étudiés, nous pouvons conclure qu'une fois l'interface graphique construite avec VS.NET, le travail du développeur consiste à écrire les gestionnaires des événements qu'il veut gérer pour cette interface graphique.

4.4 Quelques composants utiles

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 application, nous présentons l'interface graphique et le code intéressant, notamment les gestionnaires d'événements.

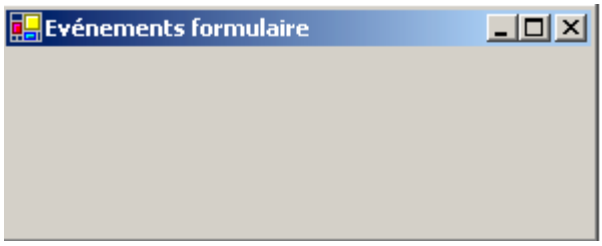
4.4.1 formulaire Form

Nous commençons par présenter le composant indispensable, le formulaire sur lequel on dépose des composants. Nous avons déjà présenté quelques-unes de ses propriétés de base. Nous nous attardons ici sur quelques événements importants d'un formulaire.

Load	le formulaire est en cours de chargement
Closing	le formulaire est en cours de fermeture
Closed	le formulaire est fermé

L'événement *Load* se produit avant même que le formulaire ne soit affiché. L'événement *Closing* se produit lorsque le formulaire est en cours de fermeture. On peut encore arrêter cette fermeture par programmation.

Nous construisons un formulaire de nom *Form1* sans composant :



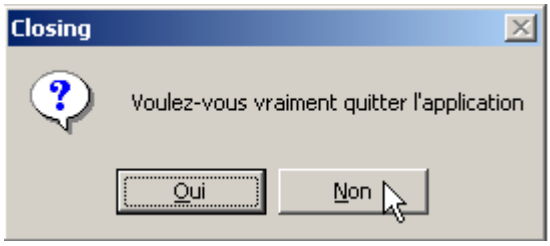
Nous traitons les trois événements précédents :

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

    private void Form1_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
      // le formulaire est en train de se fermer
      MessageBox.Show("Evt Closing","Closing");
      // on demande confirmation
      DialogResult réponse=MessageBox.Show("Voulez-vous vraiment quitter
l'application","Closing",MessageBoxButtons.YesNo,MessageBoxIcon.Question);
      if(réponse==DialogResult.No) e.Cancel=true;
    }

    private void Form1_Closed(object sender, System.EventArgs e) {
      // le formulaire est en train de se fermer
```

```
      MessageBox.Show("Evt Closed","Closed");
    }
```

Nous utilisons la fonction *MessageBox* pour être averti des différents événements. L'événement Closing va se produire lorsque l'utilisateur ferme la fenêtre.

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

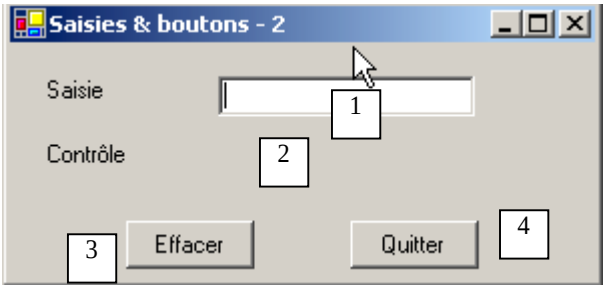


S'il répond Non, nous fixons la propriété *Cancel* de l'événement *CancelEventArgs e* que la méthode a reçu en paramètre. Si nous mettons cette propriété à *False*, la fermeture de la fenêtre est abandonnée, sinon elle se poursuit :

4.4.2 étiquettes Label et boîtes de saisie TextBox

Nous avons déjà rencontré ces deux composants. *Label* est un composant texte et *TextBox* un composant champ de saisie. Leur propriété principale est *Text* qui désigne soit le contenu du champ de saisie ou le texte du libellé. Cette propriété est en lecture/écriture.

L'événement habituellement utilisé pour *TextBox* est *TextChanged* qui signale que l'utilisateur à 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	txtSaisie	champ de saisie
2	Label	lblControle	affiche le texte de 1 en temps réel
3	Button	cmdEffacer	pour effacer les champs 1 et 2
4	Button	cmdQuitter	pour quitter l'application

Le code pertinent de cette application est celui des trois gestionnaires d'événements :

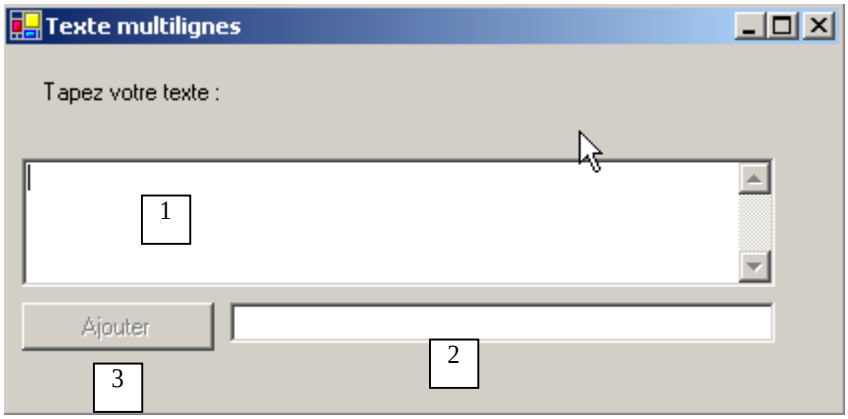
```
private void cmdQuitter_Click(object sender, System.EventArgs e) {
      // clic sur bouton Quitter
      // on quitte l'application
      Application.Exit();
    }

    private void txtSaisie_TextChanged(object sender, System.EventArgs e) {
      // le contenu du TextBox a changé
      // on le copie dans le Label lblControle
      lblControle.Text=txtSaisie.Text;
    }

    private void cmdEffacer_Click(object sender, System.EventArgs e) {
      // on efface le contenu de la boîte de saisie
```

```
      txtSaisie.Text="";
}
```

On notera la façon de terminer l'application dans la procédure *cmdQuitter_Click* : *Application.Exit()*. On se rappellera ici comment elle est lancée dans la procédure *Main* de la classe :

```
    static void Main()
    {
      // on affiche le formulaire frmSaisies
      Application.Run(new frmSaisies());
  }
```

L'exemple suivant utilise un *TextBox* multilignes :



La liste des contrôles est la suivante :

n°	type	nom	rôle
1	TextBox	txtMultiLignes	champ de saisie multilignes
2	TextBox	txtAjout	champ de saisie monoligne
3	Button	btnAjouter	Ajoute le contenu de 2 à 1

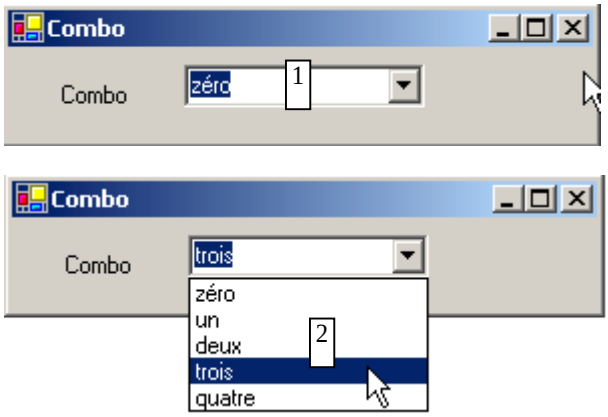
Pour qu'un TextBox devienne multilignes on positionne les propriétés suivantes du contrôle :

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

Le code utile est celui qui traite le clic sur le bouton Ajouter :

```
    private void btnAjouter_Click(object sender, System.EventArgs e) {
      // ajout du contenu de txtAjout à celui de txtMultilignes
      txtMultilignes.Text+=txtAjout.Text;
      txtAjout.Text="";
}
```

4.4.3 listes déroulantes ComboBox



Un composant *ComboBox* est une liste déroulante doublée d'une zone de saisie : l'utilisateur peut soit choisir un élément dans (2) soit taper du texte dans (1). Il existe trois sortes de ComboBox fixées par la propriété Style :

Simple	liste non déroulante avec zone d'édition
DropDown	liste déroulante avec zone d'édition
DropDownList	liste déroulante sans zone d'édition

Par défaut, le type d'un ComboBox est *DropDown*. Pour découvrir la classe *ComboBox*, tapez *ComboBox* dans l'index de l'aide (*Help/Index*).

La classe *ComboBox* a un seul constructeur :

new ComboBox()	crée un combo vide

Les éléments du ComboBox sont disponibles dans la propriété *Items* :

```
public ComboBox.ObjectCollection Items {get;}
```

C'est une propriété indexée, *Items[i]* désignant l'élément i du Combo. Elle est en lecture seule. La classe `ComboBox.ObjectCollection` est définie comme suit :

```
// from module
'c:\winnt\assembly\gac\system.windows.forms\1.0.2411.0__b77a5c561934e089\system.windows.forms.dll'
public class System.Windows.Forms.ComboBox+ObjectCollection :
    object,
    System.Collections.IList,
    System.Collections.ICollection,
    System.Collections.IEnumerable
{

    // Fields

    // Constructors
    public ObjectCollection(System.Windows.Forms.ComboBox owner);

    // Properties
    public int Count { virtual get; }
    public bool IsReadOnly { virtual get; }
    public object this[ int index ] { virtual get; virtual set; }

    // Methods
    public int Add(object item);
    public void AddRange(object[] items);
    public virtual void Clear();
    public virtual bool Contains(object value);
    public void CopyTo(object[] dest, int arrayIndex);
    public virtual bool Equals(object obj);
    public virtual System.Collections.IEnumerator GetEnumerator();
    public virtual int GetHashCode();
    public Type GetType();
    public virtual int IndexOf(object value);
    public virtual void Insert(int index, object item);
    public virtual void Remove(object value);
    public virtual void RemoveAt(int index);
    public virtual string ToString();
} // end of System.Windows.Forms.ComboBox+ObjectCollection
```

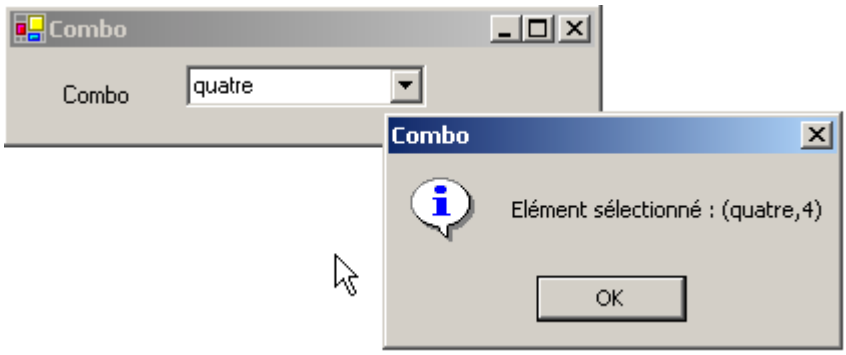
Soit *C* un combo et *C.Items* sa liste d'éléments. On a les propriétés suivantes :

C.Items.Count	nombre d'éléments du combo
C.Items[i]	élément i du combo
C.Add(object o)	ajoute l'objet o en dernier élement du combo
C.AddRange(object[] objets)	ajoute un tableau d'objets en fin de combo
C.Insert(int i, object o)	ajoute l'objet o en position i du combo
C.RemoveAt(int i)	enlève l'élément i du combo
C.Remove(object o)	enlève l'objet o du combo
C.Clear()	supprime tous les éléments du combo
C.IndexOf(object o)	rend la position i de l'objet o dans le combo

On peut s'étonner qu'un combo puisse contenir des objets alors qu'habituellement il contient des chaînes de caractères. Au niveau visuel, ce sera le cas. Si un *ComboBox* contient un objet *obj*, il affiche la chaîne *obj.ToString()*. On se rappelle que tout objet à une méthode *ToString* héritée de la classe *object* et qui rend une chaîne de caractères "représentative" de l'objet.

L'élément sélectionné dans le combo 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.

Lors du choix d'un élément dans la liste déroulante se produit l'événement *SelectedIndexChanged* qui peut être alors utilisé pour être averti du changement de sélection dans le combo. Dans l'application suivante, nous utilisons cet événement pour afficher l'élément qui a été sélectionné dans la liste.



Nous ne présentons que le code pertinent de la fenêtre.

Dans le constructeur du formulaire nous remplissons le combo :

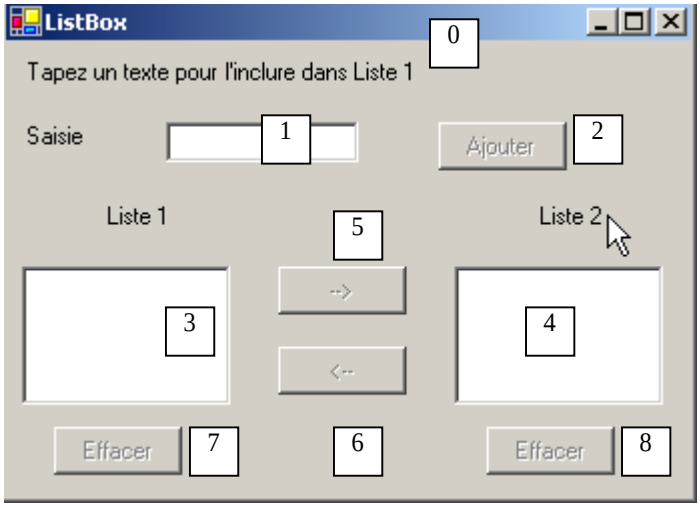
```
public frmCombo()
    {
      // création formulaire
      InitializeComponent();
      // remplissage combo
      cmbNombres.Items.AddRange(new string[] {"zéro","un","deux","trois","quatre"});
      // nous sélectionnons le 1er élément de la liste
      cmbNombres.SelectedIndex=0;
    }//constructeur
```

Nous traitons l'événement *SelectedIndexChanged* du combo qui signale un nouvel élément sélectionné :

```
private void cmbNombres_SelectedIndexChanged(object sender, System.EventArgs e) {
      // l'élément sélectionné à changé - on l'affiche
      MessageBox.Show("Elément sélectionné : (" + cmbNombres.SelectedItem +"," +
cmbNombres.SelectedIndex+")","Combo",MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

4.4.4 composant 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 BorderStyle=FixedSingle
1	TextBox	txtSaisie	champ de saisie

2	Button	btnAjouter	bouton permettant d'ajouter le contenu du champ de saisie 1 dans la liste 3
3	ListBox	listBox1	liste 1
4	ListBox	listBox2	liste 2
5	Button	btn1TO2	transfère les éléments sélectionnés de liste 1 vers liste 2
6	Button	cmd2T0	fait l'inverse
7	Button	btnEffacer1	vide la liste 1
8	Button	btnEffacer2	vide la liste 2

Fonctionnement

- 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. Ce 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 de le faire nous-mêmes dans le constructeur comme ci-dessous :

```
public Form1()
    {
      // création initiale du formulaire
      InitializeComponent();
      // initialisations complémentaires
      // on inhibe un certain nombre de boutons
      btnAjouter.Enabled=false;
      btn1TO2.Enabled=false;
      btn2TO1.Enabled=false;
      btnEffacer1.Enabled=false;
      btnEffacer2.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 txtSaisie_TextChanged(object sender, System.EventArgs e) {
      // le contenu de txtSaisie a changé
      // le bouton Ajouter n'est allumé que si la saisie est non vide
      btnAjouter.Enabled=txtSaisie.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
      btn1TO2.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
      btn2TO1.Enabled=true;
    }
```

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

```
private void btnAjouter_Click(object sender, System.EventArgs e) {
      // ajout d'un nouvel élément à la liste 1
      listBox1.Items.Add(txtSaisie.Text.Trim());
      // raz de la saisie
      txtSaisie.Text="";
      // Liste 1 n'est pas vide
      btnEffacer1.Enabled=true;
      // retour du focus sur la boîte de saisie
      txtSaisie.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 btnEffacer1_Click(object sender, System.EventArgs e) {
      // on efface la liste 1
      listBox1.Items.Clear();
    }

    private void btnEffacer2_Click(object sender, System.EventArgs e) {
      // on efface la liste 2
      listBox2.Items.Clear();
    }
```

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

```
private void btn1TO2_Click(object sender, System.EventArgs e) {
      // transfert de l'élément sélectionné dans Liste 1 dans Liste 2
      transfert(listBox1,listBox2);
      // boutons Effacer
      btnEffacer2.Enabled=true;
      btnEffacer1.Enabled=listBox1.Items.Count!=0;
      // boutons de transfert
      btn1TO2.Enabled=false;
      btn2TO1.Enabled=false;
    }

    private void btn2TO1_Click(object sender, System.EventArgs e) {
      // transfert de l'élément sélectionné dans Liste 2 dans Liste 1
      transfert(listBox2,listBox1);
      // boutons Effacer
      btnEffacer1.Enabled=true;
      btnEffacer2.Enabled=listBox2.Items.Count!=0;
      // boutons de transfert
      btn1TO2.Enabled=false;
      btn2TO1.Enabled=false;
    }

    // transfert
    private void transfert(ListBox l1, ListBox l2){
      // transfert de l'élément sélectionné de la liste 1 dans la liste l2
      // un élément sélectionné ?
      if(l1.SelectedIndex==-1) return;
      // ajout dans l2
      l2.Items.Add(l1.SelectedItem);
      // suppression dans l1
      l1.Items.RemoveAt(l1.SelectedIndex);
    }
```

Tout d'abord, on crée une méthode

```
    private void transfert(ListBox l1, ListBox l2){
```

qui transfère dans la liste *l2* l'élément sélectionné dans la liste *l1*. Cela nous permet d'avoir une seule méthode au lieu de deux pour transférer un élément de *listBox1* vers *listBox2* ou de *listBox2* vers *listBox1*. Avat de faire le transfert, on s'assure qu'il y a bien un élément sélectionné dans la liste *l1* :

```
      // un élément sélectionné ?
      if(l1.SelectedIndex==-1) return;
```

La propriété *SelectedIndex* vaut -1 si aucun élément n'est actuellement sélectionné.

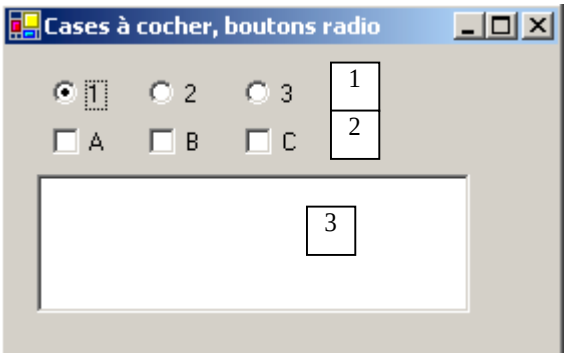
Dans les procédures

```
    private void btnXTOY_Click(object sender, System.EventArgs e)
```

on opère le transfert de la liste X vers la liste Y et on change l'état de certains boutons pour refléter le nouvel état des listes.

4.4.5 cases à cocher CheckBox, boutons radio ButtonRadio

Nous nous proposons d'écrire l'application suivante :



Les composants de la fenêtre sont les suivants :

n°	type	nom	rôle
1	RadioButton	radioButton1 radioButton2 radioButton3	3 boutons radio
2	CheckBox	chechBox1 chechBox2 chechBox3	3 cases à cocher
3	ListBox	lstValeurs	une liste

Si on construit les trois boutons radio l'un après l'autre, ils font partie par défaut d'un même groupe. Aussi lorsque l'un est coché, les autres ne le sont pas. L'événement qui nous intéresse pour ces six 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 *Check* qui à vrai signifie que le contrôle est coché. Nous avons ici utilisé une seule méthode pour traiter les six événements *CheckChanged*, la méthode *affiche*. Aussi dans la méthode *InitializeComponent* trouve-t-on les instructions suivantes :

```
this.checkBox1.CheckedChanged += new System.EventHandler(this.affiche);
this.checkBox2.CheckedChanged += new System.EventHandler(this.affiche);
this.checkBox3.CheckedChanged += new System.EventHandler(this.affiche);
this.radioButton1.CheckedChanged += new System.EventHandler(this.affiche);
this.radioButton2.CheckedChanged += new System.EventHandler(this.affiche);
this.radioButton3.CheckedChanged += new System.EventHandler(this.affiche);
```

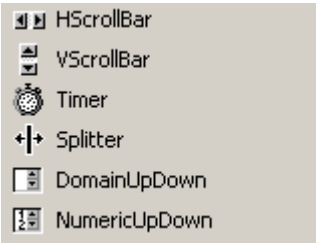
La méthode *affiche* est définie comme suit :

```
  private void affiche(object sender, System.EventArgs e) {
    // affiche l'état du bouton radio ou de la case à cocher
    // est-ce un checkbox ?
    if (sender is CheckBox) {
      CheckBox chk=(CheckBox)sender;
      lstValeurs.Items.Add(chk.Name+"="+chk.Checked);
    }
    // est-ce un radiobutton ?
    if (sender is RadioButton) {
      RadioButton rdb=(RadioButton)sender;
      lstValeurs.Items.Add(rdb.Name+"="+rdb.Checked);
    }
  }//affiche
```

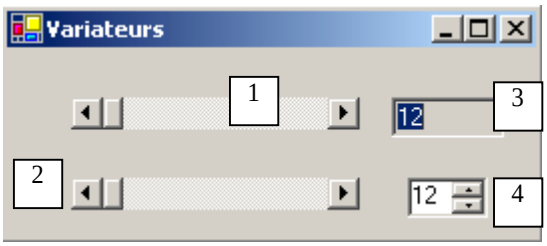
La syntaxe `if (sender is CheckBox)` permet de vérifier si l'objet *sender* est de type *CheckBox*. Cela nous permet ensuite de faire un transtypage vers le type exact de *sender*. La méthode *affiche* écrit dans la liste *lstValeurs* le nom du composant à l'origine de l'événement et la valeur de sa propriété *Checked*. A l'exécution, on voit qu'un clic sur un bouton radio provoque deux événements *CheckChanged* : l'un sur l'ancien bouton coché qui passe à "non coché" et l'autre sur le nouveau bouton qui passe à "coché".

4.4.6 variateurs ScrollBar

Il existe plusieurs types de variateur : le variateur horizontal (*hScrollBar*), le variateur vertical (*vScrollBar*), l'incrémenteur (*NumericUpDown*).



Réalisons l'application suivante :



n°	type	nom	rôle
1	hScrollBar	hScrollBar1	un variateur horizontal
2	hScrollBar	hScrollBar2	un variateur horizontal qui suit les variations du variateur 1
3	TextBox	txtValeur	affiche la valeur du variateur horizontal ReadOnly=true pour empêcher toute saisie
4	NumericUpDown	incrémenteur	permet de fixer la valeur du variateur 2

- Un variateur *ScrollBar* permet à l'utilisateur de choisir une valeur dans une plage de valeurs entières symbolisée par la "bande" du variateur sur laquelle se déplace un curseur. La valeur du variateur est disponible dans sa propriété **Value**.
- Pour un variateur horizontal, l'extrémité gauche représente la valeur minimale de la plage, l'extrémité droite la valeur maximale, le curseur la valeur actuelle choisie. Pour un variateur vertical, le minimum est représenté par l'extrémité haute, le maximum par l'extrémité basse. Ces valeurs sont représentées par les propriétés **Minimum** et **Maximum** et valent par défaut 0 et 100.
- Un clic sur les extrémités du variateur fait varier la valeur d'un incrément (positif ou négatif) selon l'extrémité cliquée appelée **SmallChange** qui est par défaut 1.
- Un clic de part et d'autre du curseur fait varier la valeur d'un incrément (positif ou négatif) selon l'extrémité cliquée appelée **LargeChange** qui est par défaut 10.
- Lorsqu'on clique sur l'extrémité supérieure d'un variateur vertical, sa valeur diminue. Cela peut surprendre l'utilisateur moyen qui s'attend normalement à voir la valeur "monter". On règle ce problème en donnant une valeur négative aux propriétés *SmallChange et LargeChange*
- Ces cinq propriétés (**Value, Minimum, Maximum, SmallChange, LargeChange**) sont accessibles en lecture et écriture.
- L'événement principal du variateur est celui qui signale un changement de valeur : l'événement **Scroll**.

Un composant **NumericUpDown** est proche du variateur : il a lui aussi les propriétés *Minimum*, *Maximum* et *Value*, par défaut 0, 100, 0. Mais ici, la propriété *Value* est affichée dans une boîte de saisie faisant partie intégrante du contrôle. L'utilisateur peut lui même modifier cette valeur sauf si on a mis la propriété *ReadOnly* du contrôle à vrai. La valeur de l'incrément est fixé par la propriété *Increment*, par défaut 1. L'événement principal du composant *NumericUpDown* est celui qui signale un changement de valeur : l'événement **ValueChanged**

Le code utile de notre application est le suivant :

Le formulaire est mis en forme lors de sa construction :

```
public Form1()
      {
        // création initiale du formulaire
        InitializeComponent();
        // on donne au variateur 2 les mêmes caractéristiques qu'au variateur 1
        hScrollBar2.Minimum=hScrollBar1.Value;
        hScrollBar2.Minimum=hScrollBar1.Minimum;
        hScrollBar2.Maximum=hScrollBar1.Maximum;
        hScrollBar2.LargeChange=hScrollBar1.LargeChange;
        hScrollBar2.SmallChange=hScrollBar1.SmallChange;
        // idem pour l'incrémenteur
        incrémenteur.Minimum=hScrollBar1.Value;
        incrémenteur.Minimum=hScrollBar1.Minimum;
        incrémenteur.Maximum=hScrollBar1.Maximum;
        incrémenteur.Increment=hScrollBar1.SmallChange;

        // on donne au TextBox la valeur du variateur 1
        txtValeur.Text=""+hScrollBar1.Value;
      }//constructeur
```

Le gestionnaire qui suit les variations de valeur du variateur 1 :

```
    private void hScrollBar1_Scroll(object sender, System.Windows.Forms.ScrollEventArgs e) {
      // changement de valeur du variateur 1
      // on répercute sa valeur sur le variateur 2 et sur le textbox TxtValeur
      hScrollBar2.Value=hScrollBar1.Value;
      txtValeur.Text=""+hScrollBar1.Value;
```

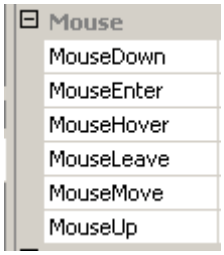
```
    }
```

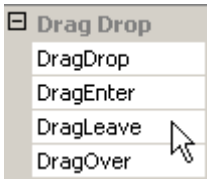
Le gestionnaire qui suit les variations du contrôle *incrémenteur* :

```
    private void incrémenteur_ValueChanged(object sender, System.EventArgs e) {
      // on fixe la valeur du variateur 2
      hScrollBar2.Value=(int)incrémenteur.Value;
    }
```

4.5 Événements souris

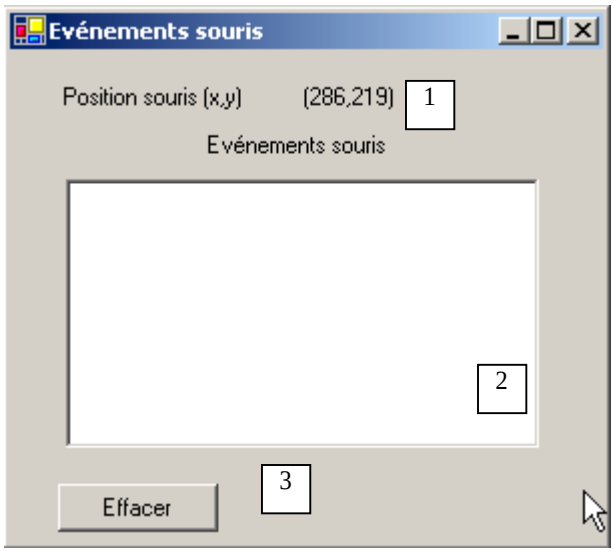
Lorsqu'on dessine dans un conteneur, il est important de connaître la position de la souris pour par exemple afficher un point lors d'un clic. Les déplacements de la souris provoquent des événements dans le conteneur dans lequel elle se déplace.





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 un programme permettant de mieux appréhender à quels moments se produisent les différents événements souris :



n°	type	nom	rôle
1	Label	lblPosition	pour afficher la position de la souris dans le formulaire 1, la liste 2 ou le bouton 3
2	ListBox	lstEvts	pour afficher les évts souris autres que MouseMove
3	Button	btnEffacer	pour effacer le contenu de 2

Les gestionnaires d'événements sont les suivants :

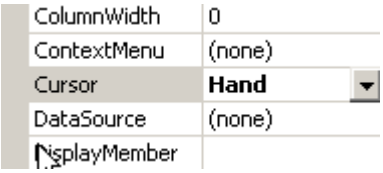
Pour suivre les déplacements de la souris sur les trois contrôles, on n'écrit qu'un seul gestionnaire :

```
    private void Form1_MouseMove(object sender, System.Windows.Forms.MouseEventArgs e) {
      // mvt souris - on affiche les coordonnées (X,Y) de celle-ci
      lblPosition.Text="("+e.X+","+e.Y+")";
    }
```

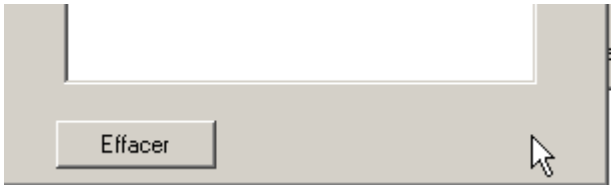
et dans le constructeur (*InitializeComponent*), on donne le même gestionnaire d'événements *Form1_MouseMove* aux trois contrôles :

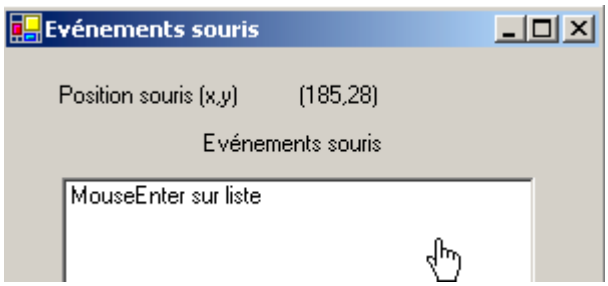
```
      this.MouseMove += new System.Windows.Forms.MouseEventHandler(this.Form1_MouseMove);
      this.btnEffacer.MouseMove += new System.Windows.Forms.MouseEventHandler(this.Form1_MouseMove);
      this.lstEvts.MouseMove += new System.Windows.Forms.MouseEventHandler(this.Form1_MouseMove);
```

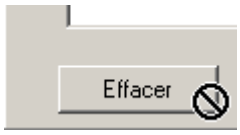
Il faut savoir ici qu'à chaque fois que la souris entre dans le domaine d'un contrôle son système de coordonnées change. Son origine (0,0) est le coin supérieur gauche du contrôle sur lequel elle se trouve. Ainsi à l'exécution, lorsqu'on passe la souris du formulaire au bouton, on voit clairement le changement de coordonnées. Afin de mieux voir ces changements de domaine de la souris, on peut utiliser la propriété *Cursor* des contrôles :



Cette propriété permet de fixer la forme du curseur de souris lorsque celle-ci entre dans le domaine du contrôle. Ainsi dans notre exemple, nous avons fixé le curseur à **Default** pour le formulaire lui-même, **Hand** pour la liste 2 et à **No** pour le bouton 3 comme le montrent les copies d'écran ci-dessous.





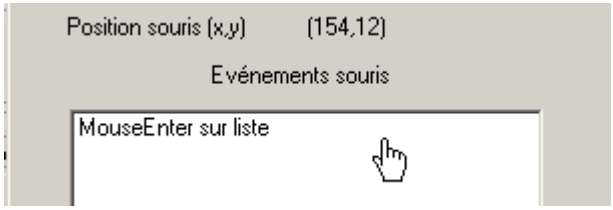


Dans le code du constructeur, le code généré par ces choix est le suivant :

```
      this.lstEvts.Cursor = System.Windows.Forms.Cursors.Hand;
      this.btnEffacer.Cursor = System.Windows.Forms.Cursors.No;
```

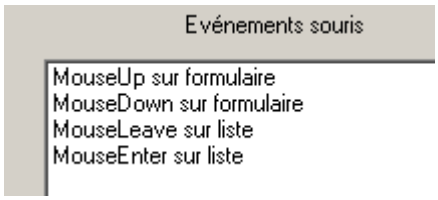
Par ailleurs, pour détecter les entrées et sorties de la souris sur la liste 2, nous traitons les événements MouseEnter et MouseLeave de cette même liste :

```
    // affiche
    private void affiche(String message){
      // on affiche le message en haut de la liste des evts
      lstEvts.Items.Insert(0,message);
    }

    private void lstEvts_MouseEnter(object sender, System.EventArgs e) {
      affiche ("MouseEnter sur liste");
    }

    private void lstEvts_MouseLeave(object sender, System.EventArgs e) {
      affiche ("MouseLeave sur liste");
    }
```



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

```
private void Form1_MouseDown(object sender, System.Windows.Forms.MouseEventArgs e) {
  affiche("MouseDown sur formulaire");
}

private void Form1_MouseUp(object sender, System.Windows.Forms.MouseEventArgs e) {
  affiche("MouseUp sur formulaire");
}
```

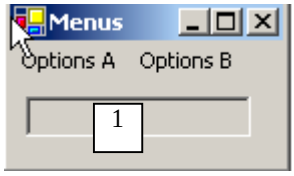


Enfin, le code du gestionnaire de clic sur le bouton *Effacer* :

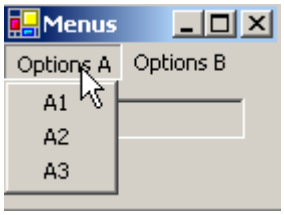
```
private void btnEffacer_Click(object sender, System.EventArgs e) {
  // efface la liste des evts
  lstEvts.Items.Clear();
}
```

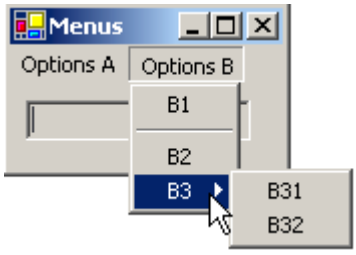
4.6 Créer une fenêtre avec menu

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

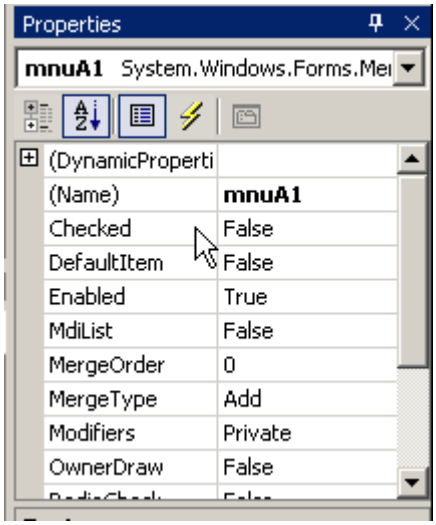


Le contrôle 1 est un *TextBox* en lecture seule (*ReadOnly=true*) et de nom *txtStatut*. L'arborescence du menu est la suivante :





Les options de menu sont des contrôles comme les autres composants visuels et ont des propriétés et événements. Par exemple le tableau des propriétés de l'option de menu A1 :



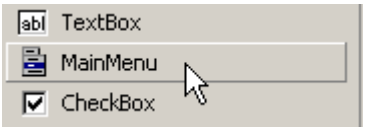
Deux propriétés sont utilisées dans notre exemple :

Name le nom du contrôle menu
Text le libellé de l'option de menu

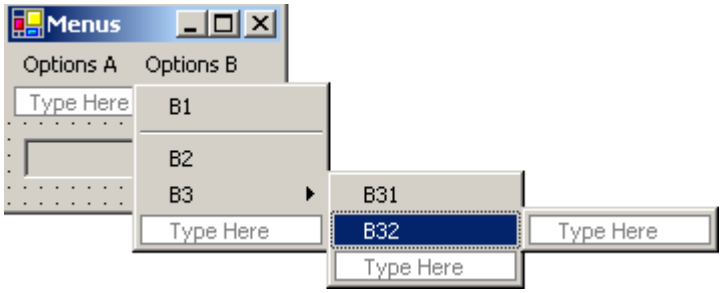
Les propriétés des différentes options de menu de notre exemple sont les suivantes :

Name	Text
mnuA	options A
mnuA1	A1
mnuA2	A2
mnuA3	A3
mnuB	options B
mnuB1	B1
mnuSep1	- (séparateur)
mnuB2	B2
mnuB3	B3
mnuB31	B31
mnuB32	B32

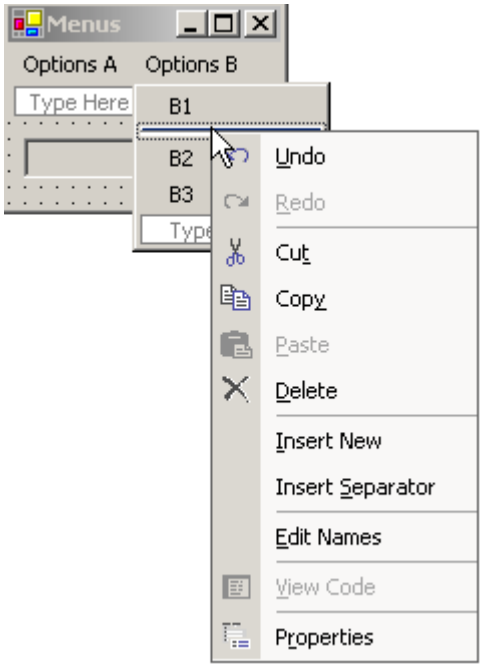
Pour créer un menu, on choisit le composant "*MainMenu*" dans la barre "*ToolBox*" :



On a alors un menu vide qui s'installe sur le formulaire avec des cases vides intitulées "Type Here". Il suffit d'y indiquer les différentes options du menu :

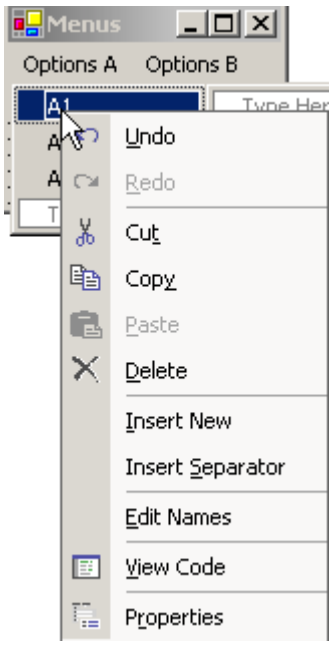


Pour insérer un séparateur entre deux options comme ci-dessus entre les options B1 et B2, positionnez-vous à l'emplacement du séparateur dans le menu, cliquez droit et prenez l'option *Insert Separator* :

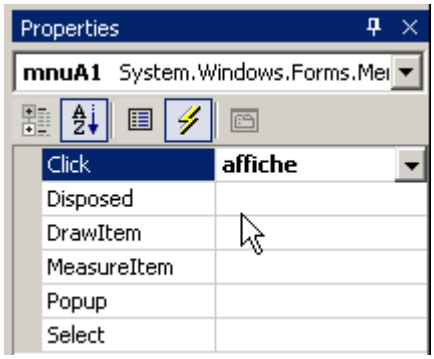


Si on lance l'application par F5, on obtient un formulaire avec un menu qui pour l'instant ne fait rien.

Les options de menu sont traitées comme des composants : elles ont des propriétés et des événements. Dans la structure du menu, sélectionnez l'option *A1* et cliquez droit pour avoir accès aux propriétés du contrôle :



Vous avez alors accès à la fenêtre des propriétés dans laquelle on sélectionnera l'onglet *événements*. Sélectionnez les événements et tapez **affiche** en face de l'événement *Click*. Cela signifie que l'on souhaite que le clic sur l'option A1 soit traitée par une méthode appelée *affiche*.



VS.NET génère automatiquement la méthode *affiche* dans la fenêtre de code :

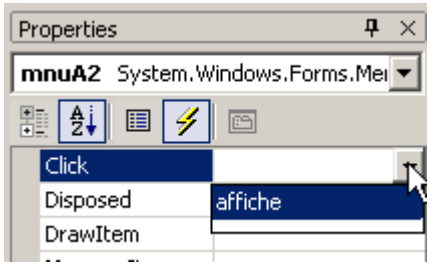
```
    private void affiche(object sender, System.EventArgs e) {
    }
```

Dans cette méthode, nous nous contenterons d'afficher la propriété *Text* de l'option de menu à la source de l'événement :

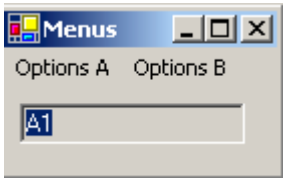
```
    private void affiche(object sender, System.EventArgs e) {
      // affiche dans le TextBox le nom du sous-menu choisi
      txtStatut.Text=((MenuItem)sender).Text;
    }
```

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

Pour toutes les options de menu, on fixe le gestionnaire du clic à la méthode *affiche*. Sélectionnez par exemple l'option A2 et ses événements. En face de l'événement *Click*, on a une liste déroulante dans laquelle sont présentes les méthodes existantes pouvant traiter cet événement. Ici on n'a que la méthode *affiche* qu'on sélectionne. On répète ce processus pour tous les composants menu.



Exécutez l'application et sélectionnez l'option A1 pour obtenir le message suivant :



Le code utile de cette application outre celui de la méthode *affiche* est celui de la construction du menu dans le constructeur du formulaire (*InitializeComponent*) :

```
private void InitializeComponent()
    {
      this.mainMenu1 = new System.Windows.Forms.MainMenu();
      this.mnuA = new System.Windows.Forms.MenuItem();
      this.mnuA1 = new System.Windows.Forms.MenuItem();
      this.mnuA2 = new System.Windows.Forms.MenuItem();
      this.mnuA3 = new System.Windows.Forms.MenuItem();
      this.mnuB = new System.Windows.Forms.MenuItem();
      this.mnuB1 = new System.Windows.Forms.MenuItem();
      this.mnuB2 = new System.Windows.Forms.MenuItem();
      this.mnuB3 = new System.Windows.Forms.MenuItem();
      this.mnuB31 = new System.Windows.Forms.MenuItem();
      this.mnuB32 = new System.Windows.Forms.MenuItem();
      this.txtStatut = new System.Windows.Forms.TextBox();
      this.mnuSep1 = new System.Windows.Forms.MenuItem();
      this.SuspendLayout();
      //
      // mainMenu1
      //
      this.mainMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
```

```
                                                                    this.mnuA,
                                                                    this.mnuB});
    //
    // mnuA
    //
    this.mnuA.Index = 0;
    this.mnuA.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
                                                                    this.mnuA1,
                                                                    this.mnuA2,
                                                                    this.mnuA3});
    this.mnuA.Text = "Options A";
    //
    // mnuA1
    //
    this.mnuA1.Index = 0;
    this.mnuA1.Text = "A1";
    this.mnuA1.Click += new System.EventHandler(this.affiche);
    //
    // mnuA2
    //
    this.mnuA2.Index = 1;
    this.mnuA2.Text = "A2";
    this.mnuA2.Click += new System.EventHandler(this.affiche);
    //
    // mnuA3
    //
    this.mnuA3.Index = 2;
    this.mnuA3.Text = "A3";
    this.mnuA3.Click += new System.EventHandler(this.affiche);
    //
    // mnuB
    //
    this.mnuB.Index = 1;
    this.mnuB.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
                                                                    this.mnuB1,
                                                                    this.mnuSep1,
                                                                    this.mnuB2,
                                                                    this.mnuB3});
    this.mnuB.Text = "Options B";
    //
    // mnuB1
    //
    this.mnuB1.Index = 0;
    this.mnuB1.Text = "B1";
    this.mnuB1.Click += new System.EventHandler(this.affiche);
    //
    // mnuB2
    //
    this.mnuB2.Index = 2;
    this.mnuB2.Text = "B2";
    this.mnuB2.Click += new System.EventHandler(this.affiche);
    //
    // mnuB3
    //
    this.mnuB3.Index = 3;
    this.mnuB3.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {
                                                                     this.mnuB31,
                                                                     this.mnuB32});
    this.mnuB3.Text = "B3";
    this.mnuB3.Click += new System.EventHandler(this.affiche);
    //
    // mnuB31
    //
    this.mnuB31.Index = 0;
    this.mnuB31.Text = "B31";
    this.mnuB31.Click += new System.EventHandler(this.affiche);
    //
    // mnuB32
    //
    this.mnuB32.Index = 1;
    this.mnuB32.Text = "B32";
    this.mnuB32.Click += new System.EventHandler(this.affiche);
    //
    // txtStatut
    //
    this.txtStatut.Location = new System.Drawing.Point(8, 8);
    this.txtStatut.Name = "txtStatut";
    this.txtStatut.ReadOnly = true;
    this.txtStatut.Size = new System.Drawing.Size(112, 20);
    this.txtStatut.TabIndex = 0;
    this.txtStatut.Text = "";
    //
    // mnuSep1
    //
    this.mnuSep1.Index = 1;
    this.mnuSep1.Text = "-";
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(136, 42);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
```

```
                                                                this.txtStatut});
    this.Menu = this.mainMenu1;
    this.Name = "Form1";
    this.Text = "Menus";
    this.ResumeLayout(false);

  }
```

On notera l'instruction qui associe le menu au formulaire :

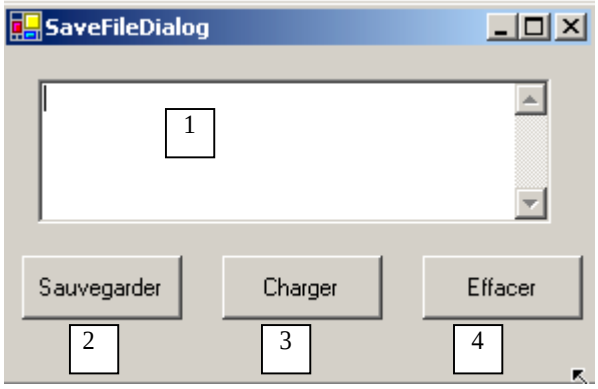
```
    this.Menu = this.mainMenu1;
```

4.7 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.

