Apprentissage du langage C# version 3.0 avec le framework .NET 3.5

Serge Tahé, mai 2008

Ce site a été créé avec le convertisseur [Word ou ODT - > HTML] créé par l'IA Gemini 3 en janvier 2026.

1 Introduction

Le PDF de ce document est disponible |ICI|.

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

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

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

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

Ce document n'est pas un cours exhaustif. Par exemple, LINQ n'y est pas abordé. Il est destiné à des personnes connaissant déjà la programmation et qui veulent découvrir les bases de C#. Il est une révision du document originel paru en 2002.

Plusieurs livres m'ont aidé à écrire ce cours :

Pour la version 2002 :
- Professional C# programming, Editions Wrox
- C# et .NET, Gérard Leblanc, Editions Eyrolles

A l'époque j'avais trouvé excellents ces deux ouvrages. Depuis Gérard Leblanc a publié des versions mises à jour dont la suivante :

- C# et .NET 2005, Gérard Leblanc, Editions Eyrolles

Pour la révision de 2008, j'ai utilisé les sources suivantes :
- le document initial de 2002. Ce document issu d'un copier / coller de mon cours Java comportait à la fois des erreurs de typographie et des erreurs plus sérieuses telles que dire que les types primitifs comme *System.Int32* étaient des classes alors que ce sont des structures. Mea culpa...
- la documentation MSDN de Visual Studio Express 2008
- le livre C# 3.0 in a Nutshell de Joseph et Ben Albahari aux éditions O'Reilly, l'un des meilleurs livres de programmation qu'il m'ait été donné de lire.

Les codes source des exemples de ce document sont disponibles sur le site [http://tahe.developpez.com/dotnet/csharp/].

Le PDF du document est disponible |ICI|.

Serge Tahé, mai 2008

Depuis mai 2008, C# a bien sûr évolué. Ce document reste cependant toujours d'actualité pour l'apprentissage de ce langage. En complément de ce cours, on trouvera une présentation de l'ORM (Object Relational Mapper) **Entity Framework** dans l'article **" Introduction à l'ORM Entity Framework 5 Code First "** à l'URL [http://tahe.developpez.com/dotnet/ef5cf/].

2 Installation de Visual C# 2008

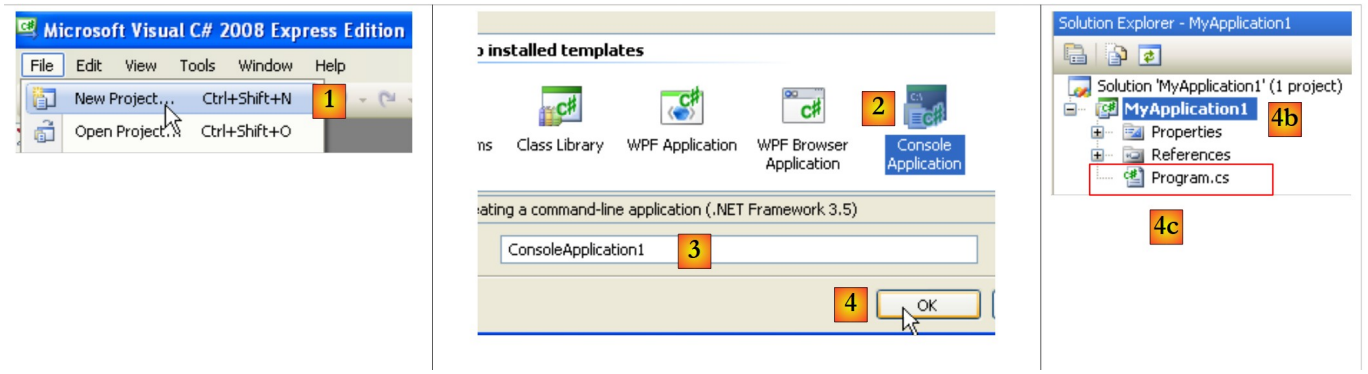
Fin janvier 2008, les versions Express de Visual Studio 2008 sont téléchargeables [2] à l'adresse suivante [1] : [http://msdn2.microsoft.com/fr-fr/express/future/default(en-us).aspx] :

- [1] : l'adresse de téléchargement
- [2] : l'onglet des téléchargements
- [3] : télécharger C# 2008

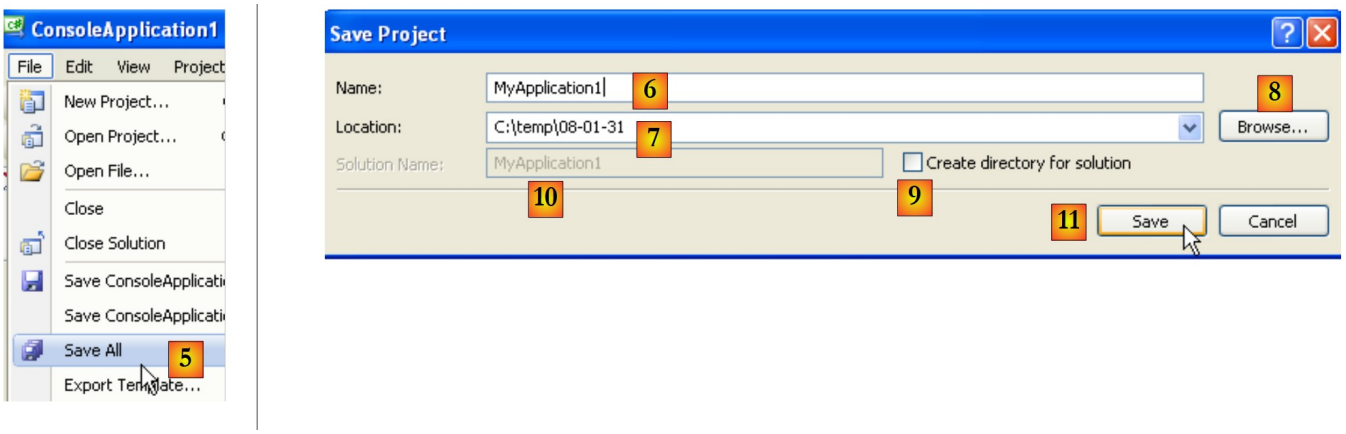
L'installation de C# 2008 entraînera celle d'autres éléments :

- le framework .NET 3.5
- le SGBD SQL Server Compact 3.5
- la documentation MSDN

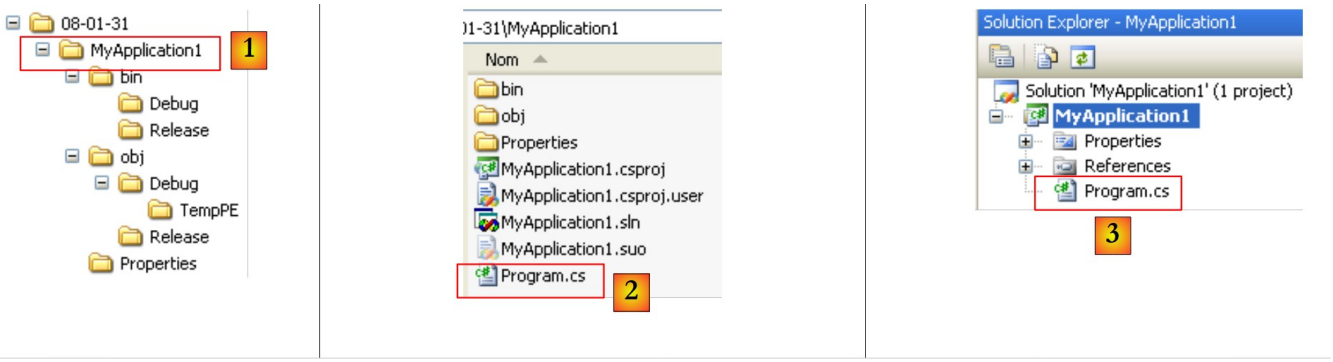
Pour créer un premier programme avec C# 2008, on pourra procéder comme suit, après avoir lancé C# :



- [1] : prendre l'option File / New Project
- [2] : choisir une application de type Console
- [3] : donner un nom au projet - il va être changé ci-dessous
- [4] : valider
- [4b] : le projet créé
- [4c] : *Program.cs* est le programme C# généré par défaut dans le projet.



- la 1ère étape n'a pas demandé où placer le projet. Si on ne fait rien, il sera sauvegardé à un endroit par défaut qui ne nous conviendra probablement pas. L'option [5] permet de sauvegarder le projet dans un dossier précis.
- on peut donner un nouveau nom au projet dans [6], préciser dans [7] son dossier. Pour cela on peut utiliser [8]. Ici, le projet sera au final dans le dossier [C:\temp\08-01-31\MyApplication1].
- en cochant [9], on peut créer un dossier pour la solution nommée dans [10]. Si *Solution1* est le nom de la solution :
- un dossier [C:\temp\08-01-31\Solution1] sera créé pour la solution *Solution1*
- un dossier [C:\temp\08-01-31\Solution1\MyApplication1] sera créé pour le projet *MyApplication1*. Cette solution convient bien aux solutions composées de plusieurs projets. Chaque projet aura un sous-dossier dans le dossier de la solution.



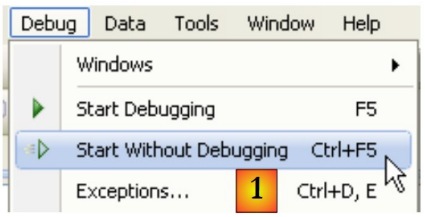
- en [1] : le dossier *windows* du projet *MyApplication1*
- en [2] : son contenu
- en [3] : le projet dans l'explorateur de projets de Visual Studio

Modifions le code du fichier [Program.cs] [3] comme suit :

```
1.  using System;
2.
3.  namespace ConsoleApplication1 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Console.WriteLine("1er essai avec C# 2008");
7.      }
8.   }
9.  }
```

- ligne 3 : l'espace de noms de la classe définie ligne 4. Ainsi le nom complet de la classe, définie ligne 4, est-il ici *ConsoleApplication1.Program*.
- lignes 5-7 : la méthode statique *Main* qui est exécutée lorsqu'on demande l'exécution d'une classe
- ligne 6 : un affichage écran

Le programme peut être exécuté de la façon suivante :

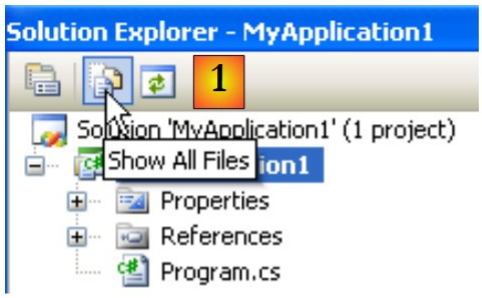


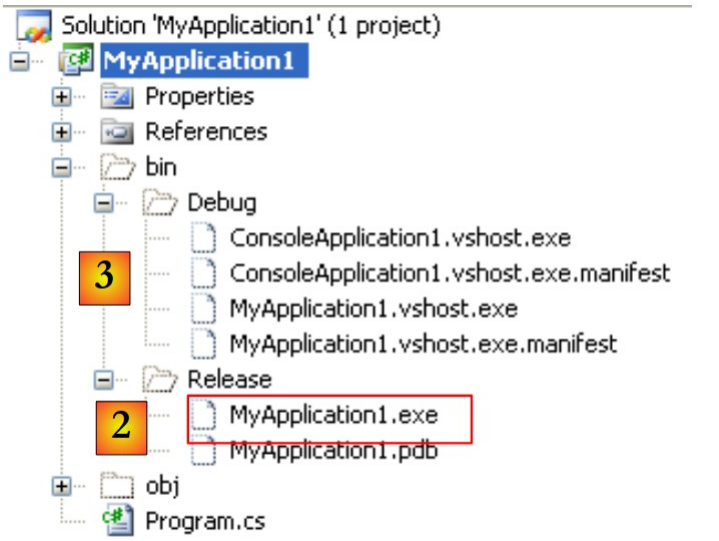

```
1.  1er essai avec C# 2008
2.  Appuyez sur une touche pour continuer...
```

2

- [Ctrl-F5] pour exécuter le projet, en [1]
- en [2], l'affichage console obtenu.

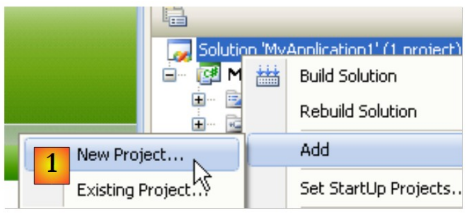
L'exécution a rajouté des fichiers au projet :

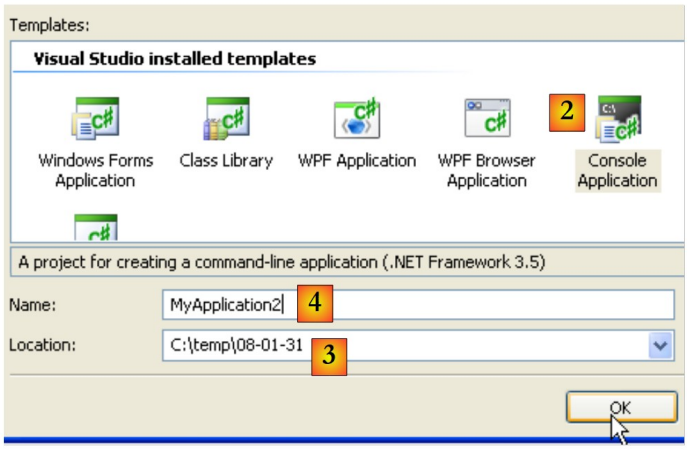




- en [1], on fait afficher tous les fichiers du projet
- en [2] : le dossier [Release] contient l'exécutable [MyApplication1.exe] du projet.
- en [3] : le dossier [Debug] qui contiendrait lui aussi un exécutable [MyApplication1.exe] du projet si on avait exécuté le projet en mode [Debug] (touche F5 au lieu de Ctrl-F5). Ce n'est pas le même exécutable que celui obtenu en mode [Release]. Il contient des informations complémentaires permettant au processus de débogage d'avoir lieu.

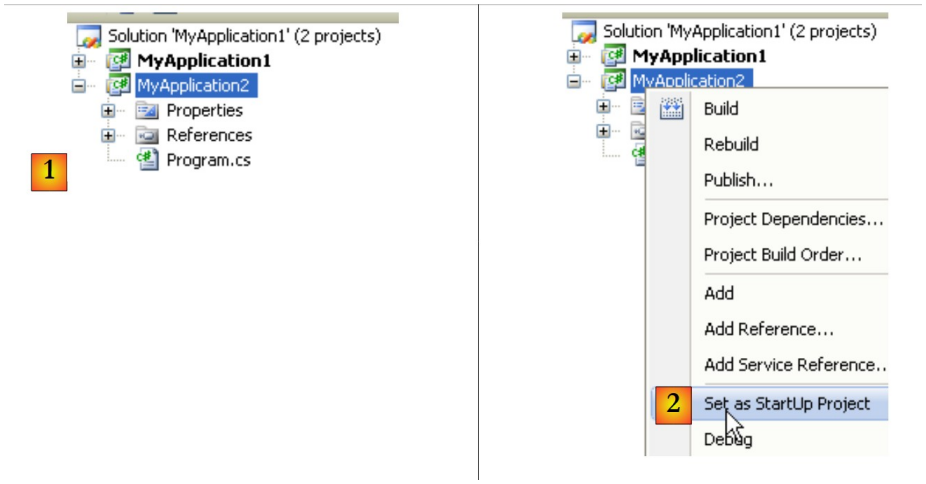
On peut rajouter un nouveau projet à la solution courante :





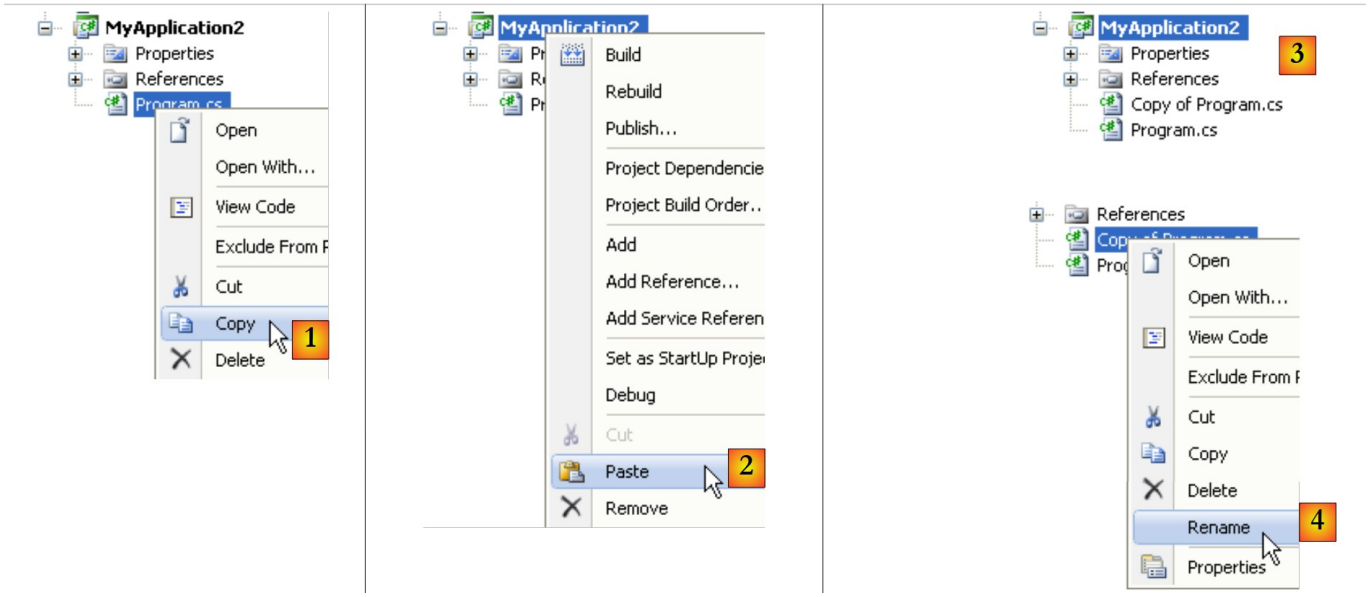
- [1] : clic droit sur la solution (pas le projet) / Add / New Project
- [2] : choix d'un type d'application
- [3] : le dossier proposé par défaut est celui contenant le dossier du projet déjà existant [MyApplication1]
- [4] : donner un nom au nouveau projet

La solution a alors deux projets :

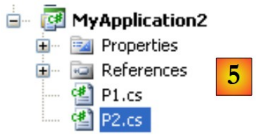


- [1] : le nouveau projet
- [2] : lorsque la solution est exécutée par (F5 ou Ctrl-F5), l'un des projets est exécuté. C'est celui qu'on désigne par [2].

Un projet peut avoir plusieurs classes exécutables (contenant une méthode Main). Dans ce cas, on doit préciser la classe à exécuter lorsque le projet est exécuté :



- [1, 2] : on copie / colle le fichier [Program.cs]
- [3] : le résultat du copier / coller
- [4,5] : on renomme les deux fichiers



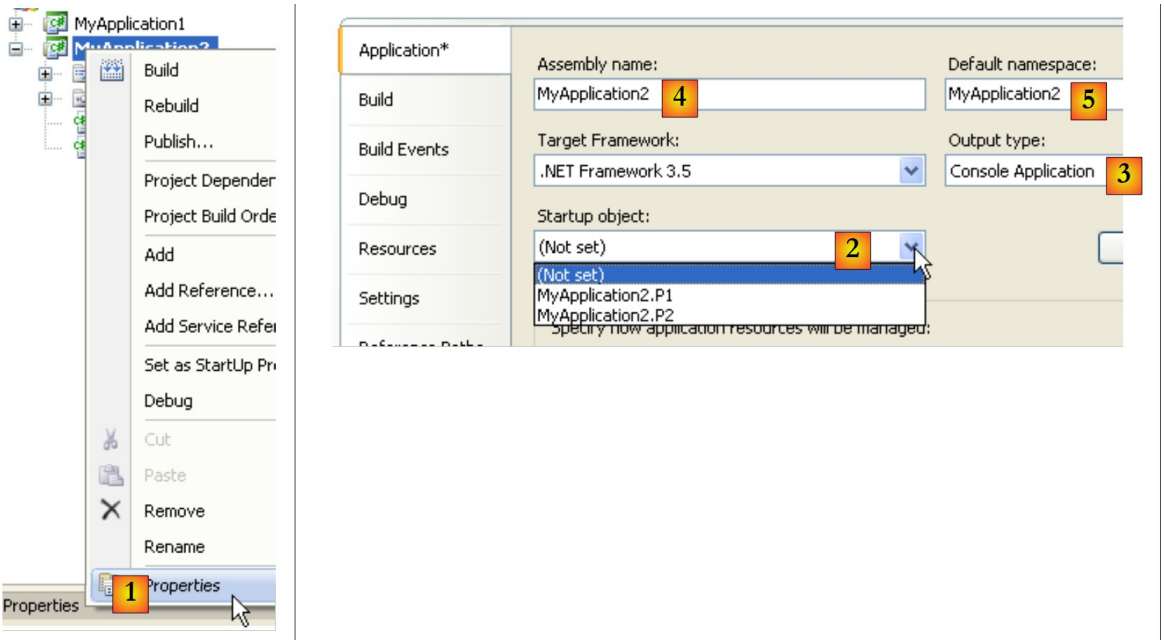
La classe P1 (ligne 4) :

```
1.  using System;
2.
3.  namespace MyApplication2 {
4.   class P1 {
5.      static void Main(string[] args) {
6.      }
7.   }
8.  }
```

La classe P2 (ligne 4) :

```
1.  using System;
2.
3.  namespace MyApplication2 {
4.   class P2 {
5.      static void Main(string[] args) {
6.      }
7.   }
8.  }
```

Le projet [MyApplication2] a maintenant deux classes ayant une méthode statique *Main*. Il faut indiquer au projet laquelle exécuter :



- en [1] : les propriétés du projet [MyApplication2]
- en [2] : le choix de la classe à exécuter lorsque le projet est exécuté (F5 ou Ctrl-F5)
- en [3] : le type d'exécutable produit - ici une application Console produira un fichier .exe.
- en [4] : le nom de l'exécutable produit (sans le .exe)
- en [5] : l'espace de noms par défaut. C'est celui qui sera généré dans le code de chaque nouvelle classe ajoutée au projet. Il peut alors être changé directement dans le code, si besoin est.

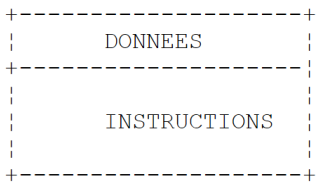
3 Les bases du langage C#

3.1 Introduction

Nous traitons C# d'abord comme un langage de programmation classique. Nous aborderons les classes 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 :



3.2 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

3.2.1 Les types de données prédéfinis

Type C#	Type .NET	Donnée représentée	Suffixe des valeurs littérales	Codage	Domaine de valeurs
`char`	Char (S)	caractère		2 octets	caractère Unicode (UTF-16)
`string`	String (C)	chaîne de caractères			référence sur une séquence de caractères Unicode
`int`	Int32 (S)	nombre entier		4 octets	$[-2^{31}, 2^{31}-1]$ [–2147483648, 2147483647]
`uint`	UInt32 (S)	..	U	4 octets	$[0, 2^{32}-1]$ [0, 4294967295]
`long`	Int64 (S)	..	L	8 octets	$[-2^{63}, 2^{63}-1]$ [–9223372036854775808, 9223372036854775807]
`ulong`	UInt64 (S)	..	UL	8 octets	$[0, 2^{64}-1]$ [0, 18446744073709551615]
`sbyte`		..		1 octet	$[-2^{7}, 2^{7}-1]$ [-128,+127]
`byte`	Byte (S)	..		1 octet	$[0, 2^{8}-1]$ [0,255]
`short`	Int16 (S)	..		2 octets	$[-2^{15}, 2^{15}-1]$ [-32768, 32767]
`ushort`	UInt16 (S)	..		2 octets	$[0, 2^{16}-1]$ [0,65535]
`float`	Single (S)	nombre réel	F	4 octets	$[1.5\ 10^{-45}, 3.4\ 10^{+38}]$ en valeur absolue
`double`	Double (S)	..	D	8 octets	$[-1.7\ 10^{+308}, 1.7\ 10^{+308}]$ en valeur absolue
`decimal`	Decimal (S)	nombre décimal	M	16 octets	$[1.0\ 10^{-28}, 7.9\ 10^{+28}]$ en valeur absolue avec 28 chiffres significatifs
`bool`	Boolean (S)	..		1 octet	true, false
`object`	Object (C)	référence d'objet			référence d'objet

Ci-dessus, on a mis en face des types C#, leur type .NET équivalent avec le commentaire (S) si ce type est une structure et (C) si le type est une classe. On découvre qu'il y a deux types possibles pour un entier sur 32 bits : *int* et *Int32*. Le type *int* est un type C#. *Int32* est une structure appartenant à l'espace de noms *System*. Son nom complet est ainsi *System.Int32*. Le type *int* est un alias C# qui désigne la **structure** .NET *System.Int32*. De même, le type C# *string* est un alias pour le type .NET *System.String*. *System.String* est une **classe** et non une structure. Les deux notions sont proches avec cependant la différence fondamentale suivante :

- une variable de type *Structure* se manipule via sa **valeur**
- une variable de type *Classe* se manipule via son **adresse** (**référence** en langage objet).

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

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

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

Voici un programme illustrant ces différents points :

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

- ligne 7 : déclaration d'un entier *ent*
- ligne 19 : le type d'une variable v peut être obtenue par *v.GetType().FullName*. La taille d'une structure S peut être obtenue par *sizeof(S)*. L'instruction *Console.WriteLine("... {0} ... {1} ...",param0, param1, ...)* écrit à l'écran, le texte qui est son premier paramètre, en remplaçant chaque notation *{i}* par la valeur de l'expression *parami*.
- ligne 22 : le type *string* désignant une classe et non une structure, on ne peut utiliser l'opérateur *sizeof*.

Voici le résultat de l'exécution :

```
1.  Type de ent[2] : [System.Int32,4]
2.  Type de fl[10,5]: [System.Single,4]
3.  Type de d[-4,6] : [System.Double,8]
4.  Type de s[essai] : [System.String]
5.  Type de ui[5] : [System.UInt32,4]
6.  Type de l[1000] : [System.Int64,8]
7.  Type de ul[1001] : [System.UInt64,8]
8.  Type de b[5] : [System.Byte,1]
```

```
9.  Type de sh[-4] : [System.Int16,2]
10. Type de ush[10] : [System.UInt16,2]
11. Type de dec[10,67] : [System.Decimal,16]
12. Type de b[True] : [System.Boolean,1]
```

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

3.2.2 Notation des données littérales

entier *int* (32 bits)	145, -7, 0xFF (hexadécimal)
entier *long* (64 bits) - suffixe L	100000L
réel *double*	134.789, -45E-18 (-45 10^{-18})
réel *float* (suffixe F)	134.789**F**, -45E-18**F** (-45 10^{-18})
réel *decimal* (suffixe M)	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 interprétée comme un seul caractère \. On pourrait écrire aussi **@"\n"** pour avoir le même résultat. La syntaxe *@"texte"* demande que *texte* soit pris exactement comme il est écrit. On appelle parfois cela une chaîne **verbatim**.

3.2.3 Déclaration des données

3.2.3.1 Rôle des déclarations

Un programme manipule des données caractérisées par un nom et un type. Ces données sont stockées en mémoire. Au moment de la traduction du programme, le compilateur affecte à chaque donnée un emplacement en mémoire caractérisé par une adresse et une taille. Il le fait en s'aidant des déclarations faites par le programmeur.
Par ailleurs celles-ci permettent au compilateur de détecter des erreurs de programmation. Ainsi l'opération

*x=x*2;*

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

3.2.3.2 Déclaration des constantes

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

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

Par exemple :

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

Pourquoi déclarer des constantes ?

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

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

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

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

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

3.2.3.3 Déclaration des variables

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

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

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

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

On peut également ne pas préciser le type exact d'une variable en utilisant le mot clé **var** en lieu et place de *Identificateur_de_type* :

```
var variable1=valeur1,variable2=valeur2,...;
```

Le mot clé **var** ne veut pas dire que les variables n'ont pas un type précis. La variable *variablei* a le type de la donnée *valeuri* qui lui est affectée. L'initialisation est ici **obligatoire** afin que le compilateur puisse en déduire le type de la variable.

Voici un exemple :

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P00 {
5.      static void Main(string[] args) {
6.         int i=2;
7.         Console.WriteLine("Type de int i=2 : {0},{1}",i.GetType().Name,i.GetType().FullName);
8.         var j = 3;
9.         Console.WriteLine("Type de var j=3 : {0},{1}", j.GetType().Name, j.GetType().FullName);
10.        var aujourdhui = DateTime.Now;
11.        Console.WriteLine("Type de var aujourdhui : {0},{1}", aujourdhui.GetType().Name,
    aujourdhui.GetType().FullName);
12.     }
13.  }
14. }
```

- ligne 6 : une donnée typée explicitement
- ligne 7 : *(donnée).GetType().Name* est le nom court de (donnée), *(donnée).GetType().FullName* est le nom complet de (donnée)
- ligne 8 : une donnée typée implicitement. Parce que 3 est de type *int*, j sera de type *int*.
- ligne 10 : une donnée typée implicitement. Parce que *DateTime.Now* est de type *DateTime*, *aujourdhui* sera de type *DateTime*.

A l'exécution, on obtient le résultat suivant :

```
1.  Type de int i=2 : Int32,System.Int32
2.  Type de var j=3 : Int32,System.Int32
3.  Type de var aujourdhui : DateTime,System.DateTime
```

Une variable typée implicitement par le mot clé *var* ne peut pas ensuite changer de type. Ainsi, on ne pourrait écrire après la ligne 10 du code, la ligne :

```
        var aujourdhui = "aujourd'hui";
```

Nous verrons ultérieurement qu'il est possible de déclarer un type "à la volée" dans une expression. C'est alors un type **anonyme**, un type auquel l'utilisateur n'a pas donné de nom. C'est le compilateur qui donnera un nom à ce nouveau type. Si une donnée de type **anonyme** doit être affectée à une variable, la seule façon de déclarer celle-ci est d'utiliser le mot clé **var**.

3.2.4 Les conversions entre nombres et chaînes de caractères

`nombre -> chaîne`	*nombre.ToString()*
`chaine -> int`	*int.Parse(chaine) ou System.Int32.Parse*
`chaîne -> long`	*long.Parse(chaine) ou System.Int64.Parse*
`chaîne -> double`	*double.Parse(chaîne) ou System.Double.Parse(chaîne)*
`chaîne -> float`	*float.Parse(chaîne) ou System.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**. 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 quelques 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 où S est de type *string*.

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P01 {
5.      static void Main(string[] args) {
6.
7.         // données
8.         const int i = 10;
9.         const long l = 100000;
10.        const float f = 45.78F;
11.        double d = -14.98;
12.
13.        // nombre --> chaîne
14.        affiche(i.ToString());
15.        affiche(l.ToString());
16.        affiche(f.ToString());
17.        affiche(d.ToString());
18.
19.        //boolean --> chaîne
20.        const bool b = false;
21.        affiche(b.ToString());
22.
23.        // chaîne --> int
24.        int i1;
25.        i1 = int.Parse("10");
26.        affiche(i1.ToString());
27.        try {
28.           i1 = int.Parse("10.67");
29.           affiche(i1.ToString());
30.        } catch (Exception e) {
31.           affiche("Erreur : " + e.Message);
32.        }
33.
34.        // chaîne --> long
35.        long l1;
36.        l1 = long.Parse("100");
37.        affiche(l1.ToString());
38.        try {
39.           l1 = long.Parse("10.675");
40.           affiche(l1.ToString());
41.        } catch (Exception e) {
42.           affiche("Erreur : " + e.Message);
43.        }
44.
45.        // chaîne --> double
46.        double d1;
47.        d1 = double.Parse("100,87");
48.        affiche(d1.ToString());
49.        try {
50.           d1 = double.Parse("abcd");
51.           affiche(d1.ToString());
52.        } catch (Exception e) {
53.           affiche("Erreur : " + e.Message);
54.        }
55.
56.        // chaîne --> float
57.        float f1;
58.        f1 = float.Parse("100,87");
59.        affiche(f1.ToString());
60.        try {
61.           d1 = float.Parse("abcd");
62.           affiche(f1.ToString());
```

```
63.         } catch (Exception e) {
64.            affiche("Erreur : " + e.Message);
65.         }
66.
67.     }// fin main
68.
69.     public static void affiche(string S) {
70.        Console.Out.WriteLine("S={0}",S);
71.     }
72.  }// fin classe
73. }
```

Lignes 30-32, on gère l'éventuelle exception qui peut se produire. *e.Message* est le message d'erreur lié à l'exception *e*.

Les résultats obtenus sont les suivants :

```
1.  S=10
2.  S=100000
3.  S=45,78
4.  S=-14,98
5.  S=False
6.  S=10
7.  S=Erreur : Input string was not in a correct format.
8.  S=100
9.  S=Erreur : Input string was not in a correct format.
10. S=100,87
11. S=Erreur : Input string was not in a correct format.
12. S=100,87
13. S=Erreur : 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");
```

3.2.5 Les tableaux de données

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

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

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

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

ou plus simplement :

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

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

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

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

où *n* est le nombre de lignes, *m* le nombre de colonnes. La syntaxe *Tableau[i,j]* désigne l'élément j de la ligne i de t*ableau*. Le tableau à deux dimensions peut lui aussi être initialisé en même temps qu'il est déclaré :

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

ou plus simplement :

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

Le nombre d'éléments dans chacune des dimensions peut être obtenue par la méthode **GetLength(i)** où i=0 représente la dimension correspondant au 1er indice, i=1 la dimension correspondant au 2ième indice, …
Le nombre total de dimensions est obtenu avec la propriété **Rank**, le nombre total d'éléments avec la propriété **Length**.

Un **tableau de tableaux** est déclaré comme suit :

Type[][] tableau=new Type[n][];

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

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

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

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

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

```
12. using System;
13.
14. namespace Chap1 {
15.  // tableaux
16.
17.  using System;
18.
19.  // classe de test
20.  public class P02 {
21.     public static void Main() {
22.        // un tableau à 1 dimension initialisé
23.        int[] entiers = new int[] { 0, 10, 20, 30 };
24.        for (int i = 0; i < entiers.Length; i++) {
25.           Console.Out.WriteLine("entiers[{0}]={1}", i, entiers[i]);
26.        }//for
27.
28.        // un tableau à 2 dimensions initialisé
29.        double[,] réels = new double[,] { { 0.5, 1.7 }, { 8.4, -6 } };
30.        for (int i = 0; i < réels.GetLength(0); i++) {
31.           for (int j = 0; j < réels.GetLength(1); j++) {
32.              Console.Out.WriteLine("réels[{0},{1}]={2}", i, j, réels[i, j]);
33.           }//for j
34.        }//for i
35.
36.        // un tableau de tableaux
37.        string[][] noms = new string[3][];
38.        for (int i = 0; i < noms.Length; i++) {
39.           noms[i] = new string[i + 1];
40.        }//for
41.        // initialisation
42.        for (int i = 0; i < noms.Length; i++) {
43.           for (int j = 0; j < noms[i].Length; j++) {
44.              noms[i][j] = "nom" + i + j;
```

```
45.          }//for j
46.       }//for i
47.       // affichage
48.       for (int i = 0; i < noms.Length; i++) {
49.          for (int j = 0; j < noms[i].Length; j++) {
50.             Console.Out.WriteLine("noms[{0}][{1}]={2}", i, j, noms[i][j]);
51.          }//for j
52.       }//for i
53.    }//Main
54. }//class
55. }//namespace
```

A l'exécution, nous obtenons les résultats suivants :

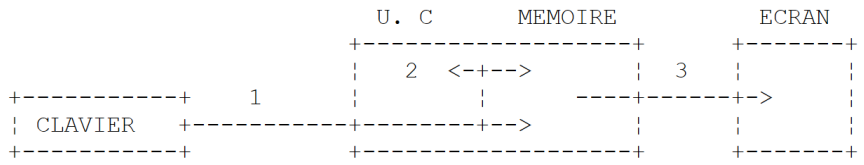
```
1.  entiers[0]=0
2.  entiers[1]=10
3.  entiers[2]=20
4.  entiers[3]=30
5.  réels[0,0]=0,5
6.  réels[0,1]=1,7
7.  réels[1,0]=8,4
8.  réels[1,1]=-6
9.  noms[0][0]=nom00
10. noms[1][0]=nom10
11. noms[1][1]=nom11
12. noms[2][0]=nom20
13. noms[2][1]=nom21
14. noms[2][2]=nom22
```

3.3 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

3.3.1 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. Tous les objets de C# ou .NET ont une méthode *ToString()* qui est utilisée pour faire cette conversion.

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 *TextWriter* :

- *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 à l'écran.

Les flux *Out* et *Error* peuvent être redirigés vers des fichiers texte au moment de l'exécution du programme comme nous le verrons prochainement.

3.3.2 Lecture de données tapées au clavier

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

```
string ligne=Console.In.ReadLine();
```

La classe Console offre une méthode ReadLine associée par défaut au flux *In*. On peut donc écrire écrire :

```
string ligne=Console.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**.

3.3.3 Exemple d'entrées-sorties

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   // classe de test
5.   public class P03 {
6.      public static void Main() {
7.
8.         // écriture sur le flux Out
9.         object obj = new object();
10.        Console.Out.WriteLine(obj);
11.
12.        // écriture sur le flux Error
13.        int i = 10;
14.        Console.Error.WriteLine("i=" + i);
15.
16.        // lecture d'une ligne saisie au clavier
17.        Console.Write("Tapez une ligne : ");
18.        string ligne = Console.ReadLine();
19.        Console.WriteLine("ligne={0}", ligne);
20.     }//fin main
21.  }//fin classe
22. }
```

- ligne 9 : *obj* est une référence d'objet
- ligne 10 : *obj* est écrit à l'écran. Par défaut, c'est la méthode *obj.ToString()* qui est appelée.
- ligne 14 : on peut aussi écrire :

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

  Le 1er paramètre "i={0}" est le format d'affichage, les autres paramètres, les expressions à afficher. Les éléments {n} sont des paramètres "positionnels". A l'exécution, le paramètre {n} est remplacé par la valeur de l'expression n° n.

Le résultat de l'exécution est le suivant :

```
1.  System.Object
2.  i=10
3.  Tapez une ligne : je suis là
4.  ligne=je suis là
```

- ligne 1 : l'affichage produit par la ligne 10 du code. La méthode *obj.ToString()* a affiché le nom du type de la variable *obj* : *System.Object*. Le type *object* est un alias C# du type .NET *System.Object*.

3.3.4 Redirection des E/S

Il existe sous DOS et UNIX trois périphériques standard 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
```

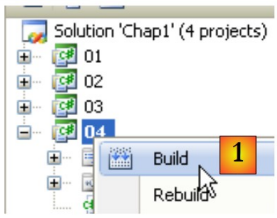
Derrière les arguments *argi* du programme *pg*, on peut rediriger les périphériques d'E/S standard vers des fichiers:

`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 le système d'exploitation qui fixe la nature des périphériques 0,1 et 2. Considérons le programme suivant :

```
1.  using System;
2.
3.  namespace Chap1 {
4.
5.   // redirections
6.   public class P04 {
7.      public static void Main(string[] args) {
8.         // lecture flux In
9.         string data = Console.In.ReadLine();
10.        // écriture flux Out
11.        Console.Out.WriteLine("écriture dans flux Out : " + data);
12.        // écriture flux Error
13.        Console.Error.WriteLine("écriture dans flux Error : " + data);
14.     }//Main
15.  }//classe
16. }
```

Générons l'exécutable de ce code source :




```
1.  C:\data\travail\2007-2008\c# 2008\poly\Chap1\04\bin\Release>dir
2.  29/01/2008  15:01             4 608 04.exe
3.  29/01/2008  15:01            11 776 04.pdb
```

2

- en [1] : l'exécutable est créé par clic droit sur le projet / Build
- en [2] : dans une fenêtre Dos, l'exécutable *04.exe* a été créé dans le dossier *bin/Release* du projet.

Emettons les commandes suivantes dans la fenêtre Dos [2] :

```
1.  ...\04\bin\Release>echo test >in.txt
2.  ...\04\bin\Release>more in.txt
3.  test
4.  ...\04\bin\Release>04 0<in.txt 1>out.txt 2>err.txt
5.  ...\04\bin\Release>more out.txt
```

```
6.  écriture dans flux Out : test
7.  ...\04\bin\Release>more err.txt
8.  écriture dans flux Error : test
```

- ligne 1 : on met la chaîne *test* dans le fichier *in.txt*
- lignes 2-3 : on affiche le contenu du fichier *in.txt* pour vérification
- ligne 4 : exécution du programme *04.exe*. Le flux *In* est redirigé vers le fichier *in.txt*, le flux *Out* vers le fichier *out.txt*, le flux *Error* vers le fichier *err.txt*. L'exécution ne provoque aucun affichage.
- lignes 5-6 : contenu du fichier *out.txt*. Ce contenu nous montre que :
  - le fichier *in.txt* a été lu
  - l'affichage écran a été redirigé vers *out.txt*
- lignes 7-8 : vérification analogue pour le fichier *err.txt*

On voit clairement que les flux *Out* et *In* n'écrivent pas sur les mêmes périphériques puisqu'on a pu les rediriger séparément.

3.3.5 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

3.3.5.1 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
```

3.3.5.2 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 :

Name	Description
Abs	Overloaded. Returns the absolute value of a specified number.
Acos	Returns the angle whose cosine is the specified number.
Asin	Returns the angle whose sine is the specified number.
Atan	Returns the angle whose tangent is the specified number.
Atan2	Returns the angle whose tangent is the quotient of two specified numbers.
BigMul	Produces the full product of two 32-bit numbers.
Ceiling	Overloaded. Returns the smallest integer greater than or equal to the specified number.
Cos	Returns the cosine of the specified angle.
Cosh	Returns the hyperbolic cosine of the specified angle.
DivRem	Overloaded. Calculates the quotient of two numbers and also returns the remainder in an output parameter.
Exp	Returns e raised to the specified power.
Floor	Overloaded. Returns the largest integer less than or equal to the specified number.
IEEERemainder	Returns the remainder resulting from the division of a specified number by another specified number.
Log	Overloaded. Returns the logarithm of a specified number.
Log10	Returns the base 10 logarithm of a specified number.
Max	Overloaded. Returns the larger of two specified numbers.
Min	Overloaded. Returns the smaller of two numbers.
Pow	Returns a specified number raised to the specified power.
Round	Overloaded. Rounds a value to the nearest integer or specified number of decimal places.
Sign	Overloaded. Returns a value indicating the sign of a number.
Sin	Returns the sine of the specified angle.

Sqrt	Returns the square root of a specified number.
Tan	Returns the tangent of the specified angle.
Tanh	Returns the hyperbolic tangent of the specified angle.
Truncate	Overloaded. Calculates the integral part of a number.

Name	Description
E	Represents the natural logarithmic base, specified by the constant, e.
PI	Represents the ratio of the circumference of a circle to its diameter, specified by the constant, π.

3.3.5.3 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.

3.3.5.4 Expressions relationnelles

Les opérateurs sont les suivants :

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

priorités des opérateurs

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

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

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

Comparaison de deux caractères

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

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

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

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

Comparaison de deux chaînes de caractères

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

Exemples :

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

```
"Chat"      "Chien"
-----------------------
'C'    =      'C'
'h'    =      'h'
'a'    <      'i'
```

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

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

```
"Chat" < "Chaton".
```

Fonctions de comparaison de deux chaînes

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P05 {
5.      static void Main(string[] args) {
6.         string chaine1="chat", chaine2="chien";
7.         int n = chaine1.CompareTo(chaine2);
8.         bool egal = chaine1.Equals(chaine2);
9.         Console.WriteLine("i={0}, egal={1}", n, egal);
10.        Console.WriteLine("chien==chaine1:{0},chien!=chaine2:{1}", "chien"==chaine1,"chien" !=
    chaine2);
11.     }
12.  }
13. }
```

Ligne 7, 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

Ligne 8, la variable *egal* aura la valeur *true* si les deux chaînes sont égales, *false* sinon. Ligne 10, on utilise les opérateurs == et != pour vérifier l'égalité ou non de deux chaînes.

Les résultats de l'exécution :

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

3.3.5.5 Expressions booléennes

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

priorités des opérateurs :

1. !
2. &&
3. ||

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

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

3.3.5.6 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 le code suivant :

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

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

Les résultats de l'exécution sont les suivants :

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

3.3.5.7 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;*

3.3.5.8 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*.

3.3.5.9 L'opérateur ternaire ?

L'expression

```
expr_cond ? expr1:expr2
```

est évaluée de la façon suivante :

1 l'expression *expr_cond* est évaluée. C'est une expression conditionnelle à valeur *vrai* ou *faux*
2 Si elle est vraie, la valeur de l'expression est celle de *expr1* et *expr2* n'est pas évaluée.
3 Si elle est fausse, c'est l'inverse qui se produit : la valeur de l'expression est celle de *expr2* et *expr1* n'est pas évaluée.

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

3.3.5.10 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.

3.3.5.11 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.

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P06 {
5.      static void Main(string[] args) {
6.         int i = 3, j = 4;
7.         float f1=i/j;
8.         float f2=(float)i/j;
9.         Console.WriteLine("f1={0}, f2={1}",f1,f2);
10.     }
11.  }
12. }
```

- ligne 7, *f1* aura la valeur 0.0. La division 3/4 est une division entière puisque les deux opérandes sont de type *int*.
- ligne 8, *(float)i* est la valeur de i transformée en *float*. Maintenant, on a une division entre un réel de type *float* et un entier de type *int*. C'est la division entre nombres réels qui est alors faite. La valeur de *j* sera elle également transformée en type *float*, puis la division des deux réels sera faite. *f2* aura alors la valeur 0,75.

Voici les résultats de l'exécution :

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

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

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

3.4 Les instructions de contrôle du déroulement du programme

3.4.1 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 dans la variable système **ERRORLEVEL** dont la valeur peut être testée dans un fichier batch. Sous Unix, avec l'interpréteur de commandes Shell Bourne, c'est la variable **$?** qui récupère la valeur de *status*.

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

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

3.4.2 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 :

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P07 {
5.      static void Main(string[] args) {
6.         int n = 5;
7.         if (n > 1)
8.            if (n > 6)
9.               Console.Out.WriteLine(">6");
10.           else Console.Out.WriteLine("<=6");
11.     }
12.  }
13. }
```

Dans l'exemple précédent, le *else* de la ligne 10 se rapporte à quel *if* ? La règle est qu'un *else* se rapporte toujours au *if* le plus proche : *if(n>6),* ligne 8, dans l'exemple. Considérons un autre exemple :

```
1.           if (n2 > 1) {
2.              if (n2 > 6) Console.Out.WriteLine(">6");
```

```
3.          } else Console.Out.WriteLine("<=1");
```

Ici nous voulions mettre un *else* au *if(n2>1)* et pas de *else* au *if(n2>6)*. A cause de la remarque précédente, nous sommes obligés de mettre des accolades au *if(n2>1)* {...} *else* ...

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

exemple

En algorithmique

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

En C#

```
1.  int choix = 2;
2.           bool erreur = false;
3.           switch (choix) {
4.              case 0: return;
5.              case 1: M1(); break;
6.              case 2: M2(); break;
7.              default: erreur = true; break;
8.           }
9.        }// fin Main
10.
11.       static void M1() {
12.          Console.WriteLine("M1");
13.       }
14.
15.       static void M2() {
16.          Console.WriteLine("M2");
17.       }
18. }
```

3.4.4 Structures de répétition

3.4.4.1 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 collection)
  instructions;
}
```

Notes

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

Ainsi le code suivant :

```
1.          string[] amis = { "paul", "hélène", "jacques", "sylvie" };
2.          foreach (string nom in amis) {
3.             Console.WriteLine(nom);
4.          }
```

afficherait :

```
paul
hélène
jacques
sylvie
```

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

```
1.          int i, somme, n=10;
2.          for (i = 1, somme = 0; i <= n; i = i + 1)
3.             somme = somme + i;
4.
5.          for (i = 1, somme = 0; i <= n; somme = somme + i, i = i + 1) ;
6.
7.          i = 1; somme = 0;
8.          while (i <= n) { somme += i; i++; }
9.
10.         i = 1; somme = 0;
11.         do somme += i++;
12.         while (i <= n);
13.
```

3.4.4.3 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

3.5 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{
  code susceptible de générer une 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 de type Exception ou dérivé. On peut être plus précis en utilisant des types tels que *IndexOutOfRangeException, FormatException, SystemException,* etc… : il existe plusieurs types d'exceptions. En écrivant *catch (Exception e)*, on indique qu'on veut gérer toutes les types d'exceptions. Si le code de la clause *try* est susceptible de générer plusieurs types d'exceptions, on peut vouloir être plus précis en gérant l'exception avec plusieurs clauses *catch* :

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

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

```
try{
  code susceptible de générer une 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 plus simplement *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.WriteLine("L'erreur suivante s'est produite : {0}",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.WriteLine("L'erreur suivante s'est produite : {0}", ex.ToString());
  ...
}//catch
```

On peut écrire aussi :

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

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

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P08 {
5.      static void Main(string[] args) {
6.         // déclaration & initialisation d'un tableau
7.         int[] tab = { 0, 1, 2, 3 };
8.         int i;
9.         // affichage tableau avec un for
10.        for (i = 0; i < tab.Length; i++)
11.           Console.WriteLine("tab[{0}]={1}", i, tab[i]);
12.        // affichage tableau avec un for each
13.        foreach (int élmt in tab) {
14.           Console.WriteLine(élmt);
15.        }
16.        // génération d'une exception
17.        try {
18.           tab[100] = 6;
19.        } catch (Exception e) {
20.           Console.Error.WriteLine("L'erreur suivante s'est produite : " + e);
21.           return;
22.        }//try-catch
23.        finally {
24.           Console.WriteLine("finally ...");
25.        }
26.     }
27.  }
28. }
```

Ci-dessus, la ligne 18 va générer une exception parce que le tableau *tab* n'a pas d'élément n° 100. L'exécution du programme donne les résultats suivants :

```
1.  tab[0]=0
2.  tab[1]=1
3.  tab[2]=2
4.  tab[3]=3
5.  0
```

```
6.  1
7.  2
8.  3
9.  L'erreur suivante s'est produite : System.IndexOutOfRangeException: L'index se trouve en dehors
    des limites du tableau.
10.    à Chap1.P08.Main(String[] args) dans C:\data\travail\2007-2008\c#
    2008\poly\Chap1\08\Program.cs:ligne 7
11. finally ...
```

- ligne 9 : l'exception [System.IndexOutOfRangeException] s'est produite
- ligne 11 : la clause *finally* (lignes 23-25) du code a été exécutée, alors même que ligne 21, on avait une instruction *return* pour sortir de la méthode. On retiendra que la clause *finally* est toujours exécutée.

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P08 {
5.      static void Main(string[] args) {
6.
7.         // exemple 2
8.         // On demande le nom
9.         Console.Write("Nom : ");
10.        // lecture réponse
11.        string nom = Console.ReadLine();
12.        // on demande l'âge
13.        int age = 0;
14.        bool ageOK = false;
15.        while (!ageOK) {
16.           // question
17.           Console.Write("âge : ");
18.           // lecture-vérification réponse
19.           try {
20.              age = int.Parse(Console.ReadLine());
21.              ageOK = age>=1;
22.           } catch {
23.           }//try-catch
24.           if (!ageOK) {
25.              Console.WriteLine("Age incorrect, recommencez...");
26.           }
27.        }//while
28.        // affichage final
29.        Console.WriteLine("Vous vous appelez {0} et vous avez {1} an(s)",nom,age);
30.     }
31.  }
32. }
```

- lignes 15-27 : la boucle de saisie de l'âge d'une personne
- ligne 20 : la ligne tapée au clavier est transformée en nombre entier par la méthode *int.Parse*. Cette méthode lance une exception si la conversion n'est pas possible. C'est pourquoi, l'opération a été placée dans un try / catch.
- lignes 22-23 : si une exception est lancée, on va dans le catch où rien n'est fait. Ainsi, le booléen *ageOK* positionné à *false*, ligne 14, va-t-il rester à *false*.
- ligne 21 : si on arrive à cette ligne, c'est que la conversion string -> int a réussi. On vérifie cependant que l'entier obtenu est bien supérieur ou égal à 1.
- lignes 24-26 : un message d'erreur est émis si l'âge est incorrect.