4.7.1 Boîtes de dialogue OpenFileDialog et SaveFileDialog

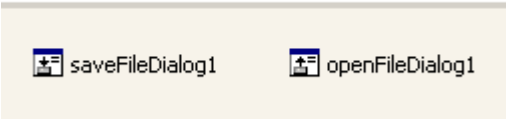
Nous allons construire l'application suivante :



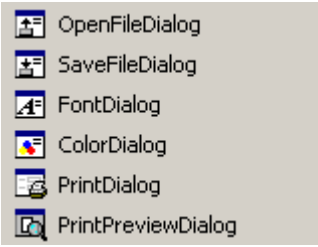
Les contrôles sont les suivants :

N°	type	nom	rôle
1	TextBox multilignes	txtTexte	texte tapé par l'utilisateur ou chargé à partir d'un fichier
2	Button	btnSauvegarder	permet de sauvegarder le texte de 1 dans un fichier texte
3	Button	btnCharger	permet de charger le contenu d'un fichier texte dans 1
4	Button	btnEffacer	efface le contenu de 1

Deux contrôles non visuels sont utilisés :



Lorsqu'ils sont pris dans le "ToolBox " et déposés sur le formulaire, ils sont placés dans une zone à part en bas du formulaire. Les composants "Dialog" sont pris dans le "ToolBox" :



Le code du bouton *Effacer* est simple :

```
    private void btnEffacer_Click(object sender, System.EventArgs e) {
      // on efface la boîte de saisie
      txtTexte.Text="";
    }
```

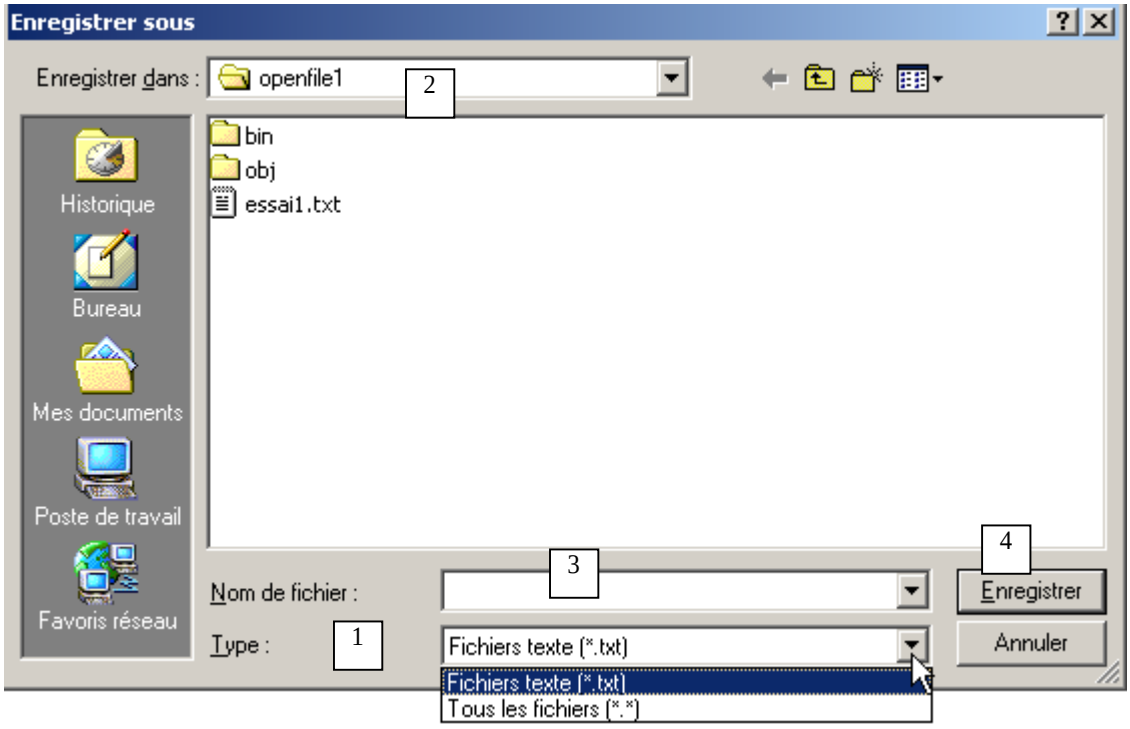
La classe **SaveFileDialog** est définie comme suit :

```
// from module
'c:\winnt\assembly\gac\system.windows.forms\1.0.2411.0__b77a5c561934e089\system.windows.forms.dll'
public sealed class System.Windows.Forms.SaveFileDialog :
    System.Windows.Forms.FileDialog,
    System.ComponentModel.IComponent,
    IDisposable
{

    // Fields

    // Constructors
    public SaveFileDialog();

    // Properties
    public bool AddExtension { get; set; }
    public bool CheckFileExists { virtual get; virtual set; }
    public bool CheckPathExists { get; set; }
    public IContainer Container { get; }
    public bool CreatePrompt { get; set; }
    public string DefaultExt { get; set; }
    public bool DereferenceLinks { get; set; }
    public string FileName { get; set; }
    public string[] FileNames { get; }
    public string Filter { get; set; }
    public int FilterIndex { get; set; }
    public string InitialDirectory { get; set; }
    public bool OverwritePrompt { get; set; }
    public bool RestoreDirectory { get; set; }
    public bool ShowHelp { get; set; }
    public ISite Site { virtual get; virtual set; }
    public string Title { get; set; }
    public bool ValidateNames { get; set; }

    // Events
    public event EventHandler Disposed;
    public event CancelEventHandler FileOk;
    public event EventHandler HelpRequest;

    // Methods
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public virtual void Dispose();
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public System.IO.Stream OpenFile();
    public virtual void Reset();
    public System.Windows.Forms.DialogResult ShowDialog();
    public virtual string ToString();
} // end of System.Windows.Forms.SaveFileDialog
```

De ces propriétés et méthodes nous retiendrons les suivantes :

string Filter	les types de fichiers proposés dans la liste déroulante des types de fichiers de la boîte de dialogue
int FilterIndex	le n° du type de fichier proposé par défaut dans la liste ci-dessus. Commence à 0.
string InitialDirectory	le dossier présenté initialement pour la sauvegarde du fichier
string FileName	le nom du fichier de sauvegarde indiqué par l'utilisateur
DialogResult.ShowDialog()	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 btnSauvegarder_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(txtTexte.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
          try{fichier.Close();} catch (Exception){}
        }//finally
      }//if
    }
```

- On fixe le dossier initial au dossier qui contient l'exécutable de l'application :

```
      saveFileDialog1.InitialDirectory=Application.ExecutablePath;
```

- On fixe les types de fichiers à présenter

```
      saveFileDialog1.Filter = "Fichiers texte (*.txt)|*.txt|Tous les fichiers (*.*)|*.*";
```

  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 au début

```
      saveFileDialog1.FilterIndex = 0;
```

Ici, ce sont les fichiers de type *.txt qui seront présentés tout d'abord à l'utilisateur.

- La boîte de dialogue est affichée et son résultat récupéré

```
if(saveFileDialog1.ShowDialog() == DialogResult.OK) {
```

- 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* : *txtTexte.Text* tout en gérant les exceptions qui peuvent se produire.

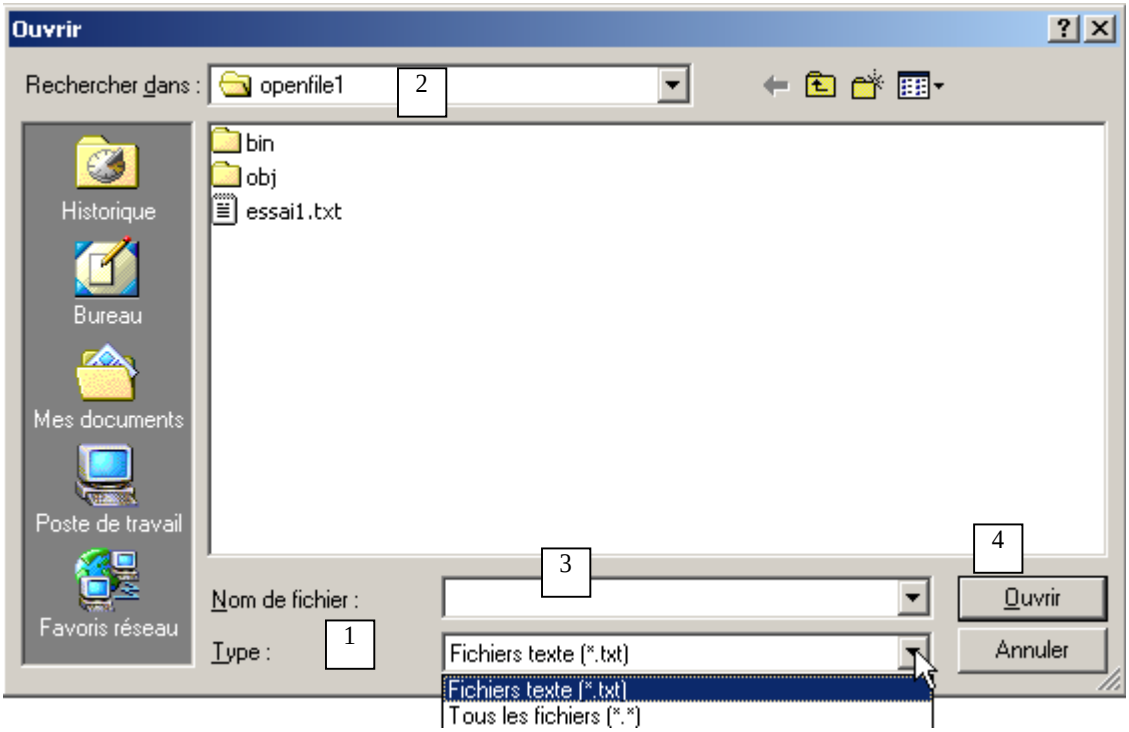
La classe **OpenFileDialog** est très proche de la classe *SaveFileDialog* et est définie comme suit :

```
// from module
'c:\winnt\assembly\gac\system.windows.forms\1.0.2411.0__b77a5c561934e089\system.windows.forms.dll'
public sealed class System.Windows.Forms.OpenFileDialog :
    System.Windows.Forms.FileDialog,
    System.ComponentModel.IComponent,
    IDisposable
{

    // Fields

    // Constructors
    public OpenFileDialog();

    // Properties
    public bool AddExtension { get; set; }
    public bool CheckFileExists { virtual get; virtual set; }
    public bool CheckPathExists { get; set; }
    public IContainer Container { get; }
    public string DefaultExt { get; set; }
    public bool DereferenceLinks { get; set; }
    public string FileName { get; set; }
    public string[] FileNames { get; }
    public string Filter { get; set; }
    public int FilterIndex { get; set; }
    public string InitialDirectory { get; set; }
    public bool Multiselect { get; set; }
    public bool ReadOnlyChecked { get; set; }
    public bool RestoreDirectory { get; set; }
    public bool ShowHelp { get; set; }
    public bool ShowReadOnly { get; set; }
    public ISite Site { virtual get; virtual set; }
    public string Title { get; set; }
    public bool ValidateNames { get; set; }

    // Events
    public event EventHandler Disposed;
    public event CancelEventHandler FileOk;
    public event EventHandler HelpRequest;

    // Methods
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public virtual void Dispose();
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public System.IO.Stream OpenFile();
    public virtual void Reset();
    public System.Windows.Forms.DialogResult ShowDialog();
    public virtual string ToString();
} // end of System.Windows.Forms.OpenFileDialog
```

De ces propriétés et méthodes nous retiendrons les suivantes :

string Filter	les types de fichiers proposés dans la liste déroulante des types de fichiers de la boîte de dialogue
int FilterIndex	le n° du type de fichier proposé par défaut dans la liste ci-dessus. Commence à 0.
string InitialDirectory	le dossier présenté initialement pour la recherche du fichier à ouvrir
string FileName	le nom du fichier à ouvrir indiqué par l'utilisateur
DialogResult.ShowDialog()	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 proprité **FileName**
4 boutons **Ouvrir / Annuler**. Si le bouton *Ouvrir* est utilisé, la fonction *ShowDialog* rend le résultat **DialogResult.OK**

La procédure d'ouverture peut s'écrire ainsi :

```
    private void btnCharger_Click(object sender, System.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
          txtTexte.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
          try{fichier.Close();} catch (Exception){}
        }//finally
      }//if
    }
```

- On fixe le dossier initial au dossier qui contient l'exécutable de l'application :

```
      saveFileDialog1.InitialDirectory=Application.ExecutablePath;
```

- On fixe les types de fichiers à présenter

```
      saveFileDialog1.Filter = "Fichiers texte (*.txt)|*.txt|Tous les fichiers (*.*)|*.*";
```

- On fixe le type de fichier à présenter au début

```
      saveFileDialog1.FilterIndex = 0;
```

Ici, ce sont les fichiers de type *.txt qui seront présentés tout d'abord à l'utilisateur.

- La boîte de dialogue est affichée et son résultat récupéré

```
if(openFileDialog1.ShowDialog() == DialogResult.OK) {
```

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 ouvrir 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 *Ouvrir* pour quitter la boîte de dialogue.

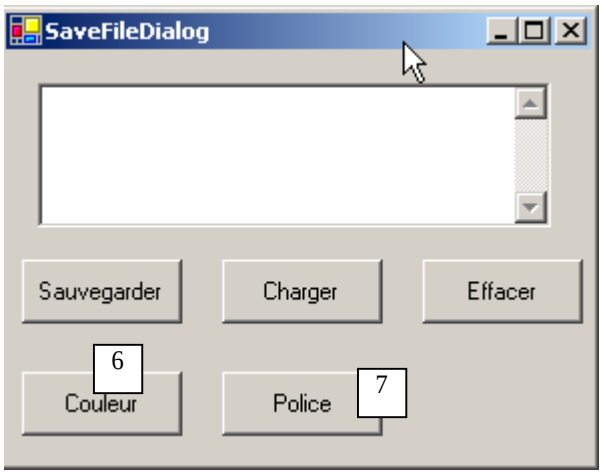
- Ceci fait, le nom du fichier à créer est maintenant dans la propriété *FileName* de l'objet openFileDialog1. On est alors ramené à la lecture classique d'un fichier texte. On notera la méthode qui permet de lire la totalité d'un fichier :

```
txtTexte.Text=fichier.ReadToEnd();
```

- le contenu du fichier est mis dans le *TextBox txtTexte.* On gère les exceptions qui peuvent se produire.

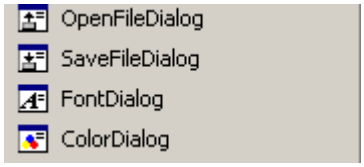
4.7.2 Boîtes de dialogue FontColor et ColorDialog

Nous continuons l'exemple précédent en présentant deux nouveaux boutons :



N°	type	nom	rôle
6	Button	btnCouleur	pour fixer la couleur des caractères du TextBox
7	Button	btnPolice	pour fixer la police de caractères du TextBox

Nous déposons sur le formulaire un contrôle *ColorDialog* et un contrôle *FontDialog* :

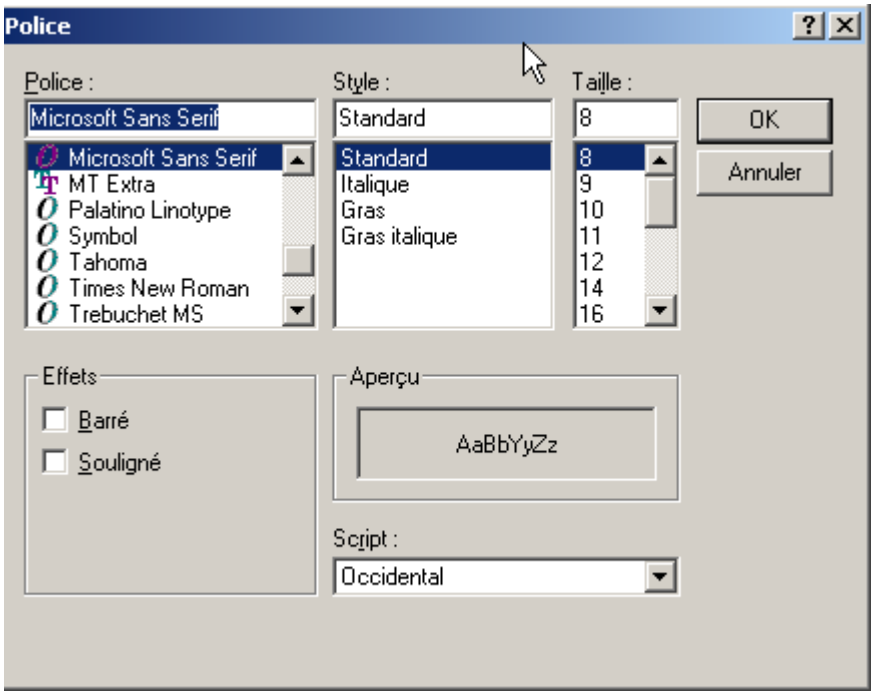


Les classes *FontDialog* et *ColorDialog* ont une méthode *ShowDialog* analogue à la méthode *ShowDialog* des classes *OpenFileDialog* et *SaveFileDialog.*

La méthode *ShowDialog* de la classe *ColorDialog* permet de choisir une couleur :

Si l'utilisateur quitte la boîte de dialogue avec le bouton *OK*, le résultat de la méthode *ShowDialog* est *DialogResult.OK* et la couleur choisie est dans la propriété *Color* de l'objet *ColorDialog* utilisé.

La méthode *ShowDialog* de la classe *FontDialog* permet de choisir une police de caractères :



Si l'utilisateur quitte la boîte de dialogue avec le bouton *OK*, le résultat de la méthode *ShowDialog* est *DialogResult.OK* et la police choisie est dans la propriété *Font* de l'objet *FontDialog* utilisé.

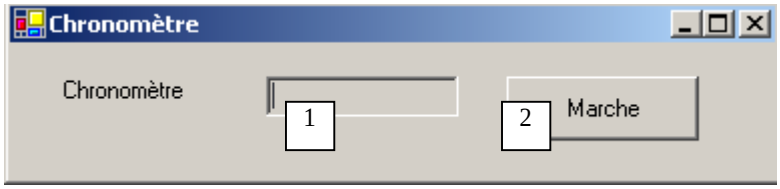
Nous avons les éléments pour traiter les clics sur les boutons *Couleur* et *Police* :

```
    private void btnCouleur_Click(object sender, System.EventArgs e) {
      // choix d'une couleur de texte
      if(colorDialog1.ShowDialog()==DialogResult.OK){
        // on change la propriété forecolor du TextBox
        txtTexte.ForeColor=colorDialog1.Color;
      }//if
    }

    private void btnPolice_Click(object sender, System.EventArgs e) {
      // choix d'une police de caractères
      if(fontDialog1.ShowDialog()==DialogResult.OK){
        // on change la propriété font du TextBox
        txtTexte.Font=fontDialog1.Font;
```

```
      }//if
    }
```

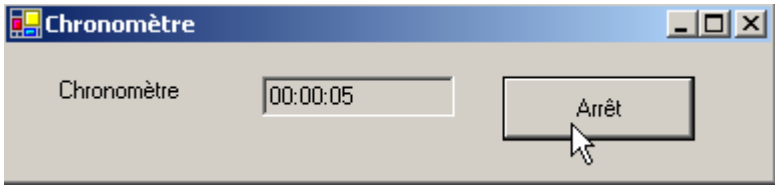
4.7.3 Timer

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

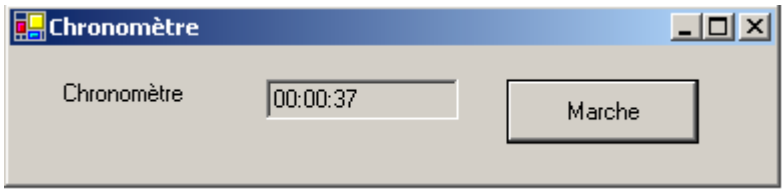


n°	Type	Nom	Rôle
1	TextBox ReadOnly=true	txtChrono	affiche un chronomètre
2	Button	btnArretMarche	bouton Arrêt/Marche du chronomètre
3	Timer	timer1	composant émettant ici un événement toutes les secondes

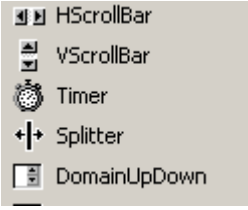
Le chronomètre en marche :



Le chronomètre arrêté :



Pour changer toutes les secondes le contenu du TextBox *txtChrono*, 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* :



Une fois ce composant installé sur le formulaire (dans la partie des composants non visuels), un objet de type *Timer* est créé dans le constructeur du formulaire. La classe *System.Windows.Forms.Timer* est définie comme suit :

```
// from module
'c:\winnt\assembly\gac\system.windows.forms\1.0.2411.0__b77a5c561934e089\system.windows.forms.dll'
public class System.Windows.Forms.Timer :
    System.ComponentModel.Component,
    System.ComponentModel.IComponent,
    IDisposable
{

    // Fields

    // Constructors
    public Timer();
    public Timer(System.ComponentModel.IContainer container);

    // Properties
    public IContainer Container { get; }
    public bool Enabled { virtual get; virtual set; }
```

```
    public int Interval { get; set; }
    public ISite Site { virtual get; virtual set; }

    // Events
    public event EventHandler Disposed;
    public event EventHandler Tick;

    // Methods
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public virtual void Dispose();
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public void Start();
    public void Stop();
    public virtual string ToString();
} // end of System.Windows.Forms.Timer
```

Les propriétés suivantes nous suffisent ici :

Interval	nombre de millisecondes au bout duquel un événement *Tick* est émis.
Tick	l'événement produit à la fin de *Interval* millisecondes
Enabled	rend le timer actif (true) ou inactif (false)

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 suivante :

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

Le libellé du bouton Arret/Marche est soit "Arrêt" soit "Marche". On est donc obligé de faire un test sur ce libellé pour savoir quoi faire.

- dans le cas de "Marche", on note l'heure de début dans une variable qui est une variable globale de l'objet formulaire, le timer est lancé (Enabled=true) et le libellé du bouton passe à "Arrêt".
- dans le cas de "Arrêt", on arrête le timer (Enabled=false) et on passe le libellé du bouton à "Marche".

```
  public class Form1 : System.Windows.Forms.Form {
    private System.Windows.Forms.Timer timer1;
    private System.Windows.Forms.Button btnArretMarche;
    private System.ComponentModel.IContainer components;
    private System.Windows.Forms.TextBox txtChrono;
    private System.Windows.Forms.Label label1;

    // variables d'instance
    private DateTime début;
```

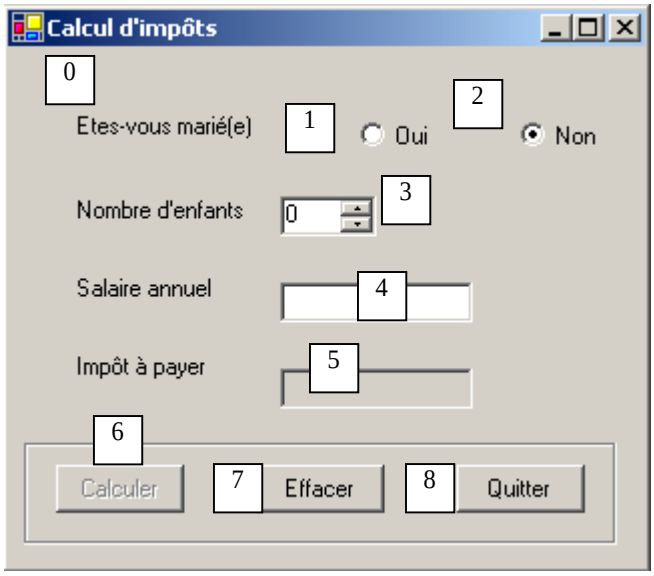
L'attribut *début* ci-dessus est connu dans toutes les méthodes de la classe. Il nous reste à traiter l'événement *Tick* sur l'objet *timer1*, événement qui se produit toutes les secondes :

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

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. 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.

4.8 L'exemple IMPOTS

On reprend l'application IMPOTS déjà traitée deux fois. Nous y ajoutons maintennat une interface graphique :

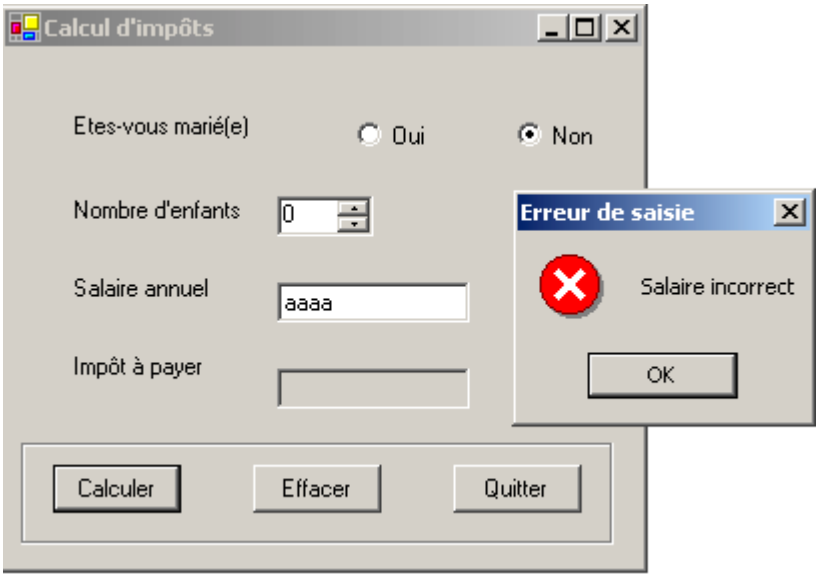


Les contrôles sont les suivants

n°	type	nom	rôle
1	RadioButton	rdOui	coché si marié
2	RadioButton	rdNon	coché si pas marié
3	NumericUpDown	incEnfants	nombre d'enfants du contribuable Minimum=0, Maximum=20, Increment=1
4	TextBox	txtSalaire	salaire annuel du contribuable en F
5	TextBox	txtImpots	montant de l'impôt à payer ReadOnly=true
6	Button	btnCalculer	lance le calcul de l'impôt
7	Button	btnEffacer	remet le formulaire dans son état initial lors du chargement
8	Button	btnQuitter	pour quitter l'application

Règles de fonctionnement

- le bouton Calculer reste éteint tant qu'il n'y a rien dans le champ du salaire
- si lorsque le calcul est lancé, il s'avère que le salaire est incorrect, l'erreur est signalée :



Le programme est donné ci-dessous. Il utilise la classe *impot* créée dans le chapitre sur les classes. Une partie du code produit automatiquement pas VS.NET n'a pas été ici reproduit.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.RadioButton rdOui;
    private System.Windows.Forms.RadioButton rdNon;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.TextBox txtSalaire;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.Label label4;
    private System.Windows.Forms.GroupBox groupBox1;
    private System.Windows.Forms.Button btnCalculer;
    private System.Windows.Forms.Button btnEffacer;
    private System.Windows.Forms.Button btnQuitter;
    private System.Windows.Forms.TextBox txtImpots;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;
    private System.Windows.Forms.NumericUpDown incEnfants;

    // tableaux de données nécessaires au calcul de l'impôt
    private decimal[] limites=new decimal[]
{12620M,13190M,15640M,24740M,31810M,39970M,48360M,55790M,92970M,127860M,151250M,172040M,195000M,0M};
    private decimal[] coeffR=new decimal[]
{0M,0.05M,0.1M,0.15M,0.2M,0.25M,0.3M,0.35M,0.4M,0.45M,0.55M,0.5M,0.6M,0.65M};
    private decimal[] coeffN=new decimal[]
{0M,631M,1290.5M,2072.5M,3309.5M,4900M,6898.5M,9316.5M,12106M,16754.5M,23147.5M,30710M,39312M,49062M};
    // objet impôt
    impôt objImpôt=null;

    public Form1()
    {
      //
      InitializeComponent();
      // initialisation du formulaire
      btnEffacer_Click(null,null);
      btnCalculer.Enabled=false;
      // création d'un objet impôt
      try{
        objImpôt=new impôt(limites,coeffR,coeffN);
      }catch (Exception ex){
        MessageBox.Show("Impossible de créer l'objet impôt
("+ex.Message+")","Erreur",MessageBoxButtons.OK,MessageBoxIcon.Error);
        // on inhibe le champ de saisie du salaire
        txtSalaire.Enabled=false;
      }//try-catch
    }//constructeur


    protected override void Dispose( bool disposing )
    {
....
    }
```

```
    #region Windows Form Designer generated code
    private void InitializeComponent()
    {
....

    }
    #endregion

    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }

    private void btnEffacer_Click(object sender, System.EventArgs e)
    {
      // raz du formulaire
      incEnfants.Value=0;
      txtSalaire.Text="";
      txtImpots.Text="";
      rdNon.Checked=true;
    }

    private void txtSalaire_TextChanged(object sender, System.EventArgs e)
    {
      // état du bouton Calculer
      btnCalculer.Enabled=txtSalaire.Text.Trim()!="";
    }

    private void btnQuitter_Click(object sender, System.EventArgs e)
    {
      // fin application
      Application.Exit();
    }

    private void btnCalculer_Click(object sender, System.EventArgs e)
    {
      // le salaire est-il correct ?
      int intSalaire=0;
      try
      {
        // récupération du salaire
        intSalaire=int.Parse(txtSalaire.Text);
        // il doit être >=0
        if(intSalaire<0) throw new Exception("");
      } catch(Exception ex){
        // msg d'erreur
        MessageBox.Show(this,"Salaire incorrect","Erreur de
saisie",MessageBoxButtons.OK,MessageBoxIcon.Error);
        // focus sur champ erroné
        txtSalaire.Focus();
        // sélection du texte du champ de saisie
        txtSalaire.SelectAll();
        // retour à l'interface visuelle
        return;
      }//try-catch
      // le salaire est correct - on calcule l'impôt
      txtImpots.Text=""+(int)objImpôt.calculer(rdOui.Checked,(int)incEnfants.Value,intSalaire);
    }//btnCalculer_Click
  }//classe
```

5. Gestion d'événements

Nous avons dans le chapitre précédent abordé la notion d'événements liés à des composants. Nous voyons maintenant comment créer des événements dans nos propres classes.

5.1 Objets delegate

L'instruction

```
  public delegate int opération(int n1, int n2);
```

définit un type appelé *opération* qui est en fait un prototype de fonction acceptant deux entiers et rendant un entier. C'est le mot clé `delegate` qui fait de *opération* une définition de prototype de fonction.

Une variable *op* de type *opération* aura pour rôle d'enregistrer une liste de fonctions correspondant au prototype *opération* :

int f1(int,int)
int f2(int,int)
...
int fn(int,int)

L'enregistrement d'une fonction *fi* dans la variable *op* se fait par *op=new opération(fi)*. Pour ajouter une fonction *fj* à la liste des fonctions déjà enregistrées, on écrit *op+= new opération(fj)*. Pour enlever une fonction *fk* déjà enregistrée on écrit *op-=new opération(fk)*. Si dans notre exemple on écrit *n=op(n1,n2)*, l'ensemble des fonctions enregistrées dans la variable *op* seront exécutées avec les paramètres *n1* et *n2*. Le résultat *n* récupéré sera celui de la dernière fonction exécutée. Il n'est pas possible d'obtenir les résultats produits par l'ensemble des fonctions. Pour cette raison, si on enregistre une liste de fonctions dans une fonction déléguée, celles-ci rendent le plus souvent le résultat *void*.

Considérons l'exemple suivant :

```
//fonctions déléguées

using System;

public class class1{

  // définition d'un prototype de fonction
  // accepte 2 entiers en paramètre et rend un entier
  public delegate int opération(int n1, int n2);

  // deux méthodes d'instance correspondant au prototype
  public int ajouter(int n1, int n2){
    Console.Out.WriteLine("ajouter("+n1+","+n2+")");
    return n1+n2;
  }//ajouter

  public int soustraire(int n1, int n2){
    Console.Out.WriteLine("soustraire("+n1+","+n2+")");
    return n1-n2;
  }//soustraire

  // une méthode statique correspondant au prototype
  public static int augmenter(int n1, int n2){
    Console.Out.WriteLine("augmenter("+n1+","+n2+")");
    return n1+2*n2;
  }//augmenter

  // programme de test
  public static void Main(){

    // on définit un objet de type opération pour y enregistrer des fonctions
    // on enregistre la fonction statique augmenter
    opération op=new opération(class1.augmenter);
    // on exécute le délégué
    int n=op(4,7);
    Console.Out.WriteLine("n="+n);

    // création d'un objet c1 de type class1
    class1 c1=new class1();
    // on enregistre dans le délégué la méthode ajouter de c1
```

```
    op=new class1.opération(c1.ajouter);
    // exécution de l'objet délégué
    n=op(2,3);
    Console.Out.WriteLine("n="+n);
    // on enregistre dans le délégué la méthode soustraire de c1
    op=new class1.opération(c1.soustraire);
    n=op(2,3);
    Console.Out.WriteLine("n="+n);
    //enregistrement de deux fonctions dans le délégué
    op=new class1.opération(c1.ajouter);
    op+=new class1.opération(c1.soustraire);
    // exécution de l'objet délégué
    op(0,0);
    // on retire une fonction du délégué
    op-=new opération(c1.soustraire);
    // on exécute le délégué
    op(1,1);
  }//Main
}//classe
```

Les résultats de l'exécution sont les suivants :

```
augmenter(4,7)
n=18
ajouter(2,3)
n=5
soustraire(2,3)
n=-1
ajouter(0,0)
soustraire(0,0)
ajouter(1,1)
```

5.2 Gestion d'événements

A quoi peut servir un objet de type *delegate* ?

Comme nous le verrons dans l'exemple suivant, cela sert surtout à la gestion des événements. Une classe *C1* peut générer des événements *evti*. Lors d'un événement *evti*, un objet de type *C1* lancera l'exécution d'un objet *evtiDéclenché* de type *delegate*. Toutes les fonctions enregistrées dans l'objet delegate *evtiDéclenché* seront alors exécutées. Si un objet *C2* utilisant un objet *C1* veut être averti de l'occurrence de l'événement *evti* sur l'objet *C1*, il enregistrera l'une de ses méthodes *C2.f* dans l'objet délégué *C1.evtiDéclenché* de l'objet *C1* afin que la méthode *C2.f* soit exécutée à chaque fois que l'événement *evti* se produit sur l'objet *C1*. Comme l'objet délégué *C1.evtiDéclenché* peut enregistrer plusieurs fonctions, différents objets *Ci* pourront s'enregistrer auprès du délégué *C1.evtiDéclenché* pour être prévenus de l'événement *evti* sur *C1*.

Considérons l'exemple suivant :

```
//gestion d'événements

using System;

public class myEventArgs : EventArgs{
  // la classe d'un évt

  // attribut
  private string _saisie;

  // constructeur
  public myEventArgs(string saisie){
    _saisie=saisie;
  }//constructeur

  // propriété saisie en lecture seule
  public override string ToString(){
    return _saisie;
  }//ToString
}// classe myEventArgs

public class émetteur{
  // la classe émettrice d'un évt

  // attribut
  private string _nom; // nom de l'émetteur

  // constructeur
  public émetteur(string nom){
    _nom=nom;
  }//constructeur

  // ToString
  public override string ToString(){
    return _nom;
```

```
  }//ToString

  // le prototype des fonctions chargées de traiter l'évt
  public delegate void _evtHandler(object sender, myEventArgs evt);

  // le pool des gestionnaires d'évts
  public event _evtHandler evtHandler;

  // méthode de demande d'émission d'un évt
  public void envoyerEvt(myEventArgs evt){
    // on prévient tous les abonnés
    // on fait comme si l'évt provenait d'ici
    evtHandler(this,evt);
  }//envoyerEvt
}//émetteur

// une classe de traitement de l'évt
public class souscripteur{

  // attribut
  private string nom; // nom du souscripteur
  private émetteur sender; // l'émetteur des évts

  // constructeur
  public souscripteur(string nom, émetteur e){
    // on note le nom du souscripteur
    this.nom=nom;
    // et l'émetteur des évts
    this.sender=e;
    // on s'inscrit pour recevoir les evts de l'émetteur e
    e.evtHandler+=new émetteur._evtHandler(traitementEvt);
  }//souscripteur

  // gestionnaire d'évt
  public void traitementEvt(object sender, myEventArgs evt){
    // affichage evt
    Console.Out.WriteLine("L'objet [" + sender + "] a signalé la saisie erronée [" + evt + "] au
souscripteur [" + nom + "]");
  }//traitement1
}// classe souscripteur

// un programme de test
public class test{
  public static void Main(){
    // création d'un émetteur d'evts
    émetteur émetteur1=new émetteur("émetteur1");
    // création d'un tableau de souscripteurs
    // pour les évts émis par émetteur1
    souscripteur[] souscripteurs=new souscripteur[] {new souscripteur("s1",émetteur1), new
souscripteur("s2",émetteur1)};
    // on lit une suite d'entiers au clavier
    // dès que l'un est erroné, on émet un évt
    Console.Out.Write("Nombre entier (rien pour arrêter) : ");
    string saisie=Console.In.ReadLine().Trim();
    // tant que la ligne saisie est non vide
    while(saisie!=""){
      // la saisie est-elle un nombre entier ?
      try{
        int n=int.Parse(saisie);
      }catch(Exception){
        // ce n'est pas un entier
        // on prévient tout le monde
        émetteur1.envoyerEvt(new myEventArgs(saisie));
      }//try-catch
      // nouvelle saisie
      Console.Out.Write("Nombre entier (rien pour arrêter) : ");
      saisie=Console.In.ReadLine().Trim();
    }//while

    // fin
    Environment.Exit(0);
  }//Main
}//classe test
```

Le code précédent est assez complexe. Détaillons-le. Dans une gestion d'événements, il y a un émetteur d'événements (*sender*) qui envoie le détail des événements (*EventArgs*) à des souscripteurs qui se sont déclarés intéressés par les événements en question. La classe émettrice des événements est ici la suivante :

```
public class émetteur{
  // la classe émettrice d'un évt

  // attribut
  private string _nom; // nom de l'émetteur

  // constructeur
  public émetteur(string nom){
    _nom=nom;
  }//constructeur

  // ToString
  public override string ToString(){
```

```
    return _nom;
  }//ToString

  // le prototype des fonctions chargées de traiter l'évt
  public delegate void _evtHandler(object sender, myEventArgs evt);

  // le pool des gestionnaires d'évts
  public event _evtHandler evtHandler;

  // méthode de demande d'émission d'un évt
  public void envoyerEvt(myEventArgs evt){
    // on prévient tous les abonnés
    // on fait comme si l'évt provenait d'ici
    evtHandler(this,evt);
  }//envoyerEvt
}//émetteur
```

Chaque objet *émetteur* a un *nom* fixé par construction. La méthode *ToString* a été redéfinie de telle façon que lorsqu'on transforme un objet *émetteur* en *string* c'est son nom qu'on récupère. La classe définit le prototype des fonctions qui peuvent traiter les événements qu'elle émet :

```
  public delegate void _evtHandler(object sender, myEventArgs evt);
```

Le premier argument d'une fonction de traitement d'un événement sera l'objet qui émet l'événement. Le second argument sera de type *myEventArgs*, un objet qui donnera des détails sur l'événement et sur lequel nous reviendrons.

Une fois le prototype (delegate) des gestionnaires d'événements défini, il nous faut un objet pour enregistrer des fonctions correspondant à ce prototype et qui seront exécutées lorsque l'objet qui les enregistre recevra les paramètres permettant de les exécuter.

```
  public event _evtHandler evtHandler;
```

On notera le mot clé **event** qui impose au *delegate* d'être de type *void f(object, EventArgs)*. Les fonctions enregistrées dans *evtHandler* seront exécutées par une instruction *evtHandler(o,evt)* où *o* est de type *object* ou dérivé et *evt* de type *myEventArgs* ou dérivé. Cette exécution ne peut se produire que dans un objet de la classe qui définit l'événement *evtHandler*, c.a.d. un objet de type *émetteur*. Afin de pouvoir déclencher un événement de l'extérieur d'un objet *émetteur*, nous ajoutons à la classe la méthode *envoyerEvt* :

```
  // méthode de demande d'émission d'un évt
  public void envoyerEvt(myEventArgs evt){
    // on prévient tous les abonnés
    // on fait comme si l'évt provenait d'ici
    evtHandler(this,evt);
  }//envoyerEvt
```

Nous devons définir quel sera le type d'événements déclenchés par la classe *émetteur*. Nous avons pour cela défini la classe *myEventArgs* :

```
public class myEventArgs : EventArgs{
  // la classe d'un évt

  // attribut
  private string _saisie;

  // constructeur
  public myEventArgs(string saisie){
    _saisie=saisie;
  }//constructeur

  // propriété saisie en lecture seule
  public override string ToString(){
    return _saisie;
  }//ToString
}// classe myEventArgs
```

Nous avons vu que les fonctions de traitement des événements devaient correspondre au prototype *void f(object, EventArgs)*. *EventArgs* est une classe qui donne des informations sur l'événement qui s'est produit. Pour chaque nouveau type d'événement, on est donc amené à dériver cette classe. C'est ce qui est fait ici. Les événements qui vont nous intéresser sont des saisies au clavier erronées. On va demander à un utilisateur de taper des nombres entiers au clavier et dès qu'il tapera une chaîne qui ne représente pas un entier, on déclenchera un événement. Comme détail de l'événement, nous nous conterons de donner la saisie erronée. C'est le sens de l'attribut *_saisie* de la classe. Un objet *myEventArgs* est donc construit avec pour paramètre la saisie erronée. On redéfinir par ailleurs la méthode *ToString* pour que lorsqu'on transforme un objet *myEventArgs* en chaîne, on obtienne l'attribut *_saisie*.

Nous avons défini l'émetteur des événements et le type d'événements qu'il émet. Nous définissons ensuite le type des souscripteurs intéressés par ces événements.

```
// une classe de traitement de l'évt
public class souscripteur{

  // attribut
  private string nom; // nom du souscripteur
```

```
  private émetteur sender; // l'émetteur des évts

  // constructeur
  public souscripteur(string nom, émetteur e){
    // on note le nom du souscripteur
    this.nom=nom;
    // et l'émetteur des évts
    this.sender=e;
    // on s'inscrit pour recevoir les evts de l'émetteur e
    e.evtHandler+=new émetteur._evtHandler(traitementEvt);
  }//souscripteur

  // gestionnaire d'évt
  public void traitementEvt(object sender, myEventArgs evt){
    // affichage evt
    Console.Out.WriteLine("L'objet [" + sender + "] a signalé la saisie erronée [" + evt + "] au
souscripteur [" + nom + "]");
  }//traitement1
}// classe souscripteur
```

Un souscripteur sera défini par deux paramètres : son nom (attribut *nom*) et l'objet *émetteur* dont il veut traiter les événements (attribut *sender*). Ces deux paramètres seront passés au constructeur de l'objet. Au cours de cette même construction, le souscripteur s'abonne aux événements de l'émetteur :

```
    e.evtHandler+=new émetteur._evtHandler(traitementEvt);
```

La fonction enregistrée auprès de l'émetteur est *traitementEvt*. Cette méthode de la classe *souscripteur* affiche les deux arguments qu'elle a reçues (*sender*, *evt*) ainsi que le nom du récepteur (*nom*).

Ont été définis, le type des événements produits, le type de l'émetteur de ces événements, le type des souscripteurs. Il ne nous reste plus qu'à les mettre en oeuvre :

```
// un programme de test
public class test{
  public static void Main(){
    // création d'un émetteur d'evts
    émetteur émetteur1=new émetteur("émetteur1");
    // création d'un tableau de souscripteurs
    // pour les évts émis par émetteur1
    souscripteur[] souscripteurs=new souscripteur[] {new souscripteur("s1",émetteur1), new
souscripteur("s2",émetteur1)};
    // on lit une suite d'entiers au clavier
    // dès que l'un est erroné, on émet un évt
    Console.Out.Write("Nombre entier (rien pour arrêter) : ");
    string saisie=Console.In.ReadLine().Trim();
    // tant que la ligne saisie est non vide
    while(saisie!=""){
      // la saisie est-elle un nombre entier ?
      try{
        int n=int.Parse(saisie);
      }catch(Exception){
        // ce n'est pas un entier
        // on prévient tout le monde
        émetteur1.envoyerEvt(new myEventArgs(saisie));
      }//try-catch
      // nouvelle saisie
      Console.Out.Write("Nombre entier (rien pour arrêter) : ");
      saisie=Console.In.ReadLine().Trim();
    }//while

    // fin
    Environment.Exit(0);
  }//Main
}//classe test
```

Nous créons un objet *émetteur* :

```
    // création d'un émetteur d'evts
    émetteur émetteur1=new émetteur("émetteur1");
```

Nous créons un tableau de deux souscripteurs pour les événements émis par l'objet *émetteur1* :

```
    // création d'un tableau de souscripteurs
    // pour les évts émis par émetteur1
    souscripteur[] souscripteurs=new souscripteur[] {new souscripteur("s1",émetteur1), new
souscripteur("s2",émetteur1)};
```

Nous demandons à l'utilisateur de taper des nombres entiers au clavier. Dès qu'une saisie est erronée, nous demandons à émetteur1 d'envoyer un événement à ses souscripteurs :

```
        // on prévient tout le monde
        émetteur1.envoyerEvt(new myEventArgs(saisie));
```

L'événement envoyé est de type *myEventArgs* et contient la saisie erronée. Les deux souscripteurs devraient recevoir cet événement et le signaler. C'est ce que montre l'exécution qui suit.

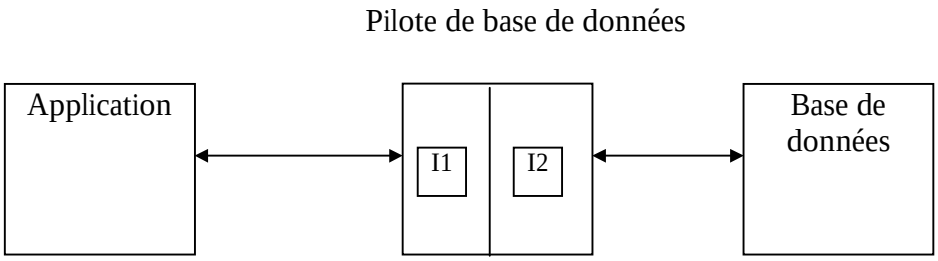
Les résultats de l'exécution sont les suivants :

```
E:\data\serge\MSNET\c#\événements\2>evt1
Nombre entier (rien pour arrêter) : 4
Nombre entier (rien pour arrêter) : a
L'objet [émetteur1] a signalé la saisie erronée [a] au souscripteur [s1]
L'objet [émetteur1] a signalé la saisie erronée [a] au souscripteur [s2]
Nombre entier (rien pour arrêter) : 1.6
L'objet [émetteur1] a signalé la saisie erronée [1.6] au souscripteur [s1]
L'objet [émetteur1] a signalé la saisie erronée [1.6] au souscripteur [s2]
Nombre entier (rien pour arrêter) :
```

6. Accès aux bases de données

6.1 Généralités

Il existe de nombreuses bases de données pour les plate-formes windows. Pour y accéder, les applications passent au travers de programmes appelés **pilotes** (drivers).



Dans le schéma ci-dessus, le pilote présente deux interfaces :

- l'interface I1 présentée à l'application
- l'interface I2 vers la base de données

Afin d'éviter qu'une application écrite pour une base de données B1 doive être ré-écrite si on migre vers une base de données B2 différente, un effort de normalisation a été fait sur l'interface I1. Si on utilise des bases de données utilisant des pilotes "normalisés", la base B1 sera fournie avec un pilote P1, la base B2 avec un pilote P2, et l'interface I1 de ces deux pilotes sera identique. Aussi n'aura-t-on pas à ré-écrire l'application. On pourra ainsi, par exemple, migrer une base de données ACCESS vers une base de données MySQL sans changer l'application.

Il existe deux types de pilotes normalisés :

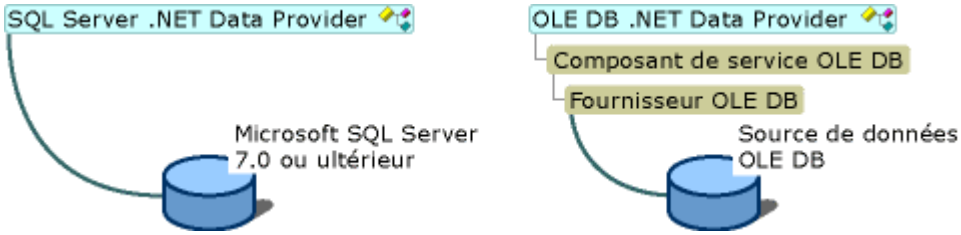
- les pilotes ODBC (Open DataBase Connectivity)
- les pilotes OLE DB (Object Linking and Embedding DataBase)

Les pilotes ODBC permettent l'accès à des bases de données. Les sources de données pour les pilotes OLE DB sont plus variées : bases de données, messageries, annuaires, ... Il n'y a pas de limite. Toute source de données peut faire l'objet d'un pilote Ole DB si un éditeur le décide. L'intérêt est évidemment grand : on a un accès uniforme à une grande variété de données.

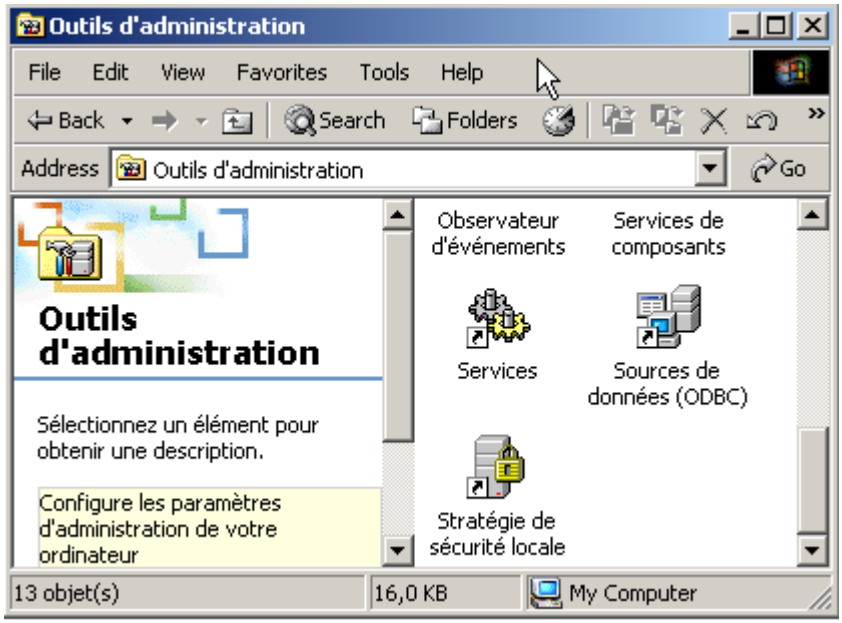
La plate-forme .NET est livrée avec deux types de classes d'accès aux données :

1. les classes SQL Server.NET
2. les classes Ole Db.NET

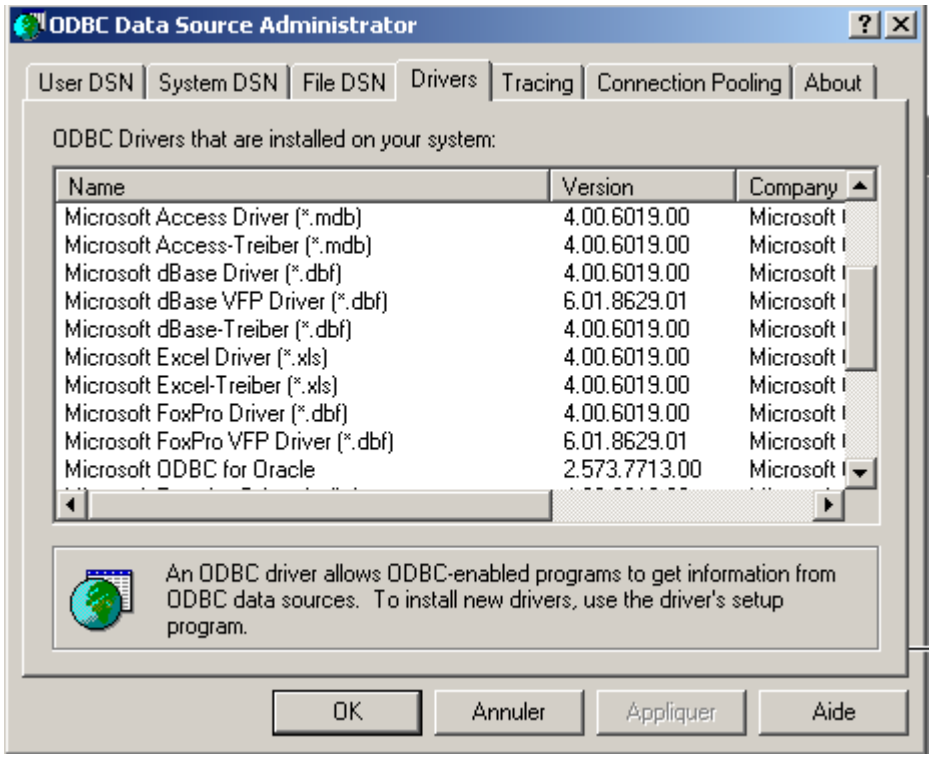
Les premières classes permettent un accès direct au SGBD SQL Server de Microsoft sans pilote intermédiaire. Les secondes permettent l'accès aux sources de données OLE DB.



La plate-forme .NET est fournie (mai 2002) avec trois pilotes OLE DB pour respectivement : SQL Server, Oracle et Microsoft Jet (Access). Si on veut travailler avec une base de données ayant un pilote ODBC mais pas de pilote OLE DB, on ne peut pas. Ainsi on ne peut pas travailler avec le SGBD MySQL qui (mai 2002) ne fournit pas de pilote OLE DB. Il existe cependant une série de classes permettant l'accès aux sources de données ODBC, les classes **odbc.net**. Elles ne sont pas livrées en standard avec le SDK et il faut aller les chercher sur le site de Microsoft. Dans les exemples qui vont suivre, nous utiliserons surtout ces classes ODBC car la plupart des bases de données sous windows sont livrées avec un tel pilote. Voici par exemple, une liste des pilotes ODBC installés sur une machine Win 2000 (*Menu Démarrer/Paramètres/Panneau de configuration/Outils d'administration*) :



On choisit l'icône *Source de données ODBC* :



6.2 Les deux modes d'exploitation d'une source de données

La plate-forme .NET permet l'exploitation d'une source de données de deux manières différentes :

1. mode connecté
2. mode déconnecté

En mode **connecté**, l'application

1. ouvre une connexion avec la source de données
2. travaille avec la source de données en lecture/écriture
3. ferme la connexion

En mode **déconnecté**, l'application

1. ouvre une connexion avec la source de données

2. obtient une copie mémoire de tout ou partie des données de la source
3. ferme la connexion
4. travaille avec la copie mémoire des données en lecture/écriture
5. lorsque le travail est fini, ouvre une connexion, envoie les données modifiées à la source de données pour qu'elle les prenne en compte, ferme la connexion

Dans les deux cas, c'est l'opération d'exploitation et de mise à jour des données qui prend du temps. Imaginons que ces mises à jour soient faites par un utilisateur faisant des saisies, cette opération peut prendre des dizaines de minutes. Pendant tout ce temps, en mode connecté, la connexion avec la base est maintenue et les modifications immédiatement répercutées. En mode déconnecté, il n'y a pas de connexion à la base pendant la mise à jour des données. Les modifications sont faites uniquement sur la copie mémoire. Elles sont répercutées sur la source de données en une seule fois lorsque tout est terminé.

Quels sont les avantages et inconvénients des deux méthodes ?

- Une connexion est coûteuse en ressources système. S'il y a beaucoup de connexions simultanées, le mode déconnecté permet de réduire leurs durées à un minimum. C'est le cas des applications web ayant des milliers d'utilisateurs.
- L'inconvénient du mode déconnecté est la gestion délicate des mises à jour simultanées. L'utilisateur U1 obtient des données au temps T1 et commence à les modifier. Au temps T2, l'utilisateur U2 accède lui aussi à la source de données et obtient les mêmes données. Entre-temps l'utilisateur U1 a modifié certaines données mais ne les a pas encore transmises à la source de données. U2 travaille donc avec des données dont certaines sont erronées. Les classes .NET offrent des solutions pour gérer ce problème mais il n'est pas simple à résoudre.
- En mode connecté, la mise à jour simultanée de données par plusieurs utilisateurs ne pose normalement pas de problème. La connexion avec la base de données étant maintenue, c'est la base de données elle-même qui gère ces mises à jour simultanées. Ainsi Oracle verrouille une ligne de la base de données dès qu'un utilisateur la modifie. Elle restera verrouillée donc inaccessible aux autres utilisateurs jusqu'à ce que celui qui l'a modifiée valide (commit) sa modification ou l'abandonne (rollback).
- Si les données doivent circuler sur le réseau, le mode déconnecté est à choisir. Il permet d'avoir une photo des données dans un objet appelé **dataset** qui représente une base de données à lui tout seul. Cet objet peut circuler sur le réseau entre machines.

Nous étudions d'abord le mode connecté.

6.3 Accès aux données en mode connecté

6.3.1 Les bases de données de l'exemple

Nous considérons une base de données ACCESS appelée *articles.mdb* et n'ayant qu'une table appelée ARTICLES avec la structure suivante :

nom	type
code	code de l'article sur 4 caractères
nom	son nom (chaîne de caractères)
prix	son prix (réel)
stock_actuel	son stock actuel (entier)
stock_minimum	le stock minimum (entier) en-deça duquel il faut réapprovisionner l'article

Son contenu de départ est le suivant :

articles : Table

code	nom	prix	stock_actuel	stock_minimu
a300	vélo	2 500,00 F	10	5
b300	pompe	56,00 F	62	45
c300	arc	3 500,00 F	10	20
d300	flèches - lot de	780,00 F	12	20
e300	combinaison de	2 800,00 F	34	7
f300	bouteilles d'oxy	800,00 F	10	5

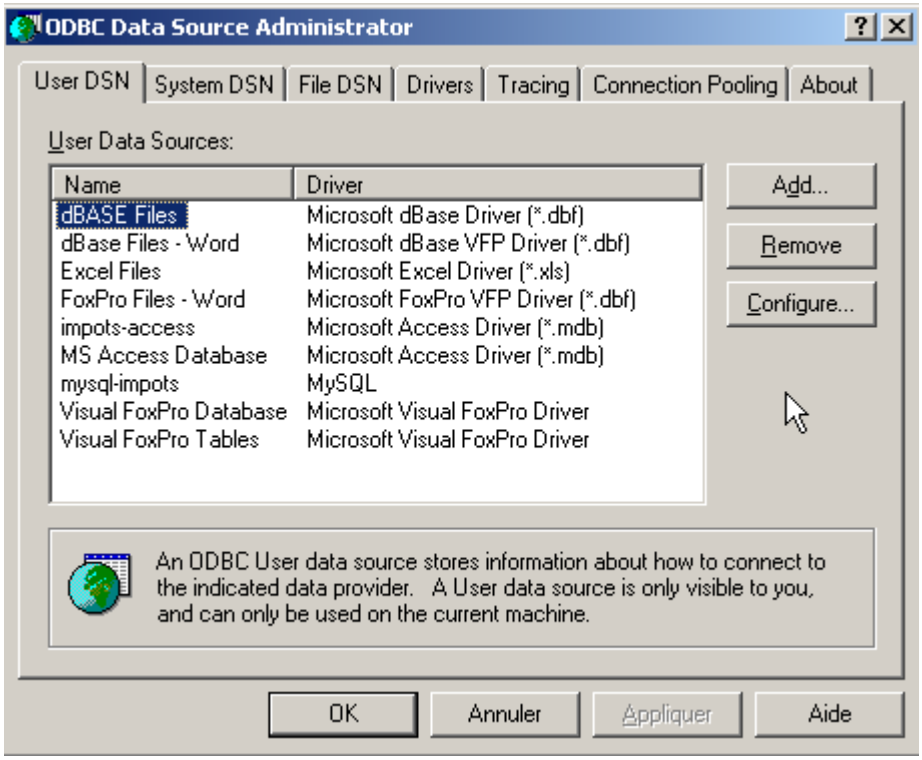
Nous utiliserons cette base aussi bien au travers d'un pilote ODBC qu'un pilote OLE DB afin de montrer la similitude des deux approches et parce que nous disposons de ces deux types de pilotes pour ACCESS.