Quelques résultats d'exécution :

```
1.  Nom : dupont
2.  âge : 23
3.  Vous vous appelez dupont et vous avez 23 an(s)
```

```
4.  Nom : durand
5.  âge : x
6.  Age incorrect, recommencez...
7.  âge : -4
8.  Age incorrect, recommencez...
9.  âge : 12
10. Vous vous appelez durand et vous avez 12 an(s)
```

3.6 Application exemple - V1

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 (chiffres 2004 pour revenus 2003) :

- 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 :

4262	0	0
8382	0.0683	291.09
14753	0.1914	1322.92
23888	0.2826	2668.39
38868	0.3738	4846.98
47932	0.4262	6883.66
0	0.4809	9505.54

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=5000 on trouvera la ligne

```
  8382      0.0683      291.09
```

L'impôt I est alors égal à **0.0683*R - 291.09*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 0.4809 9505.54

ce qui donne l'impôt I=0.4809*R - 9505.54*nbParts.

Le programme C# correspondant est le suivant :

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class Impots {
5.      static void Main(string[] args) {
6.         // tableaux de données nécessaires au calcul de l'impôt
7.         decimal[] limites = { 4962M, 8382M, 14753M, 23888M, 38868M, 47932M, 0M };
8.         decimal[] coeffR = { 0M, 0.068M, 0.191M, 0.283M, 0.374M, 0.426M, 0.481M };
9.         decimal[] coeffN = { 0M, 291.09M, 1322.92M, 2668.39M, 4846.98M, 6883.66M, 9505.54M };
10.
11.        // on récupère le statut marital
12.        bool OK = false;
13.        string reponse = null;
14.        while (!OK) {
15.           Console.Write("Etes-vous marié(e) (O/N) ? ");
16.           reponse = Console.ReadLine().Trim().ToLower();
17.           if (reponse != "o" && reponse != "n")
18.              Console.Error.WriteLine("Réponse incorrecte. Recommencez");
19.           else OK = true;
20.        }//while
21.        bool marie = reponse == "o";
22.
23.        // nombre d'enfants
24.        OK = false;
25.        int nbEnfants = 0;
26.        while (!OK) {
27.           Console.Write("Nombre d'enfants : ");
28.           try {
29.              nbEnfants = int.Parse(Console.ReadLine());
30.              OK = nbEnfants >= 0;
31.           } catch {
32.           }// try
33.           if (!OK) {
34.              Console.WriteLine("Réponse incorrecte. Recommencez");
35.           }
36.        }// while
37.
38.        // salaire
39.        OK = false;
```

```
40.         int salaire = 0;
41.         while (!OK) {
42.            Console.Write("Salaire annuel : ");
43.            try {
44.               salaire = int.Parse(Console.ReadLine());
45.               OK = salaire >= 0;
46.            } catch {
47.            }// try
48.            if (!OK) {
49.               Console.WriteLine("Réponse incorrecte. Recommencez");
50.            }
51.         }// while
52.
53.         // calcul du nombre de parts
54.         decimal nbParts;
55.         if (marie) nbParts = (decimal)nbEnfants / 2 + 2;
56.         else nbParts = (decimal)nbEnfants / 2 + 1;
57.         if (nbEnfants >= 3) nbParts += 0.5M;
58.
59.         // revenu imposable
60.         decimal revenu = 0.72M * salaire;
61.
62.         // quotient familial
63.         decimal QF = revenu / nbParts;
64.
65.         // recherche de la tranche d'impots correspondant à QF
66.         int i;
67.         int nbTranches = limites.Length;
68.         limites[nbTranches - 1] = QF;
69.         i = 0;
70.         while (QF > limites[i]) i++;
71.         // l'impôt
72.         int impots = (int)(coeffR[i] * revenu - coeffN[i] * nbParts);
73.
74.         // on affiche le résultat
75.         Console.WriteLine("Impôt à payer : {0} euros", impots);
76.      }
77.   }
78. }
```

- lignes 7-9 : les valeurs numériques sont suffixées par **M** (Money) pour qu'elles soient de type *decimal*.
- ligne 16 :
  - *Console.ReadLine()* rend la chaîne *C1* tapée au clavier
  - *C1.Trim()* enlève les espaces de début et fin de *C1* - rend une chaîne *C2*
  - *C2.ToLower()* rend la chaîne *C3* qui est la chaîne *C2* transformée en minuscules.
- ligne 21 : le booléen *marie* reçoit la valeur *true* ou *false* de la relation *reponse=="o"*
- ligne 29 : la chaîne tapée au clavier est transformée en type *int*. Si la transformation échoue, une exception est lancée.
- ligne 30 : le booléen *OK* reçoit la valeur *true* ou *false* de la relation *nbEnfants>=0*
- lignes 55-56 : on ne peut écrire simplement *nbEnfants/2*. Si *nbEnfants* était égal à 3, on aurait 3/2, une division entière qui donnerait 1 et non 1.5. Aussi, écrit-on *(decimal)nbEnfants* pour rendre réel l'un des opérandes de la division et avoir ainsi une division entre réels.

Voici des exemples d'exécution :

```
Etes-vous marié(e) (O/N) ? o
Nombre d'enfants : 2
Salaire annuel : 60000
Impôt à payer : 4282 euros
```

```
Etes-vous marié(e) (O/N) ? oui
Réponse incorrecte. Recommencez
Etes-vous marié(e) (O/N) ? o
Nombre d'enfants : trois
Réponse incorrecte. Recommencez
Nombre d'enfants : 3
Salaire annuel : 60000 euros
Réponse incorrecte. Recommencez
Salaire annuel : 60000
Impôt à payer : 2959 euros
```

3.7 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 (Dos) suivante :

```
P arg0 arg1 … argn
```

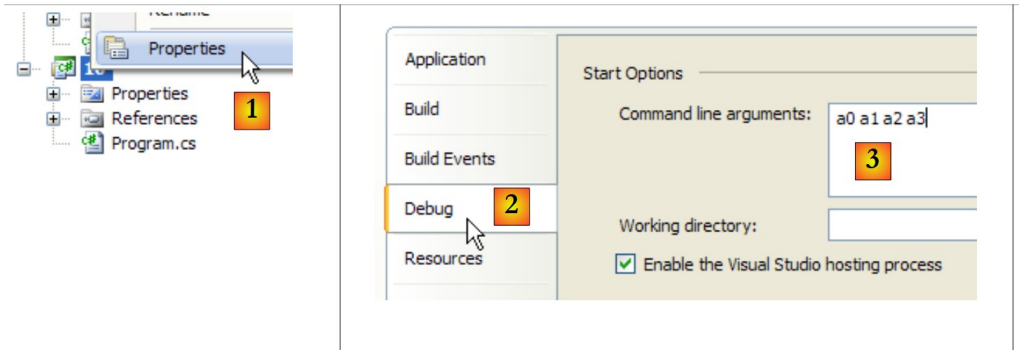
et si la fonction *Main* est déclarée comme suit :

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

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P10 {
5.      static void Main(string[] args) {
6.         // on liste les paramètres reçus
7.         Console.WriteLine("Il y a  " + args.Length + " arguments");
8.         for (int i = 0; i < args.Length; i++) {
9.            Console.Out.WriteLine("arguments[" + i + "]=" + args[i]);
10.        }
11.     }
12.  }
13. }
```

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



- en [1] : clic droit sur le projet / Properties
- en [2] : onglet [Debug]
- en [3] : mettre les arguments

L'exécution donne les résultats suivants :

```
1.  Il y a  4 arguments
2.  arguments[0]=a0
3.  arguments[1]=a1
4.  arguments[2]=a2
5.  arguments[3]=a3
```

On notera que la signature

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

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

3.8 Les énumérations

Une énumération est un type de données dont le domaine de valeurs est un ensemble de constantes 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 Mentions { Passable, AssezBien, Bien, TrèsBien, Excellent };
```

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

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

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

```
1.          if (maMention == Mentions.Passable) {
2.             Console.WriteLine("Peut mieux faire");
3.  }
```

On peut obtenir toutes les valeurs de l'énumération :

```
1.          // liste des mentions sous forme de chaînes
2.          foreach (Mentions m in Enum.GetValues(maMention.GetType())) {
3.             Console.WriteLine(m);
4.  }
```

De la même façon que le type simple *int* est équivalent à la structure *System.Int32*, le type simple *enum* est équivalent à la structure *System.Enum*. Cette structure 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'informations sur le type d'une donnée. Le type d'une variable *v* est obtenu par *v.GetType()*. Le type d'un type *T* est obtenu par *typeof(T)*. 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

```
1.         //liste des mentions sous forme d'entiers
2.         foreach (int m in Enum.GetValues(typeof(Mentions))) {
3.            Console.WriteLine(m);
4.  }
```

Ligne 2, la variable de boucle est de type entier. On obtient alors la liste des valeurs de l'énumération sous forme d'entiers. L'objet de type *System.Type* correspondant au type de données *Mentions* est obtenu par *typeof(Mentions)*. On aurait pu écrire comme précédemment, *maMention.GetType()*.

Le programme suivant met en lumière ce qui vient d'être écrit :

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P11 {
5.      enum Mentions { Passable, AssezBien, Bien, TrèsBien, Excellent };
6.      static void Main(string[] args) {
7.         // une variable qui prend ses valeurs dans l'énumération Mentions
8.         Mentions maMention = Mentions.Passable;
9.         // affichage valeur variable
10.        Console.WriteLine("mention=" + maMention);
11.        // test avec valeur de l'énumération
12.        if (maMention == Mentions.Passable) {
13.           Console.WriteLine("Peut mieux faire");
14.        }
15.        // liste des mentions sous forme de chaînes
16.        foreach (Mentions m in Enum.GetValues(maMention.GetType())) {
17.           Console.WriteLine(m);
18.        }
19.        //liste des mentions sous forme d'entiers
20.        foreach (int m in Enum.GetValues(typeof(Mentions))) {
21.           Console.WriteLine(m);
22.        }
23.     }
24.  }
```

```
25. }
```

Les résultats d'exécution sont les suivants :

```
1.  mention=Passable
2.  Peut mieux faire
3.  Passable
4.  AssezBien
5.  Bien
6.  TrèsBien
7.  Excellent
8.  0
9.  1
10. 2
11. 3
12. 4
```

3.9 Passage de paramètres à une fonction

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

```
1.        private static void ChangeInt(int a) {
2.           a = 30;
3.           Console.WriteLine("Paramètre formel a=" + a);
4.  }
```

Dans la définition de la fonction, ligne1, *a* est appelé un paramètre formel. Il n'est là que pour les besoins de la définition de la fonction *changeInt*. Il aurait tout aussi bien pu s'appeler *b*. Considérons maintenant une utilisation de cette fonction :

```
1.        public static void Main() {
2.           int age = 20;
3.           ChangeInt(age);
4.           Console.WriteLine("Paramètre effectif age=" + age);
5.  }
```

Ici dans l'instruction de la ligne 3, *ChangeInt(age)*, *age* est le paramètre effectif qui va transmettre sa valeur au paramètre formel *a*. Nous nous intéressons à la façon dont un paramètre formel récupère la valeur d'un paramètre effectif.

3.9.1 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.

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P12 {
5.      public static void Main() {
6.         int age = 20;
7.         ChangeInt(age);
8.         Console.WriteLine("Paramètre effectif age=" + age);
9.      }
10.     private static void ChangeInt(int a) {
11.        a = 30;
12.        Console.WriteLine("Paramètre formel a=" + a);
13.     }
14.  }
15. }
```

Les résultats obtenus sont les suivants :

```
1.  Paramètre formel a=30
2.  Paramètre effectif age=20
```

La valeur 20 du paramètre effectif *age* a été recopiée dans le paramètre formel *a* (ligne 10). Celui-ci a été ensuite modifié (ligne 11). Le paramètre effectif est lui resté inchangé. Ce mode de passage convient aux paramètres d'entrée d'une fonction.

3.9.2 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 :

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P12 {
5.      public static void Main() {
6.         // exemple 2
7.         int age2 = 20;
8.         ChangeInt2(ref age2);
9.         Console.WriteLine("Paramètre effectif age2=" + age2);
10.     }
11.     private static void ChangeInt2(ref int a2) {
12.        a2 = 30;
13.        Console.WriteLine("Paramètre formel a2=" + a2);
14.     }
15.  }
16. }
```

et les résultats d'exécution :

```
1.  Paramètre formel a2=30
2.  Paramètre effectif age2=30
```

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

3.9.3 Passage par référence avec le mot clé out

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P12 {
5.      public static void Main() {
6.         // exemple 2
7.         int age2;
8.         ChangeInt2(ref age2);
9.         Console.WriteLine("Paramètre effectif age2=" + age2);
10.     }
11.     private static void ChangeInt2(ref int a2) {
12.        a2 = 30;
13.        Console.WriteLine("Paramètre formel a2=" + a2);
14.     }
15.  }
16. }
```

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

```
Use of unassigned local variable 'age2'
```

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

```
1.  using System;
2.
3.  namespace Chap1 {
4.   class P12 {
5.      public static void Main() {
```

```
6.          // exemple 3
7.          int age3;
8.          ChangeInt3(out age3);
9.          Console.WriteLine("Paramètre effectif age3=" + age3);
10.     }
11.     private static void ChangeInt3(out int a3) {
12.         a3 = 30;
13.         Console.WriteLine("Paramètre formel a3=" + a3);
14.     }
15.  }
16. }
```

Les résultats de l'exécution sont les suivants :

```
1.  Paramètre formel a3=30
2.  Paramètre effectif age3=30
```

4 Classes, Stuctures, Interfaces

4.1 L' objet par l'exemple

4.1.1 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 champs, attributs, ...) et des fonctions (on les appelle des méthodes). Un objet est créé selon un modèle qu'on appelle une classe :

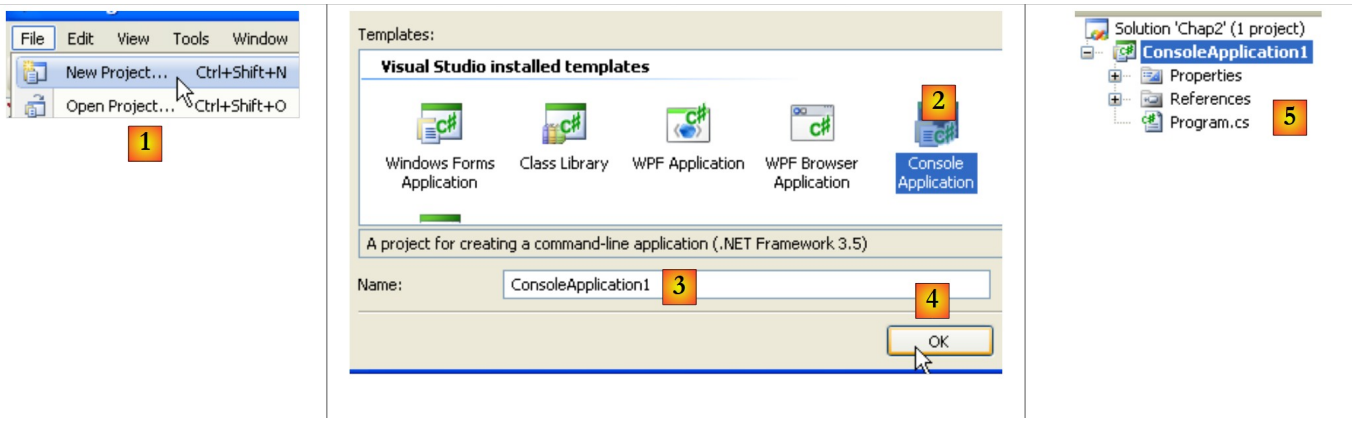
```
public class C1{
  Type1 p1;             // champ p1
  Type2 p2;             // champ p2
  …
  Type3 m3(…){          // méthode m3
        …
  }
  Type4 m4(…){          // méthode m4
        …
  }
  …
}
```

A partir de la classe *C1* précédente, on peut créer de nombreux objets *O1, O2,…* Tous auront les champs *p1, p2,…* et les méthodes *m3, m4, …* Mais ils auront des valeurs différentes pour leurs champs *pi* ayant ainsi chacun un état qui leur est propre. Si o*1* est un objet de type *C1*, o*1.p1* désigne la propriété *p1* de o*1* et o*1.m1* la méthode *m1* de *O1*.

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

4.1.2 Création du projet C#

Dans les exemples précédents, nous n'avions dans un projet qu'un unique fichier source : *Program.cs*. A partir de maintenant, nous pourrons avoir plusieurs fichiers source dans un même projet. Nous montrons comment procéder.

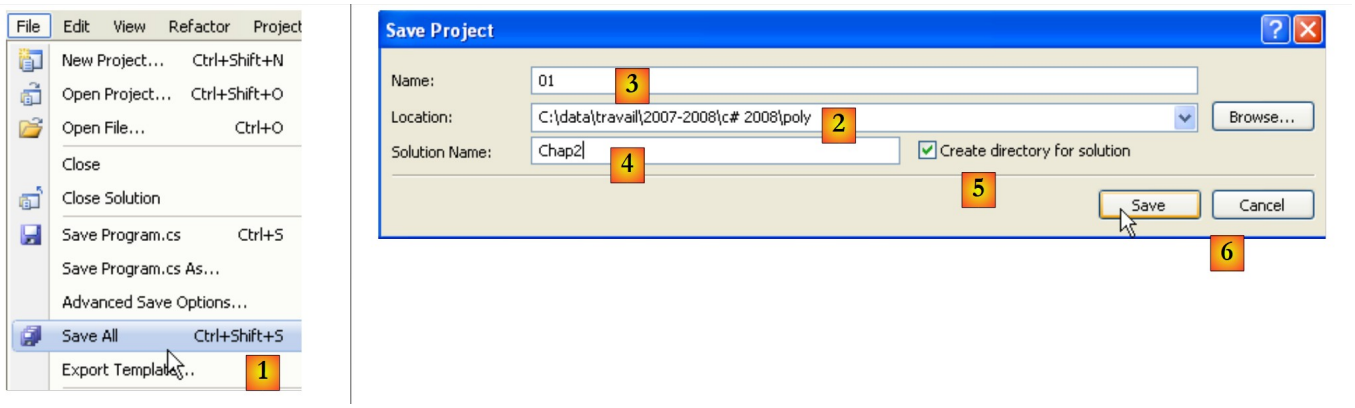


En [1], créez un nouveau projet. En [2], choisissez une Application Console. En [3], laissez la valeur par défaut. En [4], validez. En [5], le projet qui a été généré. Le contenu de *Program.cs* est le suivant :

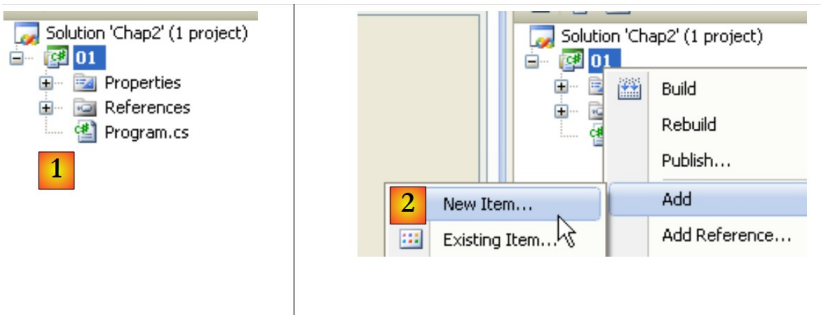
```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.
6.  namespace ConsoleApplication1 {
7.   class Program {
8.      static void Main(string[] args) {
```

```
9.       }
10.  }
11. }
```

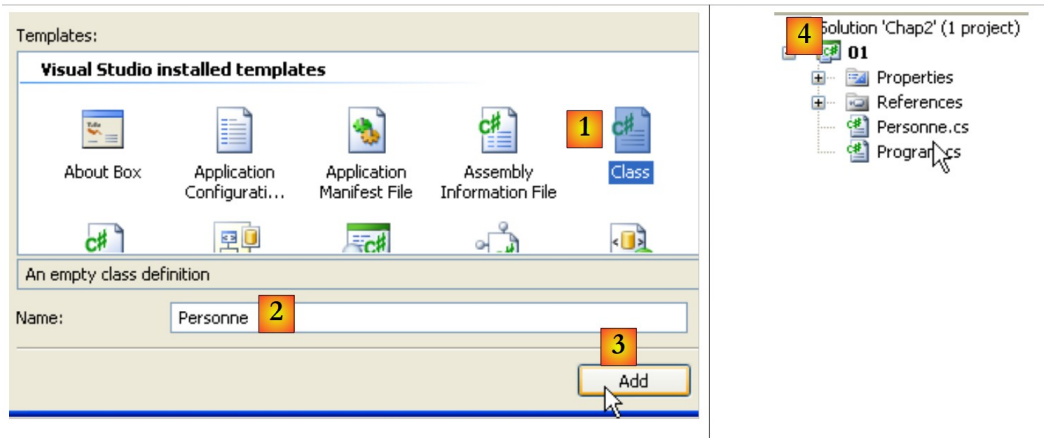
Sauvegardons le projet créé :



En [1], l'option de sauvegarde. En [2], désignez le dossier où sauvegarder le projet. En [3], donnez un nom au projet. En [5], indiquez que vous voulez créer une solution. Une solution est un ensemble de projets. En [4], donnez le nom de la solution. En [6], validez la sauvegarde.



En [1], le projet sauvegardé. En [2], ajoutez un nouvel élément au projet.



En [1], indiquez que vous voulez ajouter une classe. En [2], le nom de la classe. En [3], validez les informations. En [4], le projet [01] a un nouveau fichier source *Personne.cs* :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.
```

```
6.  namespace ConsoleApplication1 {
7.   class Personne {
8.   }
9.  }
```

On modifie l'espace de noms de chacun des fichiers source en *Chap2* et on supprime l'importation des espaces de noms inutiles :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Personne {
5.   }
6.  }
```

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.      }
7.   }
8.  }
```

4.1.3 Définition de la classe Personne

La définition de la classe *Personne* dans le fichier source [*Personne.cs*] sera la suivante :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.      // attributs
6.      private string prenom;
7.      private string nom;
8.      private int age;
9.
10.     // méthode
11.     public void Initialise(string P, string N, int age) {
12.        this.prenom = P;
13.        this.nom = N;
14.        this.age = age;
15.     }
16.
17.     // méthode
18.     public void Identifie() {
19.        Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
20.     }
21.  }
22.
23. }
```

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 classe. 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 méthode 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'une classe C est la suivante :

```
public class C{
  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.

4.1.4 La méthode Initialise

Revenons à notre classe **Personne** déclarée comme :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.      // attributs
6.      private string prenom;
7.      private string nom;
8.      private int age;
9.
10.     // méthode
11.     public void Initialise(string p, string n, int age) {
12.        this.prenom = p;
13.        this.nom = n;
14.        this.age = age;
15.     }
16.
17.     // méthode
18.     public void Identifie() {
19.        Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
20.     }
21.  }
22.
23. }
```

Quel est le rôle de la méthode I*nitialise* ? Parce que *nom*, *prenom* et *age* sont des données **privées** de la classe P*ersonne*, 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.

4.1.5 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.

4.1.6 Le mot clé this

Regardons le code de la méthode *initialise* :

```
1.        public void Initialise(string p, string n, int age) {
2.           this.prenom = p;
3.           this.nom = n;
4.           this.age = age;
5.  }
```

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 :

```
1.      public void Initialise(string p, string n, int age) {
2.         prenom = p;
3.         nom = n;
4.         this.age = age;
5.  }
```

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*.

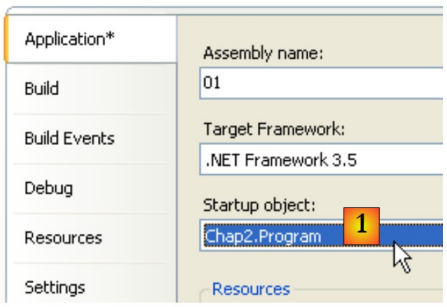
4.1.7 Un programme de test

Voici un court programme de test. Celui-ci est écrit dans le fichier source [*Program.cs*] :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class P01 {
5.      static void Main() {
6.         Personne p1 = new Personne();
7.         p1.Initialise("Jean", "Dupont", 30);
8.         p1.Identifie();
9.      }
10.  }
```

```
11. }
```

Avant d'exécuter le projet [01], il peut être nécessaire de préciser le fichier source à exécuter :



Dans les propriétés du projet [01], on indique en [1] la classe à exécuter.
Les résultats obtenus à l'exécution sont les suivants :

```
[Jean, Dupont, 30]
```

4.1.8 Une autre méthode Initialise

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

```
1.        public void Initialise(Personne p) {
2.           prenom = p.prenom;
3.           nom = p.nom;
4.           age = p.age;
5.  }
```

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 P*ersonne* :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main() {
6.         Personne p1 = new Personne();
7.         p1.Initialise("Jean", "Dupont", 30);
8.         p1.Identifie();
9.         Personne p2 = new Personne();
10.        p2.Initialise(p1);
11.        p2.Identifie();
12.     }
13.  }
14. }
```

et ses résultats :

```
1.  [Jean, Dupont, 30]
2.  [Jean, Dupont, 30]
```

4.1.9 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 (pas même *void*).

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

```
C objet =new C(arg1,arg2, ... argn);
```

ou

```
C objet;
…
objet=new C(arg1,arg2, ... argn);
```

Lorsqu'une classe *C* 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* :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.      // attributs
6.      private string prenom;
7.      private string nom;
8.      private int age;
9.
10.     // constructeurs
11.     public Personne(String p, String n, int age) {
12.        Initialise(p, n, age);
13.     }
14.     public Personne(Personne P) {
15.        Initialise(P);
16.     }
17.
18.     // méthode
19.     public void Initialise(string p, string n, int age) {
20. ...
21.     }
22.
23.     public void Initialise(Personne p) {
24. ...
25.     }
26.
27.     // méthode
28.     public void Identifie() {
29.        Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
30.     }
31.  }
32.
33. }
```

Nos deux constructeurs se contentent de faire appel aux méthodes *Initialise* étudiées précédemment. 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 :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main() {
6.         Personne p1 = new Personne("Jean", "Dupont", 30);
7.         p1.Identifie();
8.         Personne p2 = new Personne(p1);
9.         p2.Identifie();
```

```
10.     }
11.  }
12. }
13.
```

et les résultats obtenus :

```
[Jean, Dupont, 30]
[Jean, Dupont, 30]
```

4.1.10 Les références d'objets

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

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program2 {
5.      static void Main() {
6.         // p1
7.         Personne p1 = new Personne("Jean", "Dupont", 30);
8.         Console.Write("p1="); p1.Identifie();
9.         // p2 référence le même objet que p1
10.        Personne p2 = p1;
11.        Console.Write("p2="); p2.Identifie();
12.        // p3 référence un objet qui sera une copie de l'objet référencé par p1
13.        Personne p3 = new Personne(p1);
14.        Console.Write("p3="); p3.Identifie();
15.        // on change l'état de l'objet référencé par p1
16.        p1.Initialise("Micheline", "Benoît", 67);
17.        Console.Write("p1="); p1.Identifie();
18.        // comme p2=p1, l'objet référencé par p2 a du changer d'état
19.        Console.Write("p2="); p2.Identifie();
20.        // comme p3 ne référence pas le même objet que p1, l'objet référencé par p3 n'a pas du
    changer
21.        Console.Write("p3="); p3.Identifie();
22.     }
23.  }
24. }
```

Les résultats obtenus sont les suivants :

```
1.  p1=[Jean, Dupont, 30]
2.  p2=[Jean, Dupont, 30]
3.  p3=[Jean, Dupont, 30]
4.  p1=[Micheline, Benoît, 67]
5.  p2=[Micheline, Benoît, 67]
6.  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 aussi celui référencé par *p2*.

Lorsqu'on écrit :

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

il y a création d'un nouvel objet *Personne*. 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.

4.1.11 Passage de paramètres de type référence d'objet

Dans le chapitre précédent, nous avons étudié les modes de passage des paramètres d'une fonction lorsque ceux-ci représentaient un type C# simple représenté par une structure .NET. Voyons ce qui se passe lorsque la paramètre est une référence d'objet :

```
1.  using System;
2.  using System.Text;
3.
4.  namespace Chap1 {
5.   class P12 {
6.      public static void Main() {
7.         // exemple 4
8.         StringBuilder sb0 = new StringBuilder("essai0"), sb1 = new StringBuilder("essai1"), sb2 =
    new StringBuilder("essai2"), sb3;
9.         Console.WriteLine("Dans fonction appelante avant appel : sb0={0}, sb1={1}, sb2={2}",
    sb0,sb1, sb2);
10.        ChangeStringBuilder(sb0, sb1, ref sb2, out sb3);
11.        Console.WriteLine("Dans fonction appelante après appel : sb0={0}, sb1={1}, sb2={2},
    sb3={3}", sb0, sb1, sb2, sb3);
12.
13.     }
14.
15.     private static void ChangeStringBuilder(StringBuilder sbf0, StringBuilder sbf1, ref
    StringBuilder sbf2, out StringBuilder sbf3) {
16.        Console.WriteLine("Début fonction appelée : sbf0={0}, sbf1={1}, sbf2={2}", sbf0,sbf1,
    sbf2);
17.        sbf0.Append("*****");
18.        sbf1 = new StringBuilder("essai1*****");
19.        sbf2 = new StringBuilder("essai2*****");
20.        sbf3 = new StringBuilder("essai3*****");
21.        Console.WriteLine("Fin fonction appelée : sbf0={0}, sbf1={1}, sbf2={2}, sbf3={3}", sbf0,
    sbf1, sbf2, sbf3);
22.     }
23.  }
24. }
```

- ligne 8 : définit 3 objets de type *StringBuilder*. Un objet *StringBuilder* est proche d'un objet *string*. Lorsqu'on manipule un objet *string*, on obtient en retour un nouvel objet *string*. Ainsi dans la séquence de code :

```
1.  string s="une chaîne";
2.  s=s.ToUpperCase();
```

La ligne 1 crée un objet *string* en mémoire et s est son adresse. Ligne 2, *s.ToUpperCase()* crée un autre objet *string* en mémoire. Ainsi entre les lignes 1 et 2, *s* a changé de valeur (il pointe sur le nouvel objet). La classe *StringBuilder* elle, permet de transformer une chaîne sans qu'un second objet soit créé. C'est l'exemple donné plus haut :

- ligne 8 : 4 références [sb0, sb1, sb2, sb3] à des objets de type *StringBuilder*
- ligne 10 : sont passées à la méthode *ChangeStringBuilder* avec des modes différents : *sb0*, *sb1* avec le mode par défaut, *sb2* avec le mot clé **ref**, *sb3* avec le mot clé **out**.
- lignes 15-22 : une méthode qui a les paramètres formels [sbf0, sbf1, sbf2, sbf3]. Les relations entre paramètres formels formels *sbfi* et effectifs *sbi* sont les suivantes :
  - *sbf0* et *sb0* sont, au démarrage de la méthode, deux **références distinctes** qui pointent sur le même objet (passage par valeur des adresses)
  - idem pour *sbf1* et *sb1*
  - *sbf2* et *sb2* sont, au démarrage de la méthode, une **même référence** sur le même objet (mot clé *ref*)
  - *sbf3* et *sb3* sont, après exécution de la méthode, une **même référence** sur le même objet (mot clé *out*)

Les résultats obtenus sont les suivants :

```
1.  Dans fonction appelante avant appel : sb0=essai0, sb1=essai1, sb2=essai2
2.  Début fonction appelée : sbf0=essai0, sbf1=essai1, sbf2=essai2
3.  Fin fonction appelée : sbf0=essai0*****, sbf1=essai1*****, sbf2=essai2*****, sbf3=essai3*****
4.  Dans fonction appelante après appel : sb0=essai0*****, sb1=essai1, sb2=essai2*****,
    sb3=essai3*****
```

Explications :

- *sb0* et *sbf0* sont deux références distinctes sur le même objet. Celui-ci a été modifié via *sbf0* - ligne 3. Cette modification peut être vue via *sb0* - ligne 4.
- *sb1* et *sbf1* sont deux références distinctes sur le même objet. *sbf1* voit sa valeur modifiée dans la méthode et pointe désormais sur un nouvel objet - ligne 3. Cela ne change en rien la valeur de *sb1* qui continue à pointer sur le même objet - ligne 4.
- *sb2* et *sbf2* sont une même référence sur le même objet. *sbf2* voit sa valeur modifiée dans la méthode et pointe désormais sur un nouvel objet - ligne 3. Comme *sbf2* et *sb2* sont une seule et même entité, la valeur de *sb2* a été également modifiée et *sb2* pointe sur le même objet que *sbf2* - lignes 3 et 4.
- avant appel de la méthode, *sb3* n'avait pas de valeur. Après la méthode, *sb3* reçoit la valeur de *sbf3*. On a donc deux références sur le même objet - lignes 3 et 4

4.1.12 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 :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main() {
6.         new Personne(new Personne("Jean", "Dupont", 30)).Identifie();
7.      }
8.   }
9.  }
```

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

```
1.        // constructeurs
2.        public Personne(String p, String n, int age) {
3.           Console.WriteLine("Constructeur Personne(string, string, int)");
4.           Initialise(p, n, age);
5.        }
6.        public Personne(Personne P) {
7.           Console.Out.WriteLine("Constructeur Personne(Personne)");
8.           Initialise(P);
9.  }
```

Nous obtenons les résultats suivants :

```
1.  Constructeur Personne(string, string, int)
2.  Constructeur Personne(Personne)
3.  [Jean, Dupont, 30]
```

montrant la construction successive des deux objets temporaires.

4.1.13 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 :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.      // attributs
6.      private string prenom;
7.      private string nom;
8.      private int age;
9.
10.     // constructeurs
```

```
11.     public Personne(String p, String n, int age) {
12.        Console.WriteLine("Constructeur Personne(string, string, int)");
13.        Initialise(p, n, age);
14.     }
15.     public Personne(Personne p) {
16.        Console.Out.WriteLine("Constructeur Personne(Personne)");
17.        Initialise(p);
18.     }
19.
20.     // méthode
21.     public void Initialise(string p, string n, int age) {
22.        this.prenom = p;
23.        this.nom = n;
24.        this.age = age;
25.     }
26.
27.     public void Initialise(Personne p) {
28.        prenom = p.prenom;
29.        nom = p.nom;
30.        age = p.age;
31.     }
32.
33.     // accesseurs
34.     public String GetPrenom() {
35.        return prenom;
36.     }
37.     public String GetNom() {
38.        return nom;
39.     }
40.     public int GetAge() {
41.        return age;
42.     }
43.
44.     //modifieurs
45.     public void SetPrenom(String P) {
46.        this.prenom = P;
47.     }
48.     public void SetNom(String N) {
49.        this.nom = N;
50.     }
51.     public void SetAge(int age) {
52.        this.age = age;
53.     }
54.
55.     // méthode
56.     public void Identifie() {
57.        Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
58.     }
59.  }
60.
61. }
```

Nous testons la nouvelle classe avec le programme suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Personne p = new Personne("Jean", "Michelin", 34);
7.         Console.Out.WriteLine("p=(" + p.GetPrenom() + "," + p.GetNom() + "," + p.GetAge() + ")");
8.         p.SetAge(56);
9.         Console.Out.WriteLine("p=(" + p.GetPrenom() + "," + p.GetNom() + "," + p.GetAge() + ")");
10.     }
11.  }
12. }
```

et nous obtenons les résultats :

```
1.  Constructeur Personne(string, string, int)
2.  p=(Jean,Michelin,34)
3.  p=(Jean,Michelin,56)
```

4.1.14 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 P*ersonne* suivante où les accesseurs et modifieurs précédents ont été remplacés par des **propriétés** en lecture et écriture :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.      // attributs
6.      private string prenom;
7.      private string nom;
8.      private int age;
9.
10.     // constructeurs
11.     public Personne(String p, String n, int age) {
12.        Initialise(p, n, age);
13.     }
14.     public Personne(Personne p) {
15.        Initialise(p);
16.     }
17.
18.     // méthode
19.     public void Initialise(string p, string n, int age) {
20.        this.prenom = p;
21.        this.nom = n;
22.        this.age = age;
23.     }
24.
25.     public void Initialise(Personne p) {
26.        prenom = p.prenom;
27.        nom = p.nom;
28.        age = p.age;
29.     }
30.
31.     // propriétés
32.     public string Prenom {
33.        get { return prenom; }
34.        set {
35.           // prénom valide ?
36.           if (value == null || value.Trim().Length == 0) {
37.              throw new Exception("prénom (" + value + ") invalide");
38.           } else {
39.              prenom = value;
40.           }
41.        }//if
42.     }//prenom
43.
44.     public string Nom {
45.        get { return nom; }
46.        set {
47.           // nom valide ?
48.           if (value == null || value.Trim().Length == 0) {
49.              throw new Exception("nom (" + value + ") invalide");
50.           } else { nom = value; }
51.        }//if
52.     }//nom
53.
54.
55.     public int Age {
56.        get { return age; }
57.        set {
58.           // age valide ?
59.           if (value >= 0) {
60.              age = value;
61.           } else
62.              throw new Exception("âge (" + value + ") invalide");
63.        }//if
64.     }//age
65.
66.     // méthode
67.     public void Identifie() {
```

```
68.         Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);
69.     }
70.  }
71.
72. }
```

Une propriété permet de lire (**get**) ou de fixer (**set**) la valeur d'un attribut. 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.

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

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Personne p = new Personne("Jean", "Michelin", 34);
7.         Console.Out.WriteLine("p=(" + p.Prenom + "," + p.Nom + "," + p.Age + ")");
8.         p.Age = 56;
9.         Console.Out.WriteLine("p=(" + p.Prenom + "," + p.Nom + "," + p.Age + ")");
10.        try {
11.           p.Age = -4;
12.        } catch (Exception ex) {
13.           Console.Error.WriteLine(ex.Message);
14.        }//try-catch
15.     }
16.  }
17. }
```

Dans l'instruction

```
  Console.Out.WriteLine("p=(" + p.Prenom + "," + p.Nom + "," + p.Age + ")");
```

on cherche à avoir les valeurs des propriétés P*renom*, N*om* et A*ge* 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 A*ge*. 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 :

```
1.  p=(Jean,Michelin,34)
2.  p=(Jean,Michelin,56)
3.  âge (-4) invalide
```

Les propriétés nous permettent donc de manipuler des attributs privés comme s'ils étaient publics. Une autre caractéristique des propriétés est qu'elles peuvent être utilisées conjointement avec un constructeur selon la syntaxe suivante :

```
Classe objet=new Classe (...) {Propriété1=val1, Propriété2=val2, ...}
```

Cette syntaxe est équivalente au code suivant :

```
1.  Classe objet=new Classe(...);
2.  objet.Propriété1=val1;
3.  objet.Propriété2=val2;
4.  ...
```

L'ordre des propriétés n'importe pas. Voici un exemple.

La classe *Personne* se voit ajouter un nouveau constructeur sans paramètres :

```
1.        public Personne() {
2.  }
```

Le constructeur n'initialise pas les membres de l'objet. C'est ce qu'on appelle le constructeur par défaut. C'est lui qui est utilisé lorsque la classe ne définit aucun constructeur.

Le code suivant crée et initialise (ligne 6) une nouvelle *Personne* avec la syntaxe présentée précédemment :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Personne p2 = new Personne { Age = 7, Prenom = "Arthur", Nom = "Martin" };
7.         Console.WriteLine("p2=({0},{1},{2})", p2.Prenom, p2.Nom, p2.Age);
8.      }
9.   }
10. }
```

Ligne 6 ci-dessus, c'est le constructeur sans paramètres *Personne()* qui est utilisé. Dans ce cas particulier, on aurait pu aussi écrire

```
        Personne p2 = new Personne() { Age = 7, Prenom = "Arthur", Nom = "Martin" };
```

mais les parenthèses du constructeur *Personne*() sans paramètres ne sont pas obligatoires dans cette syntaxe.

Les résultats de l'exécution sont les suivants :

```
p2=(Arthur,Martin,7)
```

Dans beaucoup de cas, les méthodes *get* et *set* d'une propriété se contentent de lire et écrire un champ privé sans autre traitement. On peut alors, dans ce scénario, utiliser une propriété **automatique** déclarée comme suit :

```
public Type Propriété{ get ; set ; }
```

Le champ privé associé à la propriété n'est pas déclaré. Il est automatiquement généré par le compilateur. On y accède que via sa propriété. Ainsi, au lieu d'écrire :

```
  private string prenom;
...
  // propriété associée
     public string Prenom {
        get { return prenom; }
        set {
           // prénom valide ?
           if (value == null || value.Trim().Length == 0) {
              throw new Exception("prénom (" + value + ") invalide");
           } else {
              prenom = value;
           }
        }//if
     }//prenom
```

on pourra écrire :

```
public string Prenom {get; set;}
```

sans déclarer le champ privé *prenom*. La différence entre les deux propriétés précédentes est que la première vérifie la validité du prénom dans le *set*, alors que la deuxième ne fait aucune vérification.

Utiliser la propriété automatique *Prenom* revient à déclarer un champ *Prenom* **public** :

```
public string Prenom;
```

On peut se demander s'il y a une différence entre les deux déclarations. Déclarer *public* un champ d'une classe est déconseillé. Cela rompt avec le concept d'encapsulation de l'état d'un objet, état qui doit être privé et exposé par des méthodes publiques.

Si la propriété automatique est déclarée *virtuelle,* elle peut alors être redéfinie dans une classe fille :

```
1.   class Class1 {
2.      public virtual string Prop { get; set; }
3.   }
```

```
1.   class Class2 : Class1 {
2.      public override string Prop { get { return base.Prop; } set {... } }
3.   }
```

Ligne 2 ci-dessus, la classe fille *Class2* peut mettre dans le *set,* du code vérifiant la validité de la valeur affectée à la propriété automatique *base.Prop* de la classe mère *Class1*.

4.1.15 Les méthodes et attributs de classe

Supposons qu'on veuille compter le nombre d'objets *Personne* 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 celui d'une instance particulière de cette classe, on le déclare différemment avec le mot clé *static* :

```
     private static long nbPersonnes;
```

Pour le référencer, on écrit P*ersonne.nbPersonnes* pour montrer que c'est un attribut de la classe P*ersonne* 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 P*ersonne* particulier : en effet *nbPersonnes* est l'attribut de toute une classe. Aussi a-t-on besoin d'une propriété déclarée elle-aussi *static* :

```
1.        public static long NbPersonnes {
2.           get { return nbPersonnes; }
3.   }
```

qui de l'extérieur sera appelée avec la syntaxe P*ersonne.NbPersonnes*. Voici un exemple.

La classe P*ersonne* devient la suivante :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.
6.      // attributs de classe
7.      private static long nbPersonnes;
8.      public static long NbPersonnes {
9.         get { return nbPersonnes; }
10.     }
11.
12.     // attributs d'instance
13.     private string prenom;
14.     private string nom;
15.     private int age;
16.
17.     // constructeurs
18.     public Personne(String p, String n, int age) {
19.        Initialise(p, n, age);
20.        nbPersonnes++;
21.     }
22.     public Personne(Personne p) {
```

```
23.         Initialise(p);
24.         nbPersonnes++;
25.     }
26.
27. ...
28. }
```

Lignes 20 et 24, les constructeurs incrémentent le champ statique de la ligne 7.

Avec le programme suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Personne p1 = new Personne("Jean", "Dupont", 30);
7.         Personne p2 = new Personne(p1);
8.         new Personne(p1);
9.         Console.WriteLine("Nombre de personnes créées : " + Personne.NbPersonnes);
10.     }
11.  }
12. }
```

on obtient les résultats suivants :

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

4.1.16 Un tableau de personnes

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

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         // un tableau de personnes
7.         Personne[] amis = new Personne[3];
8.         amis[0] = new Personne("Jean", "Dupont", 30);
9.         amis[1] = new Personne("Sylvie", "Vartan", 52);
10.        amis[2] = new Personne("Neil", "Armstrong", 66);
11.        // affichage
12.        foreach (Personne ami in amis) {
13.           ami.Identifie();
14.        }
15.     }
16.  }
17. }
```

- ligne 7 : crée un tableau de 3 éléments de type P*ersonne*. 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.
- lignes 8-10 : création des 3 objets de type *Personne*
- lignes 12-14 : affichage du contenu du tableau *amis*

On obtient les résultats suivants :

```
1.  [Jean, Dupont, 30]
2.  [Sylvie, Vartan, 52]
3.  [Neil, Armstrong, 66]
```

4.2 L'héritage par l'exemple

4.2.1 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 : 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 :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   public class Personne {
5.
6.      // attributs de classe
7.      private static long nbPersonnes;
8.      public static long NbPersonnes {
9.         get { return nbPersonnes; }
10.     }
11.
12.     // attributs d'instance
13.     private string prenom;
14.     private string nom;
15.     private int age;
16.
17.     // constructeurs
18.     public Personne(String prenom, String nom, int age) {
19.        Nom = nom;
20.        Prenom = prenom;
21.        Age = age;
22.        nbPersonnes++;
23.        Console.WriteLine("Constructeur Personne(string, string, int)");
24.     }
25.     public Personne(Personne p) {
26.        Nom = p.Nom;
27.        Prenom = p.Prenom;
28.        Age = p.Age;
29.        nbPersonnes++;
30.        Console.WriteLine("Constructeur Personne(Personne)");
31.     }
32.
33.     // propriétés
34.     public string Prenom {
35.        get { return prenom; }
36.        set {
37.           // prénom valide ?
38.           if (value == null || value.Trim().Length == 0) {
39.              throw new Exception("prénom (" + value + ") invalide");
40.           } else {
41.              prenom = value;
42.           }
43.        }//if
44.     }//prenom
45.
46.     public string Nom {
47.        get { return nom; }
48.        set {
49.           // nom valide ?
50.           if (value == null || value.Trim().Length == 0) {
51.              throw new Exception("nom (" + value + ") invalide");
52.           } else { nom = value; }
53.        }//if
54.     }//nom
55.
56.     public int Age {
```

```
57.         get { return age; }
58.         set {
59.            // age valide ?
60.            if (value >= 0) {
61.               age = value;
62.            } else
63.               throw new Exception("âge (" + value + ") invalide");
64.         }//if
65.      }//age
66.
67.      // propriété
68.      public string Identite {
69.         get { return String.Format("[{0}, {1}, {2}]", prenom, nom, age);}
70.      }
71.   }
72.
73. }
```

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

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Enseignant : Personne {
5.      // attributs
6.      private int section;
7.
8.      // constructeur
9.      public Enseignant(string prenom, string nom, int age, int section)
10.        : base(prenom, nom, age) {
11.        // on mémorise la section via la propriété Section
12.        Section = section;
13.        // suivi
14.        Console.WriteLine("Construction Enseignant(string, string, int, int)");
15.     }//constructeur
16.
17.     // propriété Section
18.     public int Section {
19.        get { return section; }
20.        set { section = value; }
21.     }// Section
22.
23.  }
24. }
```

La classe *Enseignant* rajoute aux méthodes et attributs de la classe *Personne* :

- ligne 4 : la classe *Enseignant* dérive de la classe *Personne*
- ligne 6 : un attribut *section* qui est le n° de section auquel appartient l'enseignant dans le corps des enseignants (une section par discipline en gros). Cet attribut privé est accessible via la propriété publique *Section* des lignes 18-21
- ligne 9 : un nouveau constructeur permettant d'initialiser tous les attributs d'un enseignant

4.2.2 Construction d'un objet Enseignant

Une classe fille **n'hérite pas des constructeurs** de sa classe Parent. Elle doit alors définir ses propres constructeurs. Le constructeur de la classe *Enseignant* est le suivant :

```
1.        // constructeur
2.        public Enseignant(string prenom, string nom, int age, int section)
3.           : base(prenom, nom, age) {
4.           // on mémorise la section
5.           Section = section;
6.           // suivi
7.           Console.WriteLine("Construction enseignant(string, string, int, int)");
8.     }//constructeur
```

La déclaration

```
    public Enseignant(string prenom, string nom, int age, int section)
```

```
        : base(prenom, nom, age) {
```

déclare que le constructeur reçoit quatre paramètres *prenom*, *nom*, *age*, *section* et en passe trois *(prenom,nom,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 *(prenom,nom,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 :

```
        // on mémorise la section
        Section = section;
```

On notera qu'à gauche du signe =, ce n'est pas l'attribut *section* de l'objet qui a été utilisé, mais la propriété *Section* qui lui est associée. Cela permet au constructeur de profiter des éventuels contrôles de validité que pourrait faire cette méthode. Cela évite de placer ceux-ci à deux endroits différents : le constructeur et la propriété.

En résumé, le constructeur d'une classe dérivée :

- passe à sa classe de base les paramètres dont celle-ci 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 prenom, string nom, int age, int section){
    this.prenom=prenom;
       this.nom=nom;
       this.age=age;
       this.section=section;
  }
```

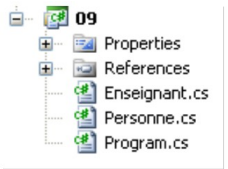
C'est impossible. La classe *Personne* a déclaré privés (*private*) ses trois champs *prenom*, *nom* et *age*. Seuls des objets de la même classe ont un accès direct à ces champs. Tous les autres objets, y compris des objets fils comme ici, doivent passer par des méthodes publiques pour y avoir accès. 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).

Tentons un premier programme de test [Program.cs] :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Console.WriteLine(new Enseignant("Jean", "Dupont", 30, 27).Identite);
7.      }
8.   }
9.  }
```

Ce programme ce contente de créer un objet *Enseignant* (new) et de l'identifier. La classe *Enseignant* n'a pas de méthode I*dentite* mais sa classe parent en a une qui de plus est publique : elle devient par héritage une méthode publique de la classe E*nseignant*.

L'ensemble du projet est le suivant :



- *Personne.cs* : la classe *Personne*
- *Enseignant.cs* : la classe *Enseignant*
- *Program.cs* : le programme de test

Les résultats obtenus sont les suivants :

```
1.  Constructeur Personne(string, string, int)
2.  Construction Enseignant(string, string, int, int)
3.  [Jean, Dupont, 30]
```

On voit que :

- un objet P*ersonne* (ligne 1) a été construit avant l'objet E*nseignant* (ligne 2)
- l'identité obtenue est celle de l'objet *Personne*

4.2.3 Redéfinition d'une méthode ou d'une propriété

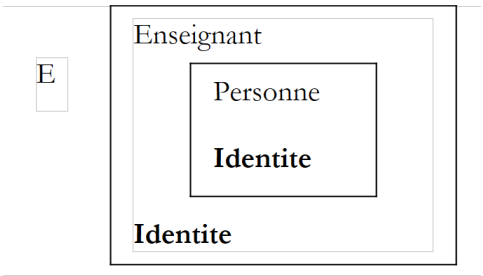
Dans l'exemple précédent, nous avons eu l'identité de la partie P*ersonne* de l'enseignant mais il manque certaines informations propres à la classe *Enseignant* (la section). On est donc amené à écrire une propriété permettant d'identifier l'enseignant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Enseignant : Personne {
5.      // attributs
6.      private int section;
7.
8.      // constructeur
9.      public Enseignant(string prenom, string nom, int age, int section)
10.        : base(prenom, nom, age) {
11.        // on mémorise la section via la propriété Section
12.        Section = section;
13.        // suivi
14.        Console.WriteLine("Construction Enseignant(string, string, int, int)");
15.     }//constructeur
16.
17.     // propriété Section
18.     public int Section {
19.        get { return section; }
20.        set { section = value; }
21.     }// section
22.
23.     // propriété Identite
24.     public new string Identite {
25.        get { return String.Format("Enseignant[{0},{1}]", base.Identite, Section); }
26.     }
27.  }
28. }
```

Lignes 24-26, la propriété *Identite* de la classe *Enseignant* s'appuie sur la propriété *Identite* de sa classe mère (*base.Identite*) (ligne 25) pour afficher sa partie "P*ersonne*" 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 **Identite** 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 E*nseignant* dispose maintenant de deux propriétés I*dentite* :

- celle héritée de la classe parent P*ersonne*
- la sienne propre

Si E est un ojet E*nseignant*, *E.Identite* désigne la propriété I*dentite* de la classe E*nseignant*. On dit que la propriété I*dentite* de la classe fille **redéfinit** ou cache la propriété I*dentite* de la classe mère. 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 redéfinir 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 redéfinition de méthodes/propriétés est le principal intérêt de l'héritage.

Considérons le même programme de test que précédemment :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Console.WriteLine(new Enseignant("Jean", "Dupont", 30, 27).Identite);
7.      }
8.   }
9.  }
```

Les résultats obtenus sont cette fois les suivants :

```
1.  Constructeur Personne(string, string, int)
2.  Construction Enseignant(string, string, int, int)
3.  Enseignant[[Jean, Dupont, 30],27]
```

4.2.4 Le polymorphisme

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

où $C_i \leftarrow C_j$ 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 :

```
Oi=Oj 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

```
Oi=Oj
```

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 *p* de type *Personne* de la méthode statique A*ffiche* va recevoir une valeur de type *Enseignant*. Comme le type *Enseignant* dérive du type *Personne*, c'est légal.

4.2.5 Redéfinition et polymorphisme

Complétons notre méthode A*ffiche* :

```
1.        public static void Affiche(Personne p) {
2.           // affiche identité de p
3.           Console.WriteLine(p.Identite);
4.  }//affiche
```

La propriété *p.Identite* rend une chaîne de caractères identifiant l'objet *Personne p*. Que se passe-t-il dans l'exemple précédent si le paramètre passé à la méthode *Affiche* est un objet de type *Enseignant* :

```
        Enseignant e = new Enseignant(...);
        Affiche(e);
```

Regardons l'exemple suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program2 {
5.      static void Main(string[] args) {
6.         // un enseignant
7.         Enseignant e = new Enseignant("Lucile", "Dumas", 56, 61);
8.         Affiche(e);
9.         // une personne
10.        Personne p = new Personne("Jean", "Dupont", 30);
11.        Affiche(p);
12.     }
13.
14.     // affiche
15.     public static void Affiche(Personne p) {
16.        // affiche identité de p
17.        Console.WriteLine(p.Identite);
18.     }//affiche
19.  }
20. }
```

Les résultats obtenus sont les suivants :

```
1.  Constructeur Personne(string, string, int)
2.  Construction Enseignant(string, string, int, int)
3.  [Lucile, Dumas, 56]
4.  Constructeur Personne(string, string, int)
5.  [Jean, Dupont, 30]
```

L'exécution montre que l'instruction *p.Identite* (ligne 17) a exécuté à chaque fois la propriété *Identite* d'une *Personne*, d'abord (ligne 7) la personne contenue dans l'*Enseignant* e, puis (ligne 10) 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 par *p*.