Nous utiliserons également une base MySQL DBARTICLES ayant la même unique table ARTICLES, le même contenu et accédé au travers d'un pilote ODBC, afin de montrer que l'application écrite pour exploiter la base ACCESS n'a pas à être modifiée pour utiliser la base MySQL. La base DBARTICLES est accessible à un utilisateur appelé *admarticles* avec le mot de passe *mdparticles*. La copie d'écran suivante montre le contenu de la base MySQL :

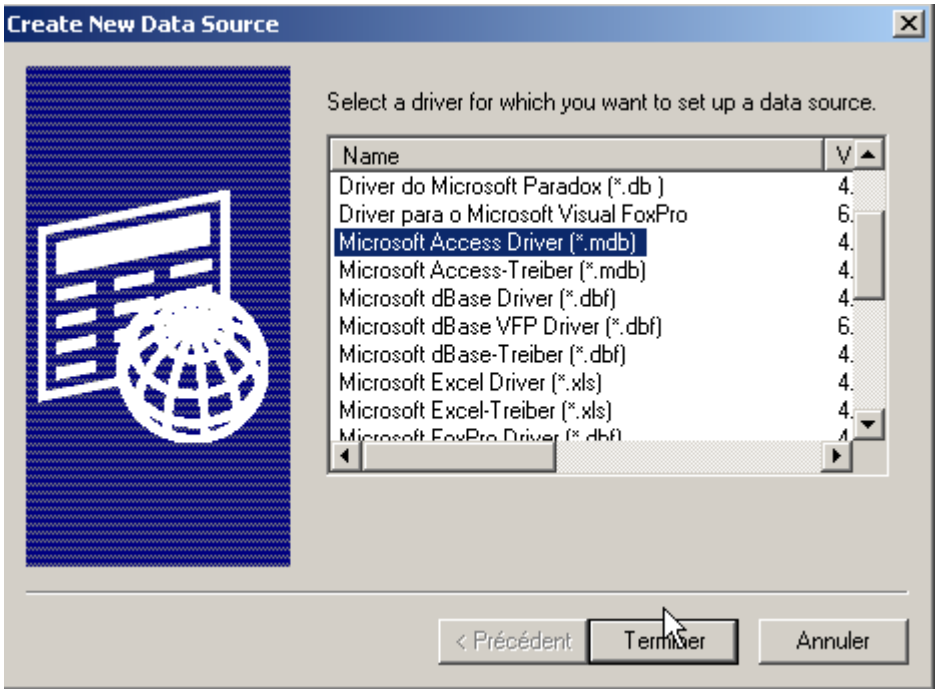
```
C:\mysql\bin>mysql --database=dbarticles --user=admarticles --password=mdparticles
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.23.49-max-debug

Type 'help' for help.


mysql> show tables;
+----------------------+
| Tables_in_dbarticles |
+----------------------+
| articles             |
+----------------------+
1 row in set (0.01 sec)

mysql> select * from articles;
+------+-------------------------------+------+--------------+---------------+
| code | nom                           | prix | stock_actuel | stock_minimum |
+------+-------------------------------+------+--------------+---------------+
| a300 | vÚlo                          | 2500 |           10 |             5 |
| b300 | pompe                         |   56 |           62 |            45 |
| c300 | arc                           | 3500 |           10 |            20 |
| d300 | flÞches - lot de 6            |  780 |           12 |            20 |
| e300 | combinaison de plongÚe        | 2800 |           34 |             7 |
| f300 | bouteilles d'oxygÞne          |  800 |           10 |             5 |
+------+-------------------------------+------+--------------+---------------+
6 rows in set (0.02 sec)

mysql> describe articles;
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| code          | text        | YES  |     | NULL    |       |
| nom           | text        | YES  |     | NULL    |       |
| prix          | double      | YES  |     | NULL    |       |
| stock_actuel  | smallint(6) | YES  |     | NULL    |       |
| stock_minimum | smallint(6) | YES  |     | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

mysql> exit
Bye
```

Pour définir la base ACCESS comme source de données ODBC, procédez comme suit :

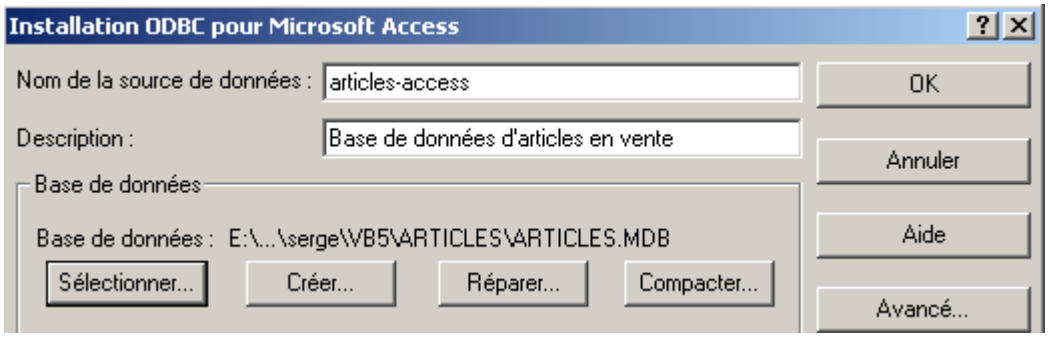
- activez l'administrateur de sources de données ODBC comme il a été montré plus haut et sélectionnez l'onglet User DSN (DSN=Data Source Name)



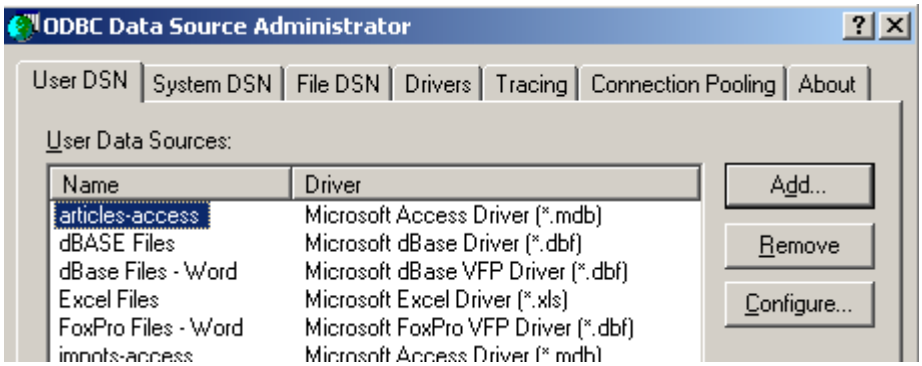
- ajoutez une source avec le bouton *Add* , indiquez que cette source est accessible via un pilote *Access* et faites *Terminer* :



- Donnez le nom *articles-access* à la source de données, mettez une description libre et utilisez le bouton *Sélectionner* pour désigner le fichier *.mdb* de la base. Terminez par *OK*.

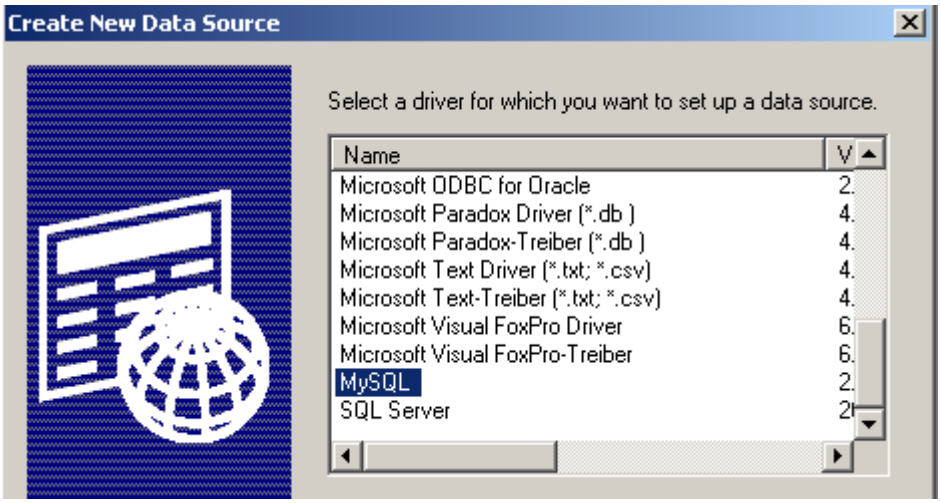


La nouvelle source de données apparaît alors dans la liste des sources DSN utilisateur :

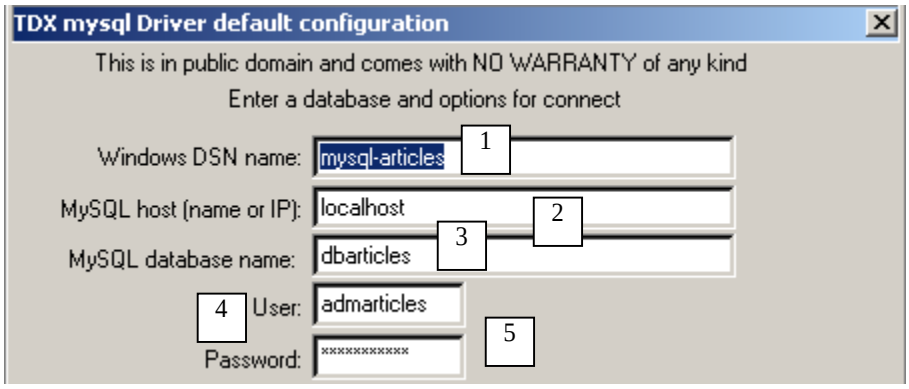


Pour définir la base MySQL DBARTICLES comme source de données ODBC, procédez comme suit :

- activez l'administrateur de sources de données ODBC comme il a été montré plus haut et sélectionnez l'onglet *User DSN*. Ajoutez une nouvelle source de données avec *Add* et sélectionnez le pilote ODBC de MySQL.



- Faites *Terminer*. Apparaît alors une page de configuration de la source MySQL :



- dans (1) on donne un nom à notre source de données ODBC
- dans (2) on indique la machine sur laquelle se trouve le serveur MySQL. Ici nous mettons *localhost* pour indiquer qu'il est sur la même machine que notre application. Si le serveur MySQL était sur une machine M distante, on mettrait là son nom et notre application fonctionnerait alors avec une base de données distante sans modification.
- dans (3) on met le nom de la base. Ici elle s'appelle DBARTICLES.
- dans (4) on met le login *admarticles* et dans (5) le mot de passe *mdparticles*.

6.3.2 Utilisation d'un pilote ODBC

Dans une application utilisant une base de données en mode connecté, on trouvera généralement les étapes suivantes :

1. Connexion à la base de données
2. Émissions de requêtes SQL vers la base
3. Réception et traitement des résultats de ces requêtes
4. Fermeture de la connexion

Les étapes 2 et 3 sont réalisées de façon répétée, la fermeture de connexion n'ayant lieu qu'à la fin de l'exploitation de la base. C'est un schéma relativement classique dont vous avez peut-être l'habitude si vous avez exploité une base de données de façon interactive. Ces étapes sont les mêmes que la base soit utilisée au travers d'un pilote ODBC ou d'un pilote OLE DB. Nous présentons ci-dessous un exemple avec les classes .NET de gestion des sources de données ODBC. Le programme s'appelle *liste* et admet comme paramètre le nom DSN d'une source de données ODBC ayant une table ARTICLES. Il affiche alors le contenu de cette table :

```
E:\data\serge\MSNET\c#\adonet\5>liste
syntaxe : pg dsnArticles

E:\data\serge\MSNET\c#\adonet\5>liste articles-access

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                           2500 10 5
b300 pompe                          56 62 45
c300 arc                            3500 10 20
d300 flèches - lot de 6             780 12 20
e300 combinaison de plongée         2800 34 7
f300 bouteilles d'oxygène           800 10 5

E:\data\serge\MSNET\c#\adonet\5>liste mysql-artices
Erreur d'exploitation de la base de données (ERROR [IM002] [Microsoft][ODBC Driver Manager] Data source
name not found and no default driver specified)

E:\data\serge\MSNET\c#\adonet\5>liste mysql-articles

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                           2500 10 5
b300 pompe                          56 62 45
c300 arc                            3500 10 20
d300 flèches - lot de 6             780 12 20
e300 combinaison de plongée         2800 34 7
f300 bouteilles d'oxygène           800 10 5
```

Sur les résultats ci-dessus, nous voyons que le programme a listé aussi bien le contenu de la base ACCESS que de la base MySQL. Etudions maintenant le code de ce programme :

```
using System;
using System.Data;
using Microsoft.Data.Odbc;

class listArticles
  {
    static void Main(string[] args){

      // application console
      // affiche le contenu d'une table ARRTICLES d'une base DSN
      // dont le nom est passé en paramètre

      const string syntaxe="syntaxe : pg dsnArticles";
```

```
      const string tabArticles="articles"; // la table des articles

      // vérification des paramètres
      // a-t-on 2 paramètres
      if(args.Length!=1){
        // msg d'erreur
        Console.Error.WriteLine(syntaxe);
        // fin
        Environment.Exit(1);
      }//if
      // on récupère le paramètre
      string dsnArticles=args[0];   // la base DSN

      // préparation de la connexion à la bd
      OdbcConnection articlesConn=null; // la connexion
      OdbcDataReader myReader=null;     // le lecteur de données

      try{
        // on tente d'accéder à la base de données
        // chaîne de connexion à la base
        string connectString="DSN="+dsnArticles+";";
        articlesConn = new OdbcConnection(connectString);
        articlesConn.Open();
        // exécution d'une commande SQL
        string sqlText = "select * from " + tabArticles;
        OdbcCommand myOdbcCommand = new OdbcCommand(sqlText);
        myOdbcCommand.Connection = articlesConn;
        myReader=myOdbcCommand.ExecuteReader();
        // Exploitation de la table récupérée
        // affichage des colonnes
        string ligne="";
        int i;
        for(i=0;i<myReader.FieldCount-1;i++){
          ligne+=myReader.GetName(i)+",";
        }//for
        ligne+=myReader.GetName(i);
        Console.Out.WriteLine("\n"+"".PadLeft(ligne.Length,'-')+"\n"+ligne+"\n"+"".PadLeft(ligne.Length,'-
')+"\n");
        // affichage des données
        while (myReader.Read()) {
          // exploitation ligne courante
          ligne="";
          for(i=0;i<myReader.FieldCount;i++){
            ligne+=myReader[i]+" ";
          }
          Console.WriteLine(ligne);
        }//while
      }//try
      catch(Exception ex){
        Console.Error.WriteLine("Erreur d'exploitation de la base de données "+ex.Message+")");
        Environment.Exit(2);
      }//catch
      finally{
        try{
          // fermeture lecteur
          myReader.Close();
          // fermeture connexion
          articlesConn.Close();
        }catch{}
      }//finally

      // fin
      Environment.Exit(0);
    }//main
  }//classe
```

Les classes de gestion des sources ODBC se trouvent dans l'espace de noms *Microsoft.Data.Odbc* qu'on doit donc importer. Par ailleurs, un certain nombre de classes se trouve dans l'espace de noms *System.Data*.

```
using System.Data;
using Microsoft.Data.Odbc;
```

L'espace de noms *Microsoft.Data.Odbc* se trouve dans "l'assembly" *microsoft.data.odbc.dll*. Aussi pour compiler le programme précédent écrit-on :

```
csc /r:microsoft.data.odbc.dll liste.cs
```

6.3.2.1 La phase de connexion

Une connexion ODBC utilise la classe *OdbcConnection*. Le constructeur de cette classe admet comme paramètre ce qu'on appelle une **chaîne de connexion**. Celle-ci est une chaîne de caractères qui définit tous les paramètres nécessaires pour que la connexion à la base de données puisse se faire. Ces paramètres peuvent être très nombreux et donc la chaîne complexe. La chaîne a la forme *"param1=valeur1;param2=valeur2;...;paramj=valeurj;"*.Voici quelques paramètres *paramj* possibles :

uid nom d'un utilisateur qui va accéder à la base de données
password mot de passe de cet utilisateur
dsn nom DSN de la base si elle en a un
data source nom de la base de données accédée
...

Si on définit une source de données comme source de données ODBC à l'aide de l'administrateur de sources de données ODBC, ces paramètres ont déjà été donnés et enregistrés. Il suffit alors de passer le paramètre DSN qui donne le nom DSN de la source de données. C'est ce qui est fait ici :

```
OdbcConnection articlesConn=null; // la connexion

try{
  // on tente d'accéder à la base de données
  // chaîne de connexion à la base
  string connectString="DSN="+dsnArticles+";";
  articlesConn = new OdbcConnection(connectString);
  articlesConn.Open();
```

Une fois l'objet *OdbcConnection* construit, on ouvre la connexion avec la méthode *Open*. Cette ouverture peut échouer comme toute autre opération sur la base. C'est pourquoi l'ensemble du code d'accès à la base est-il dans un *try-catch*. Une fois la connexion établie, on peut émettre des requêtes SQL sur la base.

6.3.2.2 Émettre des requêtes SQL

Pour émettre des requêtes SQL, il nous faut un objet *Command*, ici plus exactement un objet *OdbcCommand* puisque nous utilisons une source de données ODBC. La classe *OdbcCommand* a plusieurs constructeurs :

- *OdbcCommand()* : crée un objet *Command* vide. Il faudra pour l'utiliser préciser ultérieurement diverses propriétés :
  - **CommandText** : le texte de la requête SQL à exécuter
  - **Connection** : l'objet *OdbcConnection* représentant la connexion à la base de données sur laquelle la requête sera faite
  - **CommandType** : le type de la requête SQL. Il y a trois valeurs possibles
    1. *CommandType.Text* : la propriété *CommandText* contient le texte d'une requête SQL (valeur par défaut)
    2. *CommandType.StoredProcedure* : la propriété *CommandText* contient le nom d'une procédure stockée dans la base
    3. *CommandType.TableDirect* : la propriété *CommandText* contient le nom d'une table T. Equivalent à *select * from T*. N'existe que pour les pilotes OLE DB.
- *OdbcCommand(string sqlText)* : le paramètre *sqlText* sera affecté à la propriété *CommandText*. C'est le texte de la requête SQL à exécuter. La connexion devra être précisée dans la propriété *Connection*.
- *OdbcCommand(string sqlText, OdbcConnection connexion)* : le paramètre *sqlText* sera affecté à la propriété *CommandText* et le paramètre *connexion* à la propriété *Connection*.

Pour émettre la requête SQL, on dispose de deux méthodes :

- **OdbcdataReader ExecuteReader()** : envoie la requête SELECT de *CommandText* à la connexion *Connection* et construit un objet *OdbcDataReader* permettant l'accès à toutes les lignes de la table résultat du *select*
- **int ExecuteNOnQuery()** : envoie la requête de mise à jour (INSERT, UPDATE, DELETE) de *CommandText* à la connexion *Connection* et rend le nombre de lignes affectées par cette mise à jour.

Dans notre exemple, après avoir ouvert la connexion à la base, nous émettons une requête SQL SELECT pour avoir le contenu de la table ARTICLES :

```
const string tabArticles="articles"; // la table des articles
OdbcDataReader myReader=null;     // le lecteur de données
```

```
  // exécution d'une commande SQL
  string sqlText = "select * from " + tabArticles;
  OdbcCommand myOdbcCommand = new OdbcCommand(sqlText);
  myOdbcCommand.Connection = articlesConn;
  myReader=myOdbcCommand.ExecuteReader();
```

Une requête d'interrogation est classiquement une requête du type :

select col1, col2,... from table1, table2,...
where condition
order by expression
...

Seuls les mots clés de la première ligne sont obligatoires, les autres sont facultatifs. Il existe d'autres mots clés non présentés ici.

1. Une jointure est faite avec toutes les tables qui sont derrière le mot clé **from**
2. Seules les colonnes qui sont derrière le mot clé **select** sont conservées
3. Seules les lignes vérifiant la condition du mot clé **where** sont conservées
4. Les lignes résultantes ordonnées selon l'expression du mot clé **order by** forment le résultat de la requête.

Le résultat d'un *select* est une table. Si on considère la table ARTICLES précédente et qu'on veuille les noms des articles dont le stock actuel est au-dessous du seuil minimal, on écrira :

select nom from articles where stock_actuel<stock_minimum

Si on les veut par ordre alphabétique des noms, on écrira :

select nom from articles where stock_actuel<stock_minimum order by nom

6.3.2.3 Exploitation du résultat d'une requête SELECT

Le résultat d'une requête SELECT en mode non connecté est un objet *DataReader*, ici un objet *OdbcDataReader*. Cet objet permet d'obtenir séquentiellement toutes les lignes du résultat et d'avoir des informations sur les colonnes de ces résultats. Examinons quelques propriétés et méthodes de cette classe :

FieldCount	le nombre de colonnes de la table
Item	*Item[i]* représente la colonne n° i de la ligne courante du résultat
XXX GetXXX(i)	la valeur de la colonne n° i de la ligne courante rendue comme type XXX(Int16, Int32, Int64, Double, String, Boolean, ...)
string GetName(i)	nom de la colonne n° i
Close()	ferme l'objet *OdbcdataReader* et libère les ressources associées
bool Read()	avance d'une ligne dans la table des résultats. Rend faux si cela n'est pas possible. La nouvelle ligne devient la ligne courante du lecteur.

L'exploitation du résultat d'un *select* est typiquement une exploitation séquentielle analogue à celle des fichiers texte : on ne peut qu'avancer dans la table, pas reculer :

```
while (objetOdbcDataReadet.Read()){
  // on a une ligne - on l'exploite
  ....
  // ligne suivante
}//while
```

Ces explications suffisent à comprendre le code suivant de notre exemple :

```
        // affichage des colonnes
        string ligne="";
        int i;
        for(i=0;i<myReader.FieldCount-1;i++){
          ligne+=myReader.GetName(i)+",";
        }//for
        ligne+=myReader.GetName(i);
        Console.Out.WriteLine("\n"+"".PadLeft(ligne.Length,'-')+"\n"+ligne+"\n"+"".PadLeft(ligne.Length,'-
')+"\n");
        // affichage des données
        while (myReader.Read()) {
          // exploitation ligne courante
          ligne="";
          for(i=0;i<myReader.FieldCount;i++){
            ligne+=myReader[i]+" ";
          }
          Console.WriteLine(ligne);
        }//while
```

La seule difficulté est dans l'instruction où les valeurs des différentes colonnes de la ligne courante sont concaténées :

```
          for(i=0;i<myReader.FieldCount;i++){
            ligne+=myReader[i]+" ";
          }
```

La notation *ligne+=myReader[i] +" "* est traduit par *ligne+=myReader.Item[i].ToString()+" "* où *Item[i]* est la valeur de la colonne i de la ligne courante.

6.3.2.4 Libération des ressources

Les classes *OdbcReader* et *OdbcConnection* possèdent toutes deux une méthode **Close()** qui libère les ressources associées aux objets ainsi fermés.

```
        // fermeture lecteur
        myReader.Close();
        // fermeture connexion
        articlesConn.Close();
```

6.3.3 Utilisation d'un pilote OLE DB

Nous reprenons le même exemple, cette fois avec une base accédée via un pilote OLE DB. La plate-forme .NET fournit un tel pilote pour les bases ACCESS. Aussi allons-nous utiliser la même base *articles.mdb* que précédemment. Nous cherchons à montrer ici que si les classes changent, les concepts restent les mêmes :

- la connexion est représentée par un objet **OleDbConnection**
- une requête SQL est émise grâce à un objet **OleDbCommand**
- si cette requête est une clause SELECT, on obtiendra en retour un objet **OleDbDataReader** pour accéder aux lignes de la table résultat

Ces classes sont dans l'espace de noms *System.Data.OleDb*. Le programme précédent peut être transformé aisément pour gérer une base OLE DB :

- on remplace partout OdbcXX par OleDbXX
- on modifie la chaîne de connexion. Pour une base ACCESS sans login/mot de passe, la chaîne de connexion est *Provider=Microsoft.JET.OLEDB.4.0;Data Source=[fichier.mdb]*. La partie paramètrable de cette chaîne est le nom du fichier ACCESS à utiliser. Nous modifierons notre programme pour qu'il accepte en paramètre le nom de ce fichier.
- l'espace de noms à importer est maintenant *System.Data.OleDb*.

Notre programme devient le suivant :

```
using System;
using System.Data;
using System.Data.OleDb;

class listArticles
  {
    static void Main(string[] args){

      // application console
      // affiche le contenu d'une table ARRTICLES d'une base DSN
      // dont le nom est passé en paramètre

      const string syntaxe="syntaxe : pg base_access_articles";
      const string tabArticles="articles"; // la table des articles

      // vérification des paramètres
      // a-t-on 2 paramètres
      if(args.Length!=1){
        // msg d'erreur
        Console.Error.WriteLine(syntaxe);
        // fin
        Environment.Exit(1);
      }//if
      // on récupère le paramètre
      string dbArticles=args[0];   // la base de données

      // préparation de la connexion à la bd
      OleDbConnection articlesConn=null;  // la connexion
      OleDbDataReader myReader=null;      // le lecteur de données

      try{
        // on tente d'accéder à la base de données
        // chaîne de connexion à la base
        string connectString="Provider=Microsoft.JET.OLEDB.4.0;Data Source="+dbArticles+";";
        articlesConn = new OleDbConnection(connectString);
        articlesConn.Open();
        // exécution d'une commande SQL
        string sqlText = "select * from " + tabArticles;
        OleDbCommand myOleDbCommand = new OleDbCommand(sqlText);
        myOleDbCommand.Connection = articlesConn;
        myReader=myOleDbCommand.ExecuteReader();
        // Exploitation de la table récupérée
        // affichage des colonnes
        string ligne="";
        int i;
        for(i=0;i<myReader.FieldCount-1;i++){
          ligne+=myReader.GetName(i)+",";
```

```
      }//for
      ligne+=myReader.GetName(i);
      Console.Out.WriteLine("\n"+"".PadLeft(ligne.Length,'-')+"\n"+ligne+"\n"+"".PadLeft(ligne.Length,'-
')+"\n");
      // affichage des données
      while (myReader.Read()) {
        // exploitation ligne courante
        ligne="";
        for(i=0;i<myReader.FieldCount;i++){
          ligne+=myReader[i]+" ";
        }
        Console.WriteLine(ligne);
      }//while
    }//try
    catch(Exception ex){
      Console.Error.WriteLine("Erreur d'exploitation de la base de données ("+ex.Message+")");
      Environment.Exit(2);
    }//catch
    finally{
      try{
        // fermeture lecteur
        myReader.Close();
        // fermeture connexion
        articlesConn.Close();
      }catch{}
    }//finally

    // fin
    Environment.Exit(0);
  }//main
}//classe
```

Les résultats obtenus :

```
E:\data\serge\MSNET\c#\adonet\6>csc liste.cs

E:\data\serge\MSNET\c#\adonet\6>dir
07/05/2002  15:09                2 325 liste.CS
07/05/2002  15:09                4 608 liste.exe
20/08/2001  11:54               86 016 ARTICLES.MDB

E:\data\serge\MSNET\c#\adonet\6>liste articles.mdb

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                           2500 10 5
b300 pompe                          56 62 45
c300 arc                            3500 10 20
d300 flèches - lot de 6             780 12 20
e300 combinaison de plongée         2800 34 7
f300 bouteilles d'oxygène           800 10 5
```

6.3.4 Exemple 1 : mise à jour d'une table

Les exemples précédents se contentaient de lister le contenu d'une table. Nous modifions notre programme de gestion de la base d'articles afin qu'il puisse modifier celle-ci. Le programme s'appelle *sql*. On lui passe en paramètre le nom DSN de la base d'articles à gérer. L'utilisateur tape directement des commandes SQL au clavier que le programme exécute comme le montrent les résultats qui suivent obtenus sur la base MySQL d'articles :

```
E:\data\serge\MSNET\c#\adonet\7>csc /r:microsoft.data.odbc.dll sql.cs

E:\data\serge\MSNET\c#\adonet\7>sql mysql-articles

Requête SQL (fin pour arrêter) : select * from articles

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                           2500 10 5
b300 pompe                          56 62 45
c300 arc                            3500 10 20
d300 flèches - lot de 6             780 12 20
e300 combinaison de plongée         2800 34 7
f300 bouteilles d'oxygène           800 10 5

Requête SQL (fin pour arrêter) : select * from articles where stock_actuel<stock_minimum
```

```
----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

c300 arc                              3500 10 20
d300 flèches - lot de 6               780 12 20

Requête SQL (fin pour arrêter) : insert into articles values ("1","1",1,1,1)
Il y a eu 1 ligne(s) modifiée(s)

Requête SQL (fin pour arrêter) : select * from articles

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                             2500 10 5
b300 pompe                            56 62 45
c300 arc                              3500 10 20
d300 flèches - lot de 6               780 12 20
e300 combinaison de plongée           2800 34 7
f300 bouteilles d'oxygène             800 10 5
1 1 1 1 1

Requête SQL (fin pour arrêter) : update articles set nom="2" where nom="1"
Il y a eu 1 ligne(s) modifiée(s)

Requête SQL (fin pour arrêter) : select * from articles

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                             2500 10 5
b300 pompe                            56 62 45
c300 arc                              3500 10 20
d300 flèches - lot de 6               780 12 20
e300 combinaison de plongée           2800 34 7
f300 bouteilles d'oxygène             800 10 5
1 2 1 1 1

Requête SQL (fin pour arrêter) : delete from articles where code="1"
Il y a eu 1 ligne(s) modifiée(s)

Requête SQL (fin pour arrêter) : select * from articles

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

a300 vélo                             2500 10 5
b300 pompe                            56 62 45
c300 arc                              3500 10 20
d300 flèches - lot de 6               780 12 20
e300 combinaison de plongée           2800 34 7
f300 bouteilles d'oxygène             800 10 5

Requête SQL (fin pour arrêter) : select * from articles order by nom asc

----------------------------------------
code,nom,prix,stock_actuel,stock_minimum
----------------------------------------

c300 arc                              3500 10 20
f300 bouteilles d'oxygène             800 10 5
e300 combinaison de plongée           2800 34 7
d300 flèches - lot de 6               780 12 20
b300 pompe                            56 62 45
a300 vélo                             2500 10 5

Requête SQL (fin pour arrêter) : fin
```

Le programme est le suivant :

```
using System;
using System.Data;
using Microsoft.Data.Odbc;
using System.Text.RegularExpressions;
using System.Collections;
```

```
class sqlCommands
  {
  static void Main(string[] args){

    // application console
    // exécute des requêtes SQL tapées au clavier sur une
    // table ARTICLES d'une base DSN dont le nom est passé en paramètre

    const string syntaxe="syntaxe : pg dsnArticles";

    // vérification des paramètres
    // a-t-on 2 paramètres
    if(args.Length!=1){
      // msg d'erreur
      Console.Error.WriteLine(syntaxe);
      // fin
      Environment.Exit(1);
    }//if
    // on récupère le paramètre
    string dsnArticles=args[0];   // la base DSN
    // chaîne de connexion à la base
    string connectString="DSN="+dsnArticles+";";

    // préparation de la connexion à la bd
    OdbcConnection articlesConn=null; // la connexion
    OdbcCommand sqlCommand=null;    // la commande SQL
    try{
      // on tente d'accéder à la base de données
      articlesConn = new OdbcConnection(connectString);
      articlesConn.Open();
      // on crée un objet command
      sqlCommand=new OdbcCommand("",articlesConn);
    }//try
    catch(Exception ex){
      // msg d'erreur
      Console.Error.WriteLine("Erreur d'exploitation de la base de données ("+ex.Message+")");
      // libération des ressources
      try{articlesConn.Close();} catch {}
      // fin
      Environment.Exit(2);
    }//catch

    // on construit un dictionnaire des commandes sql acceptées
    string[] commandesSQL=new string[] {"select","insert","update","delete"};
    Hashtable dicoCommandes=new Hashtable();
    for(int i=0;i<commandesSQL.Length;i++){
      dicoCommandes.Add(commandesSQL[i],true);
    }

    // lecture-exécution des commandes SQL tapées au clavier
    string requête=null; // texte de la requête SQL
    string[] champs;     // les champs de la requête
    Regex modèle=new Regex(@"\s+"); // suite d'espaces

    // boucle de saisie-exécution des commandes SQL tapées au clavier
    while(true){
      // demande de la requête
      Console.Out.Write("\nRequête SQL (fin pour arrêter) : ");
      requête=Console.In.ReadLine().Trim().ToLower();
      // fini ?
      if(requête=="fin") break;
      // on décompose la requête en champs
      champs=modèle.Split(requête);
      // requête valide ?
      if (champs.Length==0 || ! dicoCommandes.ContainsKey(champs[0])){
        // msg d'erreur
        Console.Error.WriteLine("Requête invalide. Utilisez select, insert, update, delete");
        // requête suivante
        continue;
      }//if
      // préparation de l'objet Command pour exécuter la requête
      sqlCommand.CommandText=requête;
      // exécution de la requête
      try{
        if(champs[0]=="select"){
          executeSelect(sqlCommand);
        } else executeUpdate(sqlCommand);
      }//try
      catch(Exception ex){
        // msg d'erreur
        Console.Error.WriteLine("Erreur d'exploitation de la base de données ("+ex.Message+")");
      }//catch
      // requête suivante
    }//while
    // libération des ressources
    try{articlesConn.Close();} catch {}
    // fin
    Environment.Exit(0);
  }//main

  // exécution d'une requête de mise à jour
  static void executeUpdate(OdbcCommand sqlCommand){
```

```
    // exécute sqlCommand, requête de mise à jour
    int nbLignes=sqlCommand.ExecuteNonQuery();
    // affichage
    Console.Out.WriteLine("Il y a eu " + nbLignes + " ligne(s) modifiée(s)");
  }//executeUpdate

  // exécution d'une requête Select
  static void executeSelect(OdbcCommand sqlCommand){
    // exécute sqlCommand, requête select
    OdbcDataReader myReader=sqlCommand.ExecuteReader();
    // Exploitation de la table récupérée
    // affichage des colonnes
    string ligne="";
    int i;
    for(i=0;i<myReader.FieldCount-1;i++){
      ligne+=myReader.GetName(i)+",";
    }//for
    ligne+=myReader.GetName(i);
    Console.Out.WriteLine("\n"+"".PadLeft(ligne.Length,'-')+"\n"+ligne+"\n"+"".PadLeft(ligne.Length,'-
')+"\n");
    // affichage des données
    while (myReader.Read()) {
      // exploitation ligne courante
      ligne="";
      for(i=0;i<myReader.FieldCount;i++){
        ligne+=myReader[i]+" ";
      }
      // affichage
      Console.WriteLine(ligne);
    }//while
    // libération des ressources
    myReader.Close();
  }//executeSelect
}//classe
```

Nous ne commentons ici que ce qui est nouveau par rapport au programme précédent :

- Nous construisons un dictionnaire des commandes sql acceptées :

```
    // on construit un dictionnaire des commandes sql acceptées
    string[] commandesSQL=new string[] {"select","insert","update","delete"};
    Hashtable dicoCommandes=new Hashtable();
    for(int i=0;i<commandesSQL.Length;i++){
      dicoCommandes.Add(commandesSQL[i],true);
    }
```

ce qui nous permet ensuite de vérifier simplement si le 1er mot (*champs[0]*) de la requête tapée est l'une des quatre commandes acceptées :

```
// requête valide ?
      if (champs.Length==0 || ! dicoCommandes.ContainsKey(champs[0])){
        // msg d'erreur
        Console.Error.WriteLine("Requête invalide. Utilisez select, insert, update, delete");
        // requête suivante
        continue;
      }//if
```

- Auparavant la requête avait été décomposée en champs à l'aide de la méthode *Split* de la classe *RegEx* :

```
      Regex modèle=new Regex(@"\s+"); // suite d'espaces
      // on décompose la requête en champs
      champs=modèle.Split(requête);
```

Les mots composant la requête peuvent être séparés d'un nombre quelconque d'espaces.

- L'exécution d'une requête *select* n'utilise pas la même méthode que celle d'une requête d'une mise à jour (*insert*, *update*, *delete*). Aussi doit-on faire un test et exécuter une fonction différente pour chacu de ces deux cas :

```
      // exécution de la requête
      try{
        if(champs[0]=="select"){
          executeSelect(sqlCommand);
        } else executeUpdate(sqlCommand);
      }//try
      catch(Exception ex){
        // msg d'erreur
        Console.Error.WriteLine("Erreur d'exploitation de la base de données ("+ex.Message+")");
      }//catch
```

L'exécution d'une requête SQL peut générer une exception qui est ici gérée.

- La fonction *executeSelect* reprend tout ce qui a été vu dans les exemples précédents.
- La fonction *executeUpdate* utilise la méthode *ExecuteNonQuery* de la class *OdbcCommand* qui rend le nombre de lignes affectées par la commande.

6.3.5 Exemple 2 : IMPOTS

Nous reprenons l'objet *impôt* construit dans un chapitre précédent :

```
// création d'une classe impôt

using System;

public class impôt{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private decimal[] limites, coeffR, coeffN;

  // constructeur 1
  public impôt(decimal[] LIMITES, decimal[] COEFFR, decimal[] COEFFN){
    // on vérifie que les 3 tablaeux ont la même taille
    bool OK=LIMITES.Length==COEFFR.Length && LIMITES.Length==COEFFN.Length;
    if (! OK) throw new Exception ("Les 3 tableaux fournis n'ont pas la même taille("+
                LIMITES.Length+","+COEFFR.Length+","+COEFFN.Length+")");
    // c'est bon
    this.limites=LIMITES;
    this.coeffR=COEFFR;
    this.coeffN=COEFFN;
  }//constructeur 1


  // calcul de l'impôt
  public long calculer(bool marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
    decimal nbParts;
    if (marié) nbParts=(decimal)nbEnfants/2+2;
    else nbParts=(decimal)nbEnfants/2+1;
    if (nbEnfants>=3) nbParts+=0.5M;
    // calcul revenu imposable & Quotient familial
    decimal revenu=0.72M*salaire;
    decimal QF=revenu/nbParts;
    // calcul de l'impôt
    limites[limites.Length-1]=QF+1;
    int i=0;
    while(QF>limites[i]) i++;
    // retour résultat
    return (long)(revenu*coeffR[i]-nbParts*coeffN[i]);
  }//calculer
}//classe
```

Nous lui ajoutons un nouveau constructeur permettant d'initialiser les tableaux *limites*, *coeffR*, *coeffN* à partir d'une base de données ODBC :

```
using System.Data;
using Microsoft.Data.Odbc;
using System.Collections;
```

```
  // constructeur 2
  public impôt(string DSNimpots, string Timpots, string colLimites, string colCoeffR, string colCoeffN){
    // initialise les trois tableaux limites, coeffR, coeffN à partir
    // du contenu de la table Timpots de la base ODBC DSNimpots
    // colLimites, colCoeffR, colCoeffN sont les trois colonnes de cette table
    // peut lancer une exception

    string connectString="DSN="+DSNimpots+";"; // chaîne de connexion à la base
    OdbcConnection impotsConn=null;   // la connexion
    OdbcCommand sqlCommand=null;    // la commande SQL
    // la requête SELECT
    string selectCommand="select "+colLimites+","+colCoeffR+","+colCoeffN+" from "+Timpots;
    // tableaux pour récupérer les données
    ArrayList tLimites=new ArrayList();
    ArrayList tCoeffR=new ArrayList();
    ArrayList tCoeffN=new ArrayList();

    // on tente d'accéder à la base de données
    impotsConn = new OdbcConnection(connectString);
    impotsConn.Open();
    // on crée un objet command
    sqlCommand=new OdbcCommand(selectCommand,impotsConn);
    // on exécute la requête
    OdbcDataReader myReader=sqlCommand.ExecuteReader();
    // Exploitation de la table récupérée
    while (myReader.Read()) {
```

```
      // les données de la ligne courante sont mis dans les tableaux
      tLimites.Add(myReader[colLimites]);
      tCoeffR.Add(myReader[colCoeffR]);
      tCoeffN.Add(myReader[colCoeffN]);
    }//while
    // libération des ressources
    myReader.Close();
    impotsConn.Close();

    // les tableaux dynamiques sont mis dans des tableaux statiques
    this.limites=new decimal[tLimites.Count];
    this.coeffR=new decimal[tLimites.Count];
    this.coeffN=new decimal[tLimites.Count];
    for(int i=0;i<tLimites.Count;i++){
      limites[i]=decimal.Parse(tLimites[i].ToString());
      coeffR[i]=decimal.Parse(tCoeffR[i].ToString());
      coeffN[i]=decimal.Parse(tCoeffN[i].ToString());
    }//for
  }//constructeur 2
```

Le programme de test est le suivant : il reçoit en arguments les paramètres à transmettre au constructeur de la classe *impôt*. Après avoir construit un objet *impôt*, il fait des calculs d'impôt à payer :

```
using System;

  class test
  {
    public static void Main(string[] arguments)
    {
      // programme interactif de calcul d'impôt
      // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
      // le programme affiche alors l'impôt à payer

      const string syntaxe1="pg DSNimpots tabImpots colLimites colCoeffR colCoeffN";
      const string syntaxe2="syntaxe : marié nbEnfants salaire\n"
             +"marié : o pour marié, n pour non marié\n"
             +"nbEnfants : nombre d'enfants\n"
             +"salaire : salaire annuel en F";

      // vérification des paramètres du programme
      if(arguments.Length!=5){
        // msg d'erreur
        Console.Error.WriteLine(syntaxe1);
        // fin
        Environment.Exit(1);
      }//if
      // on récupère les arguments
      string DSNimpots=arguments[0];
      string tabImpots=arguments[1];
      string colLimites=arguments[2];
      string colCoeffR=arguments[3];
      string colCoeffN=arguments[4];

      // création d'un objet impôt
      impôt objImpôt=null;
      try{
        objImpôt=new impôt(DSNimpots,tabImpots,colLimites,colCoeffR,colCoeffN);
      }catch (Exception ex){
        Console.Error.WriteLine("L'erreur suivante s'est produite : " + ex.Message);
        Environment.Exit(2);
      }//try-catch

      // boucle infinie
      while(true){
        // on demande les paramètres du calcul de l'impôt
        Console.Out.Write("Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour
arrêter :");
        string paramètres=Console.In.ReadLine().Trim();
        // qq chose à faire ?
        if(paramètres==null || paramètres=="") break;
        // vérification du nombre d'arguments dans la ligne saisie
        string[] args=paramètres.Split(null);
        int nbParamètres=args.Length;
        if (nbParamètres!=3){
          Console.Error.WriteLine(syntaxe2);
          continue;
        }//if
        // vérification de la validité des paramètres
        // marié
        string marié=args[0].ToLower();
        if (marié!="o" && marié !="n"){
          Console.Error.WriteLine(syntaxe2+"\nArgument marié incorrect : tapez o ou n");
          continue;
        }//if
        // nbEnfants
        int nbEnfants=0;
        try{
          nbEnfants=int.Parse(args[1]);
          if(nbEnfants<0) throw new Exception();
```

```
        }catch (Exception){
          Console.Error.WriteLine(syntaxe2+"\nArgument nbEnfants incorrect : tapez un entier positif ou
nul");
          continue;
        }//if
        // salaire
        int salaire=0;
        try{
          salaire=int.Parse(args[2]);
          if(salaire<0) throw new Exception();
        }catch (Exception){
          Console.Error.WriteLine(syntaxe2+"\nArgument salaire incorrect : tapez un entier positif ou
nul");
          continue;
        }//if
        // les paramètres sont corrects - on calcule l'impôt
        Console.Out.WriteLine("impôt="+objImpôt.calculer(marié=="o",nbEnfants,salaire)+" F");
        // contribuable suivant
      }//while
    }//Main
  }//classe
```

La base utilisée est une base MySQL de nom DSN *mysql-impots* :

```
C:\mysql\bin>mysql --database=impots --user=admimpots --password=mdpimpots
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 3.23.49-max-debug

Type 'help' for help.

mysql> show tables;
+------------------+
| Tables_in_impots |
+------------------+
| timpots          |
+------------------+
1 row in set (0.05 sec)

mysql> select * from timpots;
+---------+--------+---------+
| limites | coeffR | coeffN  |
+---------+--------+---------+
|   12620 |      0 |       0 |
|   13190 |   0.05 |     631 |
|   15640 |    0.1 |  1290.5 |
|   24740 |   0.15 |  2072.5 |
|   31810 |    0.2 |  3309.5 |
|   39970 |   0.25 |    4900 |
|   48360 |    0.3 |    6898 |
|   55790 |   0.35 |  9316.5 |
|   92970 |    0.4 |   12106 |
|  127860 |   0.45 |   16754 |
|  151250 |    0.5 | 23147.5 |
|  172040 |   0.55 |   30710 |
|  195000 |    0.6 |   39312 |
|       0 |   0.65 |   49062 |
+---------+--------+---------+
14 rows in set (0.03 sec)
```

L'exécution du programme de test donne les résultats suivants :

```
E:\data\serge\MSNET\c#\impots\6>csc /r:microsoft.data.odbc.dll /t:library impots.cs


E:\data\serge\MSNET\c#\impots\6>csc /r:impots.dll test.cs

E:\data\serge\MSNET\c#\impots\6>test mysql-impots timpots limites coeffr coeffn
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 200000
impôt=22506 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :n 2 200000
impôt=33388 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 3 200000
impôt=16400 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :n 3 300000
impôt=50082 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :n 3 200000
impôt=22506 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :
```

6.4 Accès aux données en mode déconnecté

Ce sujet sera traité ultérieurement.

7. Les threads d'exécution

7.1 Introduction

Lorsqu'on lance une application, elle s'exécute dans un flux d'exécution appelé un **thread**. La classe .NET modélisant un *thread* est la classe *System.Threading.Thread* et a la définition suivante :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.3705\mscorlib.dll'
public sealed class System.Threading.Thread :
    object
{

    // Fields

    // Constructors
    public Thread(System.Threading.ThreadStart start);

    // Properties
    public ApartmentState ApartmentState { get; set; }
    public static Context CurrentContext { get; }
    public CultureInfo CurrentCulture { get; set; }
    public static IPrincipal CurrentPrincipal { get; set; }
    public static Thread CurrentThread { get; }
    public CultureInfo CurrentUICulture { get; set; }
    public bool IsAlive { get; }
    public bool IsBackground { get; set; }
    public bool IsThreadPoolThread { get; }
    public string Name { get; set; }
    public ThreadPriority Priority { get; set; }
    public ThreadState ThreadState { get; }

    // Methods
    public void Abort();
    public void Abort(object stateInfo);
    public static LocalDataStoreSlot AllocateDataSlot();
    public static LocalDataStoreSlot AllocateNamedDataSlot(string name);
    public virtual bool Equals(object obj);
    public static void FreeNamedDataSlot(string name);
    public static object GetData(LocalDataStoreSlot slot);
    public static AppDomain GetDomain();
    public static int GetDomainID();
    public virtual int GetHashCode();
    public static LocalDataStoreSlot GetNamedDataSlot(string name);
    public Type GetType();
    public void Interrupt();
    public void Join();
    public bool Join(int millisecondsTimeout);
    public bool Join(TimeSpan timeout);
    public static void ResetAbort();
    public void Resume();
    public static void SetData(LocalDataStoreSlot slot, object data);
    public static void Sleep(int millisecondsTimeout);
    public static void Sleep(TimeSpan timeout);
    public static void SpinWait(int iterations);
    public void Start();
    public void Suspend();
    public virtual string ToString();
} // end of System.Threading.Thread
```

Nous ne commenterons que certaines de ces propriétés et méthodes :

`CurrentThread - propriété statique`	donne le thread actuellement en cours d'exécution
`Name - propriété d'objet`	nom du thread
`isAlive - propriété d'objet`	indique si le thread est actif(true) ou non (false)
`Start - méthode d'objet`	lance l'exécution d'un thread
`Abort - méthode d'objet`	arrête définitivement l'exécution d'un thread
`Sleep(n) - méthode statique`	arrête l'exécution d'un thread pendant n millisecondes
`Suspend() - méthode d'objet`	suspend temporairement l'exécution d'un thread
`Resume() - méthode d'objet`	reprend l'excéution d'un thread suspendu
`Join() - méthode d'objet`	opération bloquante - attend la fin du thread pour passer à l'instruction suivante

Regardons une première application mettant en évidence l'existence d'un thread principal d'exécution, celui dans lequel s'exécute la fonction *Main* d'une classe :

```
// utilisation de threads

using System;
using System.Threading;
```

```
public class thread1{
  public static void Main(){
    // init thread courant
    Thread main=Thread.CurrentThread;
    // affichage
    Console.Out.WriteLine("Thread courant : " + main.Name);
    // on change le nom
    main.Name="main";
    // vérification
    Console.Out.WriteLine("Thread courant : " + main.Name);

    // boucle infinie
    while(true){
      // affichage
      Console.Out.WriteLine(main.Name + " : " +DateTime.Now.ToString("hh:mm:ss"));
      // arrêt temporaire
      Thread.Sleep(1000);
    }//while
  }//main
}//classe
```

Les résultats écran :

```
E:\data\serge\MSNET\c#\threads\2>thread1
Thread courant :
Thread courant : main
main : 06:13:55
main : 06:13:56
main : 06:13:57
main : 06:13:58
main : 06:13:59
^C
```

L'exemple précédent illustre les points suivants :

- la fonction *Main* s'exécute bien dans un thread
- on a accès aux caractéristiques de ce thread par *Thread.CurrentThread*
- le rôle de la méthode *Sleep*. Ici le thread exécutant *Main* se met en sommeil régulièrement pendant 1 seconde entre deux affichages.

7.2 Création de threads d'exécution

Il est possible d'avoir des applications où des morceaux de code s'exécutent de façon "simultanée" dans différents threads d'exécution. Lorsqu'on dit que des *threads* s'exécutent de façon simultanée, on commet souvent un abus de langage. Si la machine n'a qu'un processeur comme c'est encore souvent le cas, les *threads* se partagent ce processeur : ils en disposent, chacun leur tour, pendant un court instant (quelques millisecondes). C'est ce qui donne l'illusion du parallélisme d'exécution. La portion de temps accordée à un *thread* dépend de divers facteurs dont sa priorité qui a une valeur par défaut mais qui peut être fixée également par programmation. Lorsqu'un *thread* dispose du processeur, il l'utilise normalement pendant tout le temps qui lui a été accordé. Cependant, il peut le libérer avant terme :

- en se mettant en attente d'un événement (*wait*, *join*, *suspend*)
- en se mettant en sommeil pendant un temps déterminé (*sleep*)

1. Un **thread T** est d'abord créé par son constructeur

```
public Thread(System.Threading.ThreadStart start);
```

   ThreadStart est de type delegate et définit le prototype d'une fonction sans paramètres. Une construction classique est la suivante :

```
Thread T=new Thread(new ThreadStart(run));
```

   La fonction *run* passée en paramètres sera exécutée au lancement du Thread.
2. L'exécution du thread T est lancé par **T.Start()** : la fonction run passée au constructeur de T va alors être exécutée par le thread T. Le programme qui exécute l'instruction *T.start()* n'attend pas la fin de la tâche T : il passe aussitôt à l'instruction qui suit. On a alors deux tâches qui s'exécutent en parallèle. Elles doivent souvent pouvoir communiquer entre elles pour savoir où en est le travail commun à réaliser. C'est le problème de synchronisation des threads.
3. Une fois lancé, le thread s'exécute de façon autonome. Il s'arrêtera lorsque la fonction *start* qu'il exécute aura fini son travail.
4. On peut envoyer certains signaux à la tâche T :
   a. **T.Suspend()** lui dit de s'arrêter momentanément
   b. **T.Resume()** lui dit de reprendre son travail
   c. **T.Abort()** lui dit de s'arrêter définitivement
5. On peut aussi attendre la fin de son exécution par **T.join()**. On a là une instruction bloquante : le programme qui l'exécute est bloqué jusqu'à ce que la tâche T ait terminé son travail. C'est un moyen de synchronisation.

Examinons le programme suivant :

```
// utilisation de threads

using System;
using System.Threading;

public class thread1{
  public static void Main(){
    // init Thread courant
    Thread main=Thread.CurrentThread;
    // on fixe un nom au Thread
    main.Name="main";

    // création de threads d'exécution
    Thread[] tâches=new Thread[5];
    for(int i=0;i<tâches.Length;i++){
      // on crée le thread i
      tâches[i]=new Thread(new ThreadStart(affiche));
      // on fixe le nom du thread
      tâches[i].Name=""+i;
      // on lance l'exécution du thread i
      tâches[i].Start();
    }//for

    // fin de main
    Console.Out.WriteLine("fin du thread " +main.Name);
  }//Main

  public static void affiche(){
    // affichage début d'exécution
    Console.Out.WriteLine("Début d'exécution de la méthode affiche dans le Thread " +
Thread.CurrentThread.Name
      + " : " + DateTime.Now.ToString("hh:mm:ss"));
    // mise en sommeil pendant 1 s
    Thread.Sleep(1000);
    // affichage fin d'exécution
    Console.Out.WriteLine("Fin d'exécution de la méthode affiche dans le Thread " +
Thread.CurrentThread.Name
      + " : " + DateTime.Now.ToString("hh:mm:ss"));
  }//
}//classe
```

Le thread principal, celui qui exécute la fonction *Main*, crée 5 autres threads chargés d'exécuter la méthode statique *affiche*. Les résultats sont les suivants :

```
fin du thread main
Début d'exécution de la méthode affiche dans le Thread 0 : 07:01:03
Début d'exécution de la méthode affiche dans le Thread 1 : 07:01:03
Début d'exécution de la méthode affiche dans le Thread 2 : 07:01:03
Début d'exécution de la méthode affiche dans le Thread 3 : 07:01:03
Début d'exécution de la méthode affiche dans le Thread 4 : 07:01:03
Fin d'exécution de la méthode affiche dans le Thread 0 : 07:01:04
Fin d'exécution de la méthode affiche dans le Thread 1 : 07:01:04
Fin d'exécution de la méthode affiche dans le Thread 2 : 07:01:04
Fin d'exécution de la méthode affiche dans le Thread 3 : 07:01:04
Fin d'exécution de la méthode affiche dans le Thread 4 : 07:01:04
```

Ces résultats sont très instructifs :

- on voit tout d'abord que le lancement de l'exécution d'un thread n'est pas bloquante. La méthode *Main* a lancé l'exécution de 5 threads en parallèle et a terminé son exécution avant eux. L'opération

  ```
  // on lance l'exécution du thread i
  tâches[i].Start();
  ```

  lance l'exécution du thread *tâches[i]* mais ceci fait, l'exécution se poursuit immédiatement avec l'instruction qui suit sans attendre la fin d'exécution du thread.
- tous les threads créés doivent exécuter la méthode *affiche*. L'ordre d'exécution est imprévisible. Même si dans l'exemple, l'ordre d'exécution semble suivre l'ordre des demandes d'exécution, on ne peut en conclure de généralités. Le système d'exploitation a ici 6 threads et un processeur. Il va distribuer le processeur à ces 6 threads selon des règles qui lui sont propres.
- on voit dans les résultats une conséquence de la méthode *Sleep*. dans l'exemple, c'est le thread 0 qui exécute le premier la méthode *affiche*. Le message de début d'exécution est affiché puis il exécute la méthode *Sleep* qui le suspend pendant 1 seconde. Il perd alors le processeur qui devient ainsi disponible pour un autre thread. L'exemple montre que c'est le thread 1 qui va l'obtenir. Le thread 1 va suivre le même parcours ainsi que les autres threads. Lorsque la seconde de sommeil du thread 0 va être terminée, son exécution peut reprendre. Le système lui donne le processeur et il peut terminer l'exécution de la méthode *affiche*.

Modifions notre programme pour le terminer la méthode *Main* par les instructions :

```
    // fin de main
    Console.Out.WriteLine("fin du thread " +main.Name);
    Environment.Exit(0);
```

L'exécution du nouveau programme donne :

```
fin du thread main
```

Les threads créés par la fonction *Main* ne sont pas exécutés. C'est l'instruction

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

qui fait cela : elle supprime tous les threads de l'application et non simplement le thread *Main*. La solution à ce problème est que la méthode *Main* attende la fin d'exécution des threads qu'elle a créés avant de se terminer elle-même. Cela peut se faire avec la méthode *Join* de la classe *Thread* :

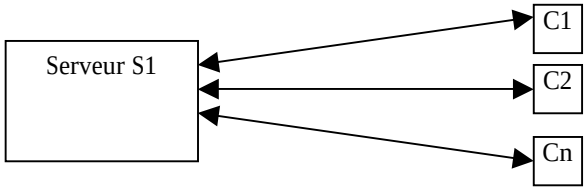
```
    // on attend la fin d'exécution de tous les threads
    for(int i=0;i<tâches.Length;i++){
      // attente de la fin d'exécution du thread i
      tâches[i].Join();
    }//for

// fin de main
    Console.Out.WriteLine("fin du thread " +main.Name + " : " + DateTime.Now.ToString("hh:mm:ss"));
    Environment.Exit(0);
```

On obtient alors les résultats suivants :

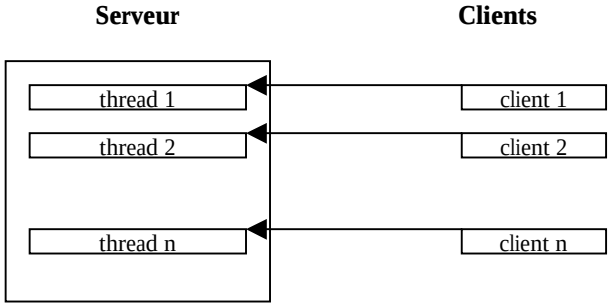
```
Début d'exécution de la méthode affiche dans le Thread 0 : 07:14:51
Début d'exécution de la méthode affiche dans le Thread 1 : 07:14:51
Début d'exécution de la méthode affiche dans le Thread 2 : 07:14:51
Début d'exécution de la méthode affiche dans le Thread 3 : 07:14:51
Début d'exécution de la méthode affiche dans le Thread 4 : 07:14:51
Fin d'exécution de la méthode affiche dans le Thread 0 : 07:14:52
Fin d'exécution de la méthode affiche dans le Thread 1 : 07:14:52
Fin d'exécution de la méthode affiche dans le Thread 2 : 07:14:52
Fin d'exécution de la méthode affiche dans le Thread 3 : 07:14:52
Fin d'exécution de la méthode affiche dans le Thread 4 : 07:14:52
fin du thread main : 07:14:52
```

7.3 Intérêt des threads

Maintenant que nous avons mis en évidence l'existence d'un thread par défaut, celui qui exécute la méthode *Main*, et que nous savons comment en créer d'autres, arrêtons-nous sur l'intérêt pour nous des threads et sur la raison pour laquelle nous les présentons ici. Il y a un type d'applications qui se prêtent bien à l'utilisation des threads, ce sont les applications client-serveur de l'internet. Dans une telle application, un serveur situé sur une machine S1 répond aux demandes de clients situés sur des machines distantes C1, C2, ..., Cn.



Nous utilisons tous les jours des applications de l'internet correspondant à ce schéma : services Web, messagerie électronique, consultation de forums, transfert de fichiers... Dans le schéma ci-dessus, le serveur S1 doit servir les clients Ci de façon simultanée. Si nous prenons l'exemple d'un serveur FTP (File Transfer Protocol) qui délivre des fichiers à ses clients, nous savons qu'un transfert de fichier peut prendre parfois plusieurs heures. Il est bien sûr hors de question qu'un client monopolise tout seul le serveur une telle durée. Ce qui est fait habituellement, c'est que le serveur crée autant de threads d'exécution qu'il y a de clients. Chaque thread est alors chargé de s'occuper d'un client particulier. Le processeur étant partagé cycliquement entre tous les threads actifs de la machine, le serveur passe alors un peu de temps avec chaque client assurant ainsi la simultanéité du service.



7.4 Accès à des ressources partagées

Dans l'exemple client-serveur évoqué ci-dessus, chaque thread sert un client de façon largement indépendante. Néanmoins, les threads peuvent être amenés à coopérer pour rendre le service demandé à leur client notamment pour l'accès à des ressources partagées. Le schéma ci-dessus fait penser aux guichets d'une grande administration, une poste par exemple où à chaque guichet un agent sert un client. Supposons que de temps en temps ces agents soient amenés à faire des photocopies de documents amenés par leurs clients et qu'il n'y ait qu'une photocopieuse. Deux agents ne peuvent utiliser la photocopieuse en même temps. Si l'agent i trouve la photocopieuse utilisée par l'agent j, il devra attendre. On appelle cette situation, l'accès à une ressource partagée et en informatique elle est assez délicate à gérer. Prenons l'exemple suivant :

- une application va générer n threads, n étant passé en paramètre
- la ressource partagée est un compteur qui devra être incrémenté par chaque thread généré
- à la fin de l'application, la valeur du compteur est affiché. On devrait donc trouver n.

Le programme est le suivant :

```
// utilisation de threads

using System;
using System.Threading;

public class thread1{

  // variables de classe
  static int cptrThreads=0;  // compteur de threads

  //main
  public static void Main(String[] args){
    // mode d'emploi
    const string syntaxe="pg nbThreads";
    const int nbMaxThreads=100;

    // vérification nbre d'arguments
    if(args.Length!=1){
      // erreur
      Console.Error.WriteLine(syntaxe);
      // arrêt
      Environment.Exit(1);
    }//if
    // vérification qualité de l'argument
    int nbThreads=0;
    try{
      nbThreads=int.Parse(args[0]);
      if(nbThreads<1 || nbThreads>nbMaxThreads)
        throw new Exception();
    }catch{
      // erreur
      Console.Error.WriteLine("Nombre de threads incorrect (entre 1 et 100)");
      // fin
      Environment.Exit(2);
    }//catch

    // création et génération des threads
    Thread[] threads=new Thread[nbThreads];
    for(int i=0;i<nbThreads;i++){
      // création
      threads[i]=new Thread(new ThreadStart(incrémente));
      // nommage
      threads[i].Name=""+i;
      // lancement
      threads[i].Start();
    }//for
    // attente de la fin des threads
    for(int i=0;i<nbThreads;i++)
      threads[i].Join();
    // affichage compteur
    Console.Out.WriteLine("Nombre de threads générés : " +cptrThreads);
```

```
  }//Main

  public static void incrémente(){
    // augmente le compteur de threads
    // lecture compteur
    int valeur=cptrThreads;
    // suivi
    Console.Out.WriteLine("A "+DateTime.Now.ToString("hh:mm:ss")+ ", le thread
"+Thread.CurrentThread.Name+" a lu la valeur du compteur : " +cptrThreads);
    // attente
    Thread.Sleep(1000);
    // incrémentation compteur
    cptrThreads=valeur+1;
    // suivi
    Console.Out.WriteLine("A "+DateTime.Now.ToString("hh:mm:ss")+ ", le thread
"+Thread.CurrentThread.Name+" a écrit la valeur du compteur : " +cptrThreads);
  }//incrémente
}//classe
```

Nous ne nous attarderons pas sur la partie génération de threads déjà étudiée. Intéressons-nous plutôt à la méthode *incrémente*, utilisée par chaque thread pour incrémenter le compteur statique *cptrThreads*.

1. le compteur est lu
2. le thread s'arrête 1 s. Il perd donc le processeur
3. le compteur est incrémenté

L'étape 2 n'est là que pour forcer le thread à perdre le processeur. Celui-ci va être donné à un autre thread. Dans la pratique, rien n'assure qu'un thread ne sera pas interrompu entre le moment où il va lire le compteur et le moment où il va l'incrémenter. Même si on écrit *cptrThreads++*, donnant ainsi l'illusion d'une instruction unique, le risque existe de perdre le processeur entre le moment où on lit la valeur du compteur et celui on écrit sa valeur incrémentée de 1. En effet, l'opération de haut niveau *cptrThreads++* va faire l'objet de plusieurs instructions élémentaires au niveau du processeur. L'étape 2 de sommeil d'une seconde n'est donc là que pour systématiser ce risque.

Les résultats obtenus sont les suivants :

```
E:\data\serge\MSNET\c#\threads\5>comptage 5
A 09:15:58, le thread 0 a lu la valeur du compteur : 0
A 09:15:58, le thread 1 a lu la valeur du compteur : 0
A 09:15:58, le thread 2 a lu la valeur du compteur : 0
A 09:15:58, le thread 3 a lu la valeur du compteur : 0
A 09:15:58, le thread 4 a lu la valeur du compteur : 0
A 09:15:59, le thread 0 a écrit la valeur du compteur : 1
A 09:15:59, le thread 1 a écrit la valeur du compteur : 1
A 09:15:59, le thread 2 a écrit la valeur du compteur : 1
A 09:15:59, le thread 3 a écrit la valeur du compteur : 1
A 09:15:59, le thread 4 a écrit la valeur du compteur : 1
Nombre de threads générés : 1
```

A la lecture de ces résultats, on voit bien ce qui se passe :

- un premier thread lit le compteur. Il trouve 0.
- il s'arrête 1 s donc perd le processeur
- un second thread prend alors le processeur et lit lui aussi la valeur du compteur. Elle est toujours à 0 puisque le thread précédent ne l'a pas encore incrémenté. Il s'arrête lui aussi 1 s.
- en 1 s, les 5 threads ont le temps de passer tous et de lire tous la valeur 0.
- lorsqu'ils vont se réveiller les uns après les autres, ils vont incrémenter la valeur 0 qu'ils ont lue et écrire la valeur 1 dans le compteur, ce que confirme le programme principal (Main).

D'où vient le problème ? Le second thread a lu une mauvaise valeur du fait que le premier avait été interrompu avant d'avoir terminé son travail qui était de mettre à jour le compteur dans la fenêtre. Cela nous amène à la notion de ressource critique et de section critique d'un programme:

- une ressource critique est une ressource qui ne peut être détenue que par un thread à la fois. Ici la ressource critique est le compteur.
- une section critique d'un programme est une séquence d'instructions dans le flux d'exécution d'un thread au cours de laquelle il accède à une ressource critique. On doit assurer qu'au cours de cette section critique, il est le seul à avoir accès à la ressource.

7.5 Accès exclusif à une ressource partagée

Dans notre exemple, la section critique est le code situé entre la lecture du compteur et l'écriture de sa nouvelle valeur :

```
    // lecture compteur
    int valeur=cptrThreads;
```

```
    // attente
    Thread.Sleep(1000);
    // incrémentation compteur
    cptrThreads=valeur+1;
```

Pour exécuter ce code, un thread doit être assuré d'être tout seul. Il peut être interrompu mais pendant cette interruption, un autre thread ne doit pas pouvoir exécuter ce même code. La plate-forme .NET offre plusieurs outils pour assurer l'entrée unitaire dans les sections critiques de code. Nous utiliserons la classe Mutex :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.3705\mscorlib.dll'
public sealed class System.Threading.Mutex :
    System.Threading.WaitHandle,
    IDisposable
{

    // Constructors
    public Mutex();
    public Mutex(bool initiallyOwned);
    public Mutex(bool initiallyOwned, string name);
    public Mutex(bool initiallyOwned, string name, ref Boolean createdNew);

    // Properties
    public IntPtr Handle { virtual get; virtual set; }

    // Methods
    public virtual void Close();
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public void ReleaseMutex();
    public virtual string ToString();
    public virtual bool WaitOne();
    public virtual bool WaitOne(int millisecondsTimeout, bool exitContext);
    public virtual bool WaitOne(TimeSpan timeout, bool exitContext);
} // end of System.Threading.Mutex
```

Nous n'utiliserons ici que les constructeurs et méthodes suivants :

`public Mutex()`	crée un objet de synchronisation M
`public bool WaitOne()`	Le thread T1 qui exécute l'opération *M.WaitOne()* demande la propriété de l'objet de synchronisation M. Si le Mutex M n'est détenu par aucun thread (le cas au départ), il est "donné" au thread T1 qui l'a demandé. Si un peu plus tard, un thread T2 fait la même opération, il sera bloqué. En effet, un Mutex ne peut appartenir qu'à un thread. Il sera débloqué lorsque le thread T1 libèrera le mutex M qu'il détient. Plusieurs threads peuvent ainsi être bloqués en attente du Mutex M.
`public void ReleaseMutex()`	Le thread T1 qui effectue l'opération *M.ReleaseMutex()* abandonne la propriété du Mutex M.Lorsque le thread T1 perdra le processeur, le système pourra le donner à l'un des threads en attente du Mutex M. Un seul l'obtiendra à son tour, les autres en attente de M restant bloqués

Un Mutex M gère l'accès à une ressource partagée R. Un thread demande la ressource R par *M.WaitOne()* et la rend par *M.ReleaseMutex()*. Une section critique de code qui ne doit être exécutée que par un seul thread à la fois est une ressource partagée. La synchronisation d'exécution de la section critique peut se faire ainsi :

```
M.WaitOne();
// le thread est seul à entrer ici
// section critique
....
M.ReleaseMutex();
```

où M est un objet *Mutex*. Il faut bien sûr ne jamais oublier de libérer un *Mutex* devenu inutile pour qu'un autre thread puisse entrer dans la section critique, sinon les threads en attente d'un Mutex jamais libéré n'auront jamais accès au processeur. Par ailleurs, il faut éviter la situation d'interblocage (*deadlock*) dans laquelle deux threads s'attendent mutuellement. Considérons les actions suivantes qui se suivent dans le temps :

- un thread T1 obtient la propriété d'un Mutex M1 pour avoir accès à une ressource partagée R1
- un thread T2 obtient la propriété d'un Mutex M2 pour avoir accès à une ressource partagée R2
- le thread T1 demande le Mutex M2. Il est bloqué.
- le thread T2 demande le Mutex M1. Il est bloqué.

Ici, les threads T1 et T2 s'attendent mutuellement. Ce cas apparaît lorsque des threads ont besoin de deux ressources partagées, la ressource R1 contrôlée par le Mutex M1 et la ressource R2 contrôlée par le Mutex M2. Une solution possible est de demander les deux ressources en même temps à l'aide d'un Mutex unique M. Mais ce n'est pas toujours possible si par exemple cela entraîne une mobilisation longue d'une ressource coûteuse. Une autre solution est qu'un thread ayant M1 et ne pouvant obtenir M2, relâche alors M1 pour éviter l'interblocage.

Si nous mettons en pratique ce que nous venons de voir sur l'exemple précédent, notre application devient la suivante :

```
// utilisation de threads

using System;
using System.Threading;

public class thread1{

  // variables de classe
  static int cptrThreads=0;    // compteur de threads
  static Mutex autorisation; // autorisation d'accès à une section critique

  //main
  public static void Main(String[] args){
    // mode d'emploi
    const string syntaxe="pg nbThreads";
    const int nbMaxThreads=100;

    // vérification nbre d'arguments
    if(args.Length!=1){
      // erreur
      Console.Error.WriteLine(syntaxe);
      // arrêt
      Environment.Exit(1);
    }//if
    // vérification qualité de l'argument
    int nbThreads=0;
    try{
      nbThreads=int.Parse(args[0]);
      if(nbThreads<1 || nbThreads>nbMaxThreads)
        throw new Exception();
    }catch{
      // erreur
      Console.Error.WriteLine("Nombre de threads incorrect (entre 1 et 100)");
      // fin
      Environment.Exit(2);
    }//catch

    // initialisation de l'autorisation d'accès à une section critique
    autorisation=new Mutex();

    // création et génération des threads
    Thread[] threads=new Thread[nbThreads];
    for(int i=0;i<nbThreads;i++){
      // création
      threads[i]=new Thread(new ThreadStart(incrémente));
      // nommage
      threads[i].Name=""+i;
      // lancement
      threads[i].Start();
    }//for
    // attente de la fin des threads
    for(int i=0;i<nbThreads;i++)
      threads[i].Join();
    // affichage compteur
    Console.Out.WriteLine("Nombre de threads générés : " +cptrThreads);
  }//Main

  public static void incrémente(){
    // augmente le compteur de threads
    // on demande l'autorisation d'entrer dans la secton critique
    autorisation.WaitOne();
    // lecture compteur
    int valeur=cptrThreads;
    // suivi
    Console.Out.WriteLine("A "+DateTime.Now.ToString("hh:mm:ss")+ ", le thread
"+Thread.CurrentThread.Name+" a lu la valeur du compteur : " +cptrThreads);
    // attente
    Thread.Sleep(1000);
    // incrémentation compteur
    cptrThreads=valeur+1;
    // suivi
    Console.Out.WriteLine("A "+DateTime.Now.ToString("hh:mm:ss")+ ", le thread
"+Thread.CurrentThread.Name+" a écrit la valeur du compteur : " +cptrThreads);
    // on rend l'autorisation d'accès
    autorisation.ReleaseMutex();
  }//incrémente
}//classe
```

Les résultats obtenus sont conformes à ce qui était attendu :

```
E:\data\serge\MSNET\c#\threads\6>comptage 5
A 10:11:37, le thread 0 a lu la valeur du compteur : 0
A 10:11:38, le thread 0 a écrit la valeur du compteur : 1
A 10:11:38, le thread 1 a lu la valeur du compteur : 1
A 10:11:39, le thread 1 a écrit la valeur du compteur : 2
A 10:11:39, le thread 2 a lu la valeur du compteur : 2
```

```
A 10:11:40, le thread 2 a écrit la valeur du compteur : 3
A 10:11:40, le thread 3 a lu la valeur du compteur : 3
A 10:11:41, le thread 3 a écrit la valeur du compteur : 4
A 10:11:41, le thread 4 a lu la valeur du compteur : 4
A 10:11:42, le thread 4 a écrit la valeur du compteur : 5
Nombre de threads générés : 5
```

7.6 Synchronisation par événements

Considérons la situation suivante, souvent appelée situation de **producteurs-consommateurs**.

1. On a un tableau dans lequel des processus viennent déposer des données (les producteurs) et d'autres viennent les lire (les consommateurs).
2. Les producteurs sont égaux entre-eux mais exclusifs : un seul producteur à la fois peut déposer ses données dans le tableau.
3. Les consommateurs sont égaux entre-eux mais exclusifs : un seul lecteur à la fois peut lire les données déposées dans le tableau.
4. Un consommateur ne peut lire les données du tableau que lorsqu'un producteur en a déposé dedans et un producteur ne peut déposer de nouvelles données dans le tableau que lorsque celles qui y sont ont été consommées.

On peut dans cet exposé distinguer deux ressources partagées :

1. le tableau en écriture
2. le tableau en lecture

L'accès à ces deux ressources partagées peut être contrôlée par des Mutex comme vu précédemment, un pour chaque ressource. Une fois qu'un consommateur a obtenu le tableau en lecture, il doit vérifier qu'il y a bien des données dedans. On utilisera un événement pour l'en avertir. De même un producteur ayant obtenu le tableau en écriture devra attendre qu'un consommateur l'ait vidé. On utilisera là encore un événement.

Les événements utilisés feront partie de la classe *AutoResetEvent* :

```
// from module 'c:\winnt\microsoft.net\framework\v1.0.3705\mscorlib.dll'
public sealed class System.Threading.AutoResetEvent :
    System.Threading.WaitHandle,
    IDisposable
{

    // Constructors
    public AutoResetEvent(bool initialState);

    // Properties
    public IntPtr Handle { virtual get; virtual set; }

    // Methods
    public virtual void Close();
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public bool Reset();
    public bool Set();
    public virtual string ToString();
    public virtual bool WaitOne();
    public virtual bool WaitOne(int millisecondsTimeout, bool exitContext);
    public virtual bool WaitOne(TimeSpan timeout, bool exitContext);
} // end of System.Threading.AutoResetEvent
```

Ce type d'événement est analogue à un booléen mais évite des attentes actives ou semi-actives. Ainsi le droit d'écriture est contrôlé par un booléen *peutEcrire*, un producteur avant d'écrire exécutera un code du genre :

```
while(peutEcrire==false);    // attente active
```

ou

```
while(peutEcrire==false){  // attente semi-active
  Thread.Sleep(100);  // attente de 100ms
}//while
```

Dans la première méthode, le thread mobilise inutilement le processeur. Dans la seconde, il vérifie l'état du booléen *peutEcrire* toutes les 100 ms. La classe *AutoResetEvent* permet encore d'améliorer les choses : le thread va demander à être réveillé lorsque l'événement qu'il attend se sera produit :

```
AutoEvent peutEcrire=new AutoResetEvent(false);   // peutEcrire=false;
....
peutEcrire.WaitOne();// le thread attend que l'évt peutEcrire passe à vrai
```

L'opération

```
AutoEvent peutEcrire=new AutoResetEvent(false);   // peutEcrire=false;
```

initialise le booléen *peutEcrire* à *false*. L'opération

```
peutEcrire.WaitOne();// le thread attend que l'évt peutEcrire passe à vrai
```

exécutée par un thread fait que celui-ci passe si le booléen *peutEcrire* est vrai ou sinon est bloqué jusqu'à ce qu'il devienne vrai. Un autre thread le passera à vrai par l'opération *peutEcrire.Set()* ou à faux par l'opération *peutEcrire.Reset()*.

Le programme de producteurs-consommateurs est le suivant :

```
// utilisation de threads lecteurs et écrivains
// illustre l'utilisation simultanée de ressources partagées et de synchronisation

using System;
using System.Threading;

public class thread1{

  // variables de classe
  static int[] data=new int[5];     // ressource partagée entre threads lecteur et threads écrivain
  static Mutex lecteur;             // variable de synchronisation pour lire le tableau
  static Mutex écrivain;            // variable de synchronisation pour écrire dans le tableau
  static Random objRandom=new Random(DateTime.Now.Second); // un générateur de nombres aléatoires
  static AutoResetEvent peutLire;   // signale qu'on peut lire le contenu de data
  static AutoResetEvent peutEcrire; // signale qu'on peut écrire le contenu de data

  //main
  public static void Main(String[] args){

    // le nbre de threads à générer
    const int nbThreads=3;

    // initialisation des drapeaux
    peutLire=new AutoResetEvent(false);  // on ne peut pas encore lire
    peutEcrire=new AutoResetEvent(true); // on peut déjà écrire

    // initialisation des variables de synchronisation
    lecteur=new Mutex();   // synchronise les lecteurs
    écrivain=new Mutex();  // synchronise les écrivains

    // création des threads lecteurs
     Thread[] lecteurs=new Thread[nbThreads];
    for(int i=0;i<nbThreads;i++){
      // création
      lecteurs[i]=new Thread(new ThreadStart(lire));
      lecteurs[i].Name=""+i;
      // lancement
      lecteurs[i].Start();
    }//for

    // création des threads écrivains
    Thread[] écrivains=new Thread[nbThreads];
    for(int i=0;i<nbThreads;i++){
      // création
      écrivains[i]=new Thread(new ThreadStart(écrire));
      écrivains[i].Name=""+i;
      // lancement
      écrivains[i].Start();
    }//for

    //fin de main
    Console.Out.WriteLine("fin de Main...");
  }//Main

  // lire le contenu du tableau
  public static void lire(){
    // section critique
    lecteur.WaitOne();   // un seul lecteur peut passer
    peutLire.WaitOne();  // on doit pouvoir lire
    // lecture tableau
    for(int i=0;i<data.Length;i++){
      //attente 1 s
      Thread.Sleep(1000);
      // affichage
      Console.Out.WriteLine(DateTime.Now.ToString("hh:mm:ss") + " : Le lecteur "+
Thread.CurrentThread.Name+ " a lu le nombre " +data[i]);
    }//for
    // on ne peut plus lire
    peutLire.Reset();
    // on peut écrire
    peutEcrire.Set();
    // fin de section critique
    lecteur.ReleaseMutex();
  }//lire

  // écrire dans le tableau
```

```
  public static void écrire(){
    // section critique
    // un seul écrivain peut passer
    écrivain.WaitOne();
    // on doit attendre l'autorisation d'écriture
    peutEcrire.WaitOne();
    // écriture tableau
    for(int i=0;i<data.Length;i++){
      //attente 1 s
      Thread.Sleep(1000);
      // affichage
      data[i]=objRandom.Next(0,1000);
      Console.Out.WriteLine(DateTime.Now.ToString("hh:mm:ss") + " : L'écrivain "+
Thread.CurrentThread.Name + " a écrit le nombre " +data[i]);
    }//for
    // on ne peut plus écrire
    peutEcrire.Reset();
    // on peut lire
    peutLire.Set();
    //fin de section critique
    écrivain.ReleaseMutex();
  }//lire
}//classe
```

L'exécution donne les résultats suivants :

```
E:\data\serge\MSNET\c#\threads\8>readwrite
fin de Main...
11:52:41 : L'écrivain 0 a écrit le nombre 623
11:52:42 : L'écrivain 0 a écrit le nombre 554
11:52:43 : L'écrivain 0 a écrit le nombre 727
11:52:44 : L'écrivain 0 a écrit le nombre 95
11:52:45 : L'écrivain 0 a écrit le nombre 265
11:52:46 : Le lecteur 0 a lu le nombre 623
11:52:47 : Le lecteur 0 a lu le nombre 554
11:52:48 : Le lecteur 0 a lu le nombre 727
11:52:49 : Le lecteur 0 a lu le nombre 95
11:52:50 : Le lecteur 0 a lu le nombre 265
11:52:51 : L'écrivain 1 a écrit le nombre 514
11:52:52 : L'écrivain 1 a écrit le nombre 828
11:52:53 : L'écrivain 1 a écrit le nombre 509
11:52:54 : L'écrivain 1 a écrit le nombre 926
11:52:55 : L'écrivain 1 a écrit le nombre 23
11:52:56 : Le lecteur 1 a lu le nombre 514
11:52:57 : Le lecteur 1 a lu le nombre 828
11:52:58 : Le lecteur 1 a lu le nombre 509
11:52:59 : Le lecteur 1 a lu le nombre 926
11:53:00 : Le lecteur 1 a lu le nombre 23
11:53:01 : L'écrivain 2 a écrit le nombre 761
11:53:02 : L'écrivain 2 a écrit le nombre 695
11:53:03 : L'écrivain 2 a écrit le nombre 130
11:53:04 : L'écrivain 2 a écrit le nombre 279
^C
```

On peut remarquer les points suivants :

- on a bien 1 seul lecteur à la fois bien que celui-ci perde le processeur dans la section critique *lire*
- on a bien 1 seul écrivain à la fois bien que celui-ci perde le processeur dans la section critique *écrire*
- un lecteur ne lit que lorsqu'il y a quelque chose à lire dans le tableau
- un écrivain n'écrit que lorsque le tableau a été entièrement lu

8. Programmation TCP-IP

8.1 Généralités

8.1.1 Les protocoles de l'Internet

Nous donnons ici une introduction aux protocoles de communication de l'Internet, appelés aussi suite de protocoles **TCP/IP** (*Transfer Control Protocol / Internet Protocol*), du nom des deux principaux protocoles. Il est bon que le lecteur ait une compréhension globale du fonctionnement des réseaux et notamment des protocoles TCP/IP avant d'aborder la construction d'applications distribuées.

Le texte qui suit est une traduction partielle d'un texte que l'on trouve dans le document *"Lan Workplace for Dos - Administrator's Guide"* de NOVELL, document du début des années 90.

Le concept général de créer un réseau d'ordinateurs hétérogènes vient de recherches effectuées par le **DARPA** (**D**efense **A**dvanced **R**esearch **P**rojects **A**gency) aux Etats-Unis. Le DARPA a développé la suite de protocoles connue sous le nom de TCP/IP qui permet à des machines hétérogènes de communiquer entre elles. Ces protocoles ont été testés sur un réseau appelé **ARPAnet**, réseau qui devint ultérieurement le réseau **INTERNET**. Les protocoles TCP/IP définissent des formats et des règles de transmission et de réception indépendants de l'organisation des réseaux et des matériels utilisés.

Le réseau conçu par le DARPA et géré par les protocoles TCP/IP est un réseau à **commutation de paquets**. Un tel réseau transmet l'information sur le réseau, en petits morceaux appelés **paquets**. Ainsi, si un ordinateur transmet un gros fichier, ce dernier sera découpé en petits morceaux qui seront envoyés sur le réseau pour être recomposés à destination. TCP/IP définit le format de ces paquets, à savoir :

. origine du paquet
. destination
. longueur
. type

8.1.2 Le modèle OSI

Les protocoles TCP/IP suivent à peu près le modèle de réseau ouvert appelé **OSI** (**O**pen **S**ystems **I**nterconnection Reference Model) défini par l'**ISO** (**I**nternational **S**tandards **O**rganisation). Ce modèle décrit un réseau idéal où la communication entre machines peut être représentée par un modèle à sept couches :

```
        |-------------------------------------|
7       |       Application                   |
        |-------------------------------------|
6       |       Présentation                  |
        |-------------------------------------|
5       |       Session                       |
        |-------------------------------------|
4       |       Transport                     |
        |-------------------------------------|
3       |       Réseau                        |
        |-------------------------------------|
2       |       Liaison                       |
        |-------------------------------------|
1       |       Physique                      |
        |-------------------------------------|
```

Chaque couche reçoit des services de la couche inférieure et offre les siens à la couche supérieure. Supposons que deux applications situées sur des machines A et B différentes veulent communiquer : elles le font au niveau de la couche *Application*. Elles n'ont pas besoin de connaître tous les détails du fonctionnement du réseau : chaque application remet l'information qu'elle souhaite

transmettre à la couche du dessous : la couche *Présentation*. L'application n'a donc à connaître que les règles d'interfaçage avec la couche *Présentation*.
Une fois l'information dans la couche *Présentation*, elle est passée selon d'autres règles à la couche *Session* et ainsi de suite, jusqu'à ce que l'information arrive sur le support physique et soit transmise physiquement à la machine destination. Là, elle subira le traitement inverse de celui qu'elle a subi sur la machine expéditeur.

A chaque couche, le processus expéditeur chargé d'envoyer l'information, l'envoie à un processus récepteur sur l'autre machine apartenant à la même couche que lui. Il le fait selon certaines règles que l'on appelle le **protocole** de la couche. On a donc le schéma de communication final suivant :

```
                 Machine A                                Machine B
        +-------------------------------------+    +----------------------------+
7       ¦       Application      v            ¦    ¦     ^   Application        ¦
        +-------------------------Î------------¦    +-----Î----------------------¦
6       ¦       Présentation     v            ¦    ¦     ^   Présentation       ¦
        +-------------------------Î------------¦    +-----Î----------------------¦
5       ¦       Session          v            ¦    ¦     ^   Session            ¦
        +-------------------------Î------------¦    +-----Î----------------------¦
4       ¦       Transport        v            ¦    ¦     ^   Transport          ¦
        +-------------------------Î------------¦    +-----Î----------------------¦
3       ¦       Réseau           v            ¦    ¦     ^   Réseau             ¦
        +-------------------------Î------------¦    +-----Î----------------------¦
2       ¦       Liaison          v            ¦    ¦     ^   Liaison            ¦
        +-------------------------Î------------¦    +-----Î----------------------¦
1       ¦       Physique         v            ¦    ¦     ^   Physique           ¦
        +-------------------------Î------------+    +-----Î----------------------+
                                  ¦                       ^
                                  +-->-------->------>-----+
```

Le rôle des différentes couches est le suivant :

Physique Assure la transmission de bits sur un support physique. On trouve dans cette couche des équipements terminaux de traitement des données (E.T.T.D.) tels que terminal ou ordinateur, ainsi que des équipements de terminaison de circuits de données (E.T.C.D.) tels que modulateur/démodulateur, multiplexeur, concentrateur. Les points d'intérêt à ce niveau sont :

. le choix du codage de l'information (analogique ou numérique)
. le choix du mode de transmission (synchrone ou asynchrone).

Liaison de données Masque les particularités physiques de la couche Physique. Détecte et corrige les erreurs de transmission.

Réseau Gère le chemin que doivent suivre les informations envoyées sur le réseau. On appelle cela le *routage* : déterminer la route à suivre par une information pour qu'elle arrive à son destinataire.

Transport Permet la communication entre deux applications alors que les couches précédentes ne permettaient que la communication entre machines. Un service fourni par cette couche peut être le multiplexage : la couche transport pourra utiliser une même connexion réseau (de machine à machine) pour transmettre des informations appartenant à plusieurs applications.

Session On va trouver dans cette couche des services permettant à une application d'ouvrir et de maintenir une session de travail sur une machine distante.

Présentation Elle vise à uniformiser la représentation des données sur les différentes machines. Ainsi des données provenant d'une machine A, vont être "habillées" par la couche *Présentation* de la machine A, selon un format standard avant d'être envoyées sur le réseau. Parvenues à la couche *Présentation* de la machine destinatrice B qui les reconnaîtra grâce à leur format standard, elles seront habillées d'une autre façon afin que l'application de la machine B les reconnaisse.

Application A ce niveau, on trouve les applications généralement proches de l'utilisateur telles que la messagerie électronique ou le transfert de fichiers.

8.1.3 Le modèle TCP/IP

Le modèle OSI est un modèle idéal encore jamais réalisé. La suite de protocoles TCP/IP s'en approche sous la forme suivante :

```
        +-----------------------+     +-------------------------------------------------------+
7       |       Application     |     |        |      |        |          |   DNS            |
        +-----------------------|     |Telnet  | FTP  | TFTP   | SMTP     +------------------|
6       |       Présentation    |     |        |      |        |          |   Autres         |
        +-----------------------|     +-------------------------+------------------------------|
5       |       Session         |     |                         |                              |
        +-----------------------|     |       TCP               |        UDP                   |
4       |       Transport       |     |                         |                              |
        +-----------------------|     +-------------------------------------------------------|
3       |       Réseau          |     |   IP   | ICMP |   ARP       |    RARP                  |
        +-----------------------|     +-------------------------------------------------------|
2       |       Liaison         |     | MLID1       |  MLID2  |  MLID3   |  MLID4              |
        +-----------------------|     +-------------------------------------------------------|
1       |       Physique        |     | Ethernet         | Token-ring   |  Autres              |
        +-----------------------+     +-------------------------------------------------------+
```

Couche Physique

En réseau local, on trouve généralement une technologie **Ethernet** ou **Token-Ring**. Nous ne présentons ici que la technologie Ethernet.

Ethernet

C'est le nom donné à une technologie de réseaux locaux à commutation de paquets inventée à PARC Xerox au début des années 1970 et normalisée par Xerox, Intel et Digital Equipment en 1978. Le réseau est physiquement constitué d'un câble coaxial d'environ 1,27 cm de diamètre et d'une longueur de 500 m au plus. Il peut être étendu au moyen de *répéteurs*, deux machines ne pouvant être séparées par plus de deux répéteurs. Le câble est passif : tous les éléments actifs sont sur les machines raccordées au câble. Chaque machine est reliée au câble par une carte d'accès au réseau comprenant :

- un transmetteur (*transceiver*) qui détecte la présence de signaux sur le câble et convertit les signaux analogiques en signaux numérique et inversement.
- un coupleur qui reçoit les signaux numériques du transmetteur et les transmet à l'ordinateur pour traitement ou inversement.

Les caractéristiques principales de la technologie Ethernet sont les suivantes :

- Capacité de 10 Mégabits/seconde.
- Topologie en bus : toutes les machines sont raccordées au même câble

```
-----------------------------------------------------------
      |                  |                       |
  +--------+        +--------+              +-----------+
  |        |        |        |              |           |
  |        |        |        |              |           |
  +--------+        +--------+              +-----------+
  Machine A             B                       C
```

- Réseau diffusant - Une machine qui émet transfère des informations sur le câble avec l'adresse de la machine destinatrice. Toutes les machines raccordées reçoivent alors ces informations et seule celle à qui elles sont destinées les conserve.

- La méthode d'accès est la suivante : le transmetteur désirant émettre écoute le câble - il détecte alors la présence ou non d'une onde porteuse, présence qui signifierait qu'une transmission est en cours. C'est la technique **CSMA** (*Carrier Sense Multiple Access*). En l'absence de porteuse, un transmetteur peut décider de transmettre à son tour. Ils peuvent être plusieurs à prendre cette décision. Les signaux émis se mélangent : on dit qu'il y a **collision**. Le transmetteur détecte cette situation : en même temps qu'il émet sur le câble, il écoute ce qui passe réellement sur celui-ci. S'il détecte que l'information transitant sur le câble n'est pas celle qu'il a émise, il en déduit qu'il y a collision et il s'arrêtera d'émettre. Les autres transmetteurs qui émettaient feront de même. Chacun reprendra son émission après un temps aléatoire dépendant de chaque transmetteur. Cette technique est appelée **CD** (*Collision Detect*). La méthode d'accès est ainsi appelée **CSMA/CD**.

- un adressage sur 48 bits. Chaque machine a une adresse, appelée ici adresse physique, qui est inscrite sur la carte qui la relie au câble. On appelle cet adresse, l'adresse *Ethernet* de la machine.

Couche Réseau

Nous trouvons au niveau de cette couche, les protocoles IP, ICMP, ARP et RARP.

IP (Internet Protocol) Délivre des paquets entre deux noeuds du réseau

ICMP (Internet Control Message Protocol)	ICMP réalise la communication entre le programme du protocole IP d'une machine et celui d'une autre machine. C'est donc un protocole d'échange de messages à l'intérieur même du protocole IP.
ARP (Address Resolution Protocol)	fait la correspondance adresse Internet machine--> adresse physique machine
RARP (Reverse Address Resolution Protocol)	fait la correspondance adresse physique machine--> adresse Internet machine

Couches Transport/Session

Dans cette couche, on trouve les protocoles suivants :

TCP (Transmission Control Protocol)	Assure une remise fiable d'informations entre deux clients
UDP (User Datagram Protocol)	Assure une remise non fiable d'informations entre deux clients

Couches Application/Présentation/Session

On trouve ici divers protocoles :

TELNET	Emulateur de terminal permettant à une machine A de se connecter à une machine B en tant que terminal
FTP (File Transfer Protocol)	permet des transferts de fichiers
TFTP (Trivial File Transfer Protocol)	permet des transferts de fichiers
SMTP (Simple Mail Transfer protocol)	permet l'échange de messages entre utilisateurs du réseau
DNS (Domain Name System)	transforme un nom de machine en adresse Internet de la machine
XDR (eXternal Data Representation)	créé par sun MicroSystems, il spécifie une représentation standard des données, indépendante des machines
RPC(Remote Procedures Call)	défini également par Sun, c'est un protocole de communication entre applications distantes, indépendant de la couche transport. Ce protocole est important : il décharge le programmeur de la connaissance des détails de la couche transport et rend les applications portables. Ce protocole s'appuie sur sur le protocole XDR
NFS (Network File System)	toujours défini par Sun, ce protocole permet à une machine, de "voir" le système de fichiers d'une autre machine. Il s'appuie sur le protocole RPC précédent

8.1.4 Fonctionnement des protocoles de l'Internet

Les applications développées dans l'environnement TCP/IP utilisent généralement plusieurs des protocoles de cet environnement. Un programme d'application communique avec la couche la plus élevée des protocoles. Celle-ci passe l'information à la couche du dessous et ainsi de suite jusqu'à arriver sur le support physique. Là, l'information est physiquement transférée à la machine destinatrice où elle retraversera les mêmes couches, en sens inverse cette fois-ci, jusqu'à arriver à l'application destinatrice des informations envoyées. Le schéma suivant montre le parcours de l'information :

```
+----------------+                        +---------------------------+
¦  Application   ¦                        ¦  Application              ¦
+----------------+                        +---------------------------+
        ¦ <----------- messages ou streams ----------> ¦
+----------------+                        +---------------------------+
¦  Transport     ¦                        ¦  Transport                ¦
¦  (Udp/Tcp)     ¦                        ¦     (Udp/tcp)             ¦
+----------------+                        +---------------------------+
        ¦ <----------- datagrammes (UDP) -----------> ¦
+----------------+          ou            +---------------------------+
¦  Réseau (IP)   ¦     segments (TCP)      ¦  Réseau (IP)              ¦
+----------------+                        +---------------------------+
        ¦ <----------- datagrammes IP --------------> ¦
+----------------+                        +----------------------------+
¦Interface réseau¦                        ¦ Interface réseau           ¦
+-------Ê--------+                        +----------------------------+
        ¦  <----------  trames réseau -------------> ¦
        +-----------------------------------------------+
                        réseau physique
```

Prenons un exemple : l'application FTP, définie au niveau de la couche *Application* et qui permet des transferts de fichiers entre machines.

. L'application délivre une suite d'octets à transmettre à la couche *transport*.
. La couche *transport* découpe cette suite d'octets en *segments* TCP, et ajoute au début de chaque segment, le numéro de celui-ci. Les segments sont passés à la couche Réseau gouvernée par le protocole **IP**.
. La couche IP crée un paquet encapsulant le segment TCP reçu. En tête de ce paquet, elle place les adresses Internet des machines source et destination. Elle détermine également l'adresse physique de la machine destinatrice. Le tout est passé à la couche *Liaison de données & Liaison physique*, c'est à dire à la carte réseau qui couple la machine au réseau physique.
. Là, le paquet IP est encapsulé à son tour dans une **trame** physique et envoyé à son destinataire sur le câble.
. Sur la machine destinatrice, la couche *Liaison de données & Liaison physique* fait l'inverse : elle désencapsule le paquet IP de la trame physique et le passe à la couche IP.
. La couche *IP* vérifie que le paquet est correct : elle calcule une somme, fonction des bits reçus (*checksum*), somme qu'elle doit retrouver dans l'en-tête du paquet. Si ce n'est pas le cas, celui-ci est rejeté.
. Si le paquet est déclaré correct, la couche IP désencapsule le segment TCP qui s'y trouve et le passe au-dessus à la couche *transport*.
. La couche *transport*, couche *TCP* dans notre exemple, examine le numéro du segment afin de restituer le bon ordre des segments.
. Elle calcule également une somme de vérification pour le segment TCP. S'il est trouvé correct, la couche TCP envoie un accusé de réception à la machine source, sinon le segment TCP est refusé.
. Il ne reste plus à la couche TCP qu'à transmettre la partie données du segment à l'application destinatrice de celles-ci dans la couche du dessus.

8.1.5 Les problèmes d'adressage dans l'Internet

Un *noeud* d'un réseau peut être un ordinateur, une imprimante intelligente, un serveur de fichiers, n'importe quoi en fait pouvant communiquer à l'aide des protocoles TCP/IP. Chaque noeud a une **adresse physique** ayant un format dépendant du type du réseau. Sur un réseau Ethernet, l'adresse physique est codée sur 6 octets. Une adresse d'un réseau X25 est un nombre à 14 chiffres.

L'**adresse Internet** d'un noeud est une adresse **logique** : elle est indépendante du matériel et du réseau utilisé. C'est une adresse sur 4 octets identifiant à la fois un réseau local et un noeud de ce réseau. L'adresse Internet est habituellement représentée sous la forme de 4 nombres, valeurs des 4 octets, séparés par un point. Ainsi l'adresse de la machine Lagaffe de la faculté des Sciences d'Angers est notée 193.49.144.1 et celle de la machine Liny 193.49.144.9. On en déduira que l'adresse Internet du réseau local est 193.49.144.0. On pourra avoir jusqu'à 254 noeuds sur ce réseau.

Parce que les adresses Internet ou adresses IP sont indépendantes du réseau, une machine d'un réseau A peut communiquer avec une machine d'un réseau B sans se préoccuper du type de réseau sur lequel elle se trouve : il suffit qu'elle connaisse son adresse IP. Le protocole IP de chaque réseau se charge de faire la conversion adresse IP <--> adresse physique, dans les deux sens.

Les adresses **IP** doivent être toutes différentes. En France, c'est l'INRIA qui s'occupe d'affecter les adresses IP. En fait, cet organisme délivre une adresse pour votre réseau local, par exemple 193.49.144.0 pour le réseau de la faculté des sciences d'Angers. L'administrateur de ce réseau peut ensuite affecter les adresses IP 193.49.144.1 à 193.49.144.254 comme il l'entend. Cette adresse est généralement inscrite dans un fichier particulier de chaque machine reliée au réseau.

8.1.5.1 Les classes d'adresses IP

Une adresse IP est une suite de 4 octets notée souvent I1.I2.I3.I4, qui contient en fait deux adresses :

. l'adresse du réseau
. l'adresse d'un noeud de ce réseau

Selon la taille de ces deux champs, les adresses IP sont divisées en 3 classes : classes A, B et C.

Classe A

L'adresse IP : I1.I2.I3.I4 a la forme R1.N1.N2.N3 où

R1	est l'adresse du réseau
N1.N2.N3	est l'adresse d'une machine dans ce réseau

Plus exactement, la forme d'une adresse IP de classe A est la suivante :

```
      1 octet                3 octets
+-----------------------------------------------------------------------------+
¦0 ¦ adr. réseau    ¦      adresse noeud                                      ¦
+-----------------------------------------------------------------------------+
```

L'adresse réseau est sur 7 bits et l'adresse du noeud sur 24 bits. On peut donc avoir 127 réseaux de classe A, chacun comportant jusqu'à 2^{24} noeuds.

Classe B

Ici, l'adresse IP : I1.I2.I3.I4 a la forme R1.R2.N1.N2 où

R1.R2	est l'adresse du réseau
N1.N2	est l'adresse d'une machine dans ce réseau

Plus exactement, la forme d'une adresse IP de classe B est la suivante :

```
2 octets                                        2 octets
+-----------------------------------------------------------------------------+
¦10 ¦adresse réseau                        ¦  adresse noeud                   ¦
+-----------------------------------------------------------------------------+
```

L'adresse du réseau est sur 2 octets (14 bits exactement) ainsi que celle du noeud. On peut donc avoir 2^{14} réseaux de classe B chacun comportant jusqu'à 2^{16} noeuds.

Classe C

Dans cette classe, l'adresse IP : I1.I2.I3.I4 a la forme R1.R2.R3.N1 où

R1.R2.R3	est l'adresse du réseau
N1	est l'adresse d'une machine dans ce réseau

Plus exactement, la forme d'une adresse IP de classe C est la suivante :

```
   3 octets                                        1 octet
+-----------------------------------------------------------------------------+
¦110¦              adresse réseau                          ¦ adr. noeud        ¦
+-----------------------------------------------------------------------------+
```

L'adresse réseau est sur 3 octets (moins 3 bits) et l'adresse du noeud sur 1 octet. On peut donc avoir 2^{21} réseaux de classe C comportant jusqu'à 256 noeuds.

L'adresse de la machine *Lagaffe* de la faculté des sciences d'Angers étant 193.49.144.1, on voit que l'octet de poids fort vaut 193, c'est à dire en binaire 11000001. On en déduit que le réseau est de classe C.

Adresses réservées

- Certaines adresses IP sont des adresses de réseaux plutôt que des adresses de noeuds dans le réseau. Ce sont celles, où l'adresse du noeud est mise à 0. Ainsi, l'adresse 193.49.144.0 est l'adresse IP du réseau de la Faculté des Sciences d'Angers. En conséquence, aucun noeud d'un réseau ne peut avoir l'adresse zéro.
- Lorsque dans une adresse IP, l'adresse du noeud ne comporte que des 1, on a alors une adresse de diffusion : cette adresse désigne **tous les noeuds du réseau**.
- Dans un réseau de classe C, permettant théoriquement 2^8=256 noeuds, si on enlève les deux adresses interdites, on n'a plus que 254 adresses autorisées.

8.1.5.2 Les protocoles de conversion Adresse Internet <--> Adresse physique

Nous avons vu que lors d'une émission d'informations d'une machine vers une autre, celles-ci à la traversée de la couche IP étaient encapsulées dans des paquets. Ceux-ci ont la forme suivante :

```
 . <---- En-tête paquet IP   ---------------------------> . <---Données paquet IP ------------>.
+-----------------------------------------------------------------------------------------------+
| Info   |Adresse Internet       |Adresse Internet       |                                      |
|        |Source                 |Destination            |                                      |
+-----------------------------------------------------------------------------------------------+
```

Le paquet IP contient donc les adresses Internet des machines source et destination. Lorsque ce paquet va être transmis à la couche chargée de l'envoyer sur le réseau physique, d'autres informations lui sont ajoutées pour former la trame physique qui sera finalement envoyée sur le réseau. Par exemple, le format d'une trame sur un réseau Ethernet est le suivant :

```
.     <---- En-tête trame Ethernet   ----------------------->     .        <-Données trame Ethernet->.
+-----------------------------------------------------------------------------------------------+
| Info   |Adresse Physique       |Adresse Physique       |longueur|  Paquet IP              |Ethernet|
|        |Source                 |Destination            | paquet |                         |  CRC   |
+-----------------------------------------------------------------------------------------------+
   8 oct         6                       6                  2         46 à 1500                4
```

Dans la trame finale, il y a l'adresse physique des machines source et destination. Comment sont-elles obtenues ?

La machine expéditrice connaissant l'adresse IP de la machine avec qui elle veut communiquer obtient l'adresse physique de celle-ci en utilisant un protocole particulier appelé **ARP** (*Address Resolution Protocol*).