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

```
1.  public virtual string Identite {
2.           get { return String.Format("[{0}, {1}, {2}]", prenom, nom, age); }
3.        }
```

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 alors utiliser le mot clé **override** au lieu de **new** pour qualifier leur propriété/méthode redéfinie. Ainsi dans la classe *Enseignant*, la propriété *Identite* est redéfinie comme suit :

```
1.        public override string Identite {
2.           get { return String.Format("Enseignant[{0},{1}]", base.Identite, Section); }
3.  }
```

Le programme précédent produit alors les résultats suivants :

```
1.  Constructeur Personne(string, string, int)
2.  Construction Enseignant(string, string, int, int)
3.  Enseignant[[Lucile, Dumas, 56],61]
4.  Constructeur Personne(string, string, int)
5.  [Jean, Dupont, 30]
```

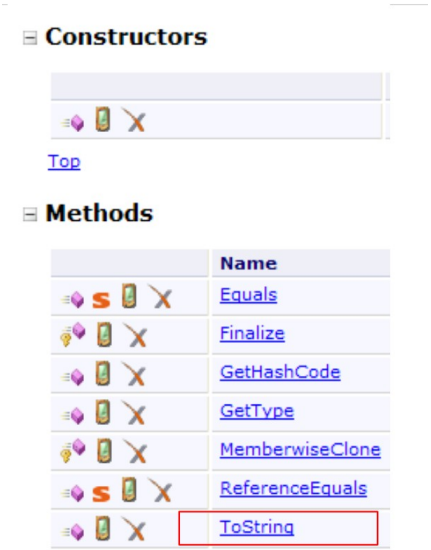
Cette fois-ci, ligne 3, on a bien eu l'identité complète de l'enseignant. Redéfinissons maintenant une méthode plutôt qu'une propriété. La classe *object* (alias C# de *System.Object*) est la classe "mère" de toutes les classes C#. Ainsi lorsqu'on écrit :

```
public class Personne
```

on écrit implicitement :

```
public class Personne : System.Object
```

La classe *System.Object* définit une méthode virtuelle **ToString** :



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

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program2 {
5.      static void Main(string[] args) {
6.         // un enseignant
7.         Console.WriteLine(new Enseignant("Lucile", "Dumas", 56, 61).ToString());
8.         // une personne
9.         Console.WriteLine(new Personne("Jean", "Dupont", 30).ToString());
10.     }
11.  }
12. }
```

Les résultats produits sont les suivants :

```
1.  Constructeur Personne(string, string, int)
2.  Construction Enseignant(string, string, int, int)
3.  Chap2.Enseignant
4.  Constructeur Personne(string, string, int)
5.  Chap2.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* a été capable d'afficher le nom réel de la classe de l'objet.

Redéfinissons la méthode *ToString* dans les classes *Personne* et *Enseignant* :

```
1.        // méthode ToString
2.        public override string ToString() {
3.           return Identite;
4.  }
```

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

```
1.  using System;
2.  namespace Chap2 {
3.   class Program3 {
4.      public static void Main() {
5.         // un enseignant
6.         Enseignant e = new Enseignant("Lucile", "Dumas", 56, 61);
7.         Affiche(e);
8.         // une personne
9.         Personne p = new Personne("Jean", "Dupont", 30);
10.        Affiche(p);
11.     }
12.     // affiche
13.     public static void Affiche(Personne p) {
14.        // affiche identité de p
15.        Console.WriteLine(p);
16.     }//Affiche
17.  }
18. }
```

Attardons-nous sur la méthode *Affiche* qui admet pour paramètre une personne *p*. Ligne 15, la méthode *WriteLine* de la classe *Console* n'a aucune variante admettant un paramètre de type *Personne*. Parmi les différentes variantes de *Writeline*, il en existe une qui admet comme paramètre un type *Object*. Le compilateur va utiliser cette méthode, *WriteLine(Object o),* parce que cette signature signifie que le paramètre *o* peut être de type *Object* ou dérivé. Puisque *Object* est la classe mère de toutes les classes, tout objet peut être passé en paramètre à *WriteLine* et donc un objet de type *Personne* ou *Enseignant*. La méthode *WriteLine(Object o)* écrit *o.ToString()* dans le flux d'écriture *Out*. La méthode *ToString* étant virtuelle, si l'objet o (de type *Object* ou dérivé) a redéfini la méthode *ToString*, ce sera cette dernière qui sera utilisée. C'est ici le cas avec les classes *Personne* et *Enseignant*.

C'est ce que montrent les résultats d'exécution :

```
1.  Constructeur Personne(string, string, int)
2.  Construction Enseignant(string, string, int, int)
3.  Enseignant[[Lucile, Dumas, 56],61]
4.  Constructeur Personne(string, string, int)
5.  [Jean, Dupont, 30]
```

4.3 Redéfir la signification d'un opérateur pour une classe

4.3.1 Introduction

Considérons l'instruction

```
op1 + op2
```

où *op1* et *op2* sont deux opérandes. Il est possible de redéfinir la signification de l'opérateur + . Si l'opérande *op1* est un objet de classe *C1,* il faut définir une méthode statique dans la classe *C1* avec la signature suivante :

```
public static [type] operator +(C1 opérande1, C2 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

4.3.2 Un exemple

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

- la méthode L.*Add(Object o)* permettant d'ajouter à la liste *L* un objet *o*. Ici l'objet *o* sera un objet *Personne*.
- la propriété L.*Count* qui donne le nombre d'éléments de la liste L
- la notation L*[i]* qui donne l'élément i de la liste L

La classe *ListeDePersonnes* va hériter de tous les attributs, méthodes et propriétés de la classe *ArrayList*. Sa définition est la suivante :

```
1.  using System;
2.  using System.Collections;
3.  using System.Text;
4.
5.  namespace Chap2 {
6.   class ListeDePersonnes : ArrayList{
7.      // redéfinition opérateur +, pour ajouter une personne à la liste
8.      public static ListeDePersonnes operator +(ListeDePersonnes l, Personne p) {
9.         // on ajoute la Personne p à la ListeDePersonnes l
10.        l.Add(p);
11.        // on rend la ListeDePersonnes l
12.        return l;
13.     }// operator +
14.
15.     // ToString
16.     public override string ToString() {
17.        // rend (él1, él2, ..., éln)
18.        // parenthèse ouvrante
19.        StringBuilder listeToString = new StringBuilder("(");
20.        // on parcourt la liste de personnes (this)
21.        for (int i = 0; i < Count - 1; i++) {
22.           listeToString.Append(this[i]).Append(",");
23.        }//for
24.        // dernier élément
25.        if (Count != 0) {
26.           listeToString.Append(this[Count-1]);
27.        }
28.        // parenthèse fermante
29.        listeToString.Append(")");
30.        // on doit rendre un string
31.        return listeToString.ToString();
32.     }//ToString
33.  }
34. }
```

- ligne 6 : la classe *ListeDePersonnes* dérive de la classe *ArrayList*
- lignes 8-13 : définition de l'opérateur + pour l'opération l + p, où l est de type *ListeDePersonnes* et p de type *Personne* ou dérivé.
- ligne 10 : la personne p est ajoutée à la liste l. C'est la méthode *Add* de la classe parent *ArrayList* qui est ici utilisée.

- ligne 12 : la référence sur la liste l est rendue afin de pouvoir enchaîner les opérateurs + tels que dans l + p1 + p2. L'opération *l+p1+p2* sera interprétée (priorité des opérateurs) comme *(l+p1)+p2*. L'opération *l+p1* rendra la référence *l*. L'opération *(l+p1)+p2* devient alors *l+p2* qui ajoute la personne *p2* à la liste de personnes *l*.
- ligne 16 : 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*.
- ligne 19 : nous utilisons un objet de type *StringBuilder*. Cette classe convient mieux que la classe *string* dès qu'il faut faire de nombreuses opérations sur la chaîne de caractères, ici des ajouts. En effet, chaque opération sur un objet *string* rend un nouvel objet *string*, alors que les mêmes opérations sur un objet *StringBuilder* modifient l'objet mais n'en créent pas un nouveau. Nous utilisons la méthode *Append* pour concaténer les chaînes de caractères.
- ligne 21 : on parcourt les éléments de la liste de personnes. Cette liste est ici désignée par **this**. C'est l'objet courant sur laquelle est exécutée la méthode *ToString*. La propriété *Count* est une propriété de la classe parent *ArrayList*.
- ligne 22 : l'élément n° i de la liste courante *this* est accessible via la notation *this[i]*. Là encore, c'est une propriété de la classe *ArrayList*. Comme il s'agit d'ajouter des chaînes, c'est la méthode *this[i].ToString()* qui va être utilisée. Comme cette méthode est virtuelle, c'est la méthode *ToString* de l'objet *this*, de type *Personne* ou dérivé, qui va être utilisée.
- ligne 31 : il nous faut rendre un objet de type *string* (ligne 16). La classe *StringBuilder* a une méthode *ToString* qui permet de passer d'un type *StringBuilder* à un type *string*.

On notera que la classe *ListeDePersonnes* n'a pas de constructeur. Dans ce cas, on sait que le constructeur

```
public ListeDePersonnes(){
}
```

sera utilisé. Ce constructeur ne fait rien si ce n'est appeler le constructeur sans paramètres de sa classe parent :

```
public ArrayList(){
...
}
```

Une classe de test pourrait être la suivante :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program1 {
5.      static void Main(string[] args) {
6.         // une liste de personnes
7.         ListeDePersonnes l = new ListeDePersonnes();
8.         // ajout de personnes
9.         l = l + new Personne("jean", "martin",10) + new Personne("pauline", "leduc",12);
10.        // affichage
11.        Console.WriteLine("l=" + l);
12.        l = l + new Enseignant("camille", "germain",27,60);
13.        Console.WriteLine("l=" + l);
14.     }
15.  }
16. }
```

- ligne 7 : création d'une liste de personnes l
- ligne 9 : ajout de 2 personnes avec l'opérateur +
- ligne 12 : ajout d'un enseignant
- lignes 11 et 13 : utilisation de la méthode redéfinie *ListeDePersonnes.ToString()*.

Les résultats :

```
1.  l=([jean, martin, 10],[pauline, leduc, 12])
2.  l=([jean, martin, 10],[pauline, leduc, 12],Enseignant[[camille, germain, 27],60])
```

4.4 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 dans la classe *ListeDePersonnes* la méthode **this** suivante :

```
1.        public Personne this[int i] {
```

```
2.          get { ... }
3.          set { ... }
4.  }
```

On appelle la méthode *this[int i]*, un **indexeur** car elle donne une signification 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 la méthode *this* 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 i] { ... }
```

Il y a un conflit entre la méthode *this* de la classe *ListeDePersonnes* :

```
public Personne this[int i]
```

et la méthode *this* de la classe *ArrayList*

```
public object this[int i]
```

parce qu'elles portent le même nom et admettent le même type de paramètre (int).Pour indiquer que la méthode *this* de la classe *ListeDePersonnes* "cache" la méthode de même nom 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 *this.get* est appelée lorsqu'on écrit *variable=l[i]* par exemple, où *l* est de type *ListeDePersonnes*. On doit alors retourner la personne n° i de la liste *l*. Ceci se fait avec la notation *base[i],* qui rend l'objet n° i de la classe *ArrayList* sous-jacente à la classe *ListeDePersonnes* . L'objet retourné étant de type *Object*, un transtypage vers la classe *Personne* est nécessaire.

```
public new Personne this[int i]{
      get { return (Personne) base[i]; }
      set { ... }
}
```

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*.

```
public new Personne this[int i]{
      get { ... }
      set { base[i]=value; }
}
```

Ici, la personne *p* représentée par le mot clé *value* est affectée à l'élément n° *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; }
}
```

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 :

```
1.        // indexeur via un nom
2.        public int this[string nom] {
3.           get {
4.              // on recherche la personne
5.              for (int i = 0; i < Count; i++) {
6.                 if (((Personne)base[i]).Nom == nom)
7.                    return i;
8.              }//for
9.              return -1;
10.          }//get
11. }
```

La première ligne

```
public int this[string nom]
```

indique qu'on indexe la classe *ListeDePersonnes* par une chaîne de caractères *nom* et que le résultat de *l[nom]* est un entier. Cet entier sera la position dans la liste, de la personne portant le nom *nom* 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 passé en paramètre. Si on le trouve en position i, on renvoie i sinon on renvoie -1.

Le programme de test précédent est complété de la façon suivante :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program2 {
5.      static void Main(string[] args) {
6.         // une liste de personnes
7.         ListeDePersonnes l = new ListeDePersonnes();
8.         // ajout de personnes
9.         l = l + new Personne("jean", "martin",10) + new Personne("pauline", "leduc",12);
10.        // affichage
11.        Console.WriteLine("l=" + l);
12.        l = l + new Enseignant("camille", "germain",27,60);
13.        Console.WriteLine("l=" + l);
14.        // changement élément 1
15.        l[1] = new Personne("franck", "gallon",5);
16.        // affichage élément 1
17.        Console.WriteLine("l[1]=" + l[1]);
18.        // affichage liste l
19.        Console.WriteLine("l=" + l);
20.        // recherche de personnes
21.        string[] noms = { "martin", "germain", "xx" };
22.        for (int i = 0; i < noms.Length; i++) {
23.           int inom = l[noms[i]];
24.           if (inom != -1)
25.              Console.WriteLine("Personne(" + noms[i] + ")=" + l[inom]);
26.           else
27.              Console.WriteLine("Personne(" + noms[i] + ") n'existe pas");
28.        }//for
29.     }
30.  }
31. }
```

Son exécution donne les résultats suivants :

```
1.  l=([jean, martin, 10],[pauline, leduc, 12])
2.  l=([jean, martin, 10],[pauline, leduc, 12],Enseignant[[camille, germain, 27],60])
3.  l[1]=[franck, gallon, 5]
4.  l=([jean, martin, 10],[franck, gallon, 5],Enseignant[[camille, germain, 27],60])
5.  Personne(martin)=[jean, martin, 10]
6.  Personne(germain)=Enseignant[[camille, germain, 27],60]
7.  Personne(xx) n'existe pas
```

4.5 Les structures

La structure C# est analogue à la structure du langage C et est très proche de la notion de 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 :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program1 {
5.      static void Main(string[] args) {
6.         // une structure sp1
7.         SPersonne sp1;
8.         sp1.Nom = "paul";
9.         sp1.Age = 10;
10.        Console.WriteLine("sp1=SPersonne(" + sp1.Nom + "," + sp1.Age + ")");
11.        // une structure sp2
12.        SPersonne sp2 = sp1;
13.        Console.WriteLine("sp2=SPersonne(" + sp2.Nom + "," + sp2.Age + ")");
14.        // sp2 est modifié
15.        sp2.Nom = "nicole";
16.        sp2.Age = 30;
17.        // vérification sp1 et sp2
18.        Console.WriteLine("sp1=SPersonne(" + sp1.Nom + "," + sp1.Age + ")");
19.        Console.WriteLine("sp2=SPersonne(" + sp2.Nom + "," + sp2.Age + ")");
20.
21.        // un objet op1
22.        CPersonne op1=new CPersonne();
23.        op1.Nom = "paul";
24.        op1.Age = 10;
25.        Console.WriteLine("op1=CPersonne(" + op1.Nom + "," + op1.Age + ")");
26.        // un objet op2
27.        CPersonne op2=op1;
28.        Console.WriteLine("op2=CPersonne(" + op2.Nom + "," + op2.Age + ")");
29.        // op2 est modifié
30.        op2.Nom = "nicole";
31.        op2.Age = 30;
32.        // vérification op1 et op2
33.        Console.WriteLine("op1=CPersonne(" + op1.Nom + "," + op1.Age + ")");
34.        Console.WriteLine("op2=CPersonne(" + op2.Nom + "," + op2.Age + ")");
35.     }
36.  }
37.  // structure SPersonne
38.  struct SPersonne {
39.     public string Nom;
40.     public int Age;
41.  }
42.
43.  // classe CPersonne
44.  class CPersonne {
45.     public string Nom;
46.     public int Age;
47.  }
48.
49. }
```

- lignes 38-41 : une structure avec deux champs publics : N*om*, A*ge*
- lignes 44-47 : une classe avec deux champs publics : N*om*, A*ge*

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

```
1.  sp1=SPersonne(paul,10)
2.  sp2=SPersonne(paul,10)
3.  sp1=SPersonne(paul,10)
4.  sp2=SPersonne(nicole,30)
5.  op1=CPersonne(paul,10)
6.  op2=CPersonne(paul,10)
7.  op1=CPersonne(nicole,30)
8.  op2=CPersonne(nicole,30)
```

Là où précédemment on utilisait une classe *Personne*, nous utilisons maintenant une structure *SPersonne* :

```
1.   struct SPersonne {
2.      public string Nom;
3.      public int Age;
4.   }
```

La structure n'a ici pas de constructeur. Elle pourrait en avoir un comme nous le montrerons plus loin. Par défaut, elle dispose toujours du constructeur sans paramètres, ici *SPersonne()*.

- ligne 7 du code : la déclaration

```
SPersonne sp1;
```

est équivalente à l'instruction :

```
SPersonne sp1=new Spersonne();
```

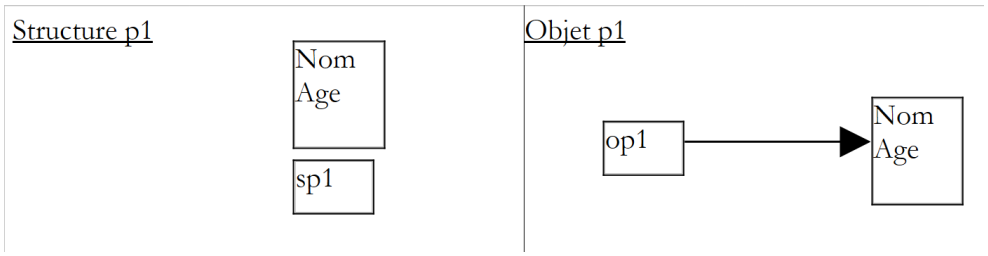
Une structure (Nom,Age) est créée et la **valeur** de sp1 est cette **structure elle-même**. Dans le cas de la classe, la création de l'objet (Nom,Age) doit se faire explicitement par l'opérateur **new** (ligne 22) :

```
CPersonne op1=new CPersonne();
```

L'instruction précédente crée un objet *CPersonne* (grosso modo l'équivalent de notre structure) et la **valeur** de *p1* est alors **l'adresse** (la référence) de cet objet.

Résumons

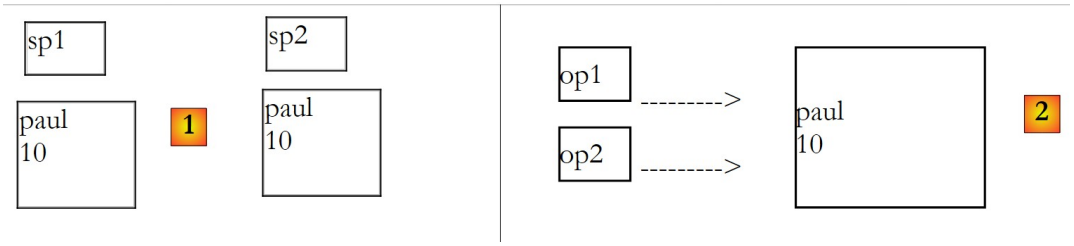
- dans le cas de la structure, la **valeur** de sp1 est la structure elle-même
- dans le cas de la classe, la **valeur** de op1 est l'**adresse** de l'objet créé



Lorsque dans le programme on écrit ligne 12 :

```
SPersonne sp2 = sp1;
```

une nouvelle structure sp2(Nom,Age) est créée et initialisée avec la valeur de s*p1,* donc la structure elle-même.



La structure de sp1 est **dupliquée** dans sp2 [1]. C'est une recopie de valeur. Considérons maintenant l'instruction, ligne 27 :

```
CPersonne op2=op1;
```

Dans le cas des classes, la valeur de op1 est recopiée dans op2, mais comme cette valeur est en fait l'adresse de l'objet, celui-ci n'est pas dupliqué [2].

Dans le cas de la structure [1], si on modifie la valeur de *sp2* on ne modifie pas la valeur de *sp1*, ce que montre le programme. Dans le cas de l'objet [2], si on modifie l'objet pointé par *op2*, celui pointé par *op1* 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 :

```
1.  using System;
2.
3.  namespace Chap2 {
4.
5.   // structure SPersonne
6.   struct SPersonne {
7.      // attributs privés
8.      private string nom;
9.      private int age;
10.
11.     // propriétés
12.     public string Nom {
13.        get { return nom; }
14.        set { nom = value; }
15.     }//nom
16.
17.     public int Age {
18.        get { return age; }
19.        set { age = value; }
20.     }//age
21.
22.     // Constructeur
23.     public SPersonne(string nom, int age) {
24.        this.nom = nom;
25.        this.age = age;
26.     }//constructeur
27.
28.     // ToString
29.     public override string ToString() {
30.        return "SPersonne(" + Nom + "," + Age + ")";
31.     }//ToString
32.  }//structure
33. }//namespace
```

- lignes 8-9 : deux champs privés
- lignes 12-20 : les propriétés publiques associées
- lignes 23-26 : on définit un constructeur. A noter que le constructeur sans paramètres *SPersonne*() est toujours présent et n'a pas à être déclaré. Sa déclaration est refusée par le compilateur. Dans le constructeur des lignes 23-26, on pourrait être tenté d'initialiser les champs privés *nom*, *age* via leurs propriétés publiques *Nom*, *Age*. C'est refusé par le compilateur. Les méthodes de la structure ne peuvent être utilisées lors de la construction de celle-ci.
- lignes 29-31 : redéfinition de la méthode *ToString*.

Un programme de test pourrait être le suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program1 {
5.      static void Main(string[] args) {
6.         // une personne p1
7.         SPersonne p1=new SPersonne();
8.         p1.Nom="paul";
9.         p1.Age= 10;
10.        Console.WriteLine("p1={0}",p1);
11.        // une personne p2
12.        SPersonne p2 = p1;
13.        Console.WriteLine("p2=" + p2);
14.        // p2 est modifié
15.        p2.Nom = "nicole";
```

```
16.         p2.Age = 30;
17.         // vérification p1 et p2
18.         Console.WriteLine("p1=" + p1);
19.         Console.WriteLine("p2=" + p2);
20.         // une personne p3
21.         SPersonne p3 = new SPersonne("amandin", 18);
22.         Console.WriteLine("p3=" + p3);
23.         // une personne p4
24.         SPersonne p4 = new SPersonne { Nom = "x", Age = 10 };
25.         Console.WriteLine("p4=" + p4);
26.     }
27.  }
28. }
```

- ligne 7 : on est obligés d'utiliser explicitement le constructeur sans paramètres, ceci parce qu'il existe un autre constructeur dans la structure. Si la structure n'avait eu aucun constructeur, l'instruction

```
        SPersonne p1;
```

  aurait suffi pour créer une structure vide.
- lignes 8-9 : la structure est initialisée via ses propriétés publiques
- ligne 10 : la méthode *p1.ToString* va être utilisée dans le *WriteLine*.
- ligne 21 : création d'une structure avec le constructeur *SPersonne(string,int)*
- ligne 24 : création d'une structure avec le constructeur sans paramètres *SPersonne()* avec, entre accolades, initialisation des champs privés via leurs propriétés publiques.

On obtient les résultats d'exécution suivants :

```
1.  p1=SPersonne(paul,10)
2.  p2=SPersonne(paul,10)
3.  p1=SPersonne(paul,10)
4.  p2=SPersonne(nicole,30)
5.  p3=SPersonne(amandin,18)
6.  p4=SPersonne(x,10)
```

La seule différence notable ici entre structure et classe, c'est qu'avec une classe les objets *p1* et *p2* auraient pointé sur le même objet à la fin du programme.

4.6 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* :

```
public interface System.Collections.IEnumerator
{

    // Properties
    Object Current { get; }

    // Methods
    bool MoveNext();
    void Reset();
}
```

Les propriétés et méthodes de l'interface ne sont définies que par leurs signatures. Elles ne sont pas implémentées (n'ont pas de code). Ce sont les classes qui implémentent l'interface qui donnent du code aux méthodes et propriétés de l'interface.