- Elle envoie un paquet d'un type spécial appelé paquet ARP contenant l'adresse IP de la machine dont on cherche l'adresse physique. Elle a pris soin également d'y placer sa propre adresse IP ainsi que son adresse physique.
- Ce paquet est envoyé à tous les noeuds du réseau.
- Ceux-ci reconnaissent la nature spéciale du paquet. Le noeud qui reconnaît son adresse IP dans le paquet, répond en envoyant à l'expéditeur du paquet son adresse physique. Comment le peut-il ? Il a trouvé dans le paquet les adresses IP et physique de l'expéditeur.
- L'expéditeur reçoit donc l'adresse physique qu'il cherchait. Il la stocke en mémoire afin de pouvoir l'utiliser ultérieurement si d'autres paquets sont à envoyer au même destinataire.

L'adresse IP d'une machine est normalement inscrite dans l'un de ses fichiers qu'elle peut donc consulter pour la connaître. Cette adresse peut être changée : il suffit d'éditer le fichier. L'adresse physique elle, est inscrite dans une mémoire de la carte réseau et ne peut être changée.

Lorsqu'un administrateur désire d'organiser son réseau différemment, il peut être amené à changer les adresses IP de tous les noeuds et donc à éditer les différents fichiers de configuration des différents noeuds. Cela peut être fastidieux et une occasion d'erreurs s'il y a beaucoup de machines. Une méthode consiste à ne pas affecter d'adresse IP aux machines : on inscrit alors un code spécial dans le fichier dans lequel la machine devrait trouver son adresse IP. Découvrant qu'elle n'a pas d'adresse IP, la machine la demande selon un protocole appelé **RARP** *(Reverse Address Resolution Protocol).* Elle envoie alors sur un réseau un paquet spécial appelé paquet RARP, analogue au paquet ARP précédent, dans lequel elle met son adresse physique. Ce paquet est envoyé à tous les noeuds qui reconnaissent alors un paquet RARP. L'un d'entre-eux, appelé **serveur RARP**, possède un fichier donnant la correspondance adresse physique <--> adresse IP de tous les noeuds. Il répond alors à l'expéditeur du paquet RARP, en lui renvoyant son adresse IP. Un administrateur désirant reconfigurer son réseau, n'a donc qu'à éditer le fichier de correspondances du serveur RARP. Celui-ci doit normalement avoir une adresse IP fixe qu'il doit pouvoir connaître sans avoir à utiliser lui-même le protocole RARP.

8.1.6 La couche réseau dite couche IP de l'internet

Le protocole IP *(Internet Protocol)* définit la forme que les paquets doivent prendre et la façon dont ils doivent être gérés lors de leur émission ou de leur réception. Ce type de paquet particulier est appelé un **datagramme IP**. Nous l'avons déjà présenté :

```
. <---- En-tête paquet IP   --------------------------> . <---Données paquet IP ------------>.
+-------------------------------------------------------------------------------------------+
¦ Info   ¦Adresse Internet       ¦Adresse Internet       ¦                                    ¦
¦        ¦Source                 ¦Destination            ¦                                    ¦
+-------------------------------------------------------------------------------------------+
```

L'important est qu'outre les données à transmettre, le datagramme IP contient les adresses Internet des machines source et destination. Ainsi la machine destinatrice sait qui lui envoie un message.
A la différence d'une trame de réseau qui a une longueur déterminée par les caractéristiques physiques du réseau sur lequel elle transite, la longueur du datagramme IP est elle fixée par le logiciel et sera donc la même sur différents réseaux physiques. Nous avons vu qu'en descendant de la couche réseau dans la couche physique le datagramme IP était encapsulé dans une trame physique. Nous avons donné l'exemple de la trame physique d'un réseau Ethernet :

```
. <---- En-tête trame Ethernet   ------------------------------>. <---Données trame Ethernet------>.
+-------------------------------------------------------------------------------------------------+
¦ Info   ¦Adresse Physique       ¦Adresse Physique       ¦ Type du¦   Paquet IP             ¦Ethernet¦
¦        ¦Source                 ¦Destination            ¦ paquet ¦                         ¦  CRC   ¦
+-------------------------------------------------------------------------------------------------+
```

Les trames physiques circulent de noeud en noeud vers leur destination qui peut ne pas être sur le même réseau physique que la machine expéditrice. Le paquet IP peut donc être encapsulé successivement dans des trames physiques différentes au niveau des noeuds qui font la jonction entre deux réseaux de type différent. Il se peut aussi que le paquet IP soit trop grand pour être encapsulé dans une trame physique. Le logiciel IP du noeud où se pose ce problème, décompose alors le paquet IP en *fragments* selon des règles précises, chacun d'eux étant ensuite envoyé sur le réseau physique. Ils ne seront réassemblés qu'à leur ultime destination.

8.1.6.1 Le routage

Le routage est la méthode d'acheminement des paquets IP à leur destination. Il y a deux méthodes : le routage direct et le routage indirect.

Routage direct

Le routage direct désigne l'acheminement d'un paquet IP directement de l'expéditeur au destinataire à l'intérieur du même réseau :

. La machine expéditrice d'un datagramme IP a l'adresse IP du destinataire.
. Elle obtient l'adresse physique de ce dernier par le protocole ARP ou dans ses tables, si cette adresse a déjà été obtenue.
. Elle envoie le paquet sur le réseau à cette adresse physique.

Routage indirect

Le routage indirect désigne l'acheminement d'un paquet IP à une destination se trouvant sur un autre réseau que celui auquel appartient l'expéditeur. Dans ce cas, les parties adresse réseau des adresses IP des machines source et destination sont différentes. La machine source reconnaît ce point. Elle envoie alors le paquet à un noeud spécial appelé **routeur** (*router*), noeud qui connecte un réseau local aux autres réseaux et dont elle trouve l'adresse IP dans ses tables, adresse obtenue initialement soit dans un fichier soit dans une mémoire permanente ou encore via des informations circulant sur le réseau.
Un **routeur** est attaché à deux réseaux et possède une adresse IP à l'intérieur de ces deux réseaux.

```
                  +------------+
        réseau 2 ¦ routeur    ¦  réseau 1
----------------|193.49.144.6|------------
 193.49.145.0   ¦193.49.145.3¦  193.49.144.0
                  +------------+
```

Dans notre exemple ci-dessus :

. Le réseau n° 1 a l'adresse Internet 193.49.144.0 et le réseau n° 2 l'adresse 193.49.145.0.
. A l'intérieur du réseau n° 1, le routeur a l'adresse 193.49.144.6 et l'adresse 193.49.145.3 à l'intérieur du réseau n° 2.

Le routeur a pour rôle de mettre le paquet IP qu'il reçoit et qui est contenu dans une trame physique typique du réseau n° 1, dans une trame physique pouvant circuler sur le réseau n° 2. Si l'adresse IP du destinataire du paquet est dans le réseau n° 2, le routeur lui enverra le paquet directement sinon il l'enverra à un autre routeur, connectant le réseau n° 2 à un réseau n° 3 et ainsi de suite.

8.1.6.2 Messages d'erreur et de contrôle

Toujours dans la couche réseau, au même niveau donc que le protocole IP, existe le protocole **ICMP** (*Internet Control Message Protocol*). Il sert à envoyer des messages sur le fonctionnement interne du réseau : noeuds en panne, embouteillage à un routeur, etc ... Les messages ICMP sont encapsulés dans des paquets IP et envoyés sur le réseau. Les couches IP des différents noeuds prennent les actions appropriées selon les messages ICMP qu'elles reçoivent. Ainsi, une application elle-même, ne voit jamais ces problèmes propres au réseau.
Un noeud utilisera les informations ICMP pour mettre à jour ses tables de routage.

8.1.7 La couche transport : les protocoles UDP et TCP

8.1.7.1 Le protocole UDP : User Datagram Protocol

Le protocole UDP permet un échange non fiable de données entre deux points, c'est à dire que le bon acheminement d'un paquet à sa destination n'est pas garanti. L'application, si elle le souhaite peut gérer cela elle-même, en attendant par exemple après l'envoi d'un message, un accusé de réception, avant d'envoyer le suivant.
Pour l'instant, au niveau réseau, nous avons parlé d'adresses IP de machines. Or sur une machine, peuvent coexister en même temps différents processus qui tous peuvent communiquer. Il faut donc indiquer, lors de l'envoi d'un message, non seulement l'adresse IP de la machine destinatrice, mais également le "nom" du processus destinataire. Ce nom est en fait un numéro, appelé **numéro de port**. Certains numéros sont réservés à des applications standard : port 69 pour l'application **tftp** *(trivial file transfer protocol)* par exemple.

Les paquets gérés par le protocole UDP sont appelés également des **datagrammes**. Ils ont la forme suivante :

```
. <---- En-tête datagramme UDP  ----------->. <---Données datagramme UDP-------->.
+-------------------------------------------------------------------------------+
¦ Port source       ¦Port destination       ¦                                     ¦
¦                   ¦                       ¦                                     ¦
+-------------------------------------------------------------------------------+
```

Ces datagrammes seront encapsulés dans des paquets IP, puis dans des trames physiques.

8.1.7.2 Le protocole TCP : Transfer Control Protocol

Pour des communications sûres, le protocole UDP est insuffisant : le développeur d'applications doit élaborer lui-même un protocole lui permettant de détecter le bon acheminement des paquets.
Le protocole **TCP** *(Transfer Control Protocol)* évite ces problèmes. Ses caractéristiques sont les suivantes :

. Le processus qui souhaite émettre établit tout d'abord une **connexion** avec le processus destinataire des informations qu'il va émettre. Cette connexion se fait entre un port de la machine émettrice et un port de la machine réceptrice. Il y a entre les deux ports un chemin virtuel qui est ainsi créé et qui sera réservé aux deux seuls processus ayant réalisé la connexion.
. Tous les paquets émis par le processus source suivent ce chemin virtuel et arrivent dans l'ordre où ils ont été émis ce qui n'était pas garanti dans le protocole UDP puisque les paquets pouvaient suivre des chemins différents.
. L'information émise a un aspect continu. Le processus émetteur envoie des informations à son rhythme. Celles-ci ne sont pas nécessairement envoyées tout de suite : le protocole TCP attend d'en avoir assez pour les envoyer. Elles sont stockées dans une structure appelée *segment TCP*. Ce segment une fois rempli sera transmis à la couche IP où il sera encapsulé dans un paquet IP.
. Chaque segment envoyé par le protocole TCP est numéroté. Le protocole TCP destinataire vérifie qu'il reçoit bien les segments en séquence. Pour chaque segment correctement reçu, il envoie un accusé de réception à l'expéditeur.
. Lorsque ce dernier le reçoit, il l'indique au processus émetteur. Celui-ci peut donc savoir qu'un segment est arrivé à bon port, ce qui n'était pas possible avec le protocole UDP.

- Si au bout d'un certain temps, le protocole TCP ayant émis un segment ne reçoit pas d'accusé de réception, il retransmet le segment en question, garantissant ainsi la qualité du service d'acheminement de l'information.
- Le circuit virtuel établi entre les deux processus qui communiquent est *full-duplex* : cela signifie que l'information peut transiter dans les deux sens. Ainsi le processus destination peut envoyer des accusés de réception alors même que le processus source continue d'envoyer des informations. Cela permet par exemple au protocole TCP source d'envoyer plusieurs segments sans attendre d'accusé de réception. S'il réalise au bout d'un certain temps qu'il n'a pas reçu l'accusé de réception d'un certain segment n° *n*, il reprendra l'émission des segments à ce point.

8.1.8 La couche Applications

Au-dessus des protocoles UDP et TCP, existent divers protocoles standard :

TELNET

Ce protocole permet à un utilisateur d'une machine A du réseau de se connecter sur une machine B (appelée souvent machine hôte). TELNET émule sur la machine A un terminal dit universel. L'utilisateur se comporte donc comme s'il disposait d'un terminal connecté à la machine B. Telnet s'appuie sur le protocole TCP.

FTP : (File Transfer protocol)

Ce protocole permet l'échange de fichiers entre deux machines distantes ainsi que des manipulations de fichiers tels que des créations de répertoire par exemple. Il s'appuie sur le protocole TCP.

TFTP: (Trivial File Transfer Control)

Ce protocole est une variante de FTP. Il s'appuie sur le protocole UDP et est moins sophistiqué que FTP.

DNS : (Domain Name System)

Lorsqu'un utilisateur désire échanger des fichiers avec une machine distante, par FTP par exemple, il doit connaître l'adresse Internet de cette machine. Par exemple, pour faire du FTP sur la machine Lagaffe de l'université d'Angers, il faudrait lancer FTP comme suit : FTP 193.49.144.1

Cela oblige à avoir un annuaire faisant la correspondance machine <--> adresse IP. Probablement que dans cet annuaire les machines seraient désignées par des noms symboliques tels que :

machine DPX2/320 de l'université d'Angers
machine Sun de l'ISERPA d'Angers

On voit bien qu'il serait plus agréable de désigner une machine par un nom plutôt que par son adresse IP. Se pose alors le problème de l'unicité du nom : il y a des millions de machines interconnectées. On pourrait imaginer qu'un organisme centralisé attribue les noms. Ce serait sans doute assez lourd. Le contrôle des noms a été en fait distribué dans des **domaines**. Chaque domaine est géré par un organisme généralement très léger qui a toute liberté quant au choix des noms de machines. Ainsi les machines en France appartiennent au domaine **fr**, domaine géré par l'Inria de Paris. Pour continuer à simplifier les choses, on distribue encore le contrôle : des domaines sont créés à l'intérieur du domaine **fr**. Ainsi l'université d'Angers appartient au domaine **univ-Angers**. Le service gérant ce domaine a toute liberté pour nommer les machines du réseau de l'Université d'Angers. Pour l'instant ce domaine n'a pas été subdivisé. Mais dans une grande université comportant beaucoup de machines en réseau, il pourrait l'être.

La machine DPX2/320 de l'université d'Angers a été nommée *Lagaffe* alors qu'un PC 486DX50 a été nommé *liny*. Comment référencer ces machines de l'extérieur ? En précisant la hiérarchie des domaines auxquelles elles appartiennent. Ainsi le nom complet de la machine Lagaffe sera :

Lagaffe.univ-Angers.fr

A l'intérieur des domaines, on peut utiliser des noms relatifs. Ainsi à l'intérieur du domaine **fr** et en dehors du domaine **univ-Angers**, la machine Lagaffe pourra être référencée par

Lagaffe.univ-Angers

Enfin, à l'intérieur du domaine *univ-Angers*, elle pourra être référencée simplement par

Lagaffe

Une application peut donc référencer une machine par son nom. Au bout du compte, il faut quand même obtenir l'adresse Internet de cette machine. Comment cela est-il réalisé ? Suposons que d'une machine A, on veuille communiquer avec une machine B.

- si la machine B appartient au même domaine que la machine A, on trouvera probablement son adresse IP dans un fichier de la machine A.
- sinon, la machine A trouvera dans un autre fichier ou le même que précédemment, une liste de quelques **serveurs de noms** avec leurs adresses IP. Un *serveur de noms* est chargé de faire la correspondance entre un nom de machine et son adresse IP. La machine A va envoyer une requête spéciale au premier serveur de nom de sa liste, appelé requête DNS incluant donc le nom de la machine recherchée. Si le serveur interrogé a ce nom dans ses tablettes, il enverra à la machine A, l'adresse IP correspondante. Sinon, le serveur trouvera lui aussi dans ses fichiers, une liste de serveurs de noms qu'il peut interroger. Il le fera alors. Ainsi un certain nombre de serveurs de noms vont être interrogés, pas de façon anarchique mais d'une façon à minimiser les requêtes. Si la machine est finalement trouvée, la réponse redescendra jusqu'à la machine A.

XDR : (eXternal Data Representation)

Créé par sun MicroSystems, ce protocole spécifie une représentation standard des données, indépendante des machines.

RPC : (Remote Procedure Call)

Défini également par sun, c'est un protocole de communication entre applications distantes, indépendant de la couche transport. Ce protocole est important : il décharge le programmeur de la connaissance des détails de la couche transport et rend les applications portables. Ce protocole s'appuie sur sur le protocole XDR

NFS : Network File System

Toujours défini par Sun, ce protocole permet à une machine, de "voir" le système de fichiers d'une autre machine. Il s'appuie sur le protocole RPC précédent.

8.1.9 Conclusion

Nous avons présenté dans cette introduction quelques grandes lignes des protocoles Internet. Pour approfondir ce domaine, on pourra lire l'excellent livre de Douglas Comer :

Titre	TCP/IP : Architecture, Protocoles, Applications.
Auteur	Douglas COMER
Editeur	InterEditions

8.2 Gestion des adresses réseau

Une machine sur le réseau Internet est définie de façon unique par une adresse IP (Internet Protocol) de la forme I1.I2.I3.I4 où I_n est un nombre entre 1 et 254. Elle peut être également définie par un nom également unique. Ce nom n'est pas obligatoire, les applications utilisant toujours au final les adresses IP des machines. lls sont là pour faciliter la vie des utilisateurs. Ainsi il est plus facile, avec un navigateur, de demander l'URL *http://www.ibm.com* que l'URL *http://129.42.17.99* bien que les deux méthodes soient possibles. L'association adresse IP <--> nomMachine est assurée par un service distribué de l'internet appelé DNS (Domain Name System). La plate-forme .NET offre la classe **Dns** pour gérer les adresses internet :

```
// from module 'c:\winnt\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll'
public sealed class System.Net.Dns :
    object
{

    // Methods
    public static IAsyncResult BeginGetHostByName(string hostName, AsyncCallback requestCallback, object
stateObject);
    public static IAsyncResult BeginResolve(string hostName, AsyncCallback requestCallback, object
stateObject);
    public static System.Net.IPHostEntry EndGetHostByName(IAsyncResult asyncResult);
    public static System.Net.IPHostEntry EndResolve(IAsyncResult asyncResult);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public static System.Net.IPHostEntry GetHostByAddress(System.Net.IPAddress address);
    public static System.Net.IPHostEntry GetHostByAddress(string address);
    public static System.Net.IPHostEntry GetHostByName(string hostName);
    public static string GetHostName();
    public Type GetType();
    public static System.Net.IPHostEntry Resolve(string hostName);
    public virtual string ToString();
} // end of System.Net.Dns
```

La plupart des méthodes offertes sont statiques. Regardons celles qui nous intéressent :

`GetHostByAddress (string address)`	rend une adresse *IPHostEntry* à partir d'une adresse IP sous la forme "I1.I2.I3.I4". Lance une exception si la machine *address* ne peut être trouvée.

`GetHostByName(string name)`	rend une adresse *IPHostEntry* à partir d'un nom de machine. Lance une exception si la machine *name* ne peut être trouvée.
`string GetHostName()`	rend le nom de la machine sur laquelle s'exécute le programme qui joue cette instruction

Les adresses réseau de type *IPHostEntry* ont la forme suivante :

```
// from module 'c:\winnt\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll'
public class System.Net.IPHostEntry :
    object
{

    // Constructors
    public IPHostEntry();

    // Properties
    public IPAddress[] AddressList { get; set; }
    public string[] Aliases { get; set; }
    public string HostName { get; set; }

    // Methods
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
} // end of System.Net.IPHostEntry
```

Les propriétés qui nous intéressent :

`IPAddress [] AddressList`	liste des adresses IP d'une machine. Si une adresse IP désigne une et une seule machine physique, une machine physique peut elle avoir plusieurs adresses IP. Ce sera le cas si elle a plusieurs cartes réseau qui la connectent à des réseaux différents.
`string [] Aliases`	liste des alias d'une machine, pouvant être désignée par un nom principal et des alias
`string HostName`	le nom de la machine si elle en a un

De la classe *IPAddress* nous retiendrons le constructeur, les propriétés et méthodes suivants :

```
// from module 'c:\winnt\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll'
public class System.Net.IPAddress :
    object
{


    // Constructors
    public IPAddress(long newAddress);

    // Properties
    public long Address { get; set; }

    // Methods
    public static System.Net.IPAddress Parse(string ipString);
    public virtual string ToString();
} // end of System.Net.IPAddress
```

De façon interne, une adresse IP est codée sous la forme d'un entier long dans la propriété :

```
    public long Address { get; set; }
```

Elle peut être transformée en chaîne *I1.I2.I3.I4* avec la méthode *ToString()*. Inversement, on peut obtenir un objet *IPAddress* à partir d'une chaîne *I1.I2.I3.I4* avec la méthode statique *IPAddress.Parse("I1.I2.I3.I4")*.

Considérons le programme suivant qui affiche le nom de la machine sur laquelle il s'exécute puis de façon interactive donne les correspondances adresse IP <--> nom Machine :

```
E:\data\serge\MSNET\c#\sockets\1>address1
Machine Locale=tahe

Machine recherchée (fin pour arrêter) : istia.univ-angers.fr
Machine : istia.univ-angers.fr
Adresses IP : 193.49.146.171

Machine recherchée (fin pour arrêter) : 193.49.146.171
Machine : istia.istia.univ-angers.fr
Adresses IP : 193.49.146.171
Alias : 171.146.49.193.in-addr.arpa

Machine recherchée (fin pour arrêter) : www.ibm.com
Machine : www.ibm.com
Adresses IP : 129.42.17.99,129.42.18.99,129.42.19.99,129.42.16.99
```

```
Machine recherchée (fin pour arrêter) : 129.42.17.99
Machine : www.ibm.com
Adresses IP : 129.42.17.99

Machine recherchée (fin pour arrêter) : x.y.z
Impossible de trouver la machine [x.y.z]

Machine recherchée (fin pour arrêter) : localhost
Machine : tahe
Adresses IP : 127.0.0.1

Machine recherchée (fin pour arrêter) : 127.0.0.1
Machine : tahe
Adresses IP : 127.0.0.1

Machine recherchée (fin pour arrêter) : tahe
Machine : tahe
Adresses IP : 127.0.0.1

Machine recherchée (fin pour arrêter) : fin
```

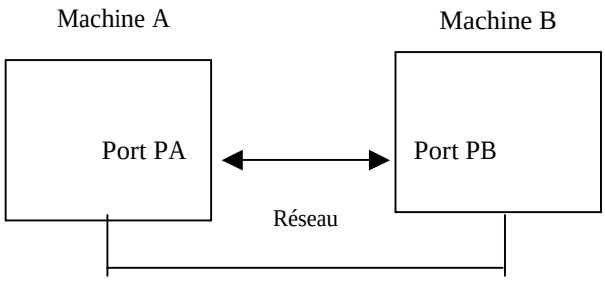
Le programme est le suivant :

```
// espaces de noms
using System;
using System.Net;
using System.Text.RegularExpressions;

public class adresses{
  public static void Main(){
    // affiche le nom de la machine locale
    // puis donne interactivement des infos sur les machines réseau
    // identifiées par un nom ou une adresse IP

    // machine locale
    string localHost=Dns.GetHostName();
    Console.Out.WriteLine("Machine Locale="+localHost);

    // question-réponses interactives
    string machine;
    IPHostEntry adresseMachine;
    while(true){
      // saisie du nom de la machine recherchée
      Console.Out.Write("Machine recherchée (fin pour arrêter) : ");
      machine=Console.In.ReadLine().Trim().ToLower();
      // fini ?
      if(machine=="fin") break;
      // qq chose à analyser ?
      if(machine=="") continue;

      // adresse I1.I2.I3.I4 ou nom de machine ?
      bool isIPV4=Regex.IsMatch(machine,@"^\s*\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\s*$");
      // gestion exception
      try{
        // recherche machine
        if (isIPV4)adresseMachine=Dns.GetHostByAddress(machine);
          else adresseMachine=Dns.GetHostByName(machine);
        // le nom
        Console.Out.WriteLine("Machine : " +adresseMachine.HostName);
        // les adresses IP
        Console.Out.Write("Adresses IP : " + adresseMachine.AddressList[0]);
        for (int i=1;i<adresseMachine.AddressList.Length;i++){
          Console.Out.Write(","+adresseMachine.AddressList[i]);
        }//for
        Console.Out.WriteLine();
        // les alias
        if(adresseMachine.Aliases.Length!=0){
          Console.Out.Write("Alias : "+adresseMachine.Aliases[0]);
          for (int i=1;i<adresseMachine.Aliases.Length;i++){
            Console.Out.Write(","+adresseMachine.Aliases[i]);
          }//for
          Console.Out.WriteLine();
        }//if
      }catch{
        // la machine n'existe pas
        Console.Out.WriteLine("Impossible de trouver la machine ["+machine+"]");
      }//catch
      // machine suivante
    }//while
  }//main
}//classe
```

8.3 Programmation TCP-IP

8.3.1 Généralités

Considérons la communication entre deux machines distantes A et B :



Lorsque une application *AppA* d'une machine A veut communiquer avec une application *AppB* d'une machine B de l'Internet, elle doit connaître plusieurs choses :

- l'adresse IP ou le nom de la machine B
- le numéro du port avec lequel travaille l'application *AppB*. En effet la machine B peut supporter de nombreuses applications qui travaillent sur l'Internet. Lorsqu'elle reçoit des informations provenant du réseau, elle doit savoir à quelle application sont destinées ces informations. Les applications de la machine B ont accès au réseau via des guichets appelés également des ports de communication. Cette information est contenue dans le paquet reçu par la machine B afin qu'il soit délivré à la bonne application.
- les protocoles de communication compris par la machine B. Dans notre étude, nous utiliserons uniquement les protocoles TCP-IP.
- le protocole de dialogue accepté par l'application *AppB*. En effet, les machines A et B vont se "parler". Ce qu'elles vont dire va être encapsulé dans les protocoles TCP-IP. Néanmoins, lorsqu'au bout de la chaîne, l'application *AppB* va recevoir l'information envoyée par l'applicaton *AppA*, il faut qu'elle soit capable de l'interpréter. Ceci est analogue à la situation où deux personnes A et B communiquent par téléphone : leur dialogue est transporté par le téléphone. La parole va être codée sous forme de signaux par le téléphone A, transportée par des lignes téléphoniques, arriver au téléphone B pour y être décodée. La personne B entend alors des paroles. C'est là qu'intervient la notion de protocole de dialogue : si A parle français et que B ne comprend pas cette langue, A et B ne pourront dialoguer utilement.

Aussi les deux applications communicantes doivent -elles être d'accord sur le type de dialogue qu'elles vont adopter. Par exemple, le dialogue avec un service *ftp* n'est pas le même qu'avec un service *pop* : ces deux services n'acceptent pas les mêmes commandes. Elles ont un protocole de dialogue différent.

8.3.2 Les caractéristiques du protocole TCP

Nous n'étudierons ici que des communications réseau utilisant le protocole de transport TCP. Rappelons ici, les caractéristiques de celui-ci :

- Le processus qui souhaite émettre établit tout d'abord une **connexion** avec le processus destinataire des informations qu'il va émettre. Cette connexion se fait entre un port de la machine émettrice et un port de la machine réceptrice. Il y a entre les deux ports un chemin virtuel qui est ainsi créé et qui sera réservé aux deux seuls processus ayant réalisé la connexion.
- Tous les paquets émis par le processus source suivent ce chemin virtuel et arrivent dans l'ordre où ils ont été émis
- L'information émise a un aspect continu. Le processus émetteur envoie des informations à son rythme. Celles-ci ne sont pas nécessairement envoyées tout de suite : le protocole TCP attend d'en avoir assez pour les envoyer. Elles sont stockées dans une structure appelée *segment TCP*. Ce segment une fois rempli sera transmis à la couche IP où il sera encapsulé dans un paquet IP.
- Chaque segment envoyé par le protocole TCP est numéroté. Le protocole TCP destinataire vérifie qu'il reçoit bien les segments en séquence. Pour chaque segment correctement reçu, il envoie un accusé de réception à l'expéditeur.
- Lorsque ce dernier le reçoit, il l'indique au processus émetteur. Celui-ci peut donc savoir qu'un segment est arrivé à bon port.
- Si au bout d'un certain temps, le protocole TCP ayant émis un segment ne reçoit pas d'accusé de réception, il retransmet le segment en question, garantissant ainsi la qualité du service d'acheminement de l'information.
- Le circuit virtuel établi entre les deux processus qui communiquent est *full-duplex* : cela signifie que l'information peut transiter dans les deux sens. Ainsi le processus destination peut envoyer des accusés de réception alors même que le processus source continue d'envoyer des informations. Cela permet par exemple au protocole TCP source d'envoyer

plusieurs segments sans attendre d'accusé de réception. S'il réalise au bout d'un certain temps qu'il n'a pas reçu l'accusé de réception d'un certain segment n° *n*, il reprendra l'émission des segments à ce point.

8.3.3 La relation client-serveur

Souvent, la communication sur Internet est dissymétrique : la machine A initie une connexion pour demander un service à la machine B : il précise qu'il veut ouvrir une connexion avec le service SB1 de la machine B. Celle-ci accepte ou refuse. Si elle accepte, la machine A peut envoyer ses demandes au service SB1. Celles-ci doivent se conformer au protocole de dialogue compris par le service SB1. Un dialogue demande-réponse s'instaure ainsi entre la machine A qu'on appelle machine **cliente** et la machine B qu'on appelle machine **serveur**. L'un des deux partenaires fermera la connexion.

8.3.4 Architecture d'un client

L'architecture d'un programme réseau demandant les services d'une application serveur sera la suivante :

```
ouvrir la connexion avec le service SB1 de la machine B
si réussite alors
 tant que ce n'est pas fini
   préparer une demande
   l'émettre vers la machine B
   attendre et récupérer la réponse
   la traiter
 fin tant que
finsi
```

8.3.5 Architecture d'un serveur

L'architecture d'un programme offrant des services sera la suivante :

```
ouvrir le service sur la machine locale
tant que le service est ouvert
   se mettre à l'écoute des demandes de connexion sur un port dit port d'écoute
   lorsqu'il y a une demande, la faire traiter par une autre tâche sur un autre port dit port de
   service
fin tant que
```

Le programme serveur traite différemment la demande de connexion initiale d'un client de ses demandes ultérieures visant à obtenir un service. Le programme n'assure pas le service lui-même. S'il le faisait, pendant la durée du service il ne serait plus à l'écoute des demandes de connexion et des clients ne seraient alors pas servis. Il procède donc autrement : dès qu'une demande de connexion est reçue sur le port d'écoute puis acceptée, le serveur crée une tâche chargée de rendre le service demandé par le client. Ce service est rendu sur un autre port de la machine serveur appelé **port de service**. On peut ainsi servir plusieurs clients en même temps.

Une tâche de service aura la structure suivante :

```
tant que le service n'a pas été rendu totalement
   attendre une demande sur le port de service
   lorsqu'il y en a une, élaborer la réponse
   transmettre la réponse via le port de service
fin tant que
libérer le port de service
```

8.3.6 La classe TcpClient

La classe T**cpClient** est la classe qui convient pour représenter le client d'un service TCP. Elle est définie comme suit :

```
// from module 'c:\winnt\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll'
public class System.Net.Sockets.TcpClient :
    object,
    IDisposable
{

    // Constructors
    public TcpClient();
    public TcpClient(string hostname, int port);
    public TcpClient(System.Net.IPEndPoint localEP);

    // Properties
    public LingerOption LingerState { get; set; }
    public bool NoDelay { get; set; }
```

```
    public int ReceiveBufferSize { get; set; }
    public int ReceiveTimeout { get; set; }
    public int SendBufferSize { get; set; }
    public int SendTimeout { get; set; }

    // Methods
    public void Close();
    public void Connect(System.Net.IPAddress address, int port);
    public void Connect(string hostname, int port);
    public void Connect(System.Net.IPEndPoint remoteEP);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public System.Net.Sockets.NetworkStream GetStream();
    public Type GetType();
    public virtual string ToString();
} // end of System.Net.Sockets.TcpClient
```

Les constructeurs, méthodes et propriétés qui nous intéressent sont les suivants :

`constructeur TcpClient(string hostname, int port)`	crée une liaison tcp avec le serveur opérant sur le port indiqué (*port*) de la machine indiquée (*hostname*). Par exemple *new TcpClient("istia.univ-angers.fr",80)* pour se connecter au port 80 de la machine *istia.univ-angers.fr*
`void Close()`	ferme la connexion au serveur Tcp
`GetStream()`	obtient un flux de lecture et d'écriture vers le serveur. C'est ce flux qui permet les échanges client-serveur.

8.3.7 La classe NetworkStream

La classe **NetworkStream** représente le flux réseau entre le client et le serveur. La classe est définie comme suit :

```
// from module 'c:\winnt\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll'
public class System.Net.Sockets.NetworkStream :
    System.IO.Stream,
    IDisposable
{

    // Fields

    // Constructors
    public NetworkStream(System.Net.Sockets.Socket socket);
    public NetworkStream(System.Net.Sockets.Socket socket, System.IO.FileAccess access);
    public NetworkStream(System.Net.Sockets.Socket socket, System.IO.FileAccess access, bool ownsSocket);
    public NetworkStream(System.Net.Sockets.Socket socket, bool ownsSocket);

    // Properties
    public bool CanRead { virtual get; }
    public bool CanSeek { virtual get; }
    public bool CanWrite { virtual get; }
    public bool DataAvailable { virtual get; }
    public long Length { virtual get; }
    public long Position { virtual get; virtual set; }

    // Methods
    public virtual IAsyncResult BeginRead(byte[] buffer, int offset, int size, AsyncCallback callback,
object state);
    public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int size, AsyncCallback callback,
object state);
    public virtual void Close();
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public virtual int EndRead(IAsyncResult asyncResult);
    public virtual void EndWrite(IAsyncResult asyncResult);
    public virtual bool Equals(object obj);
    public virtual void Flush();
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public virtual int Read(byte[] buffer, int offset, int size);
    public virtual int ReadByte();
    public virtual long Seek(long offset, System.IO.SeekOrigin origin);
    public virtual void SetLength(long value);
    public virtual string ToString();
    public virtual void Write(byte[] buffer, int offset, int size);
    public virtual void WriteByte(byte value);
} // end of System.Net.Sockets.NetworkStream
```

La classe *NetworkStream* est dérivée de la classe *Stream*. Beaucoup d'applications client-serveur échangent des lignes de texte terminées par les caractères de fin de ligne "\r\n". Aussi il est intéressant d'utiliser des objets *StreamReader* et *StreamWriter* pour lire et écrire ces lignes dans le flux réseau. Lorsque deux machines communiquent, il y a à chaque bout de la liaison un objet *TcpClient*. La méthode *GetStream* de cet objet permet d'avoir accès au flux réseau (*NetworkStream*) qui lie les deux machines. Ainsi si une machine M1 a établi une liaison avec une machine M2 à l'aide d'un objet TcpClient *client1* qu'elles échangent des lignes de texte, elle pourra créer ses flux de lecture et écriture de la façon suivante :

```
StreamReader in1=new StreamReader(client1.GetStream());
StreamWriter out1=new StreamWriter(client1.GetStream());
out1.AutoFlush=true;
```

L'instruction

```
out1.AutoFlush=true;
```

signifie que le flux d'écriture de *client1* ne transitera pas par un buffer intermédiaire mais ira directement sur le réseau. Ce point est important. En général lorsque *client1* envoie une ligne de texte à son partenaire il en attend une réponse. Celle-ci ne viendra jamais si la ligne a été en réalité bufferisée sur la machine M1 et jamais envoyée.

Pour envoyer une ligne de texte à la machine M2, on écrira :

```
client1.WriteLine("un texte");
```

Pour lire la réponse de M2, on écrira :

```
string réponse=client1.ReadLine();
```

8.3.8 Architecture de base d'un client internet

Nous avons maintenant les éléments pour écrire l'architecture de base d'un client internet :

```
  TcpClient  client=null;
  try{
      // on se connecte au service officiant sur le port P de la machine M
    client=new TcpClient(M,P);

    // on crée les flux d'entrée-sortie du client TCP
    StreamReader in=new StreamReader(client.GetStream());
    StreamWriter out=new StreamWriter(client.getStream());
    out.AutoFlush=true;

    // boucle demande - réponse
    bool  fini=false;
    String demande;
    String réponse;
    while (! fini){
      // on prépare la demande
      demande=…
      // on l'envoie
      out.WriteLine(demande);
      // on lit la réponse
      réponse=in.ReadLine();
      // on traite la réponse
      …
    }
    // c'est fini
    client.Close();
  } catch(Exception e){
    // on gère l'exception
    ….
  }
```

8.3.9 La classe TcpListener

La classe **TcpListener** est la classe qui convient pour représenter un service TCP. Elle est définie comme suit :

```
// from module 'c:\winnt\assembly\gac\system\1.0.3300.0__b77a5c561934e089\system.dll'
public class System.Net.Sockets.TcpListener :
    object
{

    // Constructors
    public TcpListener(System.Net.IPAddress localaddr, int port);
    public TcpListener(System.Net.IPEndPoint localEP);
    public TcpListener(int port);

    // Properties
    public EndPoint LocalEndpoint { get; }

    // Methods
    public System.Net.Sockets.Socket AcceptSocket();
    public System.Net.Sockets.TcpClient AcceptTcpClient();
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public bool Pending();
    public void Start();
    public void Stop();
```

```
    public virtual string ToString();
} // end of System.Net.Sockets.TcpListener
```

Les constructeurs, méthodes et propriétés qui nous intéressent sont les suivants :

`constructeur TcpListener(int port)`	crée un service TCP qui va attendre (*listen*) les demandes des clients sur un port passé en paramètre (*port*) appelé port d'écoute.
`TcpClient AcceptTcpClient()`	accepte la demande d'un client. Rend comme résultat un objet *TcpClient* associé à un autre port, appelé port de service.
`void Start()`	lance l'écoute des demandes clients
`void Stop()`	arrête d'écouter les demandes clients

8.3.10 Architecture de base d'un serveur Internet

De ce qui a été vu précédemment, on peut déduire la structure de base d'un serveur :

```
TcpListener serveur=null;
try{
  // ouverture du service
  int portEcoute=…
  serveur=new TcpListener(portEcoute);
  serveur.Start();

  // traitement des demandes de connexion
  bool fini=false;
  TcpClient client=null;
  while( ! fini){
    // attente et acceptation d'une demande
    client=serveur.AcceptTcpClient();

    // le service est rendu par une autre tâche à laquelle on passe la socket de service
    new Thread(new ThreadStart(new Service(client).Run)).Start();

    // on se remet en attente des demandes de connexion
  }
  // c'est fini - on arrête le service
  serveur.Stop();
} catch (Exception e){
  // on traite l'exception
  …
}
```

La classe *Service* est un *thread* qui pourrait avoir l'allure suivante :

```
public class Service{

  TcpClient liaisonClient;   // la liaison au client à servir

  // constructeur
  public Service(TcpClient liaisonClient){
    this.liaisonClient=liaisonClient;
  }// constructeur

// run
  public void Run(){
    StreamReader in=null;// flux d'entrée de la liaison tcp
    StreamWriter out=null; // flux de sortie de la liaison tcp
    try{
      // on crée les flux d'entrée-sortie
      in=new StreamReader(liaisonClient.getStream()));
      out=new StreamWriter(liaisonClient.getStream());

      // boucle demande - réponse
      bool  fini=false;
      String demande;
      String réponse;
      while (! fini){
        // on lit la demande du clien
        demande=in.ReadLine();

        // on la traite
        …

        // on prépare la réponse
        réponse=…

        // on l'envoie au client
        out.WriteLine(réponse);
      }//while
      // c'est fini
      liaisonClient.Close();
    } catch(Exception e){
    // on gère l'exception
```

```
    ….
    }// try
  } // run
}//classe
```

8.4 Exemples

8.4.1 Serveur d'écho

Nous nous proposons d'écrire un serveur d'écho qui sera lancé depuis une fenêtre DOS par la commande :

serveurEcho port

Le serveur officie sur le port passé en paramètre. Il se contente de renvoyer au client la demande que celui-ci lui a envoyée. Le programme est le suivant :

```
// appel : serveurEcho port
// serveur d'écho
// renvoie au client la ligne que celui-ci lui a envoyée


using System.Net.Sockets;
using System;
using System.IO;
using System.Threading;

public class serveurEcho{
  public const string syntaxe="Syntaxe : serveurEcho port";

  // programme principal
  public static void Main (string[] args){

   // y-a-t-il un argument
   if(args.Length != 1)
    erreur(syntaxe,1);

   // cet argument doit être entier >0
   int port=0;
   bool erreurPort=false;
   Exception E=null;
   try{
    port=int.Parse(args[0]);
   }catch(Exception e){
      E=e;
      erreurPort=true;
   }
   erreurPort=erreurPort || port <=0;
   if(erreurPort)
    erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

   // on crée le servive d'écoute
  TcpListener ecoute=null;
  int nbClients=0;  // nbre de clients traités
    try{
      // on crée le service
      ecoute=new TcpListener(port);
      // on le lance
      ecoute.Start();
      // suivi
      Console.Out.WriteLine("Serveur d'écho lancé sur le port " + port);
      Console.Out.WriteLine(ecoute.LocalEndpoint);

      // boucle de service
      TcpClient liaisonClient=null;
      while (true){ // boucle infinie - sera arrêtée par Ctrl-C
        // attente d'un client
        liaisonClient=ecoute.AcceptTcpClient();

        // le service est assuré par une autre tâche
        nbClients++;
        new Thread(new ThreadStart(new traiteClientEcho(liaisonClient,nbClients).Run)).Start();

        // on retourne à l'écoute des demandes
      }// fin while
    }catch(Exception ex){
      // on signale l'erreur
      erreur("L'erreur suivante s'est produite : " + ex.Message,3);
    }//catch
    // fin du service
    ecoute.Stop();
  }// fin main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
```

```
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}// fin classe serveurEcho

// -------------------------------------------------------
// assure le service à un client du serveur d'écho

public class traiteClientEcho{

  private TcpClient liaisonClient;    // liaison avec le client
  private int numClient;             // n° de client
  private StreamReader IN;           // flux d'entrée
  private StreamWriter OUT;            // flux de sortie

  // constructeur
  public traiteClientEcho(TcpClient liaisonClient, int numClient){
    this.liaisonClient=liaisonClient;
    this.numClient=numClient;
  }//constructeur

  // méthode run
  public void Run(){
    // rend le service au client
    Console.Out.WriteLine("Début de service au client "+numClient);
    try{
      // flux d'entrée
      IN=new StreamReader(liaisonClient.GetStream());
      // flux de sortie
      OUT=new StreamWriter(liaisonClient.GetStream());
      OUT.AutoFlush=true;
      // boucle lecture demande/écriture réponse
      string demande=null,reponse=null;
      while ((demande=IN.ReadLine())!=null){
        // suivi
        Console.Out.WriteLine("Client " + numClient + " : " + demande);
        // le service s'arrête lorsque le client envoie une marque de fin de fichier
        reponse="["+demande+"]";
        OUT.WriteLine(reponse);
        // le service s'arrête lorsque le client envoie "fin"
        if(demande.Trim().ToLower()=="fin") break;
      }// fin while
      // fin liaison
      liaisonClient.Close();
    } catch (Exception e){
      erreur("Erreur lors de la fermeture de la liaison client ("+e+")",2);
    }// fin try
    // fin du service
    Console.Out.WriteLine("Fin de service au client "+numClient);
  }// fin run

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}// fin class
```

La structure du serveur est conforme à l'architecture générale des serveurs tcp.

8.4.2 Un client pour le serveur d'écho

Nous écrivons maintenant un client pour le serveur précédent. Il sera appelé de la façon suivante :

clientEcho nomServeur port

Il se connecte à la machine *nomServeur* sur le port *port* puis envoie au serveur des lignes de texte que celui-ci lui renvoie en écho.

```
// espaces de noms
using System;
using System.Net.Sockets;
using System.IO;

public class clientEcho{

  // se connecte à un serveur d'écho
  // toute ligne tapée au clavier est alors reçue en écho

  public static void Main(string[] args){
    // syntaxe
    const string syntaxe="pg machine port";

    // nombre d'arguments
```

```
    if(args.Length != 2)
      erreur(syntaxe,1);

    // on note le nom du serveur
    string nomServeur=args[0];

    // le port doit être entier >0
    int port=0;
    bool erreurPort=false;
    Exception E=null;
    try{
      port=int.Parse(args[1]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

    // on peut travailler
    TcpClient  client=null;  // le client
    StreamReader IN=null;  // le flux de lecture du client
    StreamWriter OUT=null; // le flux d'écriture du client
    string demande=null;   // demande du client
    string réponse=null;   // réponse du serveur
    try{
      // on se connecte au service officiant sur le port P de la machine M
      client=new TcpClient(nomServeur,port);

      // on crée les flux d'entrée-sortie du client TCP
      IN=new StreamReader(client.GetStream());
      OUT=new StreamWriter(client.GetStream());
      OUT.AutoFlush=true;

      // boucle demande - réponse
      while (true){
        // la demande vient du clavier
        Console.Out.Write("demande (fin pour arrêter) : ");
        demande=Console.In.ReadLine();
        // on l'envoie au serveur
        OUT.WriteLine(demande);
        // on lit la réponse du serveur
        réponse=IN.ReadLine();
        // on traite la réponse
        Console.Out.WriteLine("Réponse : " + réponse);
        // fini ?
        if(demande.Trim().ToLower()=="fin") break;
      }//while
      // c'est fini
      client.Close();
    } catch(Exception e){
      // on gère l'exception
      erreur(e.Message,3);
    }//catch
  }//main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe
```

La structure de ce client est conforme à l'architecture générale des clients *tcp*.Voici les résultats obtenus dans la configuration suivante :

- le serveur est lancé sur le port 100 dans une fenêtre Dos
- sur la même machine deux clients sont lancés dans deux autres fenêtres Dos

Dans la fenêtre du client 1 on a les résultats suivants :

```
E:\data\serge\MSNET\c#\sockets\serveurEcho>clientEcho localhost 100
demande (fin pour arrêter) : ligne1
Réponse : [ligne1]
demande (fin pour arrêter) : ligne1B
Réponse : [ligne1B]
demande (fin pour arrêter) : ligne1C
Réponse : [ligne1C]
demande (fin pour arrêter) : fin
Réponse : [fin]
```

Dans celle du client 2 :

```
E:\data\serge\MSNET\c#\sockets\serveurEcho>clientEcho localhost 100
```

```
demande (fin pour arrêter) : ligne2A
Réponse : [ligne2A]
demande (fin pour arrêter) : ligne2B
Réponse : [ligne2B]
demande (fin pour arrêter) : fin
Réponse : [fin]
```

Dans celle du serveur :

```
E:\data\serge\MSNET\c#\sockets\serveurEcho>serveurEcho 100
Serveur d'écho lancé sur le port 100
0.0.0.0:100
Début de service au client 1
Client 1 : ligne1
Début de service au client 2
Client 2 : ligne2A
Client 2 : ligne2B
Client 1 : ligne1B
Client 1 : ligne1C
Client 2 : fin
Fin de service au client 2
Client 1 : fin
Fin de service au client 1
^C
```

On remarquera que le serveur a bien été capable de servir deux clients simultanément.

8.4.3 Un client TCP générique

Beaucoup de services créés à l'origine de l'Internet fonctionnent selon le modèle du serveur d'écho étudié précédemment : les échanges client-serveur se font pas échanges de lignes de texte. Nous allons écrire un client tcp générique qui sera lancé de la façon suivante : **cltgen serveur port**

Ce client TCP se connectera sur le port *port* du serveur *serveur*. Ceci fait, il créera deux threads :

1. un thread chargé de lire des commandes tapées au clavier et de les envoyer au serveur
2. un thread chargé de lire les réponses du serveur et de les afficher à l'écran

Pourquoi deux threads alors que dans l'application précédente ce besoin ne s'était pas fait ressentir ? Dans cette dernière, le protocole du dialogue était connu : le client envoyait une seule ligne et le serveur répondait par une seule ligne. Chaque service a son protocole particulier et on trouve également les situations suivantes :

- le client doit envoyer plusieurs lignes de texte avant d'avoir une réponse
- la réponse d'un serveur peut comporter plusieurs lignes de texte

Aussi la boucle envoi d'une unique ligne au seveur - réception d'une unique ligne envoyée par le serveur ne convient-elle pas toujours. On va donc créer deux boucles dissociées :

- une boucle de lecture des commandes tapées au clavier pour être envoyées au serveur. L'utilisateur signalera la fin des commandes avec le mot clé *fin*.
- une boucle de réception et d'affichage des réponses du serveur. Celle-ci sera une boucle infinie qui ne sera interrompue que par la fermeture du flux réseau par le serveur ou par l'utilisateur au clavier qui tapera la commande *fin*.

Pour avoir ces deux boucles dissociées, il nous faut deux threads indépendants. Montrons un exemple d'excécution où notre client tcp générique se connecte à un service SMTP (SendMail Transfer Protocol). Ce service est responsable de l'acheminement du courrier électronique aux destinataires. Il fonctionne sur le port 25 et a un protocole de dialogue de type échanges de lignes de texte.

```
E:\data\serge\MSNET\c#\réseau\client tcp générique>cltgen istia.univ-angers.fr 25
Commandes :
<-- 220 istia.univ-angers.fr ESMTP Sendmail 8.11.6/8.9.3; Mon, 13 May 2002 08:37:26 +0200
help
<-- 502 5.3.0 Sendmail 8.11.6 -- HELP not implemented
mail from: machin@univ-angers.fr
<-- 250 2.1.0 machin@univ-angers.fr... Sender ok
rcpt to: serge.tahe@istia.univ-angers.fr
<-- 250 2.1.5 serge.tahe@istia.univ-angers.fr... Recipient ok
data
<-- 354 Enter mail, end with "." on a line by itself
Subject: test

ligne1
ligne2
```

```
ligne3
.
<-- 250 2.0.0 g4D6bks25951 Message accepted for delivery
quit
<-- 221 2.0.0 istia.univ-angers.fr closing connection
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

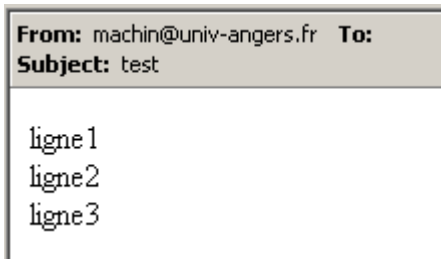
Commentons ces échanges client-serveur :

- le service SMTP envoie un message de bienvenue lorsqu'un client se connecte à lui :

```
<-- 220 istia.univ-angers.fr ESMTP Sendmail 8.11.6/8.9.3; Mon, 13 May 2002 08:37:26 +0200
```

- certains services ont une commande *help* donnant des indications sur les commandes utilisables avec le service. Ici ce n'est pas le cas. Les commandes SMTP utilisées dans l'exemple sont les suivantes :
  - **mail from:** *expéditeur*, pour indiquer l'adresse électronique de l'expéditeur du message
  - **rcpt to:** *destinataire*, pour indiquer l'adresse électronique du destinataire du message. S'il y a plusieurs destinataires, on ré-émet autant de fois que nécessaire la commande **rcpt to:** pour chacun des destinataires.
  - **data** qui signale au serveur SMTP qu'on va envoyer le message. Comme indiqué dans la réponse du serveur, celui-ci est une suite de lignes terminée par une ligne contenant le seul caractère point. Un message peut avoir des entêtes séparés du corps du message par une ligne vide. Dans notre exemple, nous avons mis un sujet avec le mot clé **Subject:**
- une fois le message envoyé, on peut indiquer au serveur qu'on a terminé avec la commande **quit**. Le serveur ferme alors la connexion réseau.Le thread de lecture peut détecter cet événement et s'arrêter.
- l'utilisateur tape alors **fin** au clavier pour arrêter également le thread de lecture des commandes tapées au clavier.

Si on vérifie le courrier reçu, nous avons la chose suivante (Outlook) :



On remarquera que le service SMTP ne peut détecter si un expéditeur est valide ou non. Aussi ne peut-on jamais faire confiance au champ *from* d'un message. Ici l'expéditeur *machin@univ-angers.fr* n'existait pas.

Ce client tcp générique peut nous permettre de découvrir le protocole de dialogue de services internet et à partir de là construire des classes spécialisées pour des clients de ces services. Découvrons le protocole de dialogue du service POP (Post Office Protocol) qui permet de retrouver ses méls stockés sur un serveur. Il travaille sur le port 110.

```
E:\data\serge\MSNET\c#\réseau\client tcp générique>cltgen istia.univ-angers.fr 110
Commandes :
<-- +OK Qpopper (version 4.0.3) at istia.univ-angers.fr starting.
help
<-- -ERR Unknown command: "help".
user st
<-- +OK Password required for st.
pass monpassword
<-- +OK st has 157 visible messages (0 hidden) in 11755927 octets.
list
<-- +OK 157 visible messages (11755927 octets)
<-- 1 892847
<-- 2 171661
...
<-- 156 2843
<-- 157 2796
<-- .
retr 157
<-- +OK 2796 octets
<-- Received: from lagaffe.univ-angers.fr (lagaffe.univ-angers.fr [193.49.144.1])
<--     by istia.univ-angers.fr (8.11.6/8.9.3) with ESMTP id g4D6wZs26600;
<--     Mon, 13 May 2002 08:58:35 +0200
<-- Received: from jaume ([193.49.146.242])
<--     by lagaffe.univ-angers.fr (8.11.1/8.11.2/GeO20000215) with SMTP id g4D6wSd37691;
```

```
<--     Mon, 13 May 2002 08:58:28 +0200 (CEST)
...
<-- ----------------------------------------------------------------------
<-- NOC-RENATER2                  Tl.  : 0800 77 47 95
<-- Fax : (+33) 01 40 78 64 00 ,  Email : noc-r2@cssi.renater.fr
<-- ----------------------------------------------------------------------
<--
<-- .
quit
<-- +OK Pop server at istia.univ-angers.fr signing off.
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Les principales commandes sont les suivantes :

- **user** *login*, où on donne son login sur la machine qui détient nos méls
- **pass** *password*, où on donne le mot de passe associé au login précédent
- **list**, pour avoir la liste des messages sous la forme numéro, taille en octets
- **retr** *i*, pour lire le message n° i
- **quit**, pour arrêter le dialogue.

Découvrons maintenant le protocole de dialogue entre un client et un serveur Web qui lui travaille habituellement sur le port 80 :

```
E:\data\serge\MSNET\c#\réseau\client tcp générique>cltgen istia.univ-angers.fr 80
Commandes :
GET /index.html HTTP/1.0

<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
<--
<-- <head>
<-- <meta http-equiv="Content-Type"
<-- content="text/html; charset=iso-8859-1">
<-- <meta name="GENERATOR" content="Microsoft FrontPage Express 2.0">
<-- <title>Bienvenue a l'ISTIA - Universite d'Angers</title>
<-- </head>
....
<-- face="Verdana"> - Dernire mise  jour le <b>10 janvier 2002</b></font></p>
<-- </body>
<-- </html>
<--
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Un client Web envoie ses commandes au serveur selon le schéma suivant :

commande1
commande2
...
commanden
[ligne vide]

Ce n'est qu'après avoir reçu la ligne vide que le serveur Web répond. Dans l'exemple nous n'avons utilisé qu'une commande :

```
GET /index.html HTTP/1.0
```

qui demande au serveur l'URL **/index.html** et indique qu'il travaille avec le protocole HTTP version 1.0. La version la plus récente de ce protocole est 1.1. L'exemple montre que le serveur a répondu en renvoyant le contenu du fichier *index.html* puis qu'il a fermé la connexion puisqu'on voit le thread de lecture des réponses se terminer. Avant d'envoyer le contenu du fichier *index.html*, le serveur web a envoyé une série d'entêtes terminée par une ligne vide :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
```

```
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

La ligne *<html>* est la première ligne du fichier */index.html*. Ce qui précède s'appelle des entêtes HTTP (HyperText Transfer Protocol). Nous n'allons pas détailler ici ces entêtes mais on se rappellera que notre client générique y donne accès, ce qui peut être utile pour les comprendre. La première ligne par exemple :

```
<-- HTTP/1.1 200 OK
```

indique que le serveur Web contacté comprend le protocole HTTP/1.1 et qu'il a bien trouvé le fichier demandé (200 OK), 200 étant un code de réponse HTTP. Les lignes

```
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
```

disent au client qu'il va recevoir 11251 octets représentant du texte HTML (HyperText Markup Language) et qu'à la fin de l'envoi, la connexion sera fermée.

On a donc là un client tcp très pratique. En fait, ce client existe déjà sur les machines où il s'appelle *telnet* mais il était intéressant de l'écrire nous-mêmes. Le programme du client tcp générique est le suivant :

```
// espaces de noms
using System;
using System.Net.Sockets;
using System.IO;
using System.Threading;

public class clientTcpGénérique{

  // reçoit en paramètre les caractéristiques d'un service sous la forme
  // serveur port
  // se connecte au service
  // crée un thread pour lire des commandes tapées au clavier
  // celles-ci seront envoyées au serveur
  // crée un thread pour lire les réponses du serveur
  // celles-ci seront affichées à l'écran
  // le tout se termine avec la commande fin tapée au clavier

  public static void Main(string[] args){

    // syntaxe
    const string syntaxe="pg serveur port";

    // nombre d'arguments
    if(args.Length != 2)
      erreur(syntaxe,1);

    // on note le nom du serveur
    string serveur=args[0];

    // le port doit être entier >0
    int port=0;
    bool erreurPort=false;
    Exception E=null;
    try{
      port=int.Parse(args[1]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

    TcpClient client=null;
    // il peut y avoir des problèmes
    try{
      // on se connecte au service
      client=new TcpClient(serveur,port);
    }catch(Exception ex){
      // erreur
      Console.Error.WriteLine("Impossible de se connecter au service ("+ serveur
        +","+port+"), erreur : "+ex.Message);
      // fin
      return;
```

```
    }//catch

    // on crée les threads de lecture/écriture
    Thread thReceive=new Thread(new ThreadStart(new clientReceive(client).Run));
    Thread thSend=new Thread(new ThreadStart(new clientSend(client).Run));

    // on lance l'exécution des deux threads
    thSend.Start();
    thReceive.Start();

    // fin thread Main
    return;
  }// Main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe

public class clientSend{
  // classe chargée de lire des commandes tapées au clavier
  // et de les envoyer à un serveur via un client tcp passé au constructeur

  TcpClient client; // le client tcp

  // constructeur
  public clientSend(TcpClient client){
    // on note le client tcp
    this.client=client;
  }//constructeur

  // méthode Run du thread
  public void Run(){

    // données locales
    StreamWriter OUT=null; // flux d'écriture réseau
    string commande=null;  // commande lue au clavier

    // gestion des erreurs
    try{
      // création du flux d'écriture réseau
      OUT=new StreamWriter(client.GetStream());
      OUT.AutoFlush=true;
      // boucle saisie-envoi des commandes
      Console.Out.WriteLine("Commandes : ");
      while(true){
        // lecture commande tapée au clavier
        commande=Console.In.ReadLine().Trim();
        // fini ?
        if (commande.ToLower()=="fin") break;
        // envoi commande au serveur
        OUT.WriteLine(commande);
        // commande suivante
      }//while
    }catch(Exception ex){
      // erreur
      Console.Error.WriteLine("L'erreur suivante s'est produite : " + ex.Message);
    }//catch
    // fin - on ferme les flux
    try{
      OUT.Close();client.Close();
    }catch{}
    // on signale la fin du thread
    Console.Out.WriteLine("[fin du thread d'envoi des commandes au serveur]");
  }//run
}//classe

public class clientReceive{
  // classe chargée de lire les lignes de texte destinées à un
  // client tcp passé au constructeur

  TcpClient client; // le client tcp

  // constructeur
  public clientReceive(TcpClient client){
    // on note le client tcp
    this.client=client;
  }//constructeur

  // méthode Run du thread
  public void Run(){

    // données locales
    StreamReader IN=null;  // flux lecture réseau
    string réponse=null;   // réponse serveur

    // gestion des erreurs
    try{
```

```
      // création du flux lecture réseau
      IN=new StreamReader(client.GetStream());
      // boucle lecture lignes de texte du flux IN
      while(true){
        // lecture flux réseau
        réponse=IN.ReadLine();
        // flux fermé ?
        if(réponse==null) break;
        // affichage
        Console.Out.WriteLine("<-- "+réponse);
      }//while
    }catch(Exception ex){
      // erreur
      Console.Error.WriteLine("L'erreur suivante s'est produite : " + ex.Message);
    }//catch
    // fin - on ferme les flux
    try{
      IN.Close();client.Close();
    }catch{}
    // on signale la fin du thread
    Console.Out.WriteLine("[fin du thread de lecture des réponses du serveur]");
  }//run
}//classe
```

8.4.4 Un serveur Tcp générique

Maintenant nous nous intéressons à un serveur

- qui affiche à l'écran les commandes envoyées par ses clients
- leur envoie comme réponse les lignes de texte tapées au clavier par un utilisateur. C'est donc ce dernier qui fait office de serveur.

Le programme est lancé par : **srvgen portEcoute**, où *portEcoute* est le port sur lequel les clients doivent se connecter. Le service au client sera assuré par deux threads :

- un thread se consacrant exclusivement à la lecture des lignes de texte envoyées par le client
- un thread se consacrant exclusivement à la lecture des réponses tapées au clavier par l'utilisateur. Celui-ci signalera par la commande **fin** qu'il clôt la connexion avec le client.

Le serveur crée deux threads par client. S'il y a n clients, il y aura 2n threads actifs en même temps. Le serveur lui ne s'arrête jamais sauf par un Ctrl-C tapé au clavier par l'utilisateur. Voyons quelques exemples.

Le serveur est lancé sur le port 100 et on utilise le client générique pour lui parler. La fenêtre du client est la suivante :

```
E:\data\serge\MSNET\c#\réseau\client tcp générique>cltgen localhost 100
Commandes :
commande 1 du client 1
<-- réponse 1 au client 1
commande 2 du client 1
<-- réponse 2 au client 1
fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du thread de lecture des réponses du serveur]
[fin du thread d'envoi des commandes au serveur]
```

Les lignes commençant par <-- sont celles envoyées du serveur au client, les autres celles du client vers le serveur. La fenêtre du serveur est la suivante :

```
E:\data\serge\MSNET\c#\réseau\serveur tcp générique>srvgen 100
Serveur générique lancé sur le port 100
Thread de lecture des réponses du serveur au client 1 lancé
1 : Thread de lecture des demandes du client 1 lancé
<-- commande 1 du client 1
réponse 1 au client 1
1 : <-- commande 2 du client 1
réponse 2 au client 1
1 : [fin du Thread de lecture des demandes du client 1]
fin
[fin du Thread de lecture des réponses du serveur au client 1]
```

Les lignes commençant par <-- sont celles envoyées du client au serveur. Les lignes *N :* sont les lignes envoyées du serveur au client n° N. Le serveur ci-dessus est encore actif alors que le client 1 est terminé. On lance un second client pour le même serveur :

```
E:\data\serge\MSNET\c#\réseau\client tcp générique>cltgen localhost 100
Commandes :
commande 3 du client 2
```

```
<-- réponse 3 au client 2
fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du thread de lecture des réponses du serveur]
[fin du thread d'envoi des commandes au serveur]
```

La fenêtre du serveur est alors celle-ci :

```
E:\data\serge\MSNET\c#\réseau\serveur tcp générique>srvgen 100
Serveur générique lancé sur le port 100
Thread de lecture des réponses du serveur au client 1 lancé
1 : Thread de lecture des demandes du client 1 lancé
<-- commande 1 du client 1
réponse 1 au client 1
1 : <-- commande 2 du client 1
réponse 2 au client 1
1 : [fin du Thread de lecture des demandes du client 1]
fin
[fin du Thread de lecture des réponses du serveur au client 1]
Thread de lecture des réponses du serveur au client 2 lancé
2 : Thread de lecture des demandes du client 2 lancé
<-- commande 3 du client 2
réponse 3 au client 2
2 : [fin du Thread de lecture des demandes du client 2]
fin
[fin du Thread de lecture des réponses du serveur au client 2]
^C
```

Simulons maintenant un serveur web en lançant notre serveur générique sur le port 88 :

```
E:\data\serge\MSNET\c#\réseau\serveur tcp générique>srvgen 88
Serveur générique lancé sur le port 88
```

Prenons maintenant un navigateur et demandons l'URL *http://localhost:88/exemple.html*. Le navigateur va alors se connecter sur le port *88* de la machine *localhost* puis demander la page */exemple.html* :

Regardons maintenant la fenêtre de notre serveur :

```
E:\data\serge\MSNET\c#\réseau\serveur tcp générique>srvgen 88
Serveur générique lancé sur le port 88
Thread de lecture des réponses du serveur au client 2 lancé
2 : Thread de lecture des demandes du client 2 lancé
<-- GET /exemple.html HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705; .NET CLR 1.0.2
914)
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
```

On découvre ainsi les entêtes HTTP envoyés par le navigateur. Cela nous permet de découvrir peu à peu le protocole HTTP. Lors d'un précédent exemple, nous avions créé un client Web qui n'envoyait que la seule commande GET. Cela avait été suffisant. On voit ici que le navigateur envoie d'autres informations au serveur. Elles ont pour but d'indiquer au serveur quel type de client il a en face de lui. On voit aussi que les entêtes HTTP se terminent par une ligne vide.

Elaborons une réponse à notre client. L'utilisateur au clavier est ici le véritable serveur et il peut élaborer une réponse à la main. Rappelons-nous la réponse faite par un serveur Web dans un précédent exemple :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
```

```
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

Essayons de donner une réponse analogue :

```
...
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
2 : HTTP/1.1 200 OK
2 : Server: serveur tcp generique
2 : Connection: close
2 : Content-Type: text/html
2 :
2 : <html>
2 :   <head><title>Serveur generique</title></head>
2 :   <body>
2 :     <center>
2 :       <h2>Reponse du serveur generique</h2>
2 :     </center>
2 :    </body>
2 : </html>
2 : fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du Thread de lecture des demandes du client 2]
[fin du Thread de lecture des réponses du serveur au client 2]
```

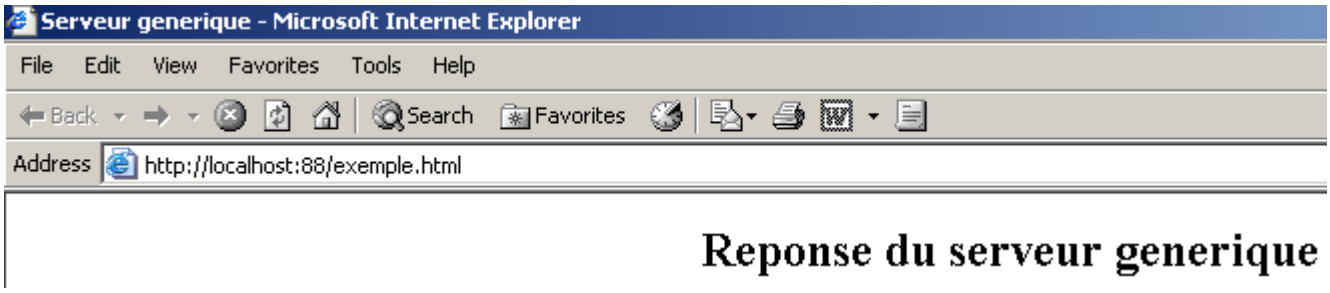
Les lignes commençant par *2 :* sont envoyées du serveur au client n° 2. La commande *fin* clôt la connexion du serveur au client. Nous nous sommes limités dans notre réponse aux entêtes HTTP suivants :

```
HTTP/1.1 200 OK
2 : Server: serveur tcp generique
2 : Connection: close
2 : Content-Type: text/html
2 :
```

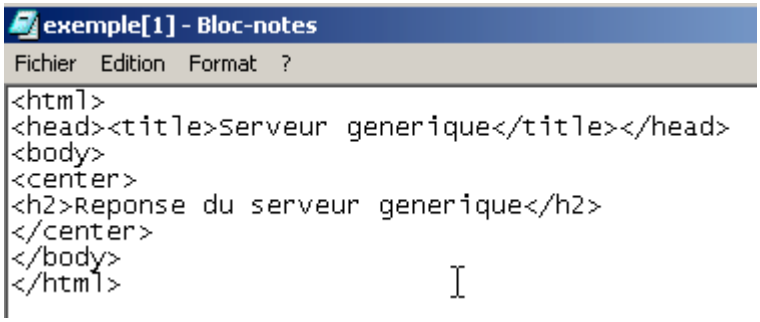
Nous ne donnons pas la taille du fichier que nous allons envoyer (*Content-Length*) mais nous contentons de dire que nous allons fermer la connexion (*Connection: close*) après envoi de celui-ci. Cela est suffisant pour le navigateur. En voyant la connexion fermée, il saura que la réponse du serveur est terminée et affichera la page HTML qui lui a été envoyée. Cette dernière est la suivante :

```
2 : <html>
2 :   <head><title>Serveur generique</title></head>
2 :   <body>
2 :     <center>
2 :       <h2>Reponse du serveur generique</h2>
2 :     </center>
2 :    </body>
2 : </html>
```

L'utilisateur ferme ensuite la connexion au client en tapant la commande *fin*. Le navigateur sait alors que la réponse du serveur est terminée et peut alors l'afficher :



Si ci-dessus, on fait *View/Source* pour voir ce qu'a reçu le navigateur, on obtient :



c'est à dire exactement ce qu'on a envoyé depuis le serveur générique.

8.4.5 Un client Web

Nous avons vu dans l'exemple précédent, certains des entêtes HTTP qu'envoyait un navigateur :

```
<-- GET /exemple.html HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705; .NET CLR 1.0.2
914)
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
```

Nous allons écrire un client Web auquel on passerait en paramètre une URL et qui afficherait à l'écran le texte envoyé par le serveur. Nous supposerons que celui-ci supporte le protocole HTTP 1.1. Des entêtes précédents, nous n'utiliserons que les suivants :

```
<-- GET /exemple.html HTTP/1.1
<-- Host: localhost:88
<-- Connection: close
```

- le premier entête indique quelle page nous désirons
- le second quel serveur nous interrogeons
- le troisième que nous souhaitons que le serveur ferme la connexion après nous avoir répondu.

Si ci-dessus, nous remplaçons GET par HEAD, le serveur ne nous enverra que les entêtes HTTP et pas la page HTML.

Notre client web sera appelé de la façon suivante : **clientweb URL cmd**, où **URL** est l'URL désirée et **cmd** l'un des deux mots clés GET ou HEAD pour indiquer si on souhaite seulement les entêtes (HEAD) ou également le contenu de la page (GET). Regardons un premier exemple. Nous lançons le serveur IIS puis le client web sur la même machine :

```
E:\data\serge\MSNET\c#\réseau\clientweb>clientweb http://localhost HEAD
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Mon, 13 May 2002 09:23:37 GMT
Connection: close
Location: /IISSamples/Default/welcome.htm
Content-Length: 189
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGUUY=HMFNCCMDECBJJBPPBHAOAJNP; path=/
Cache-control: private


E:\data\serge\MSNET\c#\réseau\clientweb>
```

La réponse

```
HTTP/1.1 302 Object moved
```

signifie que la page demandée a changé de place (donc d'URL). La nouvelle URL est donnée par l'entête **Location:**

```
Location: /IISSamples/Default/welcome.htm
```

Si nous utilisons GET au lieu de HEAD dans l'appel au client Web :

```
E:\data\serge\MSNET\c#\réseau\clientweb>clientweb http://localhost GET
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Mon, 13 May 2002 09:33:36 GMT
Connection: close
Location: /IISSamples/Default/welcome.htm
Content-Length: 189
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGUUY=IMFNCCMDAKPNNGMGMFIHENFE; path=/
Cache-control: private

<head><title>L'objet a changé d'emplacement</title></head>
<body><h1>L'objet a changé d'emplacement</h1>Cet objet peut être trouvé <a HREF="/IISSamples/Default/we
lcome.htm">ici</a>.</body>
```

Nous obtenons le même résultat qu'avec HEAD avec de plus le corps de la page HTML. Le programme est le suivant :

```
// espaces de noms
using System;
using System.Net.Sockets;
using System.IO;

public class clientWeb{

  // demande une URL
  // affiche le contenu de celle-ci à l'écran

  public static void Main(string[] args){
    // syntaxe
    const string syntaxe="pg URI GET/HEAD";

    // nombre d'arguments
    if(args.Length != 2)
      erreur(syntaxe,1);

    // on note l'URI demandée
    string URIstring=args[0];
    string commande=args[1].ToUpper();

    // vérification validité de l'URI
    Uri uri=null;
    try{
      uri=new Uri(URIstring);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.Message,2);
    }//catch
    // vérification de la commande
    if(commande!="GET" && commande!="HEAD"){
      // commande incorrecte
      erreur("Le second paramètre doit être GET ou HEAD",3);
    }

    // on peut travailler
    TcpClient  client=null;  // le client
    StreamReader IN=null;  // le flux de lecture du client
    StreamWriter OUT=null; // le flux d'écriture du client
    string réponse=null;   // réponse du serveur
    try{
      // on se connecte au serveur
      client=new TcpClient(uri.Host,uri.Port);

      // on crée les flux d'entrée-sortie du client TCP
      IN=new StreamReader(client.GetStream());
      OUT=new StreamWriter(client.GetStream());
      OUT.AutoFlush=true;

      // on demande l'URL - envoi des entêtes HTTP
      OUT.WriteLine(commande+" " + uri.PathAndQuery+ " HTTP/1.1");
      OUT.WriteLine("Host: " + uri.Host+":"+uri.Port);
      OUT.WriteLine("Connection: close");
      OUT.WriteLine();
      // on lit la réponse
      while((réponse=IN.ReadLine())!=null){
        // on traite la réponse
        Console.Out.WriteLine(réponse);
      }//while
      // c'est fini
      client.Close();
    } catch(Exception e){
      // on gère l'exception
      erreur(e.Message,4);
    }//catch
  }//main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
```

```
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe
```

La seule nouveauté dans ce programme est l'utilisation de la classe **Uri**. Le programme reçoit une URL (*Uniform Resource Locator*) ou URI (*Uniform Resource Identifier*) de la forme *http://serveur:port/cheminPageHTML?param1=val1;param2=val2;....* La classe **Uri** nous permet de décomposer la chaîne de l'URL en ses différents éléments. Un objet Uri est construit à partir de la chaîne URIstring reçue en paramètre :

```
    Uri uri=null;
    try{
      uri=new Uri(URIstring);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.Message,2);
    }//catch
```

Si la chaîne URI reçue en paramètre n'est pas une URI valide (absence du protocole, du serveur, ...), une exception est lancée. Cela nous permet de vérifier la validité du paramètre reçu. Une fois l'objet Uri construit, on a accès aux différents éléments de cette Uri. Ainsi si l'objet *uri* du code précédent a été construit à partir de la chaîne *http://serveur:port/cheminPageHTML?param1=val1;param2=val2;...* on aura :

uri.Host=*serveur*, **uri.Port**=*port*, **uri.Path**=*cheminPageHTML*, **uri.Query**=*param1=val1;param2=val2;...*, **uri.pathAndQuery**=*cheminPageHTML?param1=val1;param2=val2;...*, **uri.Scheme**=*http*.

8.4.6 Client Web gérant les redirections

Le client Web précédent ne gère pas une éventuelle redirection de l'URL qu'il a demandée. Le client suivant la gère.

1. il lit la première ligne des entêtes HTTP envoyés par le serveur pour vérifier si on y trouve la chaîne *302 Object moved* qui signale une redirection
2. il lit les entêtes suivants. S'il y a redirection, il recherche la ligne *Location: url* qui donne la nouvelle URL de la page demandée et note cette URL.
3. il affiche le reste de la réponse du serveur. S'il y a redirection, les étapes 1 à 3 sont répétées avec la nouvelle URL. Le programme n'accepte pas plus d'une redirection. Cette limite fait l'objet d'une constante qui peut être modifiée.

Voici un exemple :

```
E:\data\serge\MSNET\c#\réseau\clientweb>clientweb2 http://localhost GET
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Mon, 13 May 2002 11:38:55 GMT
Connection: close
Location: /IISSamples/Default/welcome.htm
Content-Length: 189
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGUUY=PDGNCCMDNCAOFDMPHCJNPBAI; path=/
Cache-control: private

<head><title>L'objet a chang d'emplacement</title></head>
<body><h1>L'objet a chang d'emplacement</h1>Cet objet peut tre trouv <a HREF="/IISSamples/Default/we
lcome.htm">ici</a>.</body>

<--Redirection vers l'URL http://localhost:80/IISSamples/Default/welcome.htm-->

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Connection: close
Date: Mon, 13 May 2002 11:38:55 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Mon, 16 Feb 1998 21:16:22 GMT
ETag: "0174e21203bbd1:978"
Content-Length: 4781

<html>

<head>
<title>Bienvenue dans le Serveur Web personnel</title>
</head>
....
</body>
```

```
</html>
```

Le programme est le suivant :

```
// espaces de noms
using System;
using System.Net.Sockets;
using System.IO;
using System.Text.RegularExpressions;

public class clientWeb{

  // demande une URL
  // affiche le contenu de celle-ci à l'écran

  public static void Main(string[] args){
    // syntaxe
    const string syntaxe="pg URI GET/HEAD";

    // nombre d'arguments
    if(args.Length != 2)
      erreur(syntaxe,1);

    // on note l'URI demandée
    string URIstring=args[0];
    string commande=args[1].ToUpper();

    // vérification validité de l'URI
    Uri uri=null;
    try{
      uri=new Uri(URIstring);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.Message,2);
    }//catch
    // vérification de la commande
    if(commande!="GET" && commande!="HEAD"){
      // commande incorrecte
      erreur("Le second paramètre doit être GET ou HEAD",3);
    }

    // on peut travailler
    TcpClient  client=null;       // le client
    StreamReader IN=null;       // le flux de lecture du client
    StreamWriter OUT=null;      // le flux d'écriture du client
    string réponse=null;        // réponse du serveur
    const int nbRedirsMax=1;    // pas plus d'une redirection acceptée
    int nbRedirs=0;             // nombre de redirections en cours
    string premièreLigne;       // 1ère ligne de la réponse
    bool redir=false;           // indique s'il y a redirection ou non
    string locationString="";     // la chaîne URI d'une éventuelle redirection

    // expression régulière pour trouver une URL de redirection
    Regex location=new Regex(@"^Location: (.+?)$");

    // gestion des erreurs
    try{
      // on peut avoir plusieurs URL à demander s'il y a des redirections
      while(nbRedirs<=nbRedirsMax){
        // on se connecte au serveur
        client=new TcpClient(uri.Host,uri.Port);

        // on crée les flux d'entrée-sortie du client TCP
        IN=new StreamReader(client.GetStream());
        OUT=new StreamWriter(client.GetStream());
        OUT.AutoFlush=true;

        // on envoie les entêtes HTTP pour demander l'URL
        OUT.WriteLine(commande+" " + uri.PathAndQuery+ " HTTP/1.1");
        OUT.WriteLine("Host: " + uri.Host+":"+uri.Port);
        OUT.WriteLine("Connection: close");
        OUT.WriteLine();

        // on lit la première ligne de la réponse
        premièreLigne=IN.ReadLine();
        // écho écran
        Console.Out.WriteLine(premièreLigne);

        // redirection ?
        if(Regex.IsMatch(premièreLigne,"302 Object moved$")){
          // il y a une redirection
          redir=true;
          nbRedirs++;
        }//if

        // entêtes HTTP suivants jusqu'à trouver la ligne vide signalant la fin des entêtes
        bool locationFound=false;
        while((réponse=IN.ReadLine())!=""){
          // on affiche la réponse
```

```
        Console.Out.WriteLine(réponse);
        // s'il y a redirection, on recherche l'entête Location
        if(redir && ! locationFound){
          // on compare la ligne à l'expression relationnelle location
          Match résultat=location.Match(réponse);
          if(résultat.Success){
            // si on a trouvé on note l'URL de redirection
            locationString=résultat.Groups[1].Value;
            // on note qu'on a trouvé
            locationFound=true;
          }//if
        }//if
        // entête suivant
      }//while

      // lignes suivantes de la réponse
      Console.Out.WriteLine(réponse);
      while((réponse=IN.ReadLine())!=null){
        // on affiche la réponse
        Console.Out.WriteLine(réponse);
      }//while
      // on ferme la connexion
      client.Close();
      // a-t-on fini ?
      if ( ! locationFound || nbRedirs>nbRedirsMax)
        break;
      // il y a une redirection à opérer - on construit la nouvelle Uri
      URIstring=uri.Scheme+"://"+uri.Host+":"+uri.Port+locationString;
      uri=new Uri(URIstring);
      // suivi
      Console.Out.WriteLine("\n<--Redirection vers l'URL "+URIstring+"-->\n");
    }//while
  } catch(Exception e){
    // on gère l'exception
    erreur(e.Message,4);
  }//catch
}//main

// affichage des erreurs
public static void erreur(string msg, int exitCode){
  // affichage erreur
  System.Console.Error.WriteLine(msg);
  // arrêt avec erreur
  Environment.Exit(exitCode);
}//erreur
}//classe
```

8.4.7 Serveur de calcul d'impôts

Nous reprenons l'exercice IMPOTS déjà traité sous diverses formes. Rappelons la dernière mouture :

Une classe **impôt** a été créée. Ses attributs sont trois tableaux de nombres :

```
public class impôt{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private decimal[] limites, coeffR, coeffN;
```

La classe a deux constructeurs :

- un constructeur à qui on passe les trois tableaux de données nécessaires au calcul de l'impôt

```
  // constructeur 1
  public impôt(decimal[] LIMITES, decimal[] COEFFR, decimal[] COEFFN){
    // initialise les trois tableaux limites, coeffR, coeffN à partir
    // des paramètres passés au constructeur
```

- un constructeur à qui on passe le nom DSN d'une base de données ODBC

```
  // constructeur 2
  public impôt(string DSNimpots, string Timpots, string colLimites, string colCoeffR, string colCoeffN){
    // initialise les trois tableaux limites, coeffR, coeffN à partir
    // du contenu de la table Timpots de la base ODBC DSNimpots
    // colLimites, colCoeffR, colCoeffN sont les trois colonnes de cette table
    // peut lancer une exception
```

Un programme de test avait été écrit :

```
E:\data\serge\MSNET\c#\impots\6>csc /r:impots.dll test.cs

E:\data\serge\MSNET\c#\impots\6>test mysql-impots timpots limites coeffr coeffn
```

```
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 200000
impôt=22506 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :n 2 200000
impôt=33388 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 3 200000
impôt=16400 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :n 3 300000
impôt=50082 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :n 3 200000
impôt=22506 F
```

Ici le programme de test et l'objet *impôt* étaient sur la même machine. Nous nous proposons de mettre le programme de test et l'objet *impôt* sur des machines différentes. Nous aurons une application client-serveur où l'objet *impôt* distant sera le serveur. La nouvelle classe s'appelle *ServeurImpots* et est dérivée de la classe *impôt* :

```
using System.Net.Sockets;
using System;
using System.IO;
using System.Threading;
using System.Text.RegularExpressions;
```

```
public class ServeurImpots : impôt {

  // attributs
  int portEcoute;       // le port d'écoute des demandes clients
  bool actif;           // état du serveur

  // constructeur
  public ServeurImpots(int portEcoute,string DSNimpots, string Timpots, string colLimites, string
colCoeffR, string colCoeffN)
    : base(DSNimpots, Timpots, colLimites, colCoeffR, colCoeffN) {
    // on note le port d'écoute
    this.portEcoute=portEcoute;
    // pour l'instant inactif
    actif=false;
    // crée et lance un thread de lecture des commandes tapées au clavier
    // le serveur sera géré à partir de ces commandes
    new Thread(new ThreadStart(admin)).Start();
  }//ServeurImpots
```

Le seul paramètre nouveau dans le constructeur est le port d'écoute des demandes des clients. Les autres paramètres sont passés directement à la classe de base *impôt*. Le serveur d'impôts est contrôlé par des commandes tapées au clavier. Aussi crée-t-on un thread pour lire ces commandes. Il y en aura deux possibles : *start* pour lancer le service, *stop* pour l'arrêter définitivement. La méthode *admin* qui gère ces commandes est la suivante :

```
public void admin(){
    // lit les commandes d'administration du serveur tapées au clavier
    // dans une boucle sans fin
    string commande=null;
    while(true){
      // invite
      Console.Out.Write("Serveur d'impôts>");
      // lecture commande
      commande=Console.In.ReadLine().Trim().ToLower();
      // exécution commande
      if(commande=="start"){
        // actif ?
        if(actif){
          //erreur
          Console.Out.WriteLine("Le serveur est déjà actif");
          // on continue
          continue;
        }//if
        // on lance le service d'écoute
        new Thread(new ThreadStart(ecoute)).Start();
      }//if
      else if(commande=="stop"){
        // fin de tous les threads d'exécution
        Environment.Exit(0);
      }//if
      else {
        // erreur
        Console.Out.WriteLine("Commande incorrecte. Utilisez (start,stop)");
      }//if
    }//while
  }//admin
```

Si la commande tapée au clavier est *start*, un thread d'écoute des demandes clients est lancé. Si la commande tapée est *stop*, tous les threads sont arrêtés. Le thread d'écoute exécute la méthode *ecoute* :

```
  public void ecoute(){
    // thread d'écoute des demandes des clients
    // on crée le service d'écoute
    TcpListener ecoute=null;
```

```
    try{
      // on crée le service
      ecoute=new TcpListener(portEcoute);
      // on le lance
      ecoute.Start();
      // suivi
      Console.Out.WriteLine("Serveur d'écho lancé sur le port " + portEcoute);

      // boucle de service
      TcpClient liaisonClient=null;
      while (true){ // boucle infinie
        // attente d'un client
        liaisonClient=ecoute.AcceptTcpClient();

        // le service est assuré par une autre tâche
        new Thread(new ThreadStart(new traiteClientImpots(liaisonClient,this).Run)).Start();

        // on retourne à l'écoute des demandes
      }// fin while
    }catch(Exception ex){
      // on signale l'erreur
      erreur("L'erreur suivante s'est produite : " + ex.Message,3);
    }//catch
  }//thread d'écoute

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe
```

On retrouve un serveur tcp classique écoutant sur le port *portEcoute*. Les demandes des clients sont traitées par la méthode *Run* d'un objet auquel on passe deux paramètres :

1. l'objet *TcpClient* qui va permettre d'atteindre le client
2. l'objet *impôt this* qui va donner accès à la méthode *this.calculer* de calcul de l'impôt.

```
// -------------------------------------------------------
// assure le service à un client du serveur d'impôts

public class traiteClientImpots{

  private TcpClient liaisonClient;    // liaison avec le client
  private StreamReader IN;          // flux d'entrée
  private StreamWriter OUT;           // flux de sortie
  private impôt objImpôt;             // objet Impôt

    // constructeur
  public traiteClientImpots(TcpClient liaisonClient,impôt objImpôt){
    this.liaisonClient=liaisonClient;
    this.objImpôt=objImpôt;
  }//constructeur
```

La méthode *Run* traite les demandes des clients. Celles-ci peuvent avoir deux formes :

1. calcul marié(o/n) nbEnfants salaireAnnuel
2. fincalculs

La forme 1 permet le calcul d'un impôt, la forme 2 clôt la liaison client-serveur.

```
// méthode Run
  public void Run(){
    // rend le service au client
    try{
      // flux d'entrée
      IN=new StreamReader(liaisonClient.GetStream());
      // flux de sortie
      OUT=new StreamWriter(liaisonClient.GetStream());
      OUT.AutoFlush=true;
      // envoi d'un msg de bienvenue au client
      OUT.WriteLine("Bienvenue sur le serveur d'impôts");

      // boucle lecture demande/écriture réponse
      string demande=null;
      string[] champs=null;// les éléments de la demande
      string commande=null;// la commande du client : calcul ou fincalculs
      while ((demande=IN.ReadLine())!=null){
        // on décompose la demande en champs
        champs=Regex.Split(demande.Trim().ToLower(),@"\s+");
        // deux demandes acceptées : calcul et fincalculs
        commande=champs[0];
        if(commande!="calcul" && commande!="fincalculs"){
          // erreur client
          OUT.WriteLine("Commande incorrecte. Utilisez (calcul,fincalculs).");
          // commande suivante
          continue;
```

```
      }//if
      if(commande=="calcul") calculerImpôt(champs);
      if(commande=="fincalculs"){
        // msg d'au-revoir au client
        OUT.WriteLine("Au revoir...");
        // libération des ressources
        try{ OUT.Close();IN.Close();liaisonClient.Close();}
        catch{}
        // fin
        return;
      }//if
      //demande suivante
    }//while
  }catch (Exception e){
    erreur("L'erreur suivante s'est produite ("+e+")",2);
  }// fin try
}// fin Run
```

Le calcul de l'impôt est effectué par la méthode *calculerImpôt* qui reçoit en paramètre le tableau des champs de la demande faite par le client. La validité de la demande est vérifiée et éventuellement l'impôt calculé et renvoyé au client.

```
// calcul d'impôts
  public void calculerImpôt(string[] champs){
    // traite la demande : calcul marié nbEnfants salaireAnnuel
    // décomposée en champs dans le tableau champs

    string marié=null;
    int nbEnfants=0;
    int salaireAnnuel=0;

    // validité des arguments
    try{
      // il faut au moins 4 champs
      if(champs.Length!=4) throw new Exception();
      // marié
      marié=champs[1];
      if (marié!="o" && marié !="n") throw new Exception();
      // enfants
      nbEnfants=int.Parse(champs[2]);
      // salaire
      salaireAnnuel=int.Parse(champs[3]);
    }catch{
      // erreur de format
      OUT.WriteLine(" syntaxe : calcul marié(O/N) nbEnfants salaireAnnuel");
      // fini
      return;
    }//if
    // on peut calculer l'impôt
    long impot=objImpôt.calculer(marié=="o",nbEnfants,salaireAnnuel);
    // on envoie la réponse au client
    OUT.WriteLine(""+impot);
  }//calculer
```

Cette classe est compilée par

```
csc /t:library /r:impots.dll serveurimpots.cs
```

où *impots.dll* contient le code de la classe *impôt*. Un programme de test pourrait être le suivant :

```
// appel : serveurImpots port dsnImpots Timpots colLimites colCoeffR colCoeffN

using System;
using System.IO;

public class testServeurImpots{
  public const string syntaxe="Syntaxe : pg port dsnImpots Timpots colLimites colCoeffR colCoeffN";

  // programme principal
  public static void Main (string[] args){

    // il faut6 arguments
    if(args.Length != 6)
      erreur(syntaxe,1);

    // le port doit être entier >0
    int port=0;
    bool erreurPort=false;
    Exception E=null;
    try{
      port=int.Parse(args[0]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);
```

```
    // on crée le serveur d'impôts
    try{
      new ServeurImpots(port,args[1],args[2],args[3],args[4],args[5]);
    }catch(Exception ex){
      //erreur
      Console.Error.WriteLine("L'erreur suivante s'est produite : "+ex.Message);
    }//catch
  }//Main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}// fin class
```

On passe au programme de test les données nécessaires à la construction d'un objet *ServeurImpots* et à partir de là il crée cet objet. Ce programme de test est compilé par :

```
csc /r:serveurimpots.dll /r:impots.dll testServeurImpots.cs
```

Voici un premier test :

```
dos>testserveurimpots 124 mysql-impots timpots limites coeffr coeffn
Serveur d'impôts>start
Serveur d'impôts>Serveur d'écho lancé sur le port 124
stop

E:\data\serge\MSNET\c#\impots\serveur>
```

La ligne

```
dos>testserveurimpots 124 mysql-impots timpots limites coeffr coeffn
```

crée un objet *ServeurImpots* qui n'écoute pas encore les demandes des clients. C'est la commande **start** tapée au clavier qui lance cette écoute. La commande **stop** arrête le serveur. Utilisons maintenant un client. Nous utiliserons le client générique créé précédemment. Le serveur est lancé :

```
E:\data\serge\MSNET\c#\impots\serveur>testserveurimpots 124 mysql-impots timpots limites coeffr coeffn
Serveur d'impôts>start
Serveur d'impôts>Serveur d'écho lancé sur le port 124
```

Le client générique est lancé dans une autre fenêtre Dos :

```
E:\data\serge\MSNET\c#\réseau\client tcp générique>cltgen localhost 124
Commandes :
<-- Bienvenue sur le serveur d'impôts
```

On voit que le client a bien récupéré le message de bienvenue du serveur. On envoie d'autres commandes :

```
x
<-- Commande incorrecte. Utilisez (calcul,fincalculs).
calcul
<--  syntaxe : calcul marié(O/N) nbEnfants salaireAnnuel
calcul o 2 200000
<-- 22506
calcul n 2 200000
<-- 33388
fincalculs
<-- Au revoir...
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

On retourne dans la fenêtre du serveur pour l'arrêter :

```
E:\data\serge\MSNET\c#\impots\serveur>testserveurimpots 124 mysql-impots timpots limites coeffr coeffn
Serveur d'impôts>start
Serveur d'impôts>Serveur d'écho lancé sur le port 124
stop
```

9. Services Web

9.1 Introduction

Nous avons présenté dans le chapitre précédent plusieurs applications client-serveur tcp-ip. Dans la mesure où les clients et le serveur échangent des lignes de texte, ils peuvent être écrits en n'importe quel langage. Le client doit simplement connaître le protocole de dialogue attendu par le serveur. Les services Web sont des applications serveur tcp-ip présentant les caractéristiques suivantes :

- Elles sont hébergées par des serveurs web et le protocole d'échanges client-serveur est donc HTTP (HyperText Transport Protocol), un protocole au-dessus de TCP-IP.
- Le service Web a un protocole de dialogue standard quelque soit le service assuré. Un service Web offre divers services S1, S2, .., Sn. Chacun d'eux attend des paramètres fournis par le client et rend à celui-ci un résultat. Pour chaque service, le client a besoin de savoir :
  - le nom exact du service Si
  - la liste des paramètres qu'il faut lui fournir et leur type
  - le type de résultat retourné par le service

Une fois, ces éléments connus, le dialogue client-serveur suit le même format quelque soit le service web interrogé. L'écriture des clients est ainsi normalisée.

- Pour des raisons de sécurité vis à vis des attaques venant de l'internet, beaucoup d'organisations ont des réseaux privés et n'ouvrent sur Internet que certains ports de leurs serveurs : essentiellement le port 80 du service web. Tous les autres ports sont verrouillés. Aussi les applications client-serveur telles que présentées dans le chapitre précédent sont-elles construites au sein du réseau privé (intranet) et ne sont en général pas accessibles de l'extérieur. Loger un service au sein d'un serveur web le rend accessible à toute la communauté internet.
- Le service Web peut être modélisé comme un objet distant. Les services offerts deviennent alors des méthodes de cet objet. Un client peut avoir accès à cet objet distant comme s'il était local. Cela cache toute la partie communication réseau et permet de construire un client indépendant de cette couche. Si celle-ci vient à changer, le client n'a pas à être modifié. C'est là un énorme avantage et probablement le principal atout des services Web.
- Comme pour les applications client-serveur tcp-ip présentées dans le chapitre précédent, le client et le serveur peuvent être écrits dans un langage quelconque. Ils échangent des lignes de texte. Celles-ci comportent deux parties :
  - les entêtes nécessaires au protocole HTTP
  - le corps du message. Pour une réponse du serveur au client, celui-ci est au format XML (eXtensible Markup Language). Pour une demande du client au serveur, le corps du message peut avoir plusieurs formes dont XML. La demande XML du client peut avoir un format particulier appelé SOAP (Simple Object Access Protocol). Dans ce cas, la réponse du serveur suit aussi le format SOAP.

Explicitons ces généralités sur un premier exemple.

9.2 Un premier service Web

Nous considérons un service Web qui offre cinq fonctions :

1. ajouter(a,b) qui rendra a+b
2. soustraire(a,b) qui rendra a-b
3. multiplier(a,b) qui rendra a*b
4. diviser(a,b) qui rendra a/b
5. toutfaire(a,b) qui rendra le tableau [a+b,a-b,a*b,a/b]

Le code C# de ce service est le suivant :

```
<%@ WebService Language=C# class=operations %>

using System.Web.Services;

[WebService(Namespace="st.istia.univ-angers.fr")]
public class operations : WebService{

  [WebMethod]
  public double ajouter(double a, double b){
    return a+b;
  }//ajouter
```

```
  [WebMethod]
  public double soustraire(double a, double b){
    return a-b;
  }//soustraire

  [WebMethod]
  public double multiplier(double a, double b){
    return a*b;
  }//multiplier

  [WebMethod]
  public double diviser(double a, double b){
    return a/b;
  }//diviser

  [WebMethod]
  public double[] toutfaire(double a, double b){
    return new double[] {a+b,a-b,a*b,a/b};
  }//toutfaire

}//classe
```

Ce code n'est pas destiné à être compilé directement par le compilateur C#. Ce qui explique la présence d'une directive que ne peut comprendre C# :

```
<%@ WebService Language=C# class=operations %>
```

Le code-source ci-dessus est en fait destiné au serveur Web IIS et non au compilateur C#. La directive indique que le code qui suit :

- est un service Web
- écrit en C#
- que la classe l'implémentant s'appelle *operations*

La classe *operations* ressemble à une classe C# avec cependant quelques points à noter :

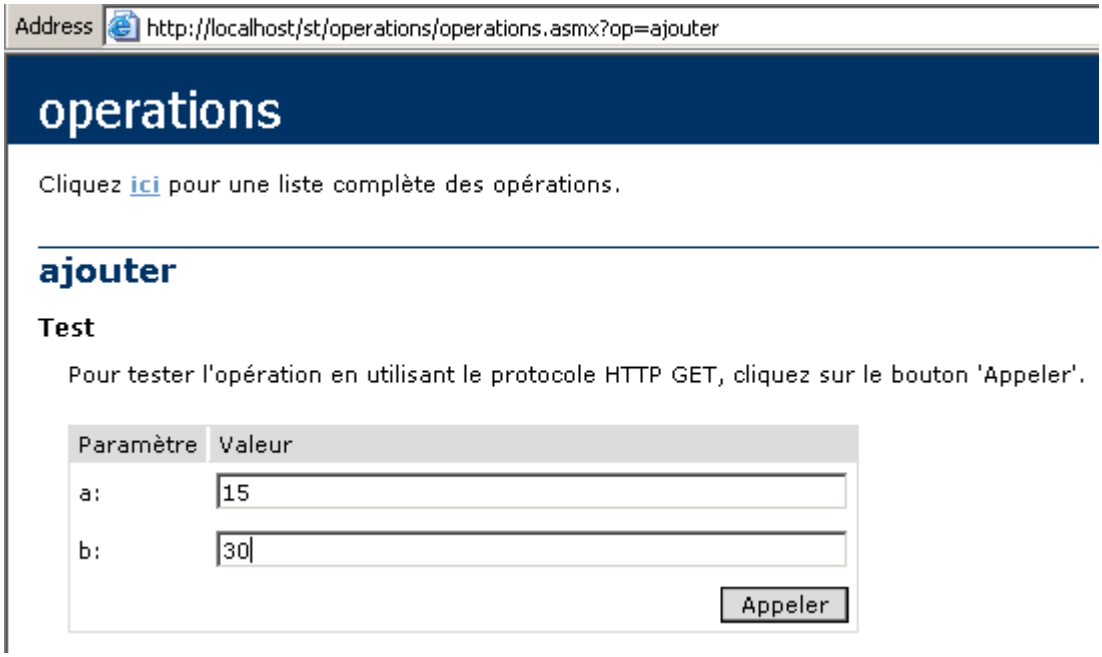
- les méthodes sont précédées d'un attribut **[WebMethod]** qui indique au compilateur les méthodes qui doivent être "publiées" c.a.d. rendues disponibles au client. Une méthode non précédée de cet attribut serait invisible aux clients distants. Ce pourrait être une méthode interne utilisée par d'autres méthodes mais pas destinée à être publiée.
- la classe dérive de la classe *WebService* définie dans l'espace de noms *System.Web.Services*. Cet héritage n'est pas toujours obligatoire. Dans cet exemple notamment on pourrait s'en passer.
- la classe elle-même est précédée d'un attribut **[WebService(Namespace="st.istia.univ-angers.fr")]** destiné à donner un espace de noms au service web. Un vendeur de classes donne un espace de noms à ses classes afin de leur donner un nom unique et éviter ainsi des conflits avec des classes d'autres vendeurs qui pourraient porter le même nom. Pour les services Web, c'est pareil. Chaque service web doit pouvoir être identifié par un nom unique, ici par **st.istia.univ-angers.fr**.
- nous n'avons pas défini de constructeur. Il est possible d'en définir mais la documentation consultée ne dit pas comment utiliser un tel constructeur.

Le code source précédent n'est pas destiné directement au compilateur C# mais au serveur Web IIS. Il doit porter le suffixe *.asmx* et sauvegardé dans l'arborescence du serveur Web. Ici nous le sauvegardons sous le nom **operations.asmx** dans le dossier **c:\inetpub\wwwroot\st\operations** :

```
E:\data\serge\MSNET\c#\webservices\clientSOAP>dir c:\inetpub\wwwroot\st\operations
14/05/2002  17:14                 549 operations.asmx
```

On rappelle que *c:\inetpub\wwwroot* est la racine des documents web délivrés par le serveur IIS. Demandons le document précédent avec un navigateur. L'URl à demander est *http://localhost/st/operations/operations.asmx*.

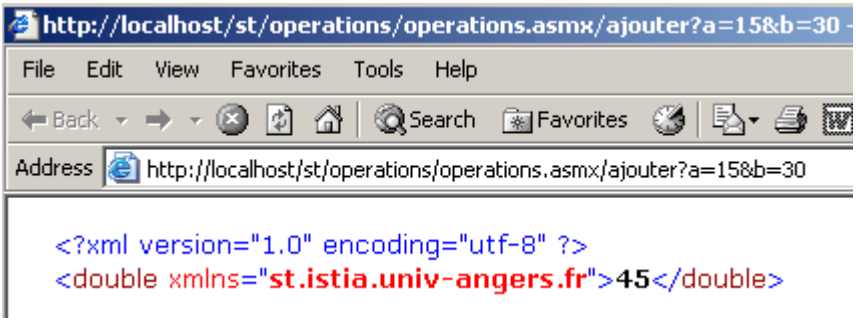
Nous obtenons un document Web avec un lien pour chacune des méthodes définies dans le service web *operations*. Suivons le lien *ajouter* :



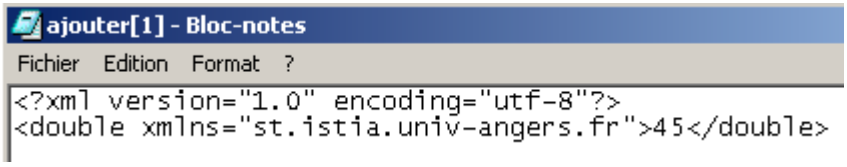
La page obtenue nous propose de tester la méthode *ajouter* en lui fournissant les deux arguments *a* et *b* dont elle a besoin. Rappelons la définition de la méthode *ajouter* :

```
[WebMethod]
public double ajouter(double a, double b){
  return a+b;
}//ajouter
```

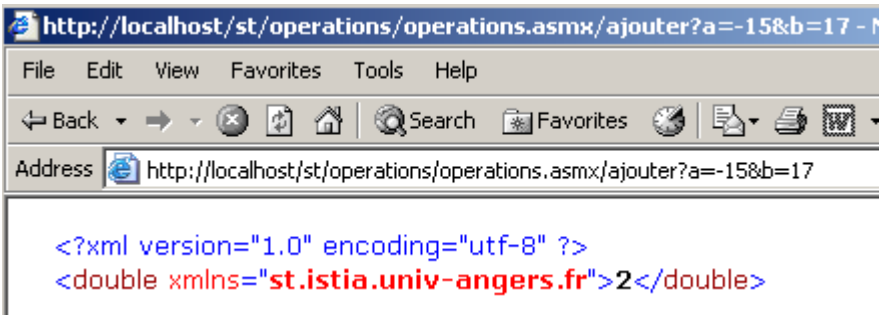
On notera que la page a repris les noms des arguments *a* et *b* utilisés dans la définition de la méthode. On utilise le bouton *Appeler* et on obtient la réponse suivante dans une fenêtre séparée du navigateur :



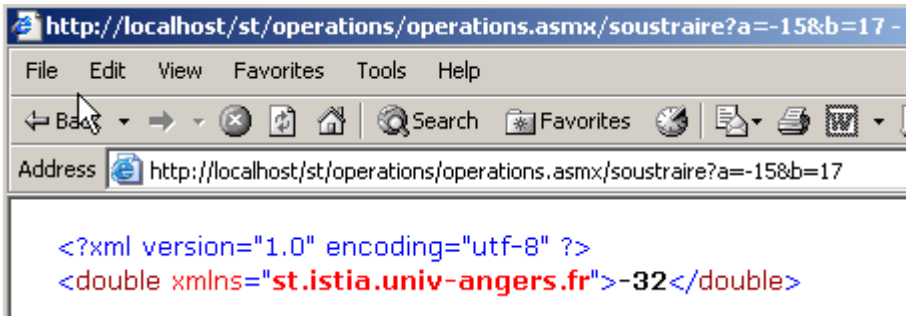
Si ci-dessus, on fait *View/Source* on obtient le code suivant :



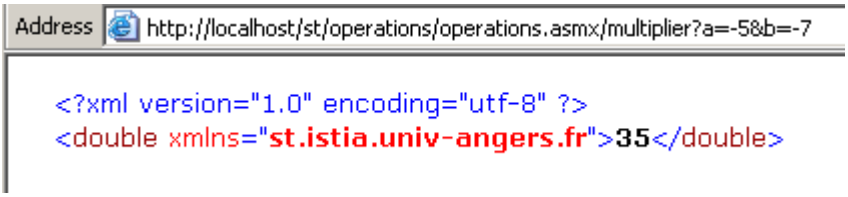
On peut remarquer que le navigateur (ici IE) n'a pas reçu du code HTML mais du code XML. Par ailleurs on voit que la réponse a été demandée à l'URL : *http://localhost/st/operations/operations.asmx/ajouter?a=15&b=30*. Si nous modifions directement les valeurs de *a* et *b* dans l'URL pour qu'elle devienne *http://localhost/st/operations/operations.asmx/ajouter?a=-15&b=17*, nous obtenons le résultat suivant :



Nous commençons à discerner une méthode pour avoir accès à une fonction F d'un service web S : on demande l'URL *http://urlServiceWeb/fonction?param1=val1&param2=val2....* Essayons dans l'URL ci-dessus de remplacer *ajouter* par *soustraire* qui est l'une des fonctions définies dans le service Web *operations*. Nous obtenons le résultat suivant :



De même pour les fonctions *multiplier*, *diviser*, *toutfaire* :



Address http://localhost/st/operations/operations.asmx/diviser?a=-5&b=-7 Go Lien

```
<?xml version="1.0" encoding="utf-8" ?>
<double xmlns="st.istia.univ-angers.fr">0.7142857142857143</double>
```

Address http://localhost/st/operations/operations.asmx/toutfaire?a=1&b=2

```
<?xml version="1.0" encoding="utf-8" ?>
- <ArrayOfDouble xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="st.istia.univ-angers.fr">
    <double>3</double>
    <double>-1</double>
    <double>2</double>
    <double>0.5</double>
  </ArrayOfDouble>
```

Dans tous les cas, la réponse du serveur a la forme :

<?xml version="1.0" encoding="utf-8"?>
[réponse au format XML]

- la réponse est au format XML
- la ligne 1 est standard et est toujours présente dans la réponse
- les lignes suivantes dépendent du type de résultat (**double,ArrayOfDouble**), du nombre de résultats, et de l'espace de noms du service web (**st.istia.univ-angers.fr** ici).