```
1.  public class C : IEnumerator{
2.   ...
3.   Object Current{ get {...}}
4.   bool MoveNext{...}
5.   void Reset(){...}
6.  }
```

- ligne 1 : la classe C implémente la classe *IEnumerator*. On notera que le signe : utilisé pour l'implémentation d'une interface est le même que celui utilisé pour la dérivation d'une classe.
- lignes 3-5 : l'implémentation des méthodes et propriétés de l'interface *IEnumerator*.

Considérons l'interface suivante :

```
1.  namespace Chap2 {
2.   public interface IStats {
3.      double Moyenne { get; }
4.      double EcartType();
5.   }
6.  }
```

L'interface *IStats* présente :
- une propriété en lecture seule *Moyenne* : pour calculer la moyenne d'une série de valeurs
- une méthode *EcartType* : pour en calculer l'écart-type

On notera qu'il n'est nulle part précisé de quelle série de valeurs il s'agit. Il peut s'agir de la moyenne des notes d'une classe, de la moyenne mensuelle des ventes d'un produit particulier, de la température moyenne dans un lieu donné, ... C'est le principe des interfaces : on suppose l'existence de méthodes dans l'objet mais pas celle de données particulières.

Une première classe d'implémentation de l'interface *IStats* pourrait une classe servant à mémoriser les notes des élèves d'une classe dans une matière donnée. Un élève serait caractérisé par la structure *Elève* suivante :

```
1.   public struct Elève {
2.      public string Nom { get; set; }
3.      public string Prénom { get; set; }
4.  }//Elève
```

L'élève serait identifié par son nom et son prénom. Lignes 2-3, on trouve les propriétés automatiques pour ces deux attributs.

Une note serait caractérisée par la structure *Note* suivante :

```
1.   public struct Note {
2.      public Elève Elève { get; set; }
3.      public double Valeur { get; set; }
4.  }//Note
```

La note serait identifiée par l'élève noté et la note elle-même. Lignes 2-3, on trouve les propriétés automatiques pour ces deux attributs.

Les notes de tous les élèves dans une matière donnée sont rassemblées dans la classe *TableauDeNotes* suivante :

```
1.  using System;
2.  using System.Text;
3.
4.  namespace Chap2 {
5.
6.   public class TableauDeNotes : IStats {
7.      // attributs
8.      public string Matière { get; set; }
9.      public Note[] Notes { get; set; }
10.     public double Moyenne { get; private set; }
11.     private double ecartType;
12.
13.     // constructeur
14.     public TableauDeNotes(string matière, Note[] notes) {
15.        // mémorisation via les propriétés publiques
16.        Matière = matière;
17.        Notes = notes;
18.        // calcul de la moyenne des notes
19.        double somme = 0;
20.        for (int i = 0; i < Notes.Length; i++) {
21.           somme += Notes[i].Valeur;
22.        }
23.        if (Notes.Length != 0) Moyenne = somme / Notes.Length;
24.        else Moyenne = -1;
25.        // écart-type
26.        double carrés = 0;
27.        for (int i = 0; i < Notes.Length; i++) {
```

```
28.           carrés += Math.Pow((Notes[i].Valeur - Moyenne), 2);
29.        }//for
30.        if (Notes.Length != 0)
31.           ecartType = Math.Sqrt(carrés / Notes.Length);
32.        else ecartType = -1;
33.     }//constructeur
34.
35.     public double EcartType() {
36.        return ecartType;
37.     }
38.
39.     // ToString
40.     public override string ToString() {
41.        StringBuilder valeur = new StringBuilder(String.Format("matière={0}, notes=(", Matière));
42.        int i;
43.        // on concatène toutes les notes
44.        for (i = 0; i < Notes.Length-1; i++) {
45. valeur.Append("[").Append(Notes[i].Elève.Prénom).Append(",").Append(Notes[i].Elève.Nom).Append(","
    ).Append(Notes[i].Valeur).Append("],");
46.        };
47.        //dernière note
48.        if (Notes.Length != 0) {
49. valeur.Append("[").Append(Notes[i].Elève.Prénom).Append(",").Append(Notes[i].Elève.Nom).Append(","
    ).Append(Notes[i].Valeur).Append("]");
50.        }
51.        valeur.Append(")");
52.        // fin
53.        return valeur.ToString();
54.     }//ToString
55.
56.  }//classe
57. }
```

- ligne 6 : la classe *TableauDeNotes* implémente l'interface *IStats*. Elle doit donc implémenter la propriété *Moyenne* et la méthode *EcartType*. Celles-ci sont implémentées lignes 10 (*Moyenne*) et 35-37 (*EcartType*)
- lignes 8-10 : trois propriétés automatiques
- ligne 8 : la matière dont l'objet mémorise les notes
- ligne 9 : le tableau des notes des élèves (Elève, Note)
- ligne 10 : la moyenne des notes - propriété implémentant la propriété *Moyenne* de l'interface IStats.
- ligne 11 : champ mémorisant l'écart-type des notes - la méthode *get* associée *EcartType* des lignes 35-37 implémente la méthode *EcartType* de l'interface *IStats*.
- ligne 9 : les notes sont mémorisées dans un tableau. Celui-ci est transmis lors de la construction de la classe *TableauDeNotes* au constructeur des lignes 14-33.
- lignes 14-33 : le constructeur. On suppose ici que les notes transmises au constructeur ne bougeront plus par la suite. Aussi utilise-t-on le constructeur pour calculer tout de suite la moyenne et l'écart-type de ces notes et les mémoriser dans les champs des lignes 10-11. La moyenne est mémorisée dans le champ privé sous-jacent à la propriété automatique *Moyenne* de la ligne 10 et l'écart-type dans le champ privé de la ligne 11.
- ligne 10 : la méthode *get* de la propriété automatique *Moyenne* rendra le champ privé sous-jacent.
- lignes 35-37 : la méthode *EcartType* rend la valeur du champ privé de la ligne 11.

Il y a quelques subtilités dans ce code :

- ligne 23 : la méthode *set* de la propriété *Moyenne* est utilisée pour faire l'affectation. Cette méthode a été déclarée privée ligne 10 afin que l'affectation d'une valeur à la propriété *Moyenne* ne soit possible qu'à l'intérieur de la classe.
- lignes 40-54 : utilisent un objet *StringBuilder* pour construire la chaîne représentant l'objet *TableauDeNotes* afin d'améliorer les performances. On peut noter que la lisibilité du code en pâtit beaucoup. C'est le revers de la médaille.

Dans la classe précédente, les notes étaient enregistrées dans un tableau. Il n'était pas possible d'ajouter une nouvelle note après construction de l'objet *TableauDeNotes*. Nous proposons maintenant une seconde implémentation de l'interface *IStats*, appelée *ListeDeNotes*, où cette fois les notes seraient enregistrées dans une liste, avec possibilité d'ajouter des notes après construction initiale de l'objet *ListeDeNotes*.

Le code de la classe *ListeDeNotes* est le suivant :

```
1.  using System;
2.  using System.Text;
3.  using System.Collections.Generic;
4.
5.  namespace Chap2 {
```

```
6.
7.   public class ListeDeNotes : IStats {
8.      // attributs
9.      public string Matière { get; set; }
10.     public List<Note> Notes { get; set; }
11.     public double moyenne = -1;
12.     public double ecartType = -1;
13.
14.     // constructeur
15.     public ListeDeNotes(string matière, List<Note> notes) {
16.        // mémorisation via les propriétés publiques
17.        Matière = matière;
18.        Notes = notes;
19.     }//constructeur
20.
21.     // ajout d'une note
22.     public void Ajouter(Note note) {
23.        // ajout de la note
24.        Notes.Add(note);
25.        // moyenne et écart type réinitialisés
26.        moyenne = -1;
27.        ecartType = -1;
28.     }
29.
30.     // ToString
31.     public override string ToString() {
32.        StringBuilder valeur = new StringBuilder(String.Format("matière={0}, notes=(", Matière));
33.        int i;
34.        // on concatène toutes les notes
35.        for (i = 0; i < Notes.Count - 1; i++) {
36. valeur.Append("[").Append(Notes[i].Elève.Prénom).Append(",").Append(Notes[i].Elève.Nom).Append(","
    ).Append(Notes[i].Valeur).Append("],");
37.        };
38.        //dernière note
39.        if (Notes.Count != 0) {
40. valeur.Append("[").Append(Notes[i].Elève.Prénom).Append(",").Append(Notes[i].Elève.Nom).Append(","
    ).Append(Notes[i].Valeur).Append("]");
41.        }
42.        valeur.Append(")");
43.        // fin
44.        return valeur.ToString();
45.     }//ToString
46.
47.     // moyenne des notes
48.     public double Moyenne {
49.        get {
50.           if (moyenne != -1) return moyenne;
51.           // calcul de la moyenne des notes
52.           double somme = 0;
53.           for (int i = 0; i < Notes.Count; i++) {
54.              somme += Notes[i].Valeur;
55.           }
56.           // on rend la moyenne
57.           if (Notes.Count != 0) moyenne = somme / Notes.Count;
58.           return moyenne;
59.        }
60.     }
61.
62.     public double EcartType() {
63.        // écart-type
64.        if (ecartType != -1) return ecartType;
65.        // moyenne
66.        double moyenne = Moyenne;
67.        double carrés = 0;
68.        for (int i = 0; i < Notes.Count; i++) {
69.           carrés += Math.Pow((Notes[i].Valeur - moyenne), 2);
70.        }//for
71.        // on rend l'écart type
72.        if (Notes.Count != 0)
73.           ecartType = Math.Sqrt(carrés / Notes.Count);
74.        return ecartType;
75.     }
76.  }//classe
77. }
```

- ligne 7 : la classe *ListeDeNotes* implémente l'interface *IStats*

- ligne 10 : les notes sont mises maintenant dans une liste plutôt qu'un tableau
- ligne 11 : la propriété automatique *Moyenne* de la classe *TableauDeNotes* a été abandonnée ici au profit d'un champ privé *moyenne*, ligne 11, associé à la propriété publique en lecture seule *Moyenne* des lignes 48-60
- lignes 22-28 : on peut désormais ajouter une note à celles déjà mémorisées, ce qu'on ne pouvait pas faire précédemment.
- lignes 15-19 : du coup, la moyenne et l'écart-type ne sont plus calculés dans le constructeur mais dans les méthodes de l'interface elles-mêmes : *Moyenne* (lignes 48-60) et *EcartType* (62-76). Le recalcul n'est cependant relancé que si la moyenne et l'écart-type sont différents de -1 (lignes 50 et 64).

Une classe de test pourrait être la suivante :

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace Chap2 {
5.   class Program1 {
6.      static void Main(string[] args) {
7.         // qqs élèves & notes d'anglais
8.         Elève[] élèves1 =  { new Elève { Prénom = "Paul", Nom = "Martin" }, new Elève { Prénom =
    "Maxime", Nom = "Germain" }, new Elève { Prénom = "Berthine", Nom = "Samin" } };
9.         Note[] notes1 = { new Note { Elève = élèves1[0], Valeur = 14 }, new Note { Elève =
    élèves1[1], Valeur = 16 }, new Note { Elève = élèves1[2], Valeur = 18 } };
10.        // qu'on enregistre dans un objet TableauDeNotes
11.        TableauDeNotes anglais = new TableauDeNotes("anglais", notes1);
12.        // affichage moyenne et écart-type
13.        Console.WriteLine("{2}, Moyenne={0}, Ecart-type={1}", anglais.Moyenne, anglais.EcartType(),
    anglais);
14.        // on met les élèves et la matière dans un objet ListeDeNotes
15.        ListeDeNotes français = new ListeDeNotes("français", new List<Note>(notes1));
16.        // affichage moyenne et écart-type
17.        Console.WriteLine("{2}, Moyenne={0}, Ecart-type={1}", français.Moyenne,
    français.EcartType(), français);
18.        // on rajoute une note
19.        français.Ajouter(new Note { Elève = new Elève { Prénom = "Jérôme", Nom = "Jaric" }, Valeur
    = 10 });
20.        // affichage moyenne et écart-type
21.        Console.WriteLine("{2}, Moyenne={0}, Ecart-type={1}", français.Moyenne,
    français.EcartType(), français);
22.     }
23.  }
24. }
```

- ligne 8 : création d'un tableau d'élèves avec utilisation du constructeur sans paramètres et initialisation via les propriétés publiques
- ligne 9 : création d'un tableau de notes selon la même technique
- ligne 11 : un objet *TableauDeNotes* dont on calcule la moyenne et l'écart-type ligne 13
- ligne 15 : un objet *ListeDeNotes* dont on calcule la moyenne et l'écart-type ligne 17. La classe *List<Note>* a un constructeur admettant un objet implémentant l'interface *IEnumerable<Note>*. Le tableau *notes1* implémente cette interface et peut être utilisé pour construire l'objet *List<Note>*.
- ligne 19 : ajout d'une nouvelle note
- ligne 21 : recalcul de la moyenne et écart-type

Les résultats de l'exécution sont les suivants :

```
1.  matière=anglais, notes=([Paul,Martin,14],[Maxime,Germain,16],[Berthine,Samin,18]), Moyenne=16,
    Ecart-type=1,63299316185545
2.  matière=français, notes=([Paul,Martin,14],[Maxime,Germain,16],[Berthine,Samin,18]), Moyenne=16,
    Ecart-type=1,63299316185545
3.  matière=français, notes=([Paul,Martin,14],[Maxime,Germain,16],[Berthine,Samin,18],
    [Jérôme,Jaric,10]), Moyenne=14,5, Ecart-type=2,95803989154981
```

Dans l'exemple précédent, deux classes implémentent l'interface *IStats*. Ceci dit, l'exemple ne fait pas apparaître l'intérêt de l'interface *IStats*. Réécrivons le programme de test de la façon suivante :

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace Chap2 {
5.   class Program2 {
6.      static void Main(string[] args) {
7.         // qqs élèves & notes d'anglais
```

```
8.          Elève[] élèves1 =  { new Elève { Prénom = "Paul", Nom = "Martin" }, new Elève { Prénom =
    "Maxime", Nom = "Germain" }, new Elève { Prénom = "Berthine", Nom = "Samin" } };
9.          Note[] notes1 = { new Note { Elève = élèves1[0], Valeur = 14 }, new Note { Elève =
    élèves1[1], Valeur = 16 }, new Note { Elève = élèves1[2], Valeur = 18 } };
10.         // qu'on enregistre dans un objet TableauDeNotes
11.         TableauDeNotes anglais = new TableauDeNotes("anglais", notes1);
12.         // affichage moyenne et écart-type
13.         AfficheStats(anglais);
14.         // on met les élèves et la matière dans un objet ListeDeNotes
15.         ListeDeNotes français = new ListeDeNotes("français", new List<Note>(notes1));
16.         // affichage moyenne et écart-type
17.         AfficheStats(français);
18.         // on rajoute une note
19.         français.Ajouter(new Note { Elève = new Elève { Prénom = "Jérôme", Nom = "Jaric" }, Valeur
    = 10 });
20.         // affichage moyenne et écart-type
21.         AfficheStats(français);
22.     }
23.
24.     // affichage moyenne et écart-type d'un type IStats
25.     static void AfficheStats(IStats valeurs) {
26.         Console.WriteLine("{2}, Moyenne={0}, Ecart-type={1}", valeurs.Moyenne, valeurs.EcartType(),
    valeurs);
27.     }
28.  }
29. }
```

- lignes 25-27 : la méthode statique *AfficheStats* reçoit pour paramètre un type *IStats*, donc un type Interface. **Cela signifie que le paramètre effectif peut être tout objet implémentant l'interface *IStats***. Quand on utilise une donnée ayant le type d'une interface, cela signifie qu'on n'utilisera que les méthodes de l'interface implémentées par la donnée. On fait abstraction du reste. On a là une propriété proche du polymorphisme vu pour les classes. Si un ensemble de classes **Ci** non liées entre-elles par héritage (donc on ne peut utiliser le polymorphisme de l'héritage) présente un ensemble de méthodes de même signature, il peut être intéressant de regrouper ces méthodes dans une interface **I** qu'implémenteraient toutes les classes concernées. Des instances de ces classes **Ci** peuvent alors être utilisées comme paramètres effectifs de fonctions admettant un paramètre formel 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**.
- ligne 13 : la méthode *AfficheStats* est appelée avec un type *TableauDeNotes* qui implémente l'interface *IStats*
- ligne 17 : idem avec un type *ListeDeNotes*

Les résultats de l'exécution sont identiques à ceux de la précédente.

Une variable **peut être du type d'une interface**. Ainsi, on peut écrire :

```
1.  IStats stats1=new TableauDeNotes(...);
2.  ...
3.  stats1=new ListeDeNotes(...);
```

La déclaration de la ligne 1 indique que ***stats1* est l'instance d'une classe implémentant l'interface *IStats***. Cette déclaration implique que le compilateur ne permettra l'accès dans *stats1* qu'aux méthodes de l'interface : la propriété *Moyenne* et la méthode *EcartType*.

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

```
public class ClasseDérivée:ClasseDeBase,I1,I2,..,In{
...
}
```

où les I_i sont des interfaces.

4.7 Les classes abstraites

Une classe abstraite est une classe qu'on ne peut instancier. Il faut créer des classes dérivées qui elles pourront être instanciées.

On peut utiliser des classes abstraites pour factoriser le code d'une lignée de classes. Examinons le cas suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
```

```
4.  abstract class Utilisateur {
5.     // champs
6.     private string login;
7.     private string motDePasse;
8.     private string role;
9.
10.    // constructeur
11.    public Utilisateur(string login, string motDePasse) {
12.       // on enregistre les informations
13.       this.login = login;
14.       this.motDePasse = motDePasse;
15.       // on identifie l'utilisateur
16.       role=identifie();
17.       // identifié ?
18.       if (role == null) {
19.          throw new ExceptionUtilisateurInconnu(String.Format("[{0},{1}]", login, motDePasse));
20.       }
21.    }
22.
23.    // toString
24.    public override string ToString() {
25.       return String.Format("Utilisateur[{0},{1},{2}]", login, motDePasse, role);
26.    }
27.
28.    // identifie
29.    abstract public string identifie();
30. }
31. }
```

- lignes 11-21 : le contructeur de la classe *Utilisateur*. Cette classe mémorise des informations sur l'utilisateur d'une application web. Celle-ci a divers types d'utilisateurs authentifiés par un login / mot de passe (lignes 6-7). Ces deux informations sont vérifiées auprès d'un service LDAP pour certains utilisateurs, auprès d'un SGBD pour d'autres, etc...
- lignes 13-14 : les informations d'authentification sont mémorisées
- ligne 16 : elles sont vérifiées par une méthode **identifie**. Parce que la méthode d'identification n'est pas connue, elle est déclarée abstraite ligne 29 avec le mot clé **abstract**. La méthode *identifie* rend une chaîne de caractères précisant le rôle de l'utilisateur (en gros ce qu'il a le droit de faire). Si cette chaîne est le pointeur *null*, une exception est lancée ligne 19.
- ligne 4 : parce qu'elle a une méthode abstraite, la classe elle-même est déclarée abstraite avec le mot clé **abstract**.
- ligne 29 : la méthode abstraite **identifie** n'a pas de définition. Ce sont les classes dérivées qui lui en donneront une.
- lignes 24-26 : la méthode *ToString* qui identifie une instance de la classe.

On suppose ici que le développeur veut avoir la maîtrise de la construction des instances de la classe *Utilisateur* et des classes dérivées, peut-être parce qu'il veut être sûr qu'une exception d'un certain type est lancée si l'utilisateur n'est pas reconnu (ligne 19). Les classes dérivées pourront s'appuyer sur ce constructeur. Elles devront pour cela fournir la méthode **identifie**.

La classe *ExceptionUtilisateurInconnu* est la suivante :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class ExceptionUtilisateurInconnu : Exception {
5.      public ExceptionUtilisateurInconnu(string message) : base(message){
6.      }
7.   }
8.  }
```

- ligne 3 : elle dérive de la classe *Exception*
- lignes 4-6 : elle n'a qu'un unique constructeur qui admet pour paramètre un message d'erreur. Celui-ci est passé à la classe parent (ligne 5) qui a ce même constructeur.

Nous dérivons maintenant la classe *Utilisateur* dans la classe fille *Administrateur* :

```
1.  namespace Chap2 {
2.   class Administrateur : Utilisateur {
3.      // constructeur
4.      public Administrateur(string login, string motDePasse)
5.         : base(login, motDePasse) {
6.      }
7.
8.      // identifie
9.      public override string identifie() {
10.        // identification LDAP
```

```
11.         // ...
12.         return "admin";
13.      }
14.   }
15. }
```

- lignes 4-6 : le constructeur se contente de passer à sa classe parent les paramètres qu'il reçoit
- lignes 9-12 : la méthode **identifie** de la classe *Administrateur*. On suppose qu'un administrateur est identifié par un système LDAP. Cette méthode redéfinit la méthode **identifie** de sa classe parent. Parce qu'elle redéfinit une méthode **abstraite**, il est inutile de mettre le mot clé *override*.

Nous dérivons maintenant la classe *Utilisateur* dans la classe fille *Observateur* :

```
1.  namespace Chap2 {
2.   class Observateur : Utilisateur{
3.      // constructeur
4.      public Observateur(string login, string motDePasse)
5.         : base(login, motDePasse) {
6.      }
7.
8.      //identifie
9.      public override string identifie() {
10.        // identification SGBD
11.        // ...
12.        return "observateur";
13.     }
14.
15.  }
16. }
```

- lignes 4-6 : le constructeur se contente de passer à sa classe parent les paramètres qu'il reçoit
- lignes 9-13 : la méthode **identifie** de la classe *Observateur*. On suppose qu'un observateur est identifié par vérification de ses données d'identification dans une base de données.

Au final, les objets *Administrateur* et *Observateur* sont instanciés par le même constructeur, celui de la classe parent *Utilisateur*. Ce constructeur va utiliser la méthode **identifie** que ces classes fournissent.

Une troisième classe *Inconnu* dérive également de la classe *Utilisateur* :

```
1.  namespace Chap2 {
2.   class Inconnu : Utilisateur{
3.
4.      // constructeur
5.      public Inconnu(string login, string motDePasse)
6.         : base(login, motDePasse) {
7.      }
8.
9.      //identifie
10.     public override string identifie() {
11.        // utilisateur pas connu
12.        // ...
13.        return null;
14.     }
15.
16.  }
17. }
```

- ligne 13 : la méthode *identifie* rend le pointeur *null* pour indiquer que l'utilisateur n'a pas été reconnu.

Un programme de test pourrait être le suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         Console.WriteLine(new Observateur("observer","mdp1"));
7.         Console.WriteLine(new Administrateur("admin", "mdp2"));
8.         try {
9.            Console.WriteLine(new Inconnu("xx", "yy"));
10.        } catch (ExceptionUtilisateurInconnu e) {
```

```
11.           Console.WriteLine("Utilisateur non connu : "+ e.Message);
12.       }
13.     }
14.  }
15. }
```

On notera que lignes 6, 7 et 9, c'est la méthode **[Utilisateur].ToString()** qui sera utilisée par la méthode W*riteLine*.

Les résultats de l'exécution sont les suivants :

```
1.  Utilisateur[observer,mdp1,observateur]
2.  Utilisateur[admin,mdp2,admin]
3.  Utilisateur non connu : [xx,yy]
```

4.8 Les classes, interfaces, méthodes génériques

Supposons qu'on veuille écrire une méthode permutant deux nombres entiers. Cette méthode pourrait être la suivante :

```
1.        public static void Echanger1(ref int value1, ref int value2){
2.           // on échange les références value1 et value2
3.           int temp = value2;
4.           value2 = value1;
5.           value1 = temp;
6.  }
```

Maintenant, si on voulait permuter deux références sur des objets *Personne*, on écrirait :

```
1.      public static void Echanger2(ref Personne value1, ref Personne value2){
2.         // on échange les références value1 et value2
3.         Personne temp = value2;
4.         value2 = value1;
5.         value1 = temp;
6.  }
```

Ce qui différencie les deux méthodes, c'est le type T des paramètres : *int* dans *Echanger1*, *Personne* dans *Echanger2*. Les classes et interfaces génériques répondent au besoin de méthodes qui ne diffèrent que par le type de certains de leurs paramètres.

Avec une classe générique, la méthode *Echanger* pourrait être réécrite de la façon suivante :

```
1.  namespace Chap2 {
2.   class Generic1<T> {
3.      public static void Echanger(ref T value1, ref T value2){
4.         // on échange les références value1 et value2
5.         T temp = value2;
6.         value2 = value1;
7.         value1 = temp;
8.      }
9.   }
10. }
```

- ligne 2 : la classe *Generic1* est paramétrée par un type noté T. On peut lui donner le nom que l'on veut. Ce type T est ensuite réutilisé dans la classe aux lignes 3 et 5. On dit que la classe *Generic1* est une classe générique.
- ligne 3 : définit les deux références sur un type T à permuter
- ligne 5 : la variable temporaire *temp* a le type T.