Nous savons maintenant comment interroger un service web et obtenir sa réponse. En fait il existe plusieurs méthodes pour faire cela. Revenons à l'URL du service :

operations Service Web - Microsoft Internet Explor

File Edit View Favorites Tools Help

Back Search Favorite

Address http://localhost/st/operations/operations.asmx

operations

Les opérations suivantes sont prises en charge. Pc

- diviser
- multiplier
- ajouter
- soustraire
- toutfaire

et suivons le lien *ajouter* :

Address http://localhost/st/operations/operations.asmx?op=ajouter

operations

Cliquez ici pour une liste complète des opérations.

ajouter

Test

Pour tester l'opération en utilisant le protocole HTTP GET, cliquez sur le bouton 'Appeler'.

Paramètre	Valeur
a:	
b:	
	Appeler

Dans page ci-dessus, sont exposées (mais non représentées dans la copie d'écran ci-dessus) trois méthodes pour interroger la fonction *ajouter* du service web :

HTTP GET

Le texte suivant est un exemple de demande et de réponse HTTP GET. Les **espaces réservés** affichés doivent être remplacés par des valeurs réelles.

```
GET /st/operations/operations.asmx/ajouter?a=string&b=string HTTP/1.1
Host: localhost
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<double xmlns="st.istia.univ-angers.fr">double</double>
```

HTTP POST

Le texte suivant est un exemple de demande et de réponse HTTP POST. Les **espaces réservés** affichés doivent être remplacés par des valeurs réelles.

```
POST /st/operations/operations.asmx/ajouter HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

a=string&b=string
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<double xmlns="st.istia.univ-angers.fr">double</double>
```

SOAP

Le texte suivant est un exemple de demande et de réponse SOAP. Les **espaces réservés** affichés doivent être remplacés par des valeurs réelles.

```
POST /st/operations/operations.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "st.istia.univ-angers.fr/ajouter"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:
  <soap:Body>
    <ajouter xmlns="st.istia.univ-angers.fr">
      <a>double</a>
      <b>double</b>
    </ajouter>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:
  <soap:Body>
    <ajouterResponse xmlns="st.istia.univ-angers.fr">
      <ajouterResult>double</ajouterResult>
    </ajouterResponse>
  </soap:Body>
</soap:Envelope>
```

Ces trois méthodes d'accès aux fonctions d'un service web sont appelées respectivement : **HTTP-GET**, **HTTP-POST** et **SOAP**. Nous les examinons maintenant l'une après l'autre.

9.3 Un client HTTP-GET

Nous nous proposons de construire un client qui interrogerait les fonctions : *ajouter*, *soustraire*, *multiplier*, *diviser* du service web *operations*. La première méthode utilisée est HTTP-GET exposée ci-dessous pour la fonction *ajouter* :

HTTP GET

Le texte suivant est un exemple de demande et de réponse HTTP GET. Les **espaces réservés** affichés doivent être remplacés par des valeurs réelles.

```
GET /st/operations/operations.asmx/ajouter?a=stringb=string HTTP/1.1
Host: localhost
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<double xmlns="st.istia.univ-angers.fr">double</double>
```

Le client HTTP doit émettre au minimum les deux commandes HTTP suivantes :

```
GET /st/operations/operations.asmx/ajouter?a=[a]&b=[b] HTTP/1.1
Host: localhost
```

où **[a]** et **[b]** doivent être remplacées par les valeurs de a et b. Le serveur web enverra la réponse suivante :

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: [longueur]

<?xml version="1.0" encoding="utf-8"?>
<double xmlns="st.istia.univ-angers.fr">[résultat]</double>
```

où **[longueur]** est le nombre de caractères envoyés par le serveur après la ligne vide qui suit les entêtes HTTP et **[résultat]** est le résultat de la fonction *ajouter*. Vérifions cela avec notre client générique défini dans le chapitre précédent :

```
E:\data\serge\MSNET\c#\webservices\clientGET>cltgen localhost 80
```

```
Commandes :
GET http://localhost/st/operations/operations.asmx/ajouter?a=10&b=20 HTTP/1.1
Connection: close
Host: localhost:80

<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:04:35 GMT
<-- Connection: close
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 91
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">30</double>
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Notons les points suivants :

- le client s'est connecté au port 80 de la machine *localhost* c'est à dire au serveur Web IIS.
- il a envoyé les entêtes HTTP nécessaires pour demander d'ajouter 10 à 20. On a rajouté l'entête **Connection: close** pour demander au serveur de fermer la connexion après avoir envoyé la réponse. Cela est nécessaire ici. Si on ne le dit pas, par défaut le serveur va garder la connexion ouverte. Or sa réponse est une suite de lignes de texte dont la dernière n'est pas terminée par une marque de fin de ligne. Or notre client TCP générique lit des lignes de texte terminées par la marque de fin de ligne avec la méthode *ReadLine*. Si le serveur ne ferme pas la connexion après envoi de la dernière ligne, le client est bloqué parce qu'il attend une marque de fin de ligne qui n'existe pas. Si le serveur ferme la connexion, la méthode *ReadLine* du client se termine et le client ne reste pas bloqué.
- le serveur a envoyé davantage d'entêtes qu'attendus mais le format de la réponse XML est bien celui qui était prévu.

Nous avons maintenant les éléments pour écrire le client suivant :

```
E:\data\serge\MSNET\c#\webservices\clientGET>clientget.exe http://localhost/st/operations/operations.asmx
Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a b

ajouter 3 4
--> GET /st/operations/operations.asmx/ajouter?a=3&b=4 HTTP/1.1
--> Host: localhost:80
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:18:24 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">7</double>
[résultat=7]

soustraire 10 15
--> GET /st/operations/operations.asmx/soustraire?a=10&b=15 HTTP/1.1
--> Host: localhost:80
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:18:30 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 91
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">-5</double>
[résultat=-5]

diviser 8 4
--> GET /st/operations/operations.asmx/diviser?a=8&b=4 HTTP/1.1
--> Host: localhost:80
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:18:38 GMT
<-- Cache-Control: private, max-age=0
```

```
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">2</double>
[résultat=2]

multiplier 10 0,2
--> GET /st/operations/operations.asmx/multiplier?a=10&b=0,2 HTTP/1.1
--> Host: localhost:80
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:18:46 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">2</double>
[résultat=2]

fin
```

Le client est appelé en lui passant l'URL du service web :

```
E:\data\serge\MSNET\c#\webservices\clientGET>clientget.exe http://localhost/st/operations/operations.asmx
```

Ensuite, le client lit les commandes tapées au clavier et les exécute. Celles-ci sont au format :

fonction a b

où **fonction** est la fonction du service web appelée (ajouter, soustraire, multiplier, diviser) et **a** et **b** les valeurs sur lesquelles va opérer cette fonction. Par exemple :

```
ajouter 3 4
```

A partir de là, le client va faire la requête HTTP nécessaire au serveur Web et obtenir une réponse. Les échanges client-serveur sont dupliqués à l'écran pour une meilleure compréhension du processus :

```
--> GET /st/operations/operations.asmx/ajouter?a=3&b=4 HTTP/1.1
--> Host: localhost:80
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:18:24 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">7</double>
```

On retrouve ci-dessus l'échange déjà rencontré avec le client tcp générique à une différence près : l'entête HTTP **Connection: Keep-Alive** demande au serveur de ne pas fermer la connexion. Celle-ci reste donc ouverte pour l'opération suivante du client qui n'a donc pas besoin de se reconnecter de nouveau au serveur. Cela l'oblige cependant à utiliser une autre méthode que *ReadLine()* pour lire la réponse du serveur puisqu'on sait que celle-ci est une suite de lignes dont la dernière n'est pas terminée par une marque de fin de ligne. Une fois toute la réponse du serveur obtenue, le client l'analyse pour y trouver le résultat de l'opération demandée et l'afficher :

```
[résultat=7]
```

Examinons le code de notre client :

```
// espaces de noms
using System;
using System.Net.Sockets;
using System.IO;
using System.Text.RegularExpressions;
using System.Collections;

public class clientGet{


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

```
    // syntaxe
    const string syntaxe="pg URI";
    string[] fonctions={"ajouter","soustraire","multiplier","diviser"};

    // nombre d'arguments
    if(args.Length != 1)
      erreur(syntaxe,1);

    // on note l'URI demandée
    string URIstring=args[0];

    // on se connecte au serveur
    Uri uri=null;           // l'URI du service web
    TcpClient client=null; // la liaison tcp du client avec le serveur
    StreamReader IN=null;  // le flux de lecture du client
    StreamWriter OUT=null; // le flux d'écriture du client
    try{
      // connexion au serveur
      uri=new Uri(URIstring);
      client=new TcpClient(uri.Host,uri.Port);
      // on crée les flux d'entrée-sortie du client TCP
      IN=new StreamReader(client.GetStream());
      OUT=new StreamWriter(client.GetStream());
      OUT.AutoFlush=true;
    }catch (Exception ex){
      // URI incorrecte ou autre problème
      erreur("L'erreur suivante s'est produite : " + ex.Message,2);
    }//catch

    // création d'un dictionnaire des fonctions du service web
    Hashtable dicoFonctions=new Hashtable();
    for (int i=0;i<fonctions.Length;i++){
      dicoFonctions.Add(fonctions[i],true);
    }//for

    // les demandes de l'utilisateur sont tapées au clavier
    // sous la forme fonction a b
    // elles se terminent avec la commande fin

    string commande=null;  // commande tapée au clavier
    string[] champs=null;  // champs d'une ligne de commande
    string fonction=null;  // nom d'une fonction du service web
    string a,b;            // les arguments des fonctions du service web

    // invite à l'utilisateur
    Console.Out.WriteLine("Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a b");

    // gestion des erreurs
    try{
      // boucle de saisie des commandes tapées au clavier
      while(true){
        // lecture commande
        commande=Console.In.ReadLine().Trim().ToLower();
        // fini ?
        if(commande==null || commande=="fin") break;
        // décomposition de la commande en champs
        champs=Regex.Split(commande,@"\s+");
        try{
          // il faut trois champs
          if(champs.Length!=3)throw new Exception();
          // le champ 0 doit être une fonction reconnue
          fonction=champs[0];
          if(! dicoFonctions.ContainsKey(fonction)) throw new Exception();
          // le champ 1 doit être un nombre valide
          a=champs[1];
          double.Parse(a);
          // le champ 2 doit être un nombre valide
          b=champs[2];
          double.Parse(b);
        }catch {
          // commande invalide
          Console.Out.WriteLine("syntaxe : [ajouter|soustraire|multiplier|diviser] a b");
          // on recommence
          continue;
        }//catch

        // on fait la demande au service web
        executeFonction(IN,OUT,uri,fonction,a,b);

        // demande suivante
      }//while
      // fin des demandes
    }catch(Exception e){
      Console.Out.WriteLine("L'erreur suivante s'est produite : " + e.Message);
    }//catch

    // fin liaison client-serveur
    try{
      IN.Close();OUT.Close();client.Close();
    }catch{}
  }//Main
```

```
....
  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe
```

On a là des choses déjà rencontrées plusieurs fois et qui ne nécessitent pas de commentaires particuliers. Examinons maintenant le code de la méthode *executeFonction* où résident les nouveautés :

```
  // executeFonction
  public static void executeFonction(StreamReader IN, StreamWriter OUT, Uri uri, string fonction, string
a, string b){
    // exécute fonction(a,b) sur le service web d'URI uri
    // les échanges client-serveur se font via les flux IN et OUT
    // le résultat de la fonction est dans la ligne
    // <double xmlns="st.istia.univ-angers.fr">double</double>
    // envoyée par le serveur

    // construction du tableau des entêtes HTTP à envoyer
    string[] entetes=new string[4];
    entetes[0]="GET " + uri.AbsolutePath+ "/"+fonction+"?a="+a+"&b="+b+" HTTP/1.1";
    entetes[1]="Host: " + uri.Host+":"+uri.Port;
    entetes[2]="Connection: Keep-Alive";
    entetes[3]="";

    // on envoie les entêtes HTTP au serveur
    for(int i=0;i<entetes.Length;i++){
      // envoi au serveur
      OUT.WriteLine(entetes[i]);
      // écho écran
      Console.Out.WriteLine("--> "+entetes[i]);
    }//for

    // construction de l'expression régulière permettant de retrouver la taille de la réponse
    // dans le flux de la réponse du serveur web
    string modèleLength=@"^Content-Length: (.+?)\s*$";
    Regex RegexLength=new Regex(modèleLength);
    Match MatchLength=null;
    int longueur=0;

    // on lit tous les entêtes de la réponse du serveur Web
    // on mémorise la valeur de la ligne Content-Length
    string ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      // écho écran
      Console.Out.WriteLine("<-- " + ligne);
      // Content-Length ?
      MatchLength=RegexLength.Match(ligne);
      if(MatchLength.Success){
        longueur=int.Parse(MatchLength.Groups[1].Value);
      }//if
    }//while
    // écho dernière ligne
    Console.Out.WriteLine("<--");

    // construction de l'expression régulière permettant de retrouver le résultat
    // dans le flux de la réponse du serveur web
    string modèle=@"<double xmlns="+"\""+@"st\.istia\.univ-angers\.fr"+"\""+@">(.+?)</double>";
    Regex ModèleRésultat=new Regex(modèle);
    Match MatchRésultat=null;

    // on lit le reste de la réponse du serveur web
    char[] chrRéponse=new char[longueur];
    IN.Read(chrRéponse,0,longueur);
    string strRéponse=new String(chrRéponse);

    // on décompose la réponse en lignes de texte
    string[] lignes=Regex.Split(strRéponse,"\n");

    // on parcourt les lignes de texte à la recherche du résultat
    string strRésultat="?";  // résultat de la fonction
    for(int i=0;i<lignes.Length;i++){
      // suivi
      Console.Out.WriteLine("<-- "+lignes[i]);
      // comparaison ligne courante au modèle
      MatchRésultat=ModèleRésultat.Match(lignes[i]);
      // a-t-on trouvé ?
      if(MatchRésultat.Success){
        // on note le résultat
        strRésultat=MatchRésultat.Groups[1].Value;
      }
    }//ligne suivante
    // on affiche le résultat
    Console.Out.WriteLine("[résultat="+strRésultat+"]\n");
  }//executeFonction
```

Le client commence par envoyer les entêtes HTTP de sa demande au serveur Web :

```
    // construction du tableau des entêtes HTTP à envoyer
    string[] entetes=new string[4];
    entetes[0]="GET " + uri.AbsolutePath+ "/"+fonction+"?a="+a+"&b="+b+" HTTP/1.1";
    entetes[1]="Host: " + uri.Host+":"+uri.Port;
    entetes[2]="Connection: Keep-Alive";
    entetes[3]="";

    // on envoie les entêtes HTTP au serveur
    for(int i=0;i<entetes.Length;i++){
      // envoi au serveur
      OUT.WriteLine(entetes[i]);
      // écho écran
      Console.Out.WriteLine("--> "+entetes[i]);
    }//for
```

Cela correspond à l'affichage écran suivant :

```
--> GET /st/operations/operations.asmx/ajouter?a=3&b=4 HTTP/1.1
--> Host: localhost:80
--> Connection: Keep-Alive
-->
```

Le serveur ayant reçu cette demande calcule sa réponse et l'envoie. Celle-ci est composée de deux parties :

1. des entêtes HTTP terminés par une ligne vide
2. la réponse au format XML

```
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:18:24 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">7</double>
```

Dans un premier temps, le client lit les entêtes HTTP pour y trouver la ligne *Content-Length* et récupérer la taille de la réponse XML (ici 90). Celle-ci est récupérée au moyen d'une expression régulière. On aurait pu faire autrement et sans doute de façon plus efficace.

```
    // construction de l'expression régulière permettant de retrouver la taille de la réponse
    // dans le flux de la réponse du serveur web
    string modèleLength=@"^Content-Length: (.+?)\s*$";
    Regex RegexLength=new Regex(modèleLength);
    Match MatchLength=null;
    int longueur=0;

    // on lit tous les entêtes de la réponse du serveur Web
    // on mémorise la valeur de la ligne Content-Length
    string ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      // écho écran
      Console.Out.WriteLine("<-- " + ligne);
      // Content-Length ?
      MatchLength=RegexLength.Match(ligne);
      if(MatchLength.Success){
        longueur=int.Parse(MatchLength.Groups[1].Value);
      }//if
    }//while
    // écho dernière ligne
    Console.Out.WriteLine("<--");
```

Une fois qu'on a la longueur N de la réponse XML, on n'a plus qu'à lire N caractères dans le flux IN de la réponse du serveur. Cette chaîne de N caractères est redécomposée en lignes de texte pour les besoins du suivi écran. Parmi ces lignes on cherche la ligne du résultat :

```
<-- <double xmlns="st.istia.univ-angers.fr">7</double>
```

au moyen là encore d'une expression régulière. Une fois le résultat trouvé, il est affiché.

```
[résultat=7]
```

La fin du code du client est la suivante :

```
    // construction de l'expression régulière permettant de retrouver le résultat
    // dans le flux de la réponse du serveur web
```

```
    string modèle=@"<double xmlns="+"\""+@"st\.istia\.univ-angers\.fr"+"\""+@">(.+?)</double>";
    Regex ModèleRésultat=new Regex(modèle);
    Match MatchRésultat=null;

    // on lit le reste de la réponse du serveur web
    char[] chrRéponse=new char[longueur];
    IN.Read(chrRéponse,0,longueur);
    string strRéponse=new String(chrRéponse);

    // on décompose la réponse en lignes de texte
    string[] lignes=Regex.Split(strRéponse,"\n");

    // on parcourt les lignes de texte à la recherche du résultat
    string strRésultat="?";  // résultat de la fonction
    for(int i=0;i<lignes.Length;i++){
      // suivi
      Console.Out.WriteLine("<-- "+lignes[i]);
      // comparaison ligne courante au modèle
      MatchRésultat=ModèleRésultat.Match(lignes[i]);
      // a-t-on trouvé ?
      if(MatchRésultat.Success){
        // on note le résultat
        strRésultat=MatchRésultat.Groups[1].Value;
      }
    }//ligne suivante
    // on affiche le résultat
    Console.Out.WriteLine("[résultat="+strRésultat+"]\n");
  }//executeFonction
```

9.4 Un client HTTP-POST

Nous reprenons le même problème avec cette fois-ci un client HTTP-POST. Nous suivons là encore la méthode proposée par le service web lui-même :

HTTP POST

Le texte suivant est un exemple de demande et de réponse HTTP POST. Les **espaces réservés** affichés doivent être remplacés par des valeurs réelles.

```
POST /st/operations/operations.asmx/ajouter HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

a=string&b=string
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<double xmlns="st.istia.univ-angers.fr">double</double>
```

On voit que la demande du client change de forme mais que la réponse du client reste la même. Utilisons de nouveau notre client tcp générique pour vérifier tout cela :

```
E:\data\serge\MSNET\c#\webservices\clientPOST>cltgen localhost 80
Commandes :
POST /st/operations/operations.asmx/ajouter HTTP/1.1
Host: localhost
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:59:17 GMT
<--
a=2&b=3
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:59:22 GMT
<-- Connection: close
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">5</double>
[fin du thread de lecture des réponses du serveur]
fin
```

```
[fin du thread d'envoi des commandes au serveur]
```

Nous avons ajouté l'entête *Connection: close* pour des raisons déjà expliquées précédemment. On peut constater que la réponse du serveur IIS n'a pas le format attendu. Aussitôt après avoir reçu la ligne vide signalant la fin des entêtes HTTP, il envoie une première réponse :

```
<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:59:17 GMT
<--
```

Cette réponse formée uniquement d'entêtes HTTP indique au client qu'il peut envoyer les 7 caractères qu'il a dit vouloir envoyer. Ce que nous faisons :

```
a=2&b=3
```

Il faut voir ici que notre client tcp envoie plus de 7 caractères puisqu'il les envoie avec une marque de fin de ligne (*WriteLine*). Ca ne gêne pas le serveur qui des caractères reçus ne prendra que les 7 premiers et parce qu'ensuite la connexion est fermée (*Connection: close*). Ces caractères en trop auraient été gênants si la connexion était restée ouverte car alors ils auraient été pris comme venant de la commande suivante du client. Une fois les paramètres reçus, le serveur envoie sa réponse, identique cette fois à celle qu'il faisait au client HTTP-GET :

```
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 13:59:22 GMT
<-- Connection: close
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 90
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">5</double>
```

Voici un exemple d'exécution :

```
E:\data\serge\MSNET\c#\webservices\clientPOST>clientpost1 http://localhost/st/operations/operations.asmx
Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a b

ajouter 6 7
--> POST /st/operations/operations.asmx/ajouter HTTP/1.1
--> Host: localhost:80
--> Content-Type: application/x-www-form-urlencoded
--> Content-Length: 7
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:11:47 GMT
<--
--> a=6&b=7
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:11:47 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 91
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">13</double>
[résultat=13]

soustraire 8 9
--> POST /st/operations/operations.asmx/soustraire HTTP/1.1
--> Host: localhost:80
--> Content-Type: application/x-www-form-urlencoded
--> Content-Length: 7
--> Connection: Keep-Alive
-->
<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:11:58 GMT
<--
--> a=8&b=9
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
```

```
<-- Date: Wed, 15 May 2002 14:11:58 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 91
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">-1</double>
[résultat=-1]

fin

E:\data\serge\MSNET\c#\webservices\clientPOST>
```

La première partie du client HTTP-POST est identique à celle correspondante du cient HTTP-GET. Seul change le code de la fonction *executeFonction* :

```
  // executeFonction
  public static void executeFonction(StreamReader IN, StreamWriter OUT, Uri uri, string fonction, string
a, string b){
    // exécute fonction(a,b) sur le service web d'URI uri
    // les échanges client-serveur se font via les flux IN et OUT
    // le résultat de la fonction est dans la ligne
    // <double xmlns="st.istia.univ-angers.fr">double</double>
    // envoyée par le serveur

    // construction de la chaîne de requête
    string requête="a="+HttpUtility.UrlEncode(a)+"&b="+HttpUtility.UrlEncode(b);
    int nbChars=requête.Length;

    // construction du tableau des entêtes HTTP à envoyer
    string[] entetes=new string[6];
    entetes[0]="POST " + uri.AbsolutePath+ "/"+fonction+" HTTP/1.1";
    entetes[1]="Host: " + uri.Host+":"+uri.Port;
    entetes[2]="Content-Type: application/x-www-form-urlencoded";
    entetes[3]="Content-Length: "+nbChars;
    entetes[4]="Connection: Keep-Alive";
    entetes[5]="";

    // on envoie les entêtes HTTP au serveur
    for(int i=0;i<entetes.Length;i++){
      // envoi au serveur
      OUT.WriteLine(entetes[i]);
      // écho écran
      Console.Out.WriteLine("--> "+entetes[i]);
    }//for

    // on lit la 1ere réponse du serveur Web HTTP/1.1 100
    string ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      //écho
      Console.Out.WriteLine("<-- "+ligne);
    }//while
    //écho dernière ligne
    Console.Out.WriteLine("<-- "+ligne);

    // envoi paramètres de la requête
    OUT.Write(requête);
    // echo
    Console.Out.WriteLine("--> "+requête);

    // construction de l'expression régulière permettant de retrouver la taille de la réponse XML
    // dans le flux de la réponse du serveur web
    string modèleLength=@"^Content-Length: (.+?)\s*$";
    Regex RegexLength=new Regex(modèleLength);
    Match MatchLength=null;
    int longueur=0;

    // lecture seconde réponse du serveur web après envoi de la requête
    // on mémorise la valeur de la ligne Content-Length
    ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      // écho écran
      Console.Out.WriteLine("<-- " + ligne);
      // Content-Length ?
      MatchLength=RegexLength.Match(ligne);
      if(MatchLength.Success){
        longueur=int.Parse(MatchLength.Groups[1].Value);
      }//if
    }//while
    // écho dernière ligne
    Console.Out.WriteLine("<--");

    // construction de l'expression régulière permettant de retrouver le résultat
    // dans le flux de la réponse du serveur web
    string modèle=@"<double xmlns="+"\""+@"st\.istia\.univ-angers\.fr"+"\""+@">(.+?)</double>";
    Regex ModèleRésultat=new Regex(modèle);
    Match MatchRésultat=null;

    // on lit le reste de la réponse du serveur web
```

```
    char[] chrRéponse=new char[longueur];
    IN.Read(chrRéponse,0,longueur);
    string strRéponse=new String(chrRéponse);

    // on décompose la réponse en lignes de texte
    string[] lignes=Regex.Split(strRéponse,"\n");

    // on parcourt les lignes de texte à la recherche du résultat
    string strRésultat="?";  // résultat de la fonction
    for(int i=0;i<lignes.Length;i++){
      // suivi
      Console.Out.WriteLine("<-- "+lignes[i]);
      // comparaison ligne courante au modèle
      MatchRésultat=ModèleRésultat.Match(lignes[i]);
      // a-t-on trouvé ?
      if(MatchRésultat.Success){
        // on note le résultat
        strRésultat=MatchRésultat.Groups[1].Value;
      }
    }//ligne suivante
    // on affiche le résultat
    Console.Out.WriteLine("[résultat="+strRésultat+"]\n");
  }//executeFonction
```

Tout d'abord, le client HTTP-POST envoie sa demande au format POST :

```
    // construction de la chaîne de requête
    string requête="a="+HttpUtility.UrlEncode(a)+"&b="+HttpUtility.UrlEncode(b);
    int nbChars=requête.Length;

    // construction du tableau des entêtes HTTP à envoyer
    string[] entetes=new string[6];
    entetes[0]="POST " + uri.AbsolutePath+ "/"+fonction+" HTTP/1.1";
    entetes[1]="Host: " + uri.Host+":"+uri.Port;
    entetes[2]="Content-Type: application/x-www-form-urlencoded";
    entetes[3]="Content-Length: "+nbChars;
    entetes[4]="Connection: Keep-Alive";
    entetes[5]="";

    // on envoie les entêtes HTTP au serveur
    for(int i=0;i<entetes.Length;i++){
      // envoi au serveur
      OUT.WriteLine(entetes[i]);
      // écho écran
      Console.Out.WriteLine("--> "+entetes[i]);
    }//for
```

Dans l'entête

```
--> Content-Length: 7
```

on doit indiquer la taille des paramètres qui seront envoyés par le client derrière les entêtes HTTP :

```
--> a=8&b=9
```

Pour cela on utilise le code suivant :

```
    // construction de la chaîne de requête
    string requête="a="+HttpUtility.UrlEncode(a)+"&b="+HttpUtility.UrlEncode(b);
    int nbChars=requête.Length;
```

La méthode *HttpUtility.UrlEncode(string chaine)* transforme certains des caractères de *chaîne* en *%n1n2* où *n1n2* est le code ASCII du caractère transformé. Les caractères visés par cette transformation sont tous les caractères ayant un sens particulier dans une requête POST (l'espace, le signe =, le signe &, ...). Ici la méthode *HttpUtility.UrlEncode* est normalement inutile puisque *a* et *b* sont des nombres qui ne contiennent aucun de ces caractères particuliers. Elle est ici employée à titre d'exemple. Elle a besoin de l'espace de noms *System.Web*.

Une fois que le client a envoyé ses entêtes HTTP

```
ajouter 6 7
--> POST /st/operations/operations.asmx/ajouter HTTP/1.1
--> Host: localhost:80
--> Content-Type: application/x-www-form-urlencoded
--> Content-Length: 7
--> Connection: Keep-Alive
-->
```

le serveur répond par l'entête HTTP *100 Continue* :

```
<-- HTTP/1.1 100 Continue
```

```
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:11:58 GMT
<--
```

Le code se contente de lire et d'afficher à l'écran cette première réponse :

```
    // on lit la 1ere réponse du serveur Web HTTP/1.1 100
    string ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      //écho
      Console.Out.WriteLine("<-- "+ligne);
    }//while
    //écho dernière ligne
    Console.Out.WriteLine("<-- "+ligne);
```

Une fois cette première réponse lue, le client doit envoyer ses paramètres :

```
--> a=8&b=9
```

Il le fait avec le code suivant :

```
    // envoi paramètres de la requête
    OUT.Write(requête);
    // echo
    Console.Out.WriteLine("--> "+requête);
```

Le serveur va alors envoyer sa réponse :

```
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:11:47 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 91
<--
<-- <?xml version="1.0" encoding="utf-8"?>
<-- <double xmlns="st.istia.univ-angers.fr">13</double>
```

Cette réponse étant identique à celle faite au client HTTP-GET, le code du client HTTP-POST pour traiter cette réponse est identique au code correspondant du client HTTP-GET.

9.5 Un client SOAP

Nous en arrivons au troisième client, celui qui utilise un dialogue client-serveur de type **SOAP** (Simple Object Access Protocol). Un exemple de dialogue nous est présenté pour la fonction *ajouter* :

SOAP

Le texte suivant est un exemple de demande et de réponse SOAP. Les **espaces réservés** affichés doivent être remplacés par des valeurs réelles.

```
POST /st/operations/operations.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "st.istia.univ-angers.fr/ajouter"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:
  <soap:Body>
    <ajouter xmlns="st.istia.univ-angers.fr">
      <a>double</a>
      <b>double</b>
    </ajouter>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:
  <soap:Body>
    <ajouterResponse xmlns="st.istia.univ-angers.fr">
      <ajouterResult>double</ajouterResult>
    </ajouterResponse>
  </soap:Body>
</soap:Envelope>
```

La demande du client est une demande POST. On va donc retrouver certains des mécanismes du client précédent. La principale différence est qu'alors que le client HTTP-POST envoyait les paramètres *a* et *b* sous la forme

```
a=A&b=B
```

le client SOAP les envoie dans un format XML plus complexe :

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ajouter xmlns="st.istia.univ-angers.fr">
      <a>double</a>
      <b>double</b>
    </ajouter>
  </soap:Body>
</soap:Envelope>
```

Il reçoit en retour une réponse XML également plus complexe que les réponses vues précédemment :

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ajouterResponse xmlns="st.istia.univ-angers.fr">
      <ajouterResult>double</ajouterResult>
    </ajouterResponse>
  </soap:Body>
</soap:Envelope>
```

Même si la demande et la réponse sont plus complexes, il s'agit bien du même mécanisme HTTP que pour le client HTTP-POST. L'écriture du client SOAP peut être ainsi calqué sur celle du client HTTP-POST. Voici un exemple d'exécution :

```
E:\data\serge\MSNET\c#\webservices\clientSOAP>clientsoap1 http://localhost/st/operations/operations.
asmx
Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a b

ajouter 3 4
--> POST /st/operations/operations.asmx HTTP/1.1
--> Host: localhost:80
--> Content-Type: text/xml; charset=utf-8
--> Content-Length: 321
--> Connection: Keep-Alive
--> SOAPAction: "st.istia.univ-angers.fr/ajouter"
-->
<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:39:17 GMT
<--
--> <?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/20
01/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ajouter xmlns="st.istia.univ-angers.fr">
<a>3</a>
<b>4</b>
</ajouter>
</soap:Body>
</soap:Envelope>
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:39:17 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
```

```
<-- Content-Length: 345
<--
<-- <?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap
/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema"><soap:Body><ajouterResponse xmlns="st.istia.univ-angers.fr"><ajouterResult>7</ajouterResu
lt></ajouterResponse></soap:Body></soap:Envelope>
[résultat=7]
```

Là encore, seule la méthode *executeFonction* change. Le client SOAP envoie les entêtes HTTP de sa demande. Ils sont simplement un peu plus complexes que ceux de HTTP-POST :

```
ajouter 3 4
--> POST /st/operations/operations.asmx HTTP/1.1
--> Host: localhost:80
--> Content-Type: text/xml; charset=utf-8
--> Content-Length: 321
--> Connection: Keep-Alive
--> SOAPAction: "st.istia.univ-angers.fr/ajouter"
-->
```

Le code qui les génère :

```
// executeFonction
  public static void executeFonction(StreamReader IN, StreamWriter OUT, Uri uri, string fonction, string
a, string b){
    // exécute fonction(a,b) sur le service web d'URI uri
    // les échanges client-serveur se font via les flux IN et OUT
    // le résultat de la fonction est dans la ligne
    // <double xmlns="st.istia.univ-angers.fr">double</double>
    // envoyée par le serveur

    // construction de la chaîne de requête SOAP
    string requêteSOAP="<?xml version="+"\"1.0\" encoding=\"utf-8\"?>\n";
    requêteSOAP+="<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">\n";
    requêteSOAP+="<soap:Body>\n";
     requêteSOAP+="<"+fonction+" xmlns=\"st.istia.univ-angers.fr\">\n";
     requêteSOAP+="<a>"+a+"</a>\n";
     requêteSOAP+="<b>"+b+"</b>\n";
     requêteSOAP+="</"+fonction+">\n";
    requêteSOAP+="</soap:Body>\n";
    requêteSOAP+="</soap:Envelope>";
    int nbCharsSOAP=requêteSOAP.Length;

    // construction du tableau des entêtes HTTP à envoyer
    string[] entetes=new string[7];
    entetes[0]="POST " + uri.AbsolutePath+" HTTP/1.1";
    entetes[1]="Host: " + uri.Host+":"+uri.Port;
    entetes[2]="Content-Type: text/xml; charset=utf-8";
    entetes[3]="Content-Length: "+nbCharsSOAP;
    entetes[4]="Connection: Keep-Alive";
    entetes[5]="SOAPAction: \"st.istia.univ-angers.fr/"+fonction+"\"";
    entetes[6]="";

    // on envoie les entêtes HTTP au serveur
    for(int i=0;i<entetes.Length;i++){
      // envoi au serveur
      OUT.WriteLine(entetes[i]);
      // écho écran
      Console.Out.WriteLine("--> "+entetes[i]);
    }//for
```

En recevant cette demande, le serveur envoie sa première réponse que le client affiche :

```
<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:39:17 GMT
<--
```

Le code :

```
    // on lit la 1ere réponse du serveur Web HTTP/1.1 100
    string ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      //écho
      Console.Out.WriteLine("<-- "+ligne);
    }//while
    //écho dernière ligne
    Console.Out.WriteLine("<-- "+ligne);
```

Le client va maintenant envoyer ses paramètres au format XML dans quelque chose qu'on appelle une enveloppe SOAP :

```
--> <?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/20
01/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ajouter xmlns="st.istia.univ-angers.fr">
<a>3</a>
<b>4</b>
</ajouter>
</soap:Body>
</soap:Envelope>
```

Le code :

```
      // envoi paramètres de la requête
    OUT.Write(requêteSOAP);
    // echo
    Console.Out.WriteLine("--> "+requêteSOAP);
```

Le serveur va alors envoyer sa réponse définitive :

```
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Wed, 15 May 2002 14:49:08 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 345
<--
<-- <?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap
/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema"><soap:Body><ajouterResponse xmlns="st.istia.univ-angers.fr"><ajouterResult>7</ajouterResu
lt></ajouterResponse></soap:Body></soap:Envelope>
[résultat=7]
```

Le client affiche à l'écran les entêtes HTTP reçus tout en y cherchant la ligne *Content-Length* :

```
    // construction de l'expression régulière permettant de retrouver la taille de la réponse XML
    // dans le flux de la réponse du serveur web
    string modèleLength=@"^Content-Length: (.+?)\s*$";
    Regex RegexLength=new Regex(modèleLength);
    Match MatchLength=null;
    int longueur=0;

    // lecture seconde réponse du serveur web après envoi de la requête
    // on mémorise la valeur de la ligne Content-Length
    ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      // écho écran
      Console.Out.WriteLine("<-- " + ligne);
      // Content-Length ?
      MatchLength=RegexLength.Match(ligne);
      if(MatchLength.Success){
        longueur=int.Parse(MatchLength.Groups[1].Value);
      }//if
    }//while
    // écho dernière ligne
    Console.Out.WriteLine("<--");
```

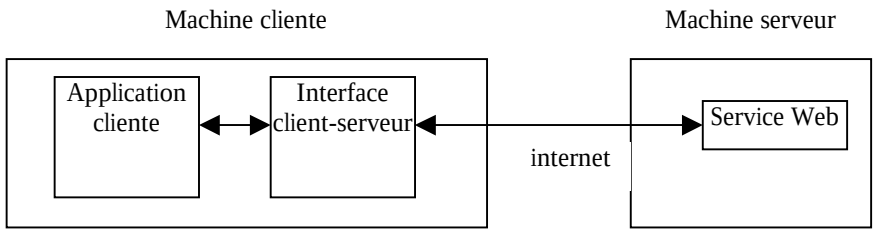
Une fois la taille N de la réponse XML connue, le client lit N caractères dans le flux de la réponse du serveur, décompose la chaîne récupérée en lignes de texte pour les afficher à l'écran et y chercher la balise XML du résultat : *<ajouterResult>7</ajouterResult>* et afficher ce dernier :

```
// construction de l'expression régulière permettant de retrouver le résultat
    // dans le flux de la réponse du serveur web
    string modèle="<"+fonction+"Result>(.+?)</"+fonction+"Result>";
    Regex ModèleRésultat=new Regex(modèle);
    Match MatchRésultat=null;

    // on lit le reste de la réponse du serveur web
    char[] chrRéponse=new char[longueur];
    IN.Read(chrRéponse,0,longueur);
    string strRéponse=new String(chrRéponse);

    // on décompose la réponse en lignes de texte
    string[] lignes=Regex.Split(strRéponse,"\n");

    // on parcourt les lignes de texte à la recherche du résultat
    string strRésultat="?";  // résultat de la fonction
    for(int i=0;i<lignes.Length;i++){
      // suivi
      Console.Out.WriteLine("<-- "+lignes[i]);
      // comparaison ligne courante au modèle
```

```
      MatchRésultat=ModèleRésultat.Match(lignes[i]);
      // a-t-on trouvé ?
      if(MatchRésultat.Success){
        // on note le résultat
        strRésultat=MatchRésultat.Groups[1].Value;
      }
    }//ligne suivante
    // on affiche le résultat
    Console.Out.WriteLine("[résultat="+strRésultat+"]\n");
  }//executeFonction
```

9.6 Encapsulation des échanges client-serveur

Imaginons que notre service web *operations* soit utilisé par diverses applications. Il serait intéressant de mettre à disposition de celles-ci une classe qui ferait l'interface entre l'application cliente et le service web et qui cacherait la majeure partie des échanges réseau qui pour la plupart des développeurs ne sont pas triviaux. On aurait ainsi le schéma suivant :



L'application cliente s'adresserait à l'interface client-serveur pour faire ses demandes au service web. Celle-ci ferait tous les échanges réseau nécessaires avec le serveur et rendrait le résultat obtenu à l'application cliente. Celle-ci n'aurait plus à s'occuper des échanges avec le serveur ce qui faciliterait grandement son écriture.

9.6.1 La classe d'encapsulation

Après ce qui a été vu dans les paragraphes précédents, nous connaissons bien maintenant les échanges réseau entre le client et le serveur. Nous avons même vu trois méthodes. Nous choisissons d'encapsuler la méthode SOAP. La classe est la suivante :

```
// espaces de noms
using System;
using System.Net.Sockets;
using System.IO;
using System.Text.RegularExpressions;
using System.Collections;
using System.Web;

  // clientSOAP du service Web operations
public class clientSOAP{

  // variables d'instance
  Uri uri=null;          // l'URI du service web
  TcpClient client=null; // la liaison tcp du client avec le serveur
  StreamReader IN=null;  // le flux de lecture du client
  StreamWriter OUT=null; // le flux d'écriture du client
  // dictionnaire des fonctions
  Hashtable dicoFonctions=new Hashtable();
  // liste des fonctions
  string[] fonctions={"ajouter","soustraire","multiplier","diviser"};
  // verbose
  bool verbose=false;    // à vrai, affiche à l'écran les échanges client-serveur

  // constructeur
  public clientSOAP(string uriString, bool verbose){

    // on note verbose
    this.verbose=verbose;

    // connexion au serveur
    uri=new Uri(uriString);
    client=new TcpClient(uri.Host,uri.Port);

    // on crée les flux d'entrée-sortie du client TCP
    IN=new StreamReader(client.GetStream());
    OUT=new StreamWriter(client.GetStream());
    OUT.AutoFlush=true;

    // création du dictionnaire des fonctions du service web
    for (int i=0;i<fonctions.Length;i++){
      dicoFonctions.Add(fonctions[i],true);
    }//for
```

```
  }//constructeur

  public void Close(){
    // fin liaison client-serveur
    IN.Close();OUT.Close();client.Close();
  }//Close

  // executeFonction
  public string executeFonction(string fonction, string a, string b){
    // exécute fonction(a,b) sur le service web d'URI uri
    // les échanges client-serveur se font via les flux IN et OUT
    // le résultat de la fonction est dans la ligne
    // <double xmlns="st.istia.univ-angers.fr">double</double>
    // envoyée par le serveur

    // fonction valide ?
    fonction=fonction.Trim().ToLower();
    if (! dicoFonctions.ContainsKey(fonction))
      return "[fonction ["+fonction+"] indisponible : (ajouter, soustraire,multiplier,diviser)]";
    // arguments a et b valides ?
    double doubleA=0;
    try{
      doubleA=double.Parse(a);
    }catch{
      return "[argument ["+a+"] incorrect (double)]";
    }//catch
    double doubleB=0;
    try{
      doubleB=double.Parse(b);
    }catch{
      return "[argument ["+b+"] incorrect (double)]";
    }//catch
    // division par zéro ?
    if(fonction=="diviser" && doubleB==0)
      return "[division par zéro]";

    // construction de la chaîne de requête SOAP
    string requêteSOAP="<?xml version="+"\"1.0\" encoding=\"utf-8\"?>\n";
    requêteSOAP+="<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">\n";
    requêteSOAP+="<soap:Body>\n";
    requêteSOAP+="<"+fonction+" xmlns=\"st.istia.univ-angers.fr\">\n";
    requêteSOAP+="<a>"+a+"</a>\n";
    requêteSOAP+="<b>"+b+"</b>\n";
    requêteSOAP+="</"+fonction+">\n";
    requêteSOAP+="</soap:Body>\n";
    requêteSOAP+="</soap:Envelope>";
    int nbCharsSOAP=requêteSOAP.Length;

    // construction du tableau des entêtes HTTP à envoyer
    string[] entetes=new string[7];
    entetes[0]="POST " + uri.AbsolutePath+" HTTP/1.1";
    entetes[1]="Host: " + uri.Host+":"+uri.Port;
    entetes[2]="Content-Type: text/xml; charset=utf-8";
    entetes[3]="Content-Length: "+nbCharsSOAP;
    entetes[4]="Connection: Keep-Alive";
    entetes[5]="SOAPAction: \"st.istia.univ-angers.fr/"+fonction+"\"";
    entetes[6]="";

    // on envoie les entêtes HTTP au serveur
    for(int i=0;i<entetes.Length;i++){
      // envoi au serveur
      OUT.WriteLine(entetes[i]);
      // écho écran
      if (verbose) Console.Out.WriteLine("--> "+entetes[i]);
    }//for

    // on lit la 1ere réponse du serveur Web HTTP/1.1 100
    string ligne=null;     // une ligne du flux de lecture
    while((ligne=IN.ReadLine())!=""){
      //écho
      if (verbose) Console.Out.WriteLine("<-- "+ligne);
    }//while
    //écho dernière ligne
    if (verbose) Console.Out.WriteLine("<-- "+ligne);

    // envoi paramètres de la requête
    OUT.Write(requêteSOAP);
    // echo
    if (verbose) Console.Out.WriteLine("--> "+requêteSOAP);

    // construction de l'expression régulière permettant de retrouver la taille de la réponse XML
    // dans le flux de la réponse du serveur web
    string modèleLength=@"^Content-Length: (.+?)\s*$";
    Regex RegexLength=new Regex(modèleLength);
    Match MatchLength=null;
    int longueur=0;

    // lecture seconde réponse du serveur web après envoi de la requête
    // on mémorise la valeur de la ligne Content-Length
    ligne=null;     // une ligne du flux de lecture
```

```
    while((ligne=IN.ReadLine())!=""){
      // écho écran
      if (verbose) Console.Out.WriteLine("<-- " + ligne);
      // Content-Length ?
      MatchLength=RegexLength.Match(ligne);
      if(MatchLength.Success){
        longueur=int.Parse(MatchLength.Groups[1].Value);
      }//if
    }//while
    // écho dernière ligne
    if (verbose) Console.Out.WriteLine("<--");

    // construction de l'expression régulière permettant de retrouver le résultat
    // dans le flux de la réponse du serveur web
    string modèle="<"+fonction+"Result>(.+?)</"+fonction+"Result>";
    Regex ModèleRésultat=new Regex(modèle);
    Match MatchRésultat=null;

    // on lit le reste de la réponse du serveur web
    char[] chrRéponse=new char[longueur];
    IN.Read(chrRéponse,0,longueur);
    string strRéponse=new String(chrRéponse);

    // on décompose la réponse en lignes de texte
    string[] lignes=Regex.Split(strRéponse,"\n");

    // on parcourt les lignes de texte à la recherche du résultat
    string strRésultat="?";  // résultat de la fonction
    for(int i=0;i<lignes.Length;i++){
      // suivi
      if (verbose) Console.Out.WriteLine("<-- "+lignes[i]);
      // comparaison ligne courante au modèle
      MatchRésultat=ModèleRésultat.Match(lignes[i]);
      // a-t-on trouvé ?
      if(MatchRésultat.Success){
        // on note le résultat
        strRésultat=MatchRésultat.Groups[1].Value;
      }
    }//ligne suivante
    // on renvoie le résultat
    return strRésultat;
  }//executeFonction
}//classe
```

Nous ne retrouvons rien de neuf par raport à ce qui a été déjà vu. Nous avons simplement repris le code du client SOAP étudié et l'avons réaménagé quelque peu pour en faire une classe. Celle-ci a un constructeur et deux méthodes :

```
  // constructeur
  public clientSOAP(string uriString, bool verbose){
```

```
  // executeFonction
  public string executeFonction(string fonction, string a, string b){
```

```
  public void Close(){
```

et a les attributs suivants :

```
  // variables d'instance
  Uri uri=null;          // l'URI du service web
  TcpClient client=null; // la liaison tcp du client avec le serveur
  StreamReader IN=null;  // le flux de lecture du client
  StreamWriter OUT=null; // le flux d'écriture du client
  // dictionnaire des fonctions
  Hashtable dicoFonctions=new Hashtable();
  // liste des fonctions
  string[] fonctions={"ajouter","soustraire","multiplier","diviser"};
  // verbose
  bool verbose=false;    // à vrai, affiche à l'écran les échanges client-serveur
```

On passe au constructeur deux paramètres :

1. l'URI du service web auquel il doit se connecter
2. un booléen *verbose* qui à vrai demande que les échanges réseau soient affichés à l'écran, sinon ils ne le seront pas.

Au cours de la construction, on construit les flux *IN* de lecture réseau, *OUT* d'écriture réseau, ainsi que le dictionnaire des fonctions gérées par le service. Une fois l'objet construit, la connexion client-serveur est ouverte et ses flux *IN* et *OUT* utilisables.

La méthode *Close* permet de fermer la liaison avec le serveur.

La méthode *ExecuteFonction* est celle qu'on a écrite pour le client SOAP étudié, à quelques détails près :

1. Les paramètres uri, IN et OUT qui étaient auparavant passés en paramètres à la méthode n'ont plus besoin de l'être puisque ce sont maintenant des attributs d'instance accessibles à toutes les méthodes de l'instance

2. La méthode *ExecuteFonction* qui rendait auparavant un type *void* et affichait le résultat de la fonction à l'écran, rend maintenant ce résultat et donc un type *string*.

Typiquement un client utilisera la classe *clientSOAP* de la façon suivante :

1. création d'un objet *clientSOAP* qui va créer la liaison avec le service web
2. utilisation répétée de la méthode *executeFonction*
3. fermeture de la liaison avec le service Web par la méthode *Close*.

Etudions un premier client.

9.6.2 Un client console

Nous reprenons ici le client SOAP étudié alors que la classe clientSOAP n'existait pas et nous le réaménageons afin qu'il utilise maintenant cette classe :

```
// espaces de noms
using System;
using System.IO;
using System.Text.RegularExpressions;

public class testClientSoap{

  // demande l'URI du service web operations
  // exécute de façon interactive les commandes tapées au clavier

  public static void Main(string[] args){
    // syntaxe
    const string syntaxe="pg URI [verbose]";

    // nombre d'arguments
    if(args.Length != 1 && args.Length!=2)
      erreur(syntaxe,1);
    // verbose ?
    bool verbose=false;
    if(args.Length==2) verbose=args[1].ToLower()=="verbose";

    // on se connecte au service web
    clientSOAP client=null;
    try{
      client=new clientSOAP(args[0],verbose);
    }catch(Exception ex){
      // erreur de connexion
      erreur("L'erreur suivante s'est produite lors de la connexion au service web : "
        + ex.Message,2);
    }//catch

    // les demandes de l'utilisateur sont tapées au clavier
    // sous la forme fonction a b
    // elles se terminent avec la commande fin

    string commande=null;  // commande tapée au clavier
    string[] champs=null;  // champs d'une ligne de commande

    // invite à l'utilisateur
    Console.Out.WriteLine("Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a
b\n");

    // gestion des erreurs
    try{
      // boucle de saisie des commandes tapées au clavier
      while(true){
        // lecture commande
        commande=Console.In.ReadLine().Trim().ToLower();
        // fini ?
        if(commande==null || commande=="fin") break;
        // décomposition de la commande en champs
        champs=Regex.Split(commande,@"\s+");
        // il faut trois champs
        if(champs.Length!=3){
          // commande invalide
          Console.Out.WriteLine("syntaxe : [ajouter|soustraire|multiplier|diviser] a b");
          // on recommence
          continue;
        }//if
        // on fait la demande au service web

  Console.Out.WriteLine("résultat="+client.executeFonction(champs[0].Trim().ToLower(),champs[1].Trim(),cha
mps[2].Trim()));
        // demande suivante
      }//while
      // fin des demandes
    }catch(Exception e){
      Console.Out.WriteLine("L'erreur suivante s'est produite : " + e.Message);
```

```
    }//catch

    // fin liaison client-serveur
    try{
      client.Close();
    }catch{}
  }//Main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe
```

La client est maintenant considérablement plus simple et on n'y retrouve aucune communication réseau. Le client admet deux paramètres :

1. l'URI du service web *operations*
2. le mot clé facultatif *verbose*. S'il est présent, les échanges réseau seront affichés à l'écran.

Ces deux paramètres sont utilisés pour construire un objet *clientSOAP* qui va assurer les échanges avec le service web.

```
    // on se connecte au service web
    clientSOAP client=null;
    try{
      client=new clientSOAP(args[0],verbose);
    }catch(Exception ex){
      // erreur de connexion
      erreur("L'erreur suivante s'est produite lors de la connexion au service web : "
        + ex.Message,2);
    }//catch
```

Une fois ouverte la connexion avec le service web, le client peut envoyer ses demandes. Celles-ci sont tapées au clavier, analysées puis envoyées au serveur par appel à la méthode *executeFonction* de l'objet *clientSOAP*.

```
        // on fait la demande au service web

  Console.Out.WriteLine("résultat="+client.executeFonction(champs[0].Trim().ToLower(),champs[1].Trim(),cha
mps[2].Trim()));
```

La classe *clientSOAP* est compilée dans un "assembly" :

```
dos> csc /t:library clientsoap.cs

dos>dir
16/05/2002  09:08                 6 065 ClientSOAP.cs
16/05/2002  09:08                 7 168 ClientSOAP.dll
```

L'application client *testClientSoap* est ensuite compilée par :

```
dos> csc /r:clientsoap.dll testclientsoap.cs
dos>dir
16/05/2002  09:08                 6 065 ClientSOAP.cs
16/05/2002  09:08                 7 168 ClientSOAP.dll
16/05/2002  10:03                 2 429 testClientSoap.cs
16/05/2002  10:03                 4 608 testClientSoap.exe
```

Voici un exemple d'exécution non verbeux :

```
E:\data\serge\>testclientsoap http://localhost/st/operations/operations.asmx
Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a b

ajouter 1 3
résultat=4
soustraire 6 7
résultat=-1
multiplier 4 5
résultat=20
diviser 1 2
résultat=0.5
x
syntaxe : [ajouter|soustraire|multiplier|diviser] a b
x 1 2
résultat=[fonction [x] indisponible : (ajouter, soustraire,multiplier,diviser)]
ajouter a b
résultat=[argument [a] incorrect (double)]
ajouter 1 b
```

```
résultat=[argument [b] incorrect (double)]
diviser 1 0
résultat=[division par zéro]
fin
```

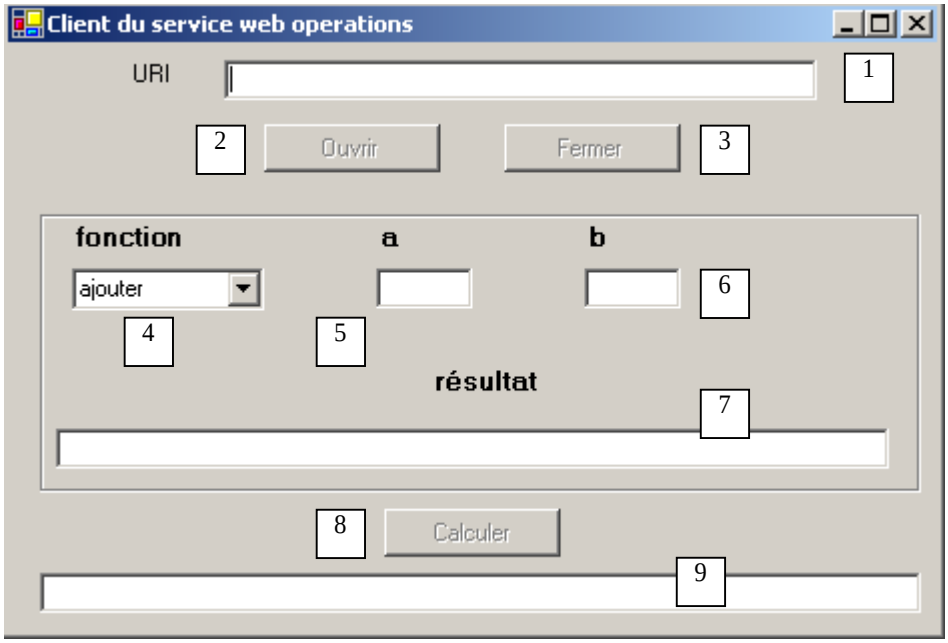
On peut suivre les échanges réseau en demandant une exécutions "verbeuse" :

```
E:\data\serge>testclientsoap http://localhost/st/operations/operations.asmx verbose
Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser] a b

ajouter 4 8
--> POST /st/operations/operations.asmx HTTP/1.1
--> Host: localhost:80
--> Content-Type: text/xml; charset=utf-8
--> Content-Length: 321
--> Connection: Keep-Alive
--> SOAPAction: "st.istia.univ-angers.fr/ajouter"
-->
<-- HTTP/1.1 100 Continue
<-- Server: Microsoft-IIS/5.0
<-- Date: Thu, 16 May 2002 08:15:15 GMT
<--
--> <?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/20
01/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ajouter xmlns="st.istia.univ-angers.fr">
<a>4</a>
<b>8</b>
</ajouter>
</soap:Body>
</soap:Envelope>
<-- HTTP/1.1 200 OK
<-- Server: Microsoft-IIS/5.0
<-- Date: Thu, 16 May 2002 08:15:15 GMT
<-- Cache-Control: private, max-age=0
<-- Content-Type: text/xml; charset=utf-8
<-- Content-Length: 346
<--
<-- <?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap
/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema"><soap:Body><ajouterResponse xmlns="st.istia.univ-angers.fr"><ajouterResult>12</ajouterRes
ult></ajouterResponse></soap:Body></soap:Envelope>
résultat=12
fin
```

Construisons maintenant un client graphique.

9.6.3 Un client graphique windows

Nous allons maintenant interroger notre service web avec un client graphique qui utilisera lui aussi la classe *clientSOAP*. L'interface graphique sera la suivante :



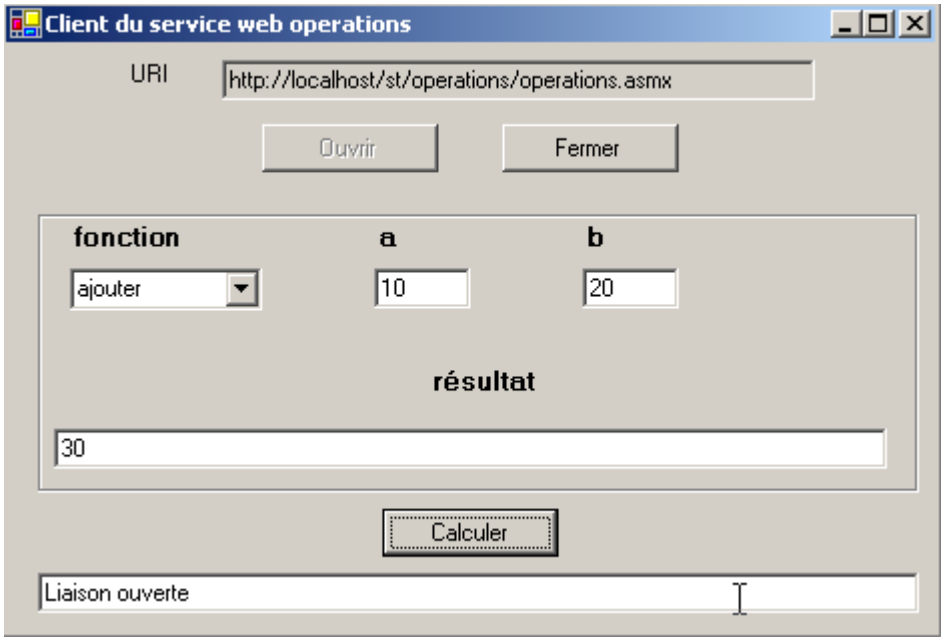
Les contrôles sont les suivants :

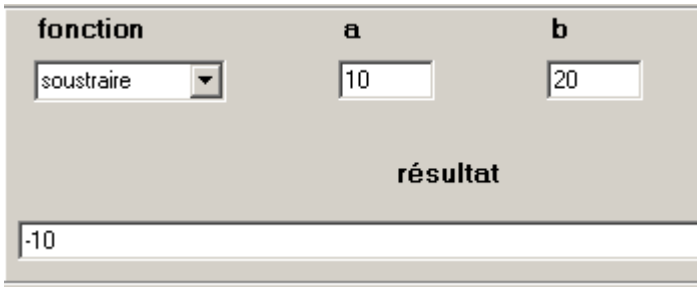
n°	type	nom	rôle
1	TextBox	txtURI	l'URI du service web operations
2	Button	btnOuvrir	ouvre la liaison avec le service Web
3	Button	btnFermer	ferme la liaison avec le service Web
4	ComboBox	cmbFonctions	la liste des fonction (ajouter, soustraire, multiplier, diviser)
5	TextBox	txtA	l'argument a des fonctions
6	TextBox	txtB	l'argument b des fonctions
7	TextBox	txtRésultat	le résultat de fonction(a,b)
8	Button	btnCalculer	lance le calcul de fonction(a,b)
9	TextBox	txtErreur	affiche un msg d'état de la liaison

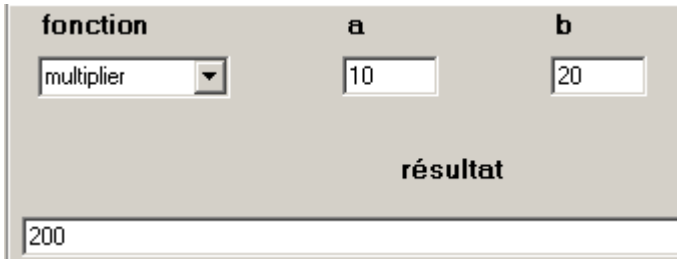
Il y a quelques contraintes de fonctionnement :

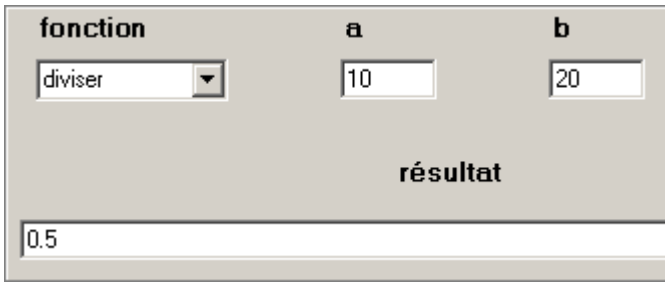
- le bouton *btnOuvrir* n'est actif que si le champ *txtURI* est non vide et qu'une liaison n'est pas déjà ouverte
- le bouton *btnFermer* n'est actif que lorsqu'une liaison avec le service web a été ouverte
- le bouton *btnCalculer* n'est actif que lorsqu'une liaison est ouverte et ques champs *txtA* et *txtB* sont non vides
- les champs *txtRésultat* et *txtErreur* ont l'attribut *ReadOnly* à vrai

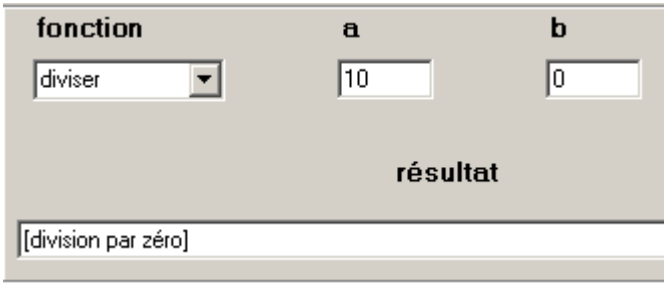
L'exécution de ce client donne les résultats suivants :











Le code de l'application suit. Nous avons omis le code du formulaire qui ne présente pas d'intérêt ici.

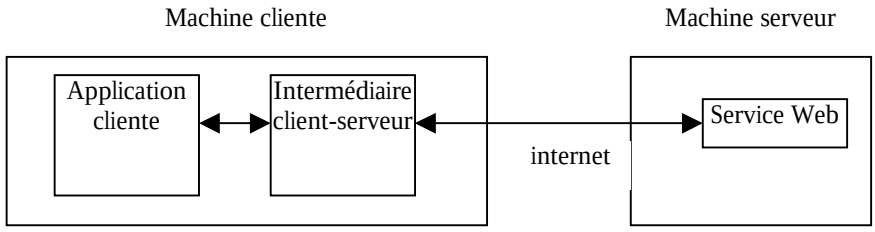
```
  public class Form1 : System.Windows.Forms.Form
  {
    // contrôles formulaire
...

    // variables globales
    clientSOAP client=null;

    public Form1()
    {
      // Required for Windows Form Designer support
      InitializeComponent();

      // init formulaire
      btnOuvrir.Enabled=false;
      btnFermer.Enabled=false;
      btnCalculer.Enabled=false;
      cmbFonctions.SelectedIndex=0;
      // client SOAP
    }

    private void InitializeComponent()
    {
      ...

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

```
    private void txtURI_TextChanged(object sender, System.EventArgs e) {
      // contrôle le bouton ouvrir
      btnOuvrir.Enabled=txtURI.Text.Trim()!="";
    }

    private void btnOuvrir_Click(object sender, System.EventArgs e) {
      try{
        // connexion au service Web
        client=new clientSOAP(txtURI.Text,false);
        // pas d'erreur
        txtErreur.Text="Liaison ouverte";
        // boutons
        btnOuvrir.Enabled=false;
        btnFermer.Enabled=true;
        // URI
        txtURI.ReadOnly=true;
      }catch(Exception ex){
        // erreur
        txtErreur.Text=ex.Message;
      }//catch
    }

    private void btnFermer_Click(object sender, System.EventArgs e) {
      // on ferme la connexion au service web
      client.Close();
      // boutons
      btnFermer.Enabled=false;
      btnOuvrir.Enabled=true;
      btnCalculer.Enabled=false;
      // URI
      txtURI.ReadOnly=false;
      // état
      txtErreur.Text="Liaison fermée";
    }

    private void Form1_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
      // on ferme la liaison avant de quitter l'application
      if(btnFermer.Enabled) btnFermer_Click(null,null);
    }

    private void btnCalculer_Click(object sender, System.EventArgs e) {
      // calcul d'une fonction(a,b)
      try{
        txtRésultat.Text=""; // efface le résultat précédent
        txtRésultat.Text=client.executeFonction(cmbFonctions.Text,txtA.Text.Trim(),txtB.Text.Trim());
      }catch(Exception ex){
        // erreur
        txtErreur.Text=ex.Message;
        // on ferme la liaison
        btnFermer_Click(null,null);
      }//catch
    }

    private void txtA_TextChanged(object sender, System.EventArgs e) {
      // contrôle l'état du bouton calculer
      btnCalculer.Enabled=txtA.Text.Trim()!="" && txtB.Text.Trim()!="" && btnFermer.Enabled;
    }

    private void txtB_TextChanged(object sender, System.EventArgs e) {
      // idem txtA
      txtA_TextChanged(null,null);
    }
  }
```

Là encore la classe *clientSOAP* cache tout l'aspect réseau de l'application.

9.7 Un client proxy

Rappelons ce qui vient d'être fait. Nous avons créé une classe intermédiaire encapsulant les échanges réseau entre un client et un service web selon le schéma ci-dessous :



La plate-forme .NET pousse cette logique plus loin. Une fois connu le service Web à atteindre, nous pouvons générer automatiquement la classe qui va nous servir d'intermédiaire pour atteindre les fonctions du service Web et qui cachera toute la partie réseau. On appelle cette classe un **proxy** pour le service web pour lequel elle a été générée.

Comment générer la classe proxy d'un service web ? Un service web est toujours accompagné d'un fichier de description au format XML. Si l'URI de notre service web operations est *http://localhost/st/operations/operations.asmx*, son fichier de description est disponible à l'URL *http://localhost/st/operations/operations.asmx**?wsdl*** comme le montre la copie d'écran suivante :

```
Address http://localhost/st/operations/operations.asmx?wsdl

  <?xml version="1.0" encoding="utf-8" ?>
- <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:soap="http://schemas.xml
    xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="st.istia.univ-angers.fr"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tm="http://microsoft.com/
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" targetNamespace="st.istia.univ-angers.fr"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <s:schema elementFormDefault="qualified" targetNamespace="st.istia.univ-angers.fr">
      - <s:element name="ajouter">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="1" maxOccurs="1" name="a" type="s:double" />
              <s:element minOccurs="1" maxOccurs="1" name="b" type="s:double" />
            </s:sequence>
          </s:complexType>
        </s:element>
```

On a là un fichier XML décrivant précisément toutes les fonctions du service web avec pour chacune d'elles le type et le nombre de paramètres, le type du résultat. On appelle ce fichier le fichier WSDL du service parce qu'il utilise le langage **WSDL** (*Web Services Description Language*). A partir de ce fichier, une classe proxy peut être générée à l'aide de l'outil *wsdl* :

```
E:\data\serge\MSNET\c#\webservices\clientproxy>wsdl http://localhost/st/operations/operations.asmx?wsdl
Microsoft (R) Web Services Description Language Utility
[Microsoft (R) .NET Framework, Version 1.0.3705.0]
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Écriture du fichier 'E:\data\serge\MSNET\c#\webservices\clientproxy\operations.cs'.

E:\data\serge\MSNET\c#\webservices\clientproxy>dir
16/05/2002  13:00                6 642 operations.cs
```

L'outil *wsdl* génère un fichier source C# portant le nom de la classe implémentant le service web, ici *operations*. Examinons une partie du code généré :

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.3705.0
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

//
// Ce code source a Ã©tÃ© automatiquement gÃ©nÃ©rÃ© par wsdl, Version=1.0.3705.0.
//
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;


/// <remarks/>
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Web.Services.WebServiceBindingAttribute(Name="operationsSoap", Namespace="st.istia.univ-
angers.fr")]
public class operations : System.Web.Services.Protocols.SoapHttpClientProtocol {

    /// <remarks/>
    public operations() {
        this.Url = "http://localhost/st/operations/operations.asmx";
    }

    /// <remarks/>
```

```
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute("st.istia.univ-angers.fr/ajouter",
RequestNamespace="st.istia.univ-angers.fr", ResponseNamespace="st.istia.univ-angers.fr",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public System.Double ajouter(System.Double a, System.Double b) {
        object[] results = this.Invoke("ajouter", new object[] {
                    a,
                    b});
        return ((System.Double)(results[0]));
    }

    /// <remarks/>
    public System.IAsyncResult Beginajouter(System.Double a, System.Double b, System.AsyncCallback
callback, object asyncState) {
        return this.BeginInvoke("ajouter", new object[] {
                    a,
                    b}, callback, asyncState);
    }

    /// <remarks/>
    public System.Double Endajouter(System.IAsyncResult asyncResult) {
        object[] results = this.EndInvoke(asyncResult);
        return ((System.Double)(results[0]));
    }
....
```

Ce code est un peu complexe au premier abord. Nous n'avons pas besoin d'en comprendre les détails pour pouvoir l'utiliser. Examinons tout d'abord la déclaration de la classe :

```
public class operations : System.Web.Services.Protocols.SoapHttpClientProtocol {
```

La classe porte le nom *operations* du service web pour lequel elle a été construite. Elle dérive de la classe *SoapHttpClientProtocol* :

```
// from module
'c:\winnt\assembly\gac\system.web.services\1.0.3300.0__b03f5f7f11d50a3a\system.web.services.dll'
public class System.Web.Services.Protocols.SoapHttpClientProtocol :
    System.Web.Services.Protocols.HttpWebClientProtocol,
    System.ComponentModel.IComponent,
    IDisposable
{

    // Fields

    // Constructors
    public SoapHttpClientProtocol();

    // Properties
    public bool AllowAutoRedirect { get; set; }
    public X509CertificateCollection ClientCertificates { get; }
    public string ConnectionGroupName { get; set; }
    public IContainer Container { get; }
    public CookieContainer CookieContainer { get; set; }
    public ICredentials Credentials { get; set; }
    public bool PreAuthenticate { get; set; }
    public IWebProxy Proxy { get; set; }
    public Encoding RequestEncoding { get; set; }
    public ISite Site { virtual get; virtual set; }
    public int Timeout { get; set; }
    public string Url { get; set; }
    public string UserAgent { get; set; }

    // Events
    public event EventHandler Disposed;

    // Methods
    public virtual void Abort();
    public virtual System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType);
    public void Discover();
    public virtual void Dispose();
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public virtual string ToString();
} // end of System.Web.Services.Protocols.SoapHttpClientProtocol
```

Notre classe proxy a un constructeur unique :

```
    public operations() {
        this.Url = "http://localhost/st/operations/operations.asmx";
    }
```

Le constructeur affecte à l'attibut *url* l'URL du service web associé au proxy. La classe *operations* ci-dessus ne définit pas elle-même l'attribut *url*. Celui-ci est hérité de la classe dont dérive le proxy : *System.Web.Services.Protocols.SoapHttpClientProtocol*. Examinons maintenant ce qui se rapporte à la méthode *ajouter* :

```
    public System.Double ajouter(System.Double a, System.Double b) {
        object[] results = this.Invoke("ajouter", new object[] {
                    a,
                    b});
        return ((System.Double)(results[0]));
    }
```

On peut constater qu'elle a la même signature que dans le service Web *operations* où elle était définie comme suit :

```
  [WebMethod]
  public double ajouter(double a, double b){
    return a+b;
  }//ajouter
```

La façon dont cette classe dialogue avec le service Web n'apparaît pas ici. Ce dialogue est entièrement pris en charge par la classe parent *System.Web.Services.Protocols.SoapHttpClientProtocol*. On ne trouve dans le proxy que ce qui le différencie des autres proxy :

- l'URL du service web associé
- la définition des méthodes du service associé.

Pour utiliser les méthodes du service web *operations*, un client n'a besoin que de la classe proxy *operations* générée précédemment. Compilons cette classe dans un fichier *assembly* :

```
E:\data\serge\MSNET\c#\webservices\clientproxy>csc /t:library operations.cs

E:\data\serge\MSNET\c#\webservices\clientproxy>dir
16/05/2002  13:00                6 642 operations.cs
16/05/2002  13:33                7 680 operations.dll
```

Maintenant écrivons un client. Nous reprenons un client console déjà utilisé plusieurs fois. Il est appelé sans paramètres et exécute les demandes tapées au clavier :

```
E:\data\serge\MSNET\c#\webservices\clientproxy>testclientproxy
Tapez vos commandes au format : [ajouter|soustraire|multiplier|diviser|toutfaire] a b

ajouter 4 5
résultat=9
soustraire 9 8
résultat=1
multiplier 10 4
résultat=40
diviser 6 7
résultat=0,857142857142857
toutfaire 10 20
résultats=[30,-10,200,0,5]
diviser 5 0
résultat=+Infini
fin
```

Le code du client est le suivant :

```
// espaces de noms
using System;
using System.IO;
using System.Text.RegularExpressions;
using System.Collections;

public class testClientProxy{

  // exécute de façon interactive les commandes tapées au clavier
  // et les envoie au service web operations

  public static void Main(){
    // il n'y a plus d'arguments
    // l'URL du service web étant codée en dur dans le proxy

    // création d'un dictionnaire des fonctions du service web
    string[] fonctions={"ajouter","soustraire","multiplier","diviser","toutfaire"};
    Hashtable dicoFonctions=new Hashtable();
    for (int i=0;i<fonctions.Length;i++){
      dicoFonctions.Add(fonctions[i],true);
    }//for

    // on crée un objet proxy operations
    operations myOperations=null;
```

```
    try{
      myOperations=new operations();
    }catch(Exception ex){
      // erreur de connexion
      erreur("L'erreur suivante s'est produite lors de la connexion au proxy dy service web : "
        + ex.Message,2);
    }//catch

    // les demandes de l'utilisateur sont tapées au clavier
    // sous la forme fonction a b
    // elles se terminent avec la commande fin

    string commande=null;  // commande tapée au clavier
    string[] champs=null;  // champs d'une ligne de commande

    // invite à l'utilisateur
    Console.Out.WriteLine("Tapez vos commandes au format :
[ajouter|soustraire|multiplier|diviser|toutfaire] a b\n");

    // gestion des erreurs
    // boucle de saisie des commandes tapées au clavier
    while(true){
      // lecture commande
      commande=Console.In.ReadLine().Trim().ToLower();
      // fini ?
      if(commande==null || commande=="fin") break;
      // décomposition de la commande en champs
      champs=Regex.Split(commande,@"\s+");
      // commande valide ?
      string fonction;
      double a,b;
      try{
        // il faut trois champs
        if(champs.Length!=3)throw new Exception();
        // le champ 0 doit être une fonction reconnue
        fonction=champs[0];
        if(! dicoFonctions.ContainsKey(fonction)) throw new Exception();
        // le champ 1 doit être un nombre valide
        a=double.Parse(champs[1]);
        // le champ 2 doit être un nombre valide
        b=double.Parse(champs[2]);
      }catch {
        // commande invalide
        Console.Out.WriteLine("syntaxe : [ajouter|soustraire|multiplier|diviser] a b");
        // on recommence
        continue;
      }//catch
      // on fait la demande au service web
      try{
        double résultat;
        double[] résultats;
        if(fonction=="ajouter"){
          résultat=myOperations.ajouter(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//ajouter
        if(fonction=="soustraire"){
          résultat=myOperations.soustraire(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//soustraire
        if(fonction=="multiplier"){
          résultat=myOperations.multiplier(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//multiplier
        if(fonction=="diviser"){
          résultat=myOperations.diviser(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//diviser
        if(fonction=="toutfaire"){
          résultats=myOperations.toutfaire(a,b);

  Console.Out.WriteLine("résultats=["+résultats[0]+","+résultats[1]+","+résultats[2]+","+résultats[3]+"]")
;
        }//toutfaire
        }catch(Exception e){
          Console.Out.WriteLine("L'erreur suivante s'est produite : " + e.Message);
        }//catch
        // demande suivante
      }//while
  }//Main

  // affichage des erreurs
  public static void erreur(string msg, int exitCode){
    // affichage erreur
    System.Console.Error.WriteLine(msg);
    // arrêt avec erreur
    Environment.Exit(exitCode);
  }//erreur
}//classe
```

Nous n'examinons que le code propre à l'utilisation de la classe proxy. Tout d'abord un objet proxy *operations* est créé :

```
    // on crée un objet proxy operations
    operations myOperations=null;
    try{
      myOperations=new operations();
    }catch(Exception ex){
      // erreur de connexion
      erreur("L'erreur suivante s'est produite lors de la connexion au proxy du service web : "
        + ex.Message,2);
    }//catch
```

Des lignes *fonction a b* sont tapées au clavier. A partir de ces informations, on appelle les méthodes appropriées du proxy :

```
      // on fait la demande au service web
      try{
        double résultat;
        double[] résultats;
        if(fonction=="ajouter"){
          résultat=myOperations.ajouter(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//ajouter
        if(fonction=="soustraire"){
          résultat=myOperations.soustraire(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//soustraire
        if(fonction=="multiplier"){
          résultat=myOperations.multiplier(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//multiplier
        if(fonction=="diviser"){
          résultat=myOperations.diviser(a,b);
          Console.Out.WriteLine("résultat="+résultat);
        }//diviser
        if(fonction=="toutfaire"){
          résultats=myOperations.toutfaire(a,b);

  Console.Out.WriteLine("résultats=["+résultats[0]+","+résultats[1]+","+résultats[2]+","+résultats[3]+"]")
;
        }//toutfaire
      }catch(Exception e){
          Console.Out.WriteLine("L'erreur suivante s'est produite : " + e.Message);
      }//catch
```

On traite ici pour la première fois, l'opération *toutfaire* qui fait les quatre opérations. Elle avait été ignorée pour l'instant car elle envoie un tableau de nombres encapsulé dans une enveloppe XML plus compliquée à gérer que les réponses XML simples des autres fonctions ne délivrant qu'un résultat. On voit qu'ici avec la classe proxy, utiliser la méthode *toutfaire* n'est pas plus compliqué qu'utiliser les autres méthodes.

9.8 Configurer un service Web

Un service Web peut avoir besoin d'informations de configuration pour s'initialiser correctement. Avec le serveur IIS, ces informations peuvent être pacées dans un fichier appelé *web.config* et situé dans le même dossier que le service web. Supposons qu'on veuille créer un service web ayant besoin de deux informations pour s'initialiser : un nom et un âge. Ces deux informations pevent être placées dans le fichier *web.config* sous la forme suivante :

```
<configuration>
  <appSettings>
    <add key="nom" value="tintin"/>
    <add key="age" value="27"/>
  </appSettings>
</configuration>
```

Les paramètres d'initialisation sont placées dans une enveloppe XML :

```
<configuration>
  <appSettings>
...
  </appSettings>
</configuration>
```

Un paramètre d'initialisation de nom *P* ayant la valeur *V* sera déclarée avec la ligne :

```
    <add key="P" value="V"/>
```

Comment le service Web récupère-t-il ces informations ? Lorsque IIS charge un service web, il regarde s'il y a dans le même dossier un fichier *web.config*. Si oui, il le lit. La valeur V d'un paramètre P est obtenue par l'instruction :

```
    string P=ConfigurationSettings.AppSettings["V"];
```

où *ConfigurationSettings* est une classe dans l'espace de noms *System.Configuration*.

Vérifions cette technique sur le service web suivant :

```
<%@ WebService Language=C# class=personne %>

using System.Web.Services;
using System.Configuration;

[WebService(Namespace="st.istia.univ-angers.fr")]
public class personne : WebService{

  // attributs
  string nom;
  int age;

  // constructeur
  public personne(){
    // init attributs
    nom=ConfigurationSettings.AppSettings["nom"];
    age=int.Parse(ConfigurationSettings.AppSettings["age"]);
  }//constructeur

  [WebMethod]
  public string id(){
    return "["+nom+","+age+"]";
  }//ajouter
}//classe
```

Le service web *personne* a deux attributs *nom* et *age* qui sont initialisés dans son constructeur sans paramètres à partir des valeurs lues dans le fichier de configuration *web.config* du service *personne*. Ce fichier est le suivant :

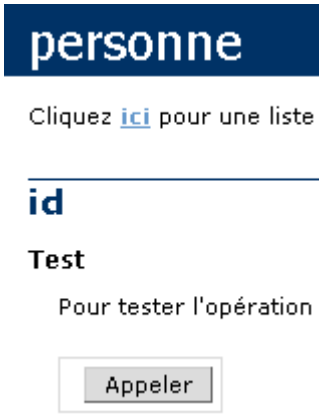
```
<configuration>
  <appSettings>
    <add key="nom" value="tintin"/>
    <add key="age" value="27"/>
  </appSettings>
</configuration>
```

Le service web a par ailleurs une **[WebMethod] id** sans paramètres et qui se contente de rendre les attributs *nom* et *age*. Le service est enregistré dans le fichier source *personne.asmx* qui est placé avec son fichier de configuration dans le dossier *c:\inetpub\wwwroot\st\personne* :

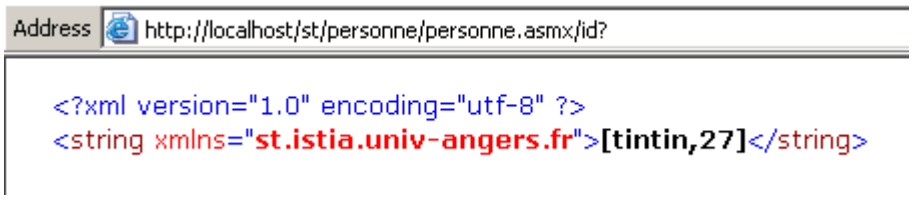
```
E:\data\serge\MSNET\c#\webservices\clientproxy>dir c:\inetpub\wwwroot\st\personne
16/05/2002  14:38                 133 web.config
16/05/2002  14:46                 521 personne.asmx
```

Prenon un navigateur et demandons l'url *http://localhost/st/personne/personne.asmx* du service *personne* :

Suivons le lien de l'unique méthode **id** :



La méthode *id* n'a pas de paramètres. Utilisons le bouton *Appeler* :



Nous avons bien récupéré les informations placées dans le fichier *web.config* du service.

9.9 Le service Web IMPOTS

Nous reprenons l'application IMPOTS maintenant bien connue. La dernière fois que nous avons travaillé avec, nous en avions fait un serveur distant qu'on pouvait appeler sur l'internet. Nous en faisons maintenant un service web.

9.9.1 Le service web

Nous partons de la classe *impôt* créée dans le chapitre sur les bases de données et qui se construit à partir des informations contenues dans une base de données ODBC :

```
// création d'une classe impôt

using System;
using System.Data;
using Microsoft.Data.Odbc;
using System.Collections;

public class impôt{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private decimal[] limites, coeffR, coeffN;

  // constructeur
  public impôt(string DSNimpots, string Timpots, string colLimites, string colCoeffR, string colCoeffN){
    // initialise les trois tableaux limites, coeffR, coeffN à partir
    // du contenu de la table Timpots de la base ODBC DSNimpots
    // colLimites, colCoeffR, colCoeffN sont les trois colonnes de cette table
    // peut lancer une exception

    string connectString="DSN="+DSNimpots+";"; // chaîne de connexion à la base
    OdbcConnection impotsConn=null;   // la connexion
    OdbcCommand sqlCommand=null;    // la commande SQL
    // la requête SELECT
    string selectCommand="select "+colLimites+","+colCoeffR+","+colCoeffN+" from "+Timpots;
    // tableaux pour récupérer les données
    ArrayList tLimites=new ArrayList();
    ArrayList tCoeffR=new ArrayList();
    ArrayList tCoeffN=new ArrayList();

    // on tente d'accéder à la base de données
    impotsConn = new OdbcConnection(connectString);
    impotsConn.Open();
    // on crée un objet command
    sqlCommand=new OdbcCommand(selectCommand,impotsConn);
    // on exécute la requête
    OdbcDataReader myReader=sqlCommand.ExecuteReader();
```

```
    // Exploitation de la table récupérée
    while (myReader.Read()) {
      // les données de la ligne courante sont mis dans les tableaux
      tLimites.Add(myReader[colLimites]);
      tCoeffR.Add(myReader[colCoeffR]);
      tCoeffN.Add(myReader[colCoeffN]);
    }//while
    // libération des ressources
    myReader.Close();
    impotsConn.Close();

    // les tableaux dynamiques sont mis dans des tableaux statiques
    this.limites=new decimal[tLimites.Count];
    this.coeffR=new decimal[tLimites.Count];
    this.coeffN=new decimal[tLimites.Count];
    for(int i=0;i<tLimites.Count;i++){
      limites[i]=decimal.Parse(tLimites[i].ToString());
      coeffR[i]=decimal.Parse(tCoeffR[i].ToString());
      coeffN[i]=decimal.Parse(tCoeffN[i].ToString());
    }//for
  }//constructeur 2

  // calcul de l'impôt
  public long calculer(bool marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
    decimal nbParts;
    if (marié) nbParts=(decimal)nbEnfants/2+2;
    else nbParts=(decimal)nbEnfants/2+1;
    if (nbEnfants>=3) nbParts+=0.5M;
    // calcul revenu imposable & Quotient familial
    decimal revenu=0.72M*salaire;
    decimal QF=revenu/nbParts;
    // calcul de l'impôt
    limites[limites.Length-1]=QF+1;
    int i=0;
    while(QF>limites[i]) i++;
    // retour résultat
    return (long)(revenu*coeffR[i]-nbParts*coeffN[i]);
  }//calculer
}//classe
```

Dans le service web, on ne peut utiliser qu'un constructeur sans paramètres. Aussi le constructeur de la classe va-t-il devenir le suivant :

```
using System.Configuration;

// constructeur
  public impôt(){
    // initialise les trois tableaux limites, coeffR, coeffN à partir
    // du contenu de la table Timpots de la base ODBC DSNimpots
    // colLimites, colCoeffR, colCoeffN sont les trois colonnes de cette table
    // peut lancer une exception

    // on récupère les paramètres de configuration du service
    string DSNimpots=ConfigurationSettings.AppSettings["DSN"];
    string Timpots=ConfigurationSettings.AppSettings["TABLE"];
    string colLimites=ConfigurationSettings.AppSettings["COL_LIMITES"];
    string colCoeffR=ConfigurationSettings.AppSettings["COL_COEFFR"];
    string colCoeffN=ConfigurationSettings.AppSettings["COL_COEFFN"];

    // on exploite la base de données
    string connectString="DSN="+DSNimpots+";"; // chaîne de connexion à la base
....
```

Les cinq paramètres du constructeur de la classe précédente sont maintenant lus dans le fichier *web.config* du service. Le code du fichier source *impots.asmx* est le suivant. Il reprend la majeure partie du code précédent. Nous nous sommes contentés d'encadrer les portions de code propres au service web :

```
<%@ WebService Language=C# class=impôt %>

// création d'un servie web impots
using System;
using System.Data;
using Microsoft.Data.Odbc;
using System.Collections;
using System.Configuration;
using System.Web.Services;

[WebService(Namespace="st.istia.univ-angers.fr")]
public class impôt : WebService{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private decimal[] limites, coeffR, coeffN;
  bool OK=false;
```

```
  string errMessage="";

  // constructeur
  public impôt(){
    // initialise les trois tableaux limites, coeffR, coeffN à partir
    // du contenu de la table Timpots de la base ODBC DSNimpots
    // colLimites, colCoeffR, colCoeffN sont les trois colonnes de cette table
    // peut lancer une exception

    // on récupère les paramètres de configuration du service
    string DSNimpots=ConfigurationSettings.AppSettings["DSN"];
    string Timpots=ConfigurationSettings.AppSettings["TABLE"];
    string colLimites=ConfigurationSettings.AppSettings["COL_LIMITES"];
    string colCoeffR=ConfigurationSettings.AppSettings["COL_COEFFR"];
    string colCoeffN=ConfigurationSettings.AppSettings["COL_COEFFN"];

    // on exploite la base de données
    string connectString="DSN="+DSNimpots+";"; // chaîne de connexion à la base
...

    // on tente d'accéder à la base de données
    try{
      impotsConn = new OdbcConnection(connectString);
      impotsConn.Open();
...

      // c'est bon
      OK=true;
      errMessage="";
    }catch(Exception ex){
      // erreur
      OK=false;
      errMessage+="["+ex.Message+"]";
    }//catch
  }//constructeur

  // calcul de l'impôt
  [WebMethod]
  public long calculer(bool marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
...
  }//calculer

  // id
  [WebMethod]
  public string id(){
    // pour voir si tout est OK
    return "["+OK+","+errMessage+"]";
  }//id

}//classe
```

Expliquons les quelques modifications faites à la classe *impôt* en-dehors de celles nécessaires pour en faire un service web :

- la lecture de la base de données dans le constructeur peut échouer. Aussi avons-nous ajouté deux attributs à notre classe et une méthode :
  - le booléen *OK* est à *vrai* si la base a pu être lue, à *faux* sinon
  - la chaîne *errMessage* contient un message d'erreur si la base de données n'a pu être lue.
  - la méthode *id* sans paramètres permet d'obtenir la valeur ces deux attributs.
- pour gérer l'erreur éventuelle d'accès à la base de données, la partie du code du constructeur concernée par cet accès a été entourée d'un *try-catch*.

Le fichier *web.config* de configuration du service est le suivant :

```
<configuration>
  <appSettings>
    <add key="DSN" value="mysql-impots" />
    <add key="TABLE" value="timpots" />
    <add key="COL_LIMITES" value="limites" />
    <add key="COL_COEFFR" value="coeffr" />
    <add key="COL_COEFFN" value="coeffn" />
  </appSettings>
</configuration>
```

Lors d'un premier essai de chargement du service *impots*, le compilateur a déclaré qu'il ne trouvait pas l'espace de noms *Microsoft.Data.Odbc* utilisé dans la directive :

```
using Microsoft.Data.Odbc;
```

Après consultation de la documentation

- une directive de compilation a été ajoutée dans *web.config* pour indiquer qu'il fallait utiliser l'assembly *Microsoft.Data.odbc*

- une copie du fichier *microsoft.data.odbc.dll* a été placée dans le dossier *c:\inetpub\wwwroot\bin* qui est systématiquement exploré par le compilateur d'un service web lorsqu'il recherche un "assembly".

D'autres solutions semblent possibles mais n'ont pas été crusées ici. Le fichier de configuration est donc devenu :

```
<configuration>
  <appSettings>
    <add key="DSN" value="mysql-impots" />
    <add key="TABLE" value="timpots" />
    <add key="COL_LIMITES" value="limites" />
    <add key="COL_COEFFR" value="coeffr" />
    <add key="COL_COEFFN" value="coeffn" />
  </appSettings>
  <system.web>
    <compilation>
      <assemblies>
        <add assembly="Microsoft.Data.Odbc" />
      </assemblies>
    </compilation>
  </system.web>
</configuration>
```

Le contenu du dossier *c:\inetpub\wwwroot\bin* :

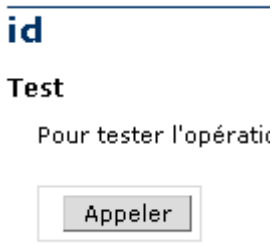
```
E:\data\serge\MSNET\c#\adonet\7>dir c:\inetpub\wwwroot\bin
30/01/2002  02:02              327 680 Microsoft.Data.Odbc.dll
```

Le service et son fichier de configuration ont été placés dans *c:\inetpub\wwwroot\st\impots* :

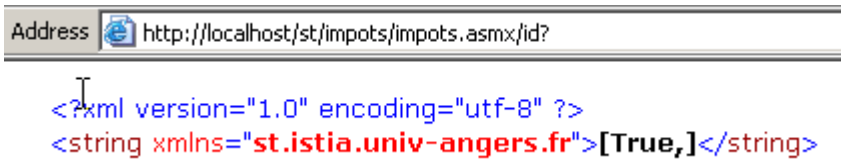
```
E:\data\serge\MSNET\c#\adonet\7>dir c:\inetpub\wwwroot\st\impots
16/05/2002  16:43                3 678 impots.asmx
16/05/2002  16:35                  425 web.config
```

La page du service est alors la suivante :

Si on suit le lien *id* :

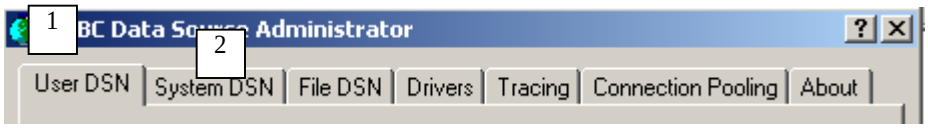


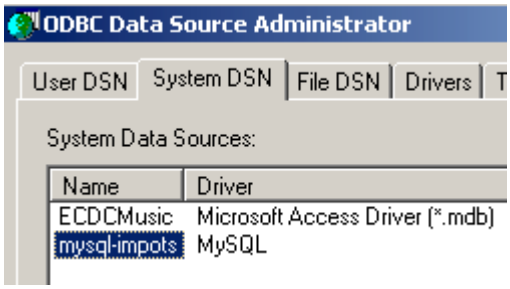
Si on utilise le bouton *Appeler* :



Le résultat précédent affiche les valeurs des attributs *OK* (true) et *errMessage* ("). Dans cet exemple, la base a été chargée correctement. Ca n'a pas toujours été le cas et c'est pourquoi nous avons ajouté la méthode *id* pour avoir accès au message d'erreur.

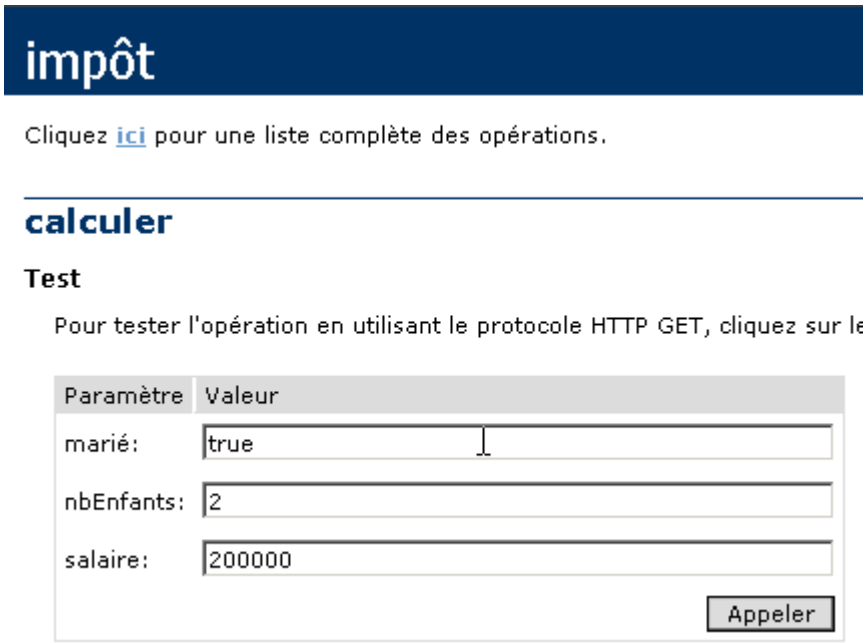
L'erreur était que le nom DSN de la base avait été définie comme **DSN utilisateur** alors qu'il fallait le définir comme **DSN système**. Cette distinction se fait dans le gestionnaire de sources ODBC 32 bits :



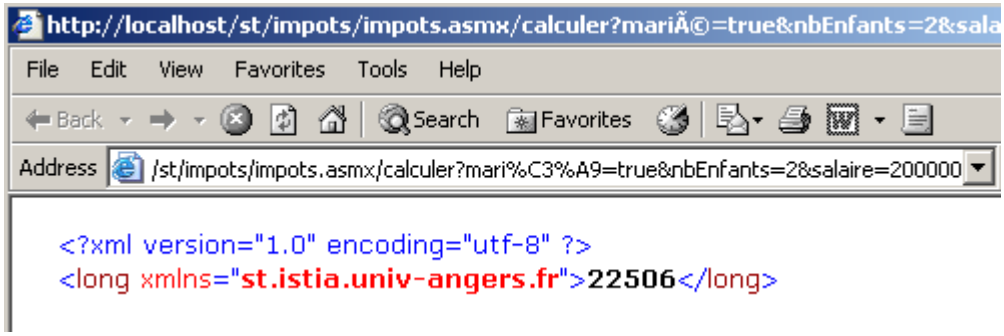


Revenons à la page du service :

Suivons le lien *calculer* :



Nous définissons les paramètres de l'appel et nous exécutons celui-ci :



Le résultat est correct.

9.9.2 Générer le proxy du service impots

Maintenant que nous avons un service web *impots* opérationnel, nous pouvons générer sa classe proxy. On rappelle que celle-ci sera utilisée par des applications clientes pour atteindre le service web *impots* de façon transparente. On utilise d'abord l'utilitaire *wsdl* pour générer le fichier source de la classe proxy puis celui-ci est compilé dans un une dll.

```
E:\data\serge\MSNET\c#\impots\webservice>wsdl http://localhost/st/impots/impots.asmx

Écriture du fichier 'E:\data\serge\MSNET\c#\impots\webservice\impôt.cs'.

E:\data\serge\MSNET\c#\impots\webservice>dir
16/05/2002  17:19                3 138 impôt.cs

E:\data\serge\MSNET\c#\impots\webservice>csc /t:library impôt.cs

E:\data\serge\MSNET\c#\impots\webservice>dir
16/05/2002  17:19                3 138 impôt.cs
16/05/2002  17:20                5 120 impôt.dll
```

9.9.3 Utiliser le proxy avec un client

Dans le chapitre sur les bases de données nous avions créé une application console permettant le calcul de l'impôt :

```
E:\data\serge\MSNET\c#\impots\6>dir
07/05/2002  18:01                3 235 impots.cs
07/05/2002  18:01                5 120 impots.dll
07/05/2002  17:50                2 854 test.cs
07/05/2002  18:01                5 632 test.exe

E:\data\serge\MSNET\c#\impots\6>test
pg DSNimpots tabImpots colLimites colCoeffR colCoeffN

E:\data\serge\MSNET\c#\impots\6>test mysql-impots timpots limites coeffr coeffn
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 200000
impôt=22506 F
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :
```

Le programme test utilisait alors la classe *impôt* classique celle contenue dans le fichier *impots.dll*. Le code du programme *test.cs* était le suivant :

```
using System;

  class test
  {
    public static void Main(string[] arguments)
    {
      // programme interactif de calcul d'impôt
      // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
      // le programme affiche alors l'impôt à payer

      const string syntaxe1="pg DSNimpots tabImpots colLimites colCoeffR colCoeffN";
      const string syntaxe2="syntaxe : marié nbEnfants salaire\n"
             +"marié : o pour marié, n pour non marié\n"
             +"nbEnfants : nombre d'enfants\n"
             +"salaire : salaire annuel en F";

      // vérification des paramètres du programme
      if(arguments.Length!=5){
```

```
        // msg d'erreur
        Console.Error.WriteLine(syntaxe1);
        // fin
        Environment.Exit(1);
      }//if
      // on récupère les arguments
      string DSNimpots=arguments[0];
      string tabImpots=arguments[1];
      string colLimites=arguments[2];
      string colCoeffR=arguments[3];
      string colCoeffN=arguments[4];

      // création d'un objet impôt
      impôt objImpôt=null;
      try{
        objImpôt=new impôt(DSNimpots,tabImpots,colLimites,colCoeffR,colCoeffN);
      }catch (Exception ex){
        Console.Error.WriteLine("L'erreur suivante s'est produite : " + ex.Message);
        Environment.Exit(2);
      }//try-catch

      // boucle infinie
      while(true){
        // on demande les paramètres du calcul de l'impôt
        Console.Out.Write("Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour
arrêter :");
        string paramètres=Console.In.ReadLine().Trim();
        // qq chose à faire ?
        if(paramètres==null || paramètres=="") break;
        // vérification du nombre d'arguments dans la ligne saisie
        string[] args=paramètres.Split(null);
        int nbParamètres=args.Length;
        if (nbParamètres!=3){
          Console.Error.WriteLine(syntaxe2);
          continue;
        }//if
        // vérification de la validité des paramètres
        // marié
        string marié=args[0].ToLower();
        if (marié!="o" && marié !="n"){
          Console.Error.WriteLine(syntaxe2+"\nArgument marié incorrect : tapez o ou n");
          continue;
        }//if
        // nbEnfants
        int nbEnfants=0;
        try{
          nbEnfants=int.Parse(args[1]);
          if(nbEnfants<0) throw new Exception();
        }catch (Exception){
          Console.Error.WriteLine(syntaxe2+"\nArgument nbEnfants incorrect : tapez un entier positif ou
nul");
          continue;
        }//if
        // salaire
        int salaire=0;
        try{
          salaire=int.Parse(args[2]);
          if(salaire<0) throw new Exception();
        }catch (Exception){
          Console.Error.WriteLine(syntaxe2+"\nArgument salaire incorrect : tapez un entier positif ou
nul");
          continue;
        }//if
        // les paramètres sont corrects - on calcule l'impôt
        Console.Out.WriteLine("impôt="+objImpôt.calculer(marié=="o",nbEnfants,salaire)+" F");
        // contribuable suivant
      }//while
    }//Main
  }//classe
```

Nous reprenons le même programme pour lui faire utiliser maintenant le service web *impots* au travers de la classe proxy *impôt* créée précédemment. Nous sommes obligés de modifier quelque peu le code :

- alors que la classe *impôt* d'origine avait un constructeur à cinq arguments, la classe proxy *impôt* a un constructeur sans paramètres. Les cinq paramètres, nous l'avons vu, sont maintenant fixés dans le fichier de configuration du service web.
- il n'y a donc plus besoin de passer ces cinq paramètres en arguments au programme test

Le nouveau code est le suivant :

```
using System;

  class test
  {
    public static void Main()
    {
      // programme interactif de calcul d'impôt
      // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
      // le programme affiche alors l'impôt à payer

      const string syntaxe2="syntaxe : marié nbEnfants salaire\n"
```

```
            +"marié : o pour marié, n pour non marié\n"
            +"nbEnfants : nombre d'enfants\n"
            +"salaire : salaire annuel en F";

      // création d'un objet impôt
      impôt objImpôt=null;
      try{
        objImpôt=new impôt();
      }catch (Exception ex){
        Console.Error.WriteLine("L'erreur suivante s'est produite : " + ex.Message);
        Environment.Exit(2);
      }//try-catch

      // boucle infinie
      while(true){
        // on demande les paramètres du calcul de l'impôt
...
      }//while
    }//Main
  }//classe
```

Nous avons le proxy *impôt.dll* et le source *test.cs* dans le même dossier.

```
E:\data\serge\MSNET\c#\impots\webservice>dir

16/05/2002  17:20                5 120 impôt.dll
16/05/2002  17:53                2 324 test.cs
```

Nous compilons le source *test.cs* :

```
E:\data\serge\MSNET\c#\impots\webservice>csc /r:impôt.dll test.cs

E:\data\serge\MSNET\c#\impots\webservice>dir
16/05/2002  17:20                5 120 impôt.dll
16/05/2002  17:53                2 324 test.cs
16/05/2002  18:11                5 632 test.exe
```

puis l'exécutons :

```
E:\data\serge\MSNET\c#\impots\webservice>test
Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 200000
impôt=22506 F
```

10. A suivre…

Il resterait des thèmes importants à couvrir. En voici trois :

1. les objets **DataSet** qui permettent de gérer une base de données en mémoire, de l'exporter ou l'importer au format XML
2. une étude de XML avec les classes .NET permettant de gérer les documents XML
3. la programmation Web avec les pages et contrôles ASP.NET

A lui seul, le point 3 mérite un polycopié. Les points 1 et 2 devraient être ajoutés progressivement à ce présent document.