Un programme de test de la classe pourrait être le suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main(string[] args) {
6.         // int
7.         int i1 = 1, i2 = 2;
8.         Generic1<int>.Echanger(ref i1, ref i2);
9.         Console.WriteLine("i1={0},i2={1}", i1, i2);
10.        // string
11.        string s1 = "s1", s2 = "s2";
```

```
12.         Generic1<string>.Echanger(ref s1, ref s2);
13.         Console.WriteLine("s1={0},s2={1}", s1, s2);
14.         // Personne
15.         Personne p1 = new Personne("jean", "clu", 20), p2 = new Personne("pauline", "dard", 55);
16.         Generic1<Personne>.Echanger(ref p1, ref p2);
17.         Console.WriteLine("p1={0},p2={1}", p1, p2);
18.
19.     }
20.  }
21. }
```

- ligne 8 : lorsqu'on utilise une classe générique paramétrée par des types T1, T2, ... ces derniers doivent être "instanciés". Ligne 8, on utilise la méthode statique *Echanger* du type *Generic1<int>* pour indiquer que les références passées à la méthode *Echanger* sont de type *int*.
- ligne 12 : on utilise la méthode statique *Echanger* du type *Generic1<string>* pour indiquer que les références passées à la méthode *Echanger* sont de type *string*.
- ligne 16 : on utilise la méthode statique *Echanger* du type *Generic1<Personne>* pour indiquer que les références passées à la méthode *Echanger* sont de type *Personne*.

Les résultats de l'exécution sont les suivants :

```
1.  i1=2,i2=1
2.  s1=s2,s2=s1
3.  p1=[pauline, dard, 55],p2=[jean, clu, 20]
```

La méthode *Echanger* aurait pu également être écrite de la façon suivante :

```
1.  namespace Chap2 {
2.   class Generic2 {
3.      public static void Echanger<T>(ref T value1, ref T value2){
4.         // on échange les références value1 et value2
5.         T temp = value2;
6.         value2 = value1;
7.         value1 = temp;
8.      }
9.   }
10. }
```

- ligne 2 : la classe *Generic2* n'est plus générique
- ligne 3 : la méthode statique *Echanger* est générique

Le programme de test devient alors le suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program2 {
5.      static void Main(string[] args) {
6.         // int
7.         int i1 = 1, i2 = 2;
8.         Generic2.Echanger<int>(ref i1, ref i2);
9.         Console.WriteLine("i1={0},i2={1}", i1, i2);
10.        // string
11.        string s1 = "s1", s2 = "s2";
12.        Generic2.Echanger<string>(ref s1, ref s2);
13.        Console.WriteLine("s1={0},s2={1}", s1, s2);
14.        // Personne
15.        Personne p1 = new Personne("jean", "clu", 20), p2 = new Personne("pauline", "dard", 55);
16.        Generic2.Echanger<Personne>(ref p1, ref p2);
17.        Console.WriteLine("p1={0},p2={1}", p1, p2);
18.     }
19.  }
20. }
```

- lignes 8, 12 et 16 : on appelle la méthode *Echanger* en précisant dans <> le type des paramètres. En fait, le compilateur est capable de déduire d'après le type des paramètres effectifs, la variante de la méthode *Echanger* à utiliser. Aussi, l'écriture suivante est-elle légale :

```
1.         Generic2.Echanger(ref i1, ref i2);
2.  ...
3.         Generic2.Echanger(ref s1, ref s2);
```

```
4.  ...
5.          Generic2.Echanger(ref p1, ref p2);
```

Lignes 1, 3 et 5 : la variante de la méthode *Echanger* appelée n'est plus précisée. Le compilateur est capable de la déduire de la nature des paramètres effectifs utilisés.

On peut mettre des contraintes sur les paramètres génériques :

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified. See Using Nullable Types (C# Programming Guide) for more information.
where T : class	The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the **new()** constraint must be specified last.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U. This is called a naked type constraint.

Considérons la nouvelle méthode générique *Echanger* suivante :

```
1.  namespace Chap2 {
2.   class Generic3 {
3.      public static void Echanger<T>(ref T value1, ref T value2) where T : class {
4.         // on échange les références value1 et value2
5.         T temp = value2;
6.         value2 = value1;
7.         value1 = temp;
8.      }
9.   }
10. }
```

- ligne 3 : on exige que le type T soit une référence (classe, interface)

Considérons le programme de test suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program4 {
5.      static void Main(string[] args) {
6.         // int
7.         int i1 = 1, i2 = 2;
8.         Generic3.Echanger<int>(ref i1, ref i2);
9.         Console.WriteLine("i1={0},i2={1}", i1, i2);
10.        // string
11.        string s1 = "s1", s2 = "s2";
12.        Generic3.Echanger(ref s1, ref s2);
13.        Console.WriteLine("s1={0},s2={1}", s1, s2);
14.        // Personne
15.        Personne p1 = new Personne("jean", "clu", 20), p2 = new Personne("pauline", "dard", 55);
16.        Generic3.Echanger(ref p1, ref p2);
17.        Console.WriteLine("p1={0},p2={1}", p1, p2);
18.
19.     }
20.  }
21. }
```

Le compilateur déclare une erreur sur la ligne 8 car le type *int* n'est pas une classe ou une interface, c'est une structure :

```
Generic3.Echanger<int>(ref i1, ref i2);
The type 'int' must be a reference type in order to use it as parameter 'T' in the generic type or method 'Chap2.Generic3.Echanger<T>(ref T, ref T)'
// string
```

Considérons la nouvelle méthode générique *Echanger* suivante :

```
1.  namespace Chap2 {
```

```
2.   class Generic4 {
3.      public static void Echanger<T>(ref T element1, ref T element2) where T : Interface1 {
4.         // on récupère la valeur des 2 éléments
5.         int value1 = element1.Value();
6.         int value2 = element2.Value();
7.         // si 1er élément > 2ième élément, on échange les éléments
8.         if (value1 > value2) {
9.            T temp = element2;
10.           element2 = element1;
11.           element1 = temp;
12.        }
13.     }
14.  }
15. }
```

- ligne 3 : le type T doit implémenter l'interface *Interface1*. Celle-ci a une méthode *Value,* utilisée lignes 5 et 6, qui donne la valeur de l'objet de type T.
- lignes 8-12 : les deux références *element1* et *element2* ne sont échangées que si la valeur de *element1* est supérieure à la valeur de *element2*.

L'interface *Interface1* est la suivante :

```
1.  namespace Chap2 {
2.   interface Interface1 {
3.      int Value();
4.   }
5.  }
```

Elle est implémentée par la classe *Class1* suivante :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap2 {
5.   class Class1 : Interface1 {
6.      // valeur de l'objet
7.      private int value;
8.
9.      // constructeur
10.     public Class1() {
11.        // attente 1 ms
12.        Thread.Sleep(1);
13.        // valeur aléatoire entre 0 et 99
14.        value = new Random(DateTime.Now.Millisecond).Next(100);
15.     }
16.
17.     // accesseur champ privé value
18.     public int Value() {
19.        return value;
20.     }
21.
22.     // état de l'instance
23.     public override string ToString() {
24.        return value.ToString();
25.     }
26.  }
27. }
```

- ligne 5 : *Class1* implémente l'interface *Interface1*
- ligne 7 : la valeur d'une instance de *Class1*
- lignes 10-14 : le champ *value* est initialisé avec une valeur aléatoire entre 0 et 99
- lignes 18-20 : la méthode *Value* de l'interface *Interface1*
- lignes 23-25 : la méthode *ToString* de la classe

L'interface *Interface1* est également implémentée par la classe *Class2* :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Class2 : Interface1 {
5.      // valeurs de l'objet
6.      private int value;
```

```
7.      private String s;
8.
9.      // constructeur
10.     public Class2(String s) {
11.        this.s = s;
12.        value = s.Length;
13.     }
14.
15.     // accesseur champ privé value
16.     public int Value() {
17.        return value;
18.     }
19.
20.     // état de l'instance
21.     public override string ToString() {
22.        return s;
23.     }
24.  }
25. }
```

- ligne 4 : *Class2* implémente l'interface *Interface1*
- ligne 6 : la valeur d'une instance de *Class2*
- lignes 10-13 : le champ *value* est initialisé avec la longueur de la chaîne de caractères passée au constructeur
- lignes 16-18 : la méthode *Value* de l'interface *Interface1*
- lignes 21-22 : la méthode *ToString* de la classe

Un programme de test pourrait être le suivant :

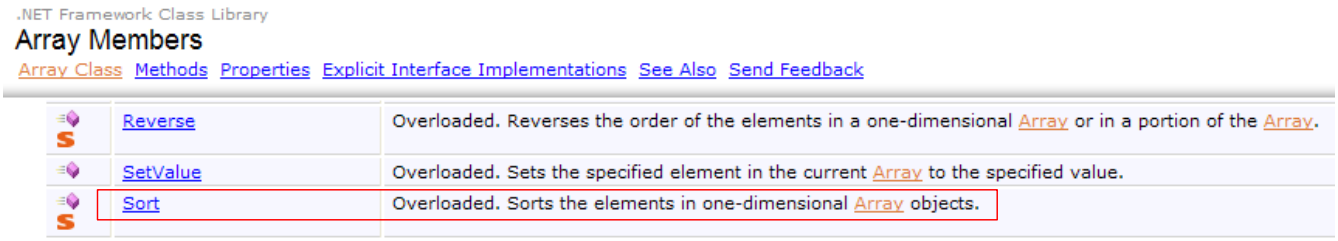
```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program5 {
5.      static void Main(string[] args) {
6.         // échanger des instances de type Class1
7.         Class1 c1, c2;
8.         for (int i = 0; i < 5; i++) {
9.            c1 = new Class1();
10.           c2 = new Class1();
11.           Console.WriteLine("Avant échange --> c1={0},c2={1}", c1, c2);
12.           Generic4.Echanger(ref c1, ref c2);
13.           Console.WriteLine("Après échange --> c1={0},c2={1}", c1, c2);
14.        }
15.        // échanger des instances de type Class2
16.        Class2 c3, c4;
17.        c3 = new Class2("xxxxxxxxxxxxx");
18.        c4 = new Class2("xx");
19.        Console.WriteLine("Avant échange --> c3={0},c4={1}", c3, c4);
20.        Generic4.Echanger(ref c3, ref c4);
21.        Console.WriteLine("Avant échange --> c3={0},c4={1}", c3, c4);
22.     }
23.  }
24. }
```

- lignes 8-14 : des instances de type *Class1* sont échangées
- lignes 16-22 : des instances de type *Class2* sont échangées

Les résultats de l'exécution sont les suivants :

```
1.  Avant échange --> c1=43,c2=79
2.  Après échange --> c1=43,c2=79
3.  Avant échange --> c1=72,c2=56
4.  Après échange --> c1=56,c2=72
5.  Avant échange --> c1=92,c2=75
6.  Après échange --> c1=75,c2=92
7.  Avant échange --> c1=11,c2=47
8.  Après échange --> c1=11,c2=47
9.  Avant échange --> c1=31,c2=67
10. Après échange --> c1=31,c2=67
11. Avant échange --> c3=xxxxxxxxxxxxx,c4=xx
12. Après échange --> c3=xx,c4=xxxxxxxxxxxxx
```

Pour illustrer la notion d'**interface générique**, nous allons trier un tableau de personnes d'abord sur leurs noms, puis sur leurs ages. La méthode qui nous permet de trier un tableau est la méthode statique **Sort** de la classe **Array** :



On rappelle qu'une méthode statique s'utilise en préfixant la méthode par le nom de la classe et non par celui d'une instance de la classe. La méthode **Sort** a différentes signatures (elle est surchargée). Nous utiliserons la signature suivante :

```
public static void Sort<T>(T[] tableau, IComparer<T> comparateur)
```

Sort une **méthode générique** où T désigne un type quelconque. La méthode reçoit deux paramètres :

- **T[] tableau** : le tableau d'éléments de type T à trier
- **IComparer<T> comparateur** : une référence d'objet implémentant l'interface *IComparer<T>*.

IComparer<T> est une **interface générique** définie comme suit :

```
1.  public interface IComparer<T>{
2.   int Compare(T t1, T t2);
3.  }
```

L'interface **IComparer<T>** n'a qu'une unique méthode. La méthode **Compare** :

- reçoit en paramètres deux éléments *t1* et *t2* de type *T*
- rend 1 si t1>t2, 0 si t1==t2, -1 si t1<t2. C'est au développeur de donner une signification aux opérateurs <, ==, >. Par exemple, si p1 et p2 sont deux objets *Personne*, on pourra dire que p1>p2 si le nom de p1 précède le nom de p2 dans l'ordre alphabétique. On aura alors un tri croissant selon le nom des personnes. Si on veut un tri selon l'âge, on dira que p1>p2 si l'âge de p1 est supérieur à l'âge de p2.
- pour avoir un tri dans l'ordre décroissant, il suffit d'inverser les résultats +1 et -1

Nous en savons assez pour trier un tabeau de personnes. Le programme est le suivant :

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace Chap2 {
5.   class Program6 {
6.      static void Main(string[] args) {
7.         // un tableau de personnes
8.         Personne[] personnes1 = { new Personne("claude", "pollon", 25), new Personne("valentine",
    "germain", 35), new Personne("paul", "germain", 32) };
9.         // affichage
10.        Affiche("Tableau à trier", personnes1);
11.        // tri selon le nom
12.        Array.Sort(personnes1, new CompareNoms());
13.        // affichage
14.        Affiche("Tableau après le tri selon les nom et prénom", personnes1);
15.        // tri selon l'âge
16.        Array.Sort(personnes1, new CompareAges());
17.        // affichage
18.        Affiche("Tableau après le tri selon l'âge", personnes1);
19.     }
20.
21.     static void Affiche(string texte, Personne[] personnes) {
22.        Console.WriteLine(texte.PadRight(50, '-'));
23.        foreach (Personne p in personnes) {
24.           Console.WriteLine(p);
25.        }
26.     }
27.  }
28.
29.  // classe de comparaison des noms et prénoms des personnes
30.  class CompareNoms : IComparer<Personne> {
31.     public int Compare(Personne p1, Personne p2) {
```

```
32.        // on compare les noms
33.        int i = p1.Nom.CompareTo(p2.Nom);
34.        if (i != 0)
35.           return i;
36.        // égalité des noms - on compare les prénoms
37.        return p1.Prenom.CompareTo(p2.Prenom);
38.     }
39.  }
40.
41.  // classe de comparaison des ages des personnes
42.  class CompareAges : IComparer<Personne> {
43.     public int Compare(Personne p1, Personne p2) {
44.        // on compare les ages
45.        if (p1.Age > p2.Age)
46.           return 1;
47.        else if (p1.Age == p2.Age)
48.           return 0;
49.        else
50.           return -1;
51.     }
52.  }
53.
54. }
```

- ligne 8 : le tableau de personnes
- ligne 12 : le tri du tableau de personnes selon les nom et prénom. Le 2ième paramètre de la méthode générique *Sort* est une instance d'une classe *CompareNoms* implémentant l'interface générique *IComparer<Personne>*.
- lignes 30-39 : la classe *CompareNoms* implémentant l'interface générique *IComparer<Personne>*.
- lignes 31-38 : implémentation de la méthode générique *int CompareTo(T,T)* de l'interface *IComparer<T>*. La méthode utilise la méthode *String.CompareTo*, présentée page 22, pour comparer deux chaînes de caractères.
- ligne 16 : le tri du tableau de personnes selon les âges. Le 2ième paramètre de la méthode générique *Sort* est une instance d'une classe *CompareAges* implémentant l'interface générique *IComparer<Personne>* et définie lignes 42-51.

Les résultats de l'exécution sont les suivants :

```
1.  Tableau à trier----------------------------------
2.  [claude, pollon, 25]
3.  [valentine, germain, 35]
4.  [paul, germain, 32]
5.  Tableau après le tri selon les nom et prénom------
6.  [paul, germain, 32]
7.  [valentine, germain, 35]
8.  [claude, pollon, 25]
9.  Tableau après le tri selon l'âge------------------
10. [claude, pollon, 25]
11. [paul, germain, 32]
12. [valentine, germain, 35]
```

4.9 Les espaces de noms

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

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

On évite cela en utilisant une clause *using* :

```
using System;
...
```

```
Console.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 et qu'on doit donner les références des différentes *dll* contenant les classes que l'on doit utiliser, cette information peut s'avérer utile. Pour référencer les *dll* nécessaires à la compilation d'une classe, on écrit :

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

où *csc* est le compilateur C#. 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*, e*1.dll* et e*2.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 P*ersonne*, le compilateur ne saura pas s'il doit prendre la classe P*ersonne* de e*1.dll* ou celle de e*2.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 P*ersonne* s'appelleront alors E*1.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
}
```

4.10 Application exemple - V2

On reprend le calcul de l'impôt déjà étudié dans le chapitre précédent page 32 et on le traite maintenant en utilisant des classes et des interfaces. Rappelons le problème :

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 (chiffres 2004 pour revenus 2003) :

- 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 :

4262	0	0
8382	0.0683	291.09
14753	0.1914	1322.92
23888	0.2826	2668.39
38868	0.3738	4846.98
47932	0.4262	6883.66
0	0.4809	9505.54

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=5000 on trouvera la ligne

8382 0.0683 291.09

L'impôt I est alors égal à **0.0683*R - 291.09*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 0.4809 9505.54

ce qui donne l'impôt I=0.4809*R - 9505.54*nbParts.

Tout d'abord, nous définissons une structure capable d'encapsuler une ligne du tableau précédent :

```
1.  namespace Chap2 {
2.   // une tranche d'impôt
3.   struct TrancheImpot {
4.      public decimal Limite { get; set; }
5.      public decimal CoeffR { get; set; }
6.      public decimal CoeffN { get; set; }
7.   }
8.  }
9.
```

Puis nous définissons une interface *IImpot* capable de calculer l'impôt :

```
1.  namespace Chap2 {
2.   interface IImpot {
3.      int calculer(bool marié, int nbEnfants, int salaire);
4.   }
5.  }
```

- ligne 3 : la méthode de calcul de l'impôt à partir de trois données : l'état marié ou non du contribuable, son nombre d'enfants, son salaire

Ensuite, nous définissons une classe abstraite implémentant cette interface :

```
1.  namespace Chap2 {
2.   abstract class AbstractImpot : IImpot {
3.
4.      // les tranches d'impôt nécessaires au calcul de l'impôt
5.      // proviennent d'une source extérieure
6.
7.      protected TrancheImpot[] tranchesImpot;
8.
9.      // calcul de l'impôt
10.     public int calculer(bool marié, int nbEnfants, int salaire) {
11.        // calcul du nombre de parts
12.        decimal nbParts;
13.        if (marié) nbParts = (decimal)nbEnfants / 2 + 2;
14.        else nbParts = (decimal)nbEnfants / 2 + 1;
15.        if (nbEnfants >= 3) nbParts += 0.5M;
16.        // calcul revenu imposable & Quotient familial
17.        decimal revenu = 0.72M * salaire;
18.        decimal QF = revenu / nbParts;
19.        // calcul de l'impôt
20.        tranchesImpot[tranchesImpot.Length - 1].Limite = QF + 1;
21.        int i = 0;
22.        while (QF > tranchesImpot[i].Limite) i++;
23.        // retour résultat
24.        return (int)(revenu * tranchesImpot[i].CoeffR - nbParts * tranchesImpot[i].CoeffN);
25.     }//calculer
26.  }//classe
27.
28. }
```

- ligne 2 : la classe *AbstractImpot* implémente l'interface *IImpot*.
- ligne 7 : les données annuelles du calcul de l'impôt sous forme d'un champ protégé. La classe *AbstractImpot* ne sait pas comment sera initialisé ce champ. Elle en laisse le soin aux classes dérivées. C'est pourquoi elle est déclarée abstraite (ligne 2) afin d'en interdire toute instanciation.
- lignes 10-25 : l'implémentation de la méthode *calculer* de l'interface *IImpot*. Les classes dérivées n'auront pas à réécrire cette méthode. La classe *AbstractImpot* sert ainsi de classe de factorisation des classes dérivées. On y met ce qui est commun à toutes les classes dérivées.

Une classe implémentant l'interface *IImpot* peut être construite en dérivant la classe *AbstractImpot*. C'est ce que nous faisons maintenant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class HardwiredImpot : AbstractImpot {
5.
```

```
6.      // tableaux de données nécessaires au calcul de l'impôt
7.      decimal[] limites = { 4962M, 8382M, 14753M, 23888M, 38868M, 47932M, 0M };
8.      decimal[] coeffR = { 0M, 0.068M, 0.191M, 0.283M, 0.374M, 0.426M, 0.481M };
9.      decimal[] coeffN = { 0M, 291.09M, 1322.92M, 2668.39M, 4846.98M, 6883.66M, 9505.54M };
10.
11.     public HardwiredImpot() {
12.           // création du tableau des tranches d'impôt
13.        tranchesImpot = new TrancheImpot[limites.Length];
14.           // remplissage
15.        for (int i = 0; i < tranchesImpot.Length; i++) {
16.           tranchesImpot[i] = new TrancheImpot { Limite = limites[i], CoeffR = coeffR[i], CoeffN =
    coeffN[i] };
17.           }
18.     }
19.  }// classe
20. }// namespace
```

La classe *HardwiredImpot* définit, lignes 7-9, en dur les données nécessaires au calcul de l'impôt. Son constructeur (lignes 11-18) utilise ces données pour initialiser le champ protégé *tranchesImpot* de la classe mère *AbstractImpot*.

Un programme de test pourait être le suivant :

```
1.  using System;
2.
3.  namespace Chap2 {
4.   class Program {
5.      static void Main() {
6.         // programme interactif de calcul d'Impot
7.         // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
8.         // le programme affiche alors l'Impot à payer
9.
10.        const string syntaxe = "syntaxe : Marié NbEnfants Salaire\n"
11.                    + "Marié : o pour marié, n pour non marié\n"
12.                    + "NbEnfants : nombre d'enfants\n"
13.                    + "Salaire : salaire annuel en F";
14.
15.        // création d'un objet IImpot
16.        IImpot impot = new HardwiredImpot();
17.
18.        // boucle infinie
19.        while (true) {
20.           // on demande les paramètres du calcul de l'impôt
21.           Console.Write("Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire
    ou rien pour arrêter :");
22.           string paramètres = Console.ReadLine().Trim();
23.           // qq chose à faire ?
24.           if (paramètres == null || paramètres == "") break;
25.           // vérification du nombre d'arguments dans la ligne saisie
26.           string[] args = paramètres.Split(null);
27.           int nbParamètres = args.Length;
28.           if (nbParamètres != 3) {
29.              Console.WriteLine(syntaxe);
30.              continue;
31.           }//if
32.           // vérification de la validité des paramètres
33.           // marié
34.           string marié = args[0].ToLower();
35.           if (marié != "o" && marié != "n") {
36.              Console.WriteLine(syntaxe + "\nArgument marié incorrect : tapez o ou n");
37.              continue;
38.           }//if
39.           // nbEnfants
40.           int nbEnfants = 0;
41.           bool dataOk = false;
42.           try {
43.              nbEnfants = int.Parse(args[1]);
44.              dataOk = nbEnfants >= 0;
45.           } catch {
46.           }//if
47.           // donnée correcte ?
48.           if (!dataOk) {
49.              Console.WriteLine(syntaxe + "\nArgument NbEnfants incorrect : tapez un entier positif
    ou nul");
50.              continue;
51.           }
52.           // salaire
```

```
53.           int salaire = 0;
54.           dataOk = false;
55.           try {
56.              salaire = int.Parse(args[2]);
57.              dataOk = salaire >= 0;
58.           } catch {
59.           }//try-catch
60.           // donnée correcte ?
61.           if (!dataOk) {
62.              Console.WriteLine(syntaxe + "\nArgument salaire incorrect : tapez un entier positif
    ou nul");
63.              continue;
64.           }
65.           // les paramètres sont corrects - on calcule l'Impot
66.           Console.WriteLine("Impot=" + impot.calculer(marié == "o", nbEnfants, salaire) + "
    euros");
67.           // contribuable suivant
68.        }//while
69.     }
70.  }
71. }
```

Le programme ci-dessus permet à l'utilisateur de faire des simulations répétées de calcul d'impôt.

- ligne 16 : création d'un objet *impot* implémentant l'interface *IImpot*. Cet objet est obtenu par instanciation d'un type *HardwiredImpot*, un type qui implémente l'interface *IImpot*. On notera qu'on n'a pas donné à la variable *impot*, le type *HardwiredImpot* mais le type *IImpot*. En écrivant cela, on indique qu'on ne s'intéresse qu'à la méthode *calculer* de l'objet *impot* et pas au reste.
- lignes 19-68 : la boucle des simulations de calcul de l'impôt
- ligne 22 : les trois paramètres nécessaires à la méthode *calculer* sont demandés en une seule ligne tapée au clavier.
- ligne 26 : la méthode [chaine].Split(null) permet de décomposer [chaine] en mots. Ceux-ci sont stockés dans un tableau *args*.
- ligne 66 : appel de la méthode *calculer* de l'objet *impot* implémentant l'interface *IImpot*.

Voici un exemple d'exécution du programme :

```
1.  Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :q
    s d
2.  syntaxe : Marié NbEnfants Salaire
3.  Marié : o pour marié, n pour non marié
4.  NbEnfants : nombre d'enfants
5.  Salaire : salaire annuel en euros
6.  Argument marié incorrect : tapez o ou n
7.  Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :o
    2 d
8.  syntaxe : Marié NbEnfants Salaire
9.  Marié : o pour marié, n pour non marié
10. NbEnfants : nombre d'enfants
11. Salaire : salaire annuel en euros
12. Argument salaire incorrect : tapez un entier positif ou nul
13. Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :q
    s d f
14. syntaxe : Marié NbEnfants Salaire
15. Marié : o pour marié, n pour non marié
16. NbEnfants : nombre d'enfants
17. Salaire : salaire annuel en euros
18. Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :o
    2 60000
19. Impot=4282 euros
```

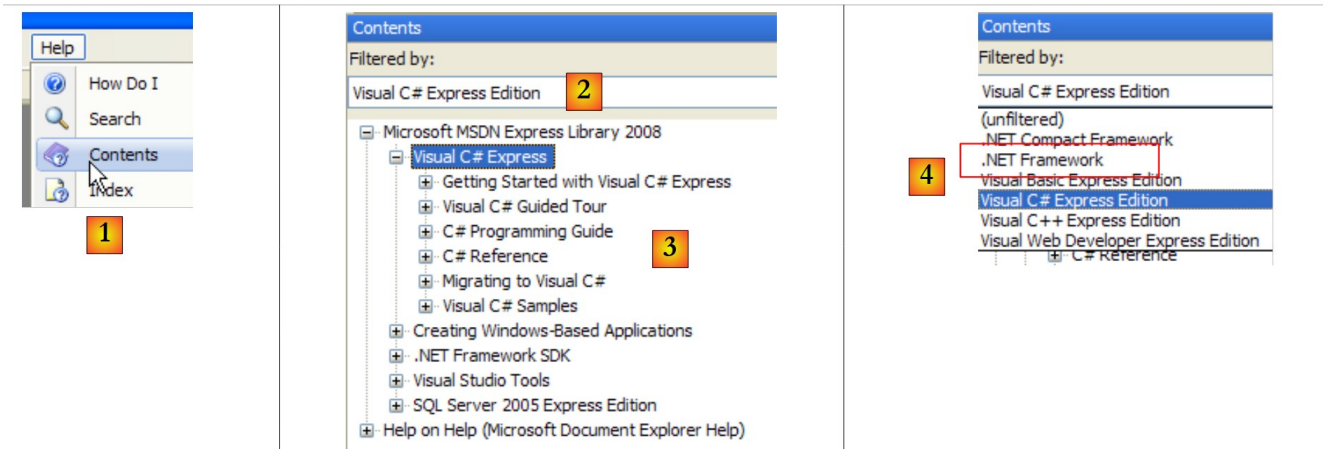
5 Classes .NET d'usage courant

Nous présentons ici quelques classes de la plate-forme .NET fréquemment utilisées. Auparavant, nous montrons comment obtenir des renseignements sur les quelques centaines de classes disponibles. Cette aide est indispensable au développeur C# même confirmé. Le niveau de qualité d'une aide (accès facile, organisation compréhensible, pertinence des informations, ...) peut faire le succès ou l'échec d'un environnement de développement.

5.1 Chercher de l'aide sur les classes .NET

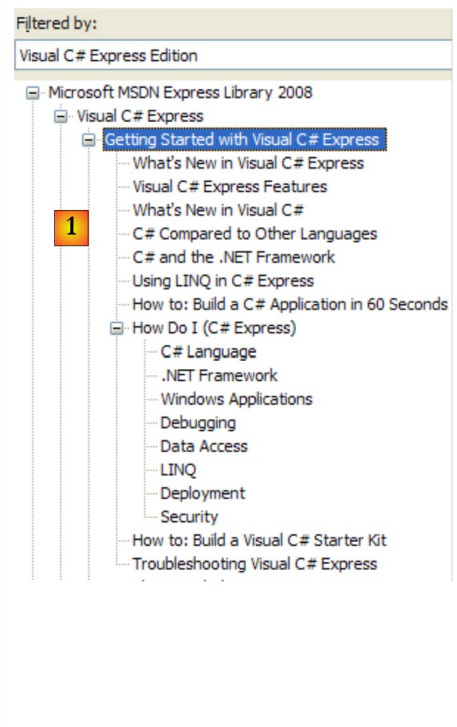
Nous donnons ici quelques indications pour trouver de l'aide avec Visual Studio.NET

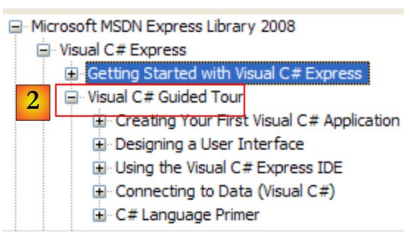
5.1.1 Help/Contents



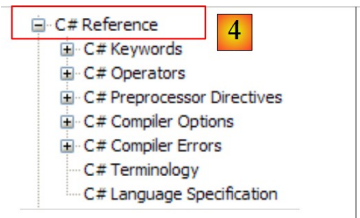
- en [1], prendre l'option *Help/Contents* du menu.
- en [2], prendre l'option Visual C# Express Edition
- en [3], l'arbre de l'aide sur C#
- en [4], une autre option utile est .NET Framework qui donne accès à toutes les classes du framework .NET.

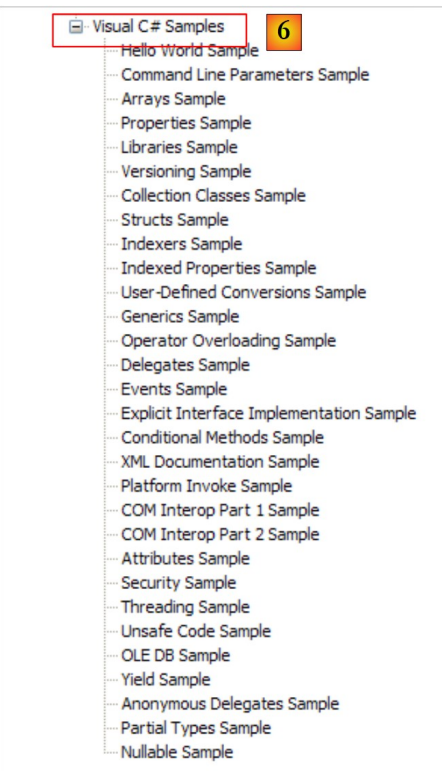
Faisons le tour des têtes de chapitre de l'aide C# :





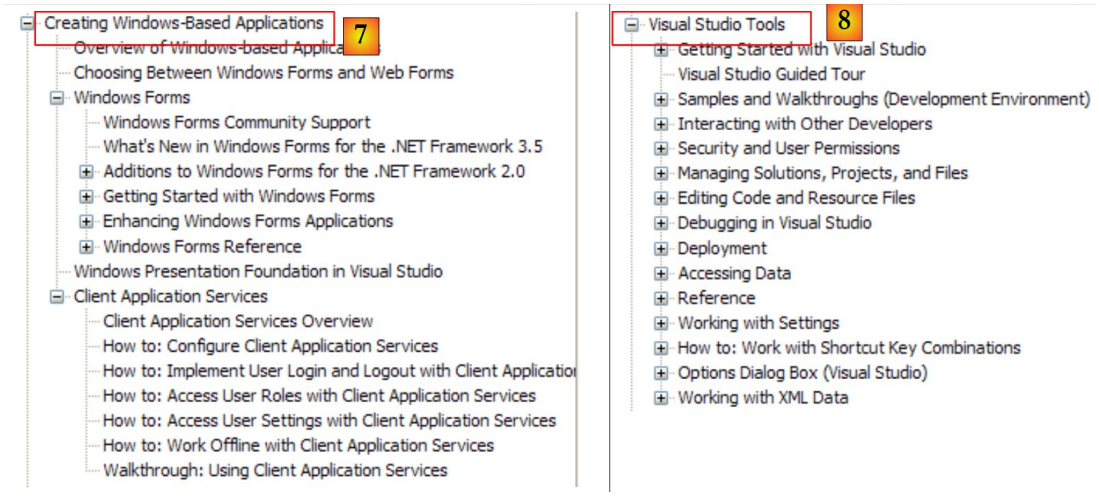
- [1] : une vue d'ensemble de C#
- [2] : une série d'exemples sur certains points de C#
- [3] : un cours C# - pourrait remplacer avantageusement le présent document…



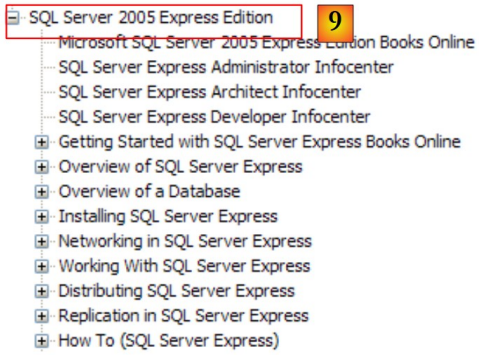


- [4] : pour aller dans les détails de C#
- [5] : utile pour les développeurs C++ ou Java. Permet d'éviter quelques pièges.

- [6] : lorsque vous cherchez des exemples, vous pouvez commencer par là.

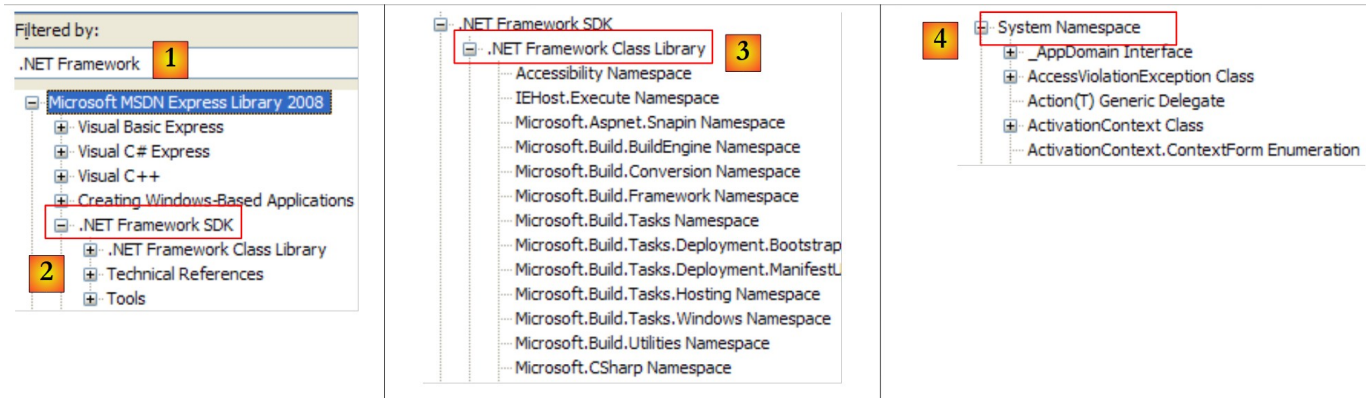


- [7] : ce qu'il faut savoir pour créer des interfaces graphiques
- [8] : pour mieux utiliser l'IDE Visual Studio Express



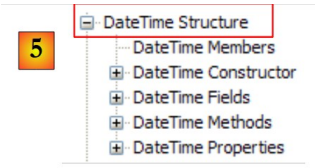
- [9] : SQL Server Express 2005 est un SGBD de qualité distribué gratuitement. Nous l'utiliserons dans ce cours.

L'aide C# n'est qu'une partie de ce dont a besoin le développeur. L'autre partie est l'aide sur les centaines de classes du framework .NET qui vont lui faciliter son travail.



- [1] : on sélectionne l'aide sur le framework .NET

- [2] : l'aide se trouve dans la branche *.NET Framework SDK*
- [3] : la branche *.NET Framework Class Library* présente toutes les classes .NET selon l'espace de noms auquel elles appartiennent
- [4] : l'espace de noms *System* qui a été le plus souvent utilisé dans les exemples des chapitres précédents

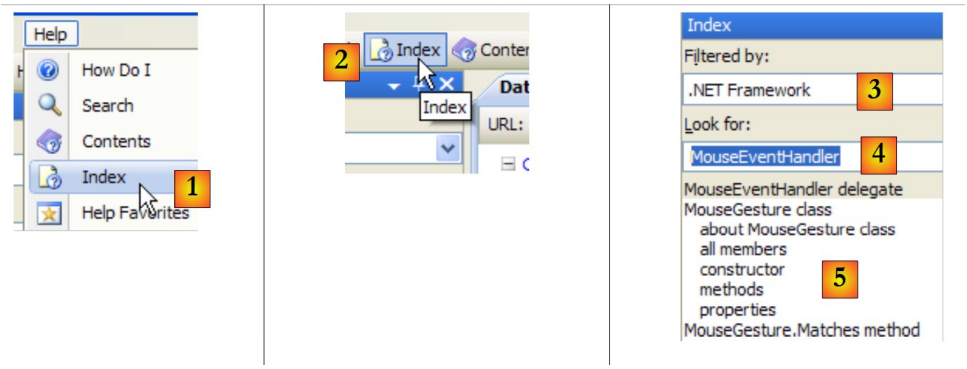


- [5] : dans l'espace de noms *System*, un exemple, ici la structure *DateTime*

- [6] : l'aide sur la structure *DateTime*

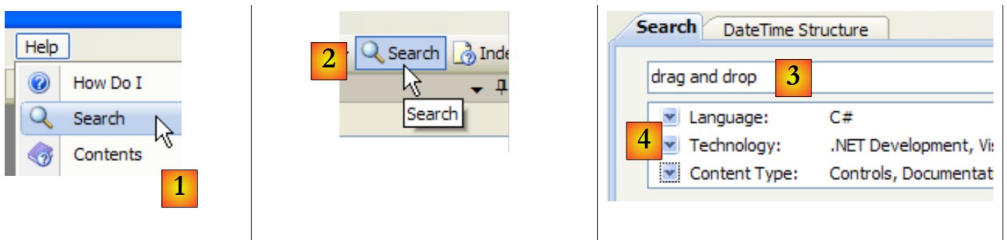
5.1.2 Help/Index/Search

L'aide fournie par MSDN est immense et on peut ne pas savoir où chercher. On peut alors utiliser l'index de l'aide :

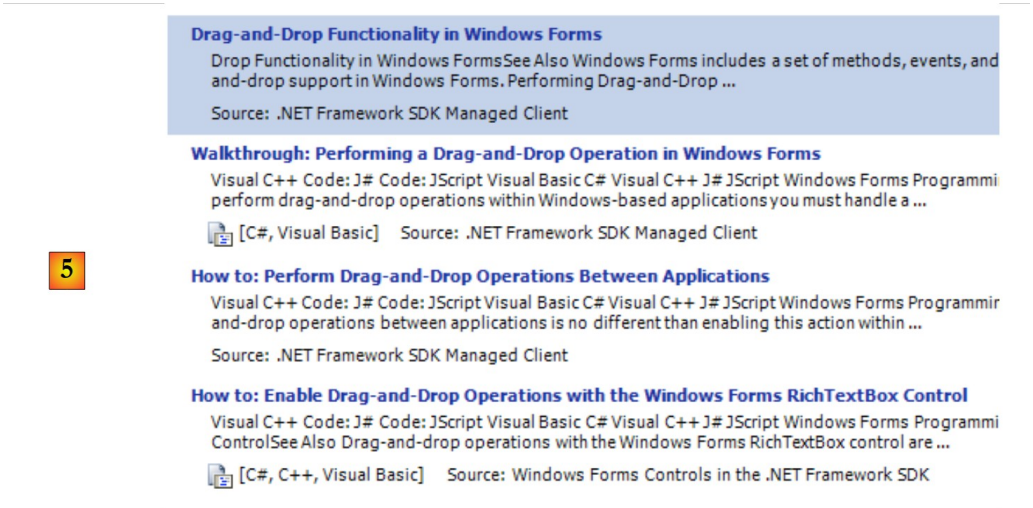


- en [1], utiliser l'option [Help/Index] si la fenêtre d'aide n'est pas déjà ouverte, sinon utiliser [2] dans une fenêtre d'aide existante.
- en [3], préciser le domaine dans lequel doit se faire la recherche
- en [4], préciser ce que vous cherchez, ici une classe
- en [5], la réponse

Une autre façon de chercher de l'aide est d'utiliser la fonction **search** de l'aide :



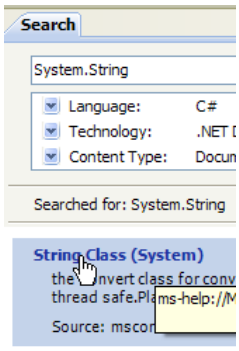
- en [1], utiliser l'option [Help/Search] si la fenêtre d'aide n'est pas déjà ouverte, sinon utiliser [2] dans une fenêtre d'aide existante.
- en [3], préciser ce qui est cherché
- en [4], filtrer les domaines de recherche

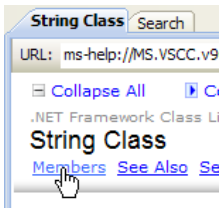


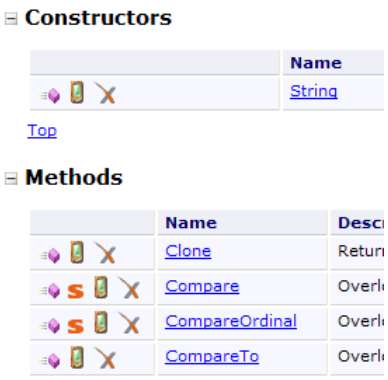
- en [5], la réponse sous forme de différents thèmes où le texte cherché a été trouvé.

5.2 Les chaînes de caractères

5.2.1 La classe System.String







La classe **System.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 courante en minuscules
`public string ToUpper()`	rend la chaîne courante en majuscules
`public string Trim()`	rend la chaîne courante débarrassée de ses espaces de début et fin

On notera un point important : lorsqu'une méthode rend une chaîne de caractères, celle-ci est une **chaîne différente** de la chaîne sur laquelle a été appliquée la méthode. Ainsi *S1.Trim()* rend une chaîne *S2*, et *S1* et *S2* sont deux chaînes différentes.

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

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

Considérons l'exemple suivant :

```
1.  using System;
2.
3.  namespace Chap3 {
4.   class Program {
5.      static void Main(string[] args) {
6.         string uneChaine = "l'oiseau vole au-dessus des nuages";
7.         affiche("uneChaine=" + uneChaine);
8.         affiche("uneChaine.Length=" + uneChaine.Length);
9.         affiche("chaine[10]=" + uneChaine[10]);
10.        affiche("uneChaine.IndexOf(\"vole\")=" + uneChaine.IndexOf("vole"));
11.        affiche("uneChaine.IndexOf(\"x\")=" + uneChaine.IndexOf("x"));
12.        affiche("uneChaine.LastIndexOf('a')=" + uneChaine.LastIndexOf('a'));
13.        affiche("uneChaine.LastIndexOf('x')=" + uneChaine.LastIndexOf('x'));
14.        affiche("uneChaine.Substring(4,7)=" + uneChaine.Substring(4, 7));
15.        affiche("uneChaine.ToUpper()=" + uneChaine.ToUpper());
16.        affiche("uneChaine.ToLower()=" + uneChaine.ToLower());
17.        affiche("uneChaine.Replace('a','A')=" + uneChaine.Replace('a', 'A'));
18.        string[] champs = uneChaine.Split(null);
19.        for (int i = 0; i < champs.Length; i++) {
20.           affiche("champs[" + i + "]=[" + champs[i] + "]");
21.        }//for
22.        affiche("Join(\":\",champs)=" + System.String.Join(":", champs));
23.        affiche("(\"  abc  \").Trim()=[" + "  abc  ".Trim() + "]");
24.     }//Main
25.
26.     public static void affiche(string msg) {
27.        // affiche msg
28.        Console.WriteLine(msg);
29.     }//affiche
30.  }//classe
31. }//namespace
```

L'exécution donne les résultats suivants :

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

Considérons un nouvel exemple :

```
1.  using System;
2.
3.  namespace Chap3 {
4.   class Program {
5.      static void Main(string[] args) {
6.         // la ligne à analyser
7.         string ligne = "un:deux::trois:";
8.         // les séparateurs de champs
9.         char[] séparateurs = new char[] { ':' };
10.        // split
11.        string[] champs = ligne.Split(séparateurs);
12.        for (int i = 0; i < champs.Length; i++) {
13.           Console.WriteLine("Champs[" + i + "]=" + champs[i]);
14.        }
```

```
15.         // join
16.         Console.WriteLine("join=[" + System.String.Join(":", champs) + "]");
17.     }
18.  }
19. }
```

et les résultats d'exécution :

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

La méthode **Split** de la classe *String* permet de mettre dans un tableau des éléments d'une chaîne de caractères. La définition de la méthode **Split** utilisée ici est la suivante :

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

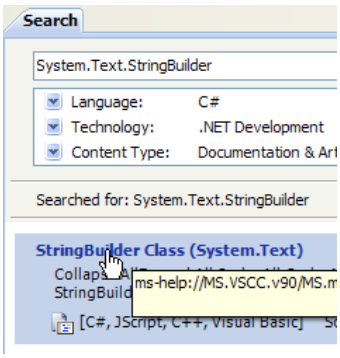
`separator`	tableau de caractères. Ces caractères représentent les caractères utilisés pour séparer les champs de la chaîne de caractères. Ainsi si la chaîne est "*champ1*, *champ2*, *champ3*" on pourra utiliser *separator=new char[] {',',}*. Si le séparateur est une suite d'espaces on utilisera *separator=null*.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

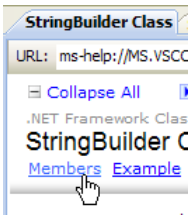
La méthode **Join** est une méthode statique de la classe *String* :

```
    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*.

5.2.2 La classe System.Text.StringBuilder





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

La classe admet divers constructeurs :

`StringBuilder()`	`constructeur par défaut`

`StringBuilder(String value)`	construction et initialisation avec *value*
`StringBuilder(String value, int capacité)`	construction et initialisation avec *value* avec une taille de bloc de *capacité* caractères.

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

Voici un exemple illustrant cette notion de *capacité* :

```
1.  using System.Text;
2.  using System;
3.  namespace Chap3 {
4.   class Program {
5.      static void Main(string[] args) {
6.         // str
7.         StringBuilder str = new StringBuilder("test");
8.         Console.WriteLine("taille={0}, capacité={1}", str.Length, str.Capacity);
9.         for (int i = 0; i < 10; i++) {
10.           str.Append("test");
11.           Console.WriteLine("taille={0}, capacité={1}", str.Length, str.Capacity);
12.        }
13.        // str2
14.        StringBuilder str2 = new StringBuilder("test",10);
15.        Console.WriteLine("taille={0}, capacité={1}", str2.Length, str2.Capacity);
16.        for (int i = 0; i < 10; i++) {
17.           str2.Append("test");
18.           Console.WriteLine("taille={0}, capacité={1}", str2.Length, str2.Capacity);
19.        }
20.     }
21.  }
22. }
```

- ligne 7 : création d'un objet *StringBuilder* avec une taille de bloc de 16 caractères
- ligne 8 : *str.Length* est le nombre actuel de caractères de la chaîne *str*. *str.Capacity* est le nombre de caractères que peut stocker la chaîne *str* actuelle avant réallocation d'un nouveau bloc.
- ligne 10 : *str.Append(String S)* permet de concaténer la chaîne *S* de type *String* à la chaîne *str* de type *StringBuilder*.
- ligne 14 : création d'un objet *StringBuilder* avec une capacité de bloc de 10 caractères

Le résultat de l'exécution :

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

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

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

Voici quelques-unes des méthodes de la classe :

`public StringBuilder Append(string value)`	ajoute la chaîne value à l'objet *StringBuilder*. Rend l'objet *StringBuilder*. Cette méthode est surchargée pour admettre différents types pour *value* : byte, int, float, double, decimal, ...
`public StringBuilder Insert(int index, string value)`	insère *value* à la position *index*. Cette méthode est surchargée comme la précédente pour accepter différents types pour *value*.
`public StringBuilder Remove(int index, int length)`	supprime *length* caractères à partir de la position *index*.
`public StringBuilder Replace(string oldValue, string newValue)`	remplace dans *StringBuilder*, la chaîne *oldValue* par la chaîne *newValue*. Il existe une version surchargée (char oldChar, char newChar).
`public String ToString()`	convertit l'objet *StringBuilder* en un objet de type *String*.

Voici un exemple :

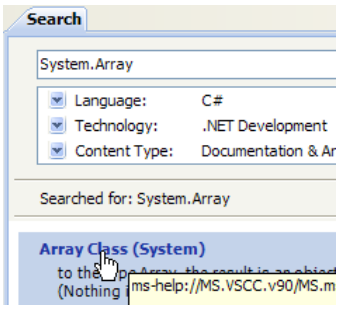
```
1.  using System.Text;
2.  using System;
3.  namespace Chap3 {
4.   class Program {
5.      static void Main(string[] args) {
6.         // str3
7.         StringBuilder str3 = new StringBuilder("test");
8.         Console.WriteLine(str3.Append("abCD").Insert(2, "xyZT").Remove(0, 2).Replace("xy", "XY"));
9.      }
10.  }
11. }
```

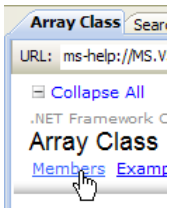
et ses résultats :

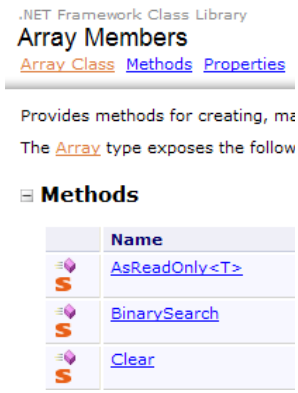
```
XYZTstabCD
```

5.3 Les tableaux

Les tableaux dérivent de la classe **Array** :







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

Propriétés

`public int Length {get;}`	nombre total d'éléments du tableau, quelque soit son nombre de dimensions

public int Rank {get;}	nombre total de dimensions du tableau

Méthodes

public static int BinarySearch<T>(T[] tableau,T value)	rend la position de *value* dans *tableau*.
public static int BinarySearch<T>(T[] tableau,int index, int length, T value)	idem mais cherche dans *tableau* à partir de la position *index* et sur *length* éléments
public static void Clear(Array tableau, int index, int length)	met les *length* éléments de *tableau* commençant au n° *index* à 0 si numériques, false si booléens, null si références
public static void Copy(Array source, Array destination, int length)	copie *length* éléments de *source* dans *destination*
public int GetLength(int i)	nombre d'éléments de la dimension n° i du tableau
public int GetLowerBound(int i)	indice du 1er élément de la dimension n° i
public int GetUpperBound(int i)	indice du dernier élément de la dimension n° i
public static int IndexOf<T>(T[] tableau, T valeur)	rend la position de *valeur* dans *tableau* ou -1 si *valeur* n'est pas trouvée.
public static void Resize<T>(ref T[] tableau, int n)	redimensionne *tableau* à *n* éléments. Les éléments déjà présents sont conservés.
public static void Sort<T>(T[] tableau, IComparer<T> comparateur)	trie *tableau* selon un ordre défini par *comparateur*. Cette méthode a été présentée page 82.

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

```
1.  using System;
2.
3.  namespace Chap3 {
4.   class Program {
5.      // type de recherche
6.      enum TypeRecherche { linéaire, dichotomique };
7.
8.      // méthode principale
9.      static void Main(string[] args) {
10.        // lecture des éléments d'un tableau tapés au clavier
11.        double[] éléments;
12.        Saisie(out éléments);
13.        // affichage tableau non trié
14.        Affiche("Tableau non trié", éléments);
15.        // Recherche linéaire dans le tableau non trié
16.        Recherche(éléments, TypeRecherche.linéaire);
17.        // tri du tableau
18.        Array.Sort(éléments);
19.        // affichage tableau trié
20.        Affiche("Tableau trié", éléments);
21.        // Recherche dichotomique dans le tableau trié
22.        Recherche(éléments, TypeRecherche.dichotomique);
23.     }
24.
25.     // saisie des valeurs du tableau éléments
26.     // éléments : référence sur tableau créé par la méthode
27.     static void Saisie(out double[] éléments) {
28.        bool terminé = false;
29.        string réponse;
30.        bool erreur;
31.        double élément = 0;
32.        int i = 0;
33.        // au départ, le tableau n'existe pas
34.        éléments = null;
35.        // boucle de saisie des éléments du tableau
36.        while (!terminé) {
37.           // question
38.           Console.Write("Elément (réel) " + i + " du tableau (rien pour terminer) : ");
39.           // lecture de la réponse
40.           réponse = Console.ReadLine().Trim();
41.           // fin de saisie si chaîne vide
42.           if (réponse.Equals(""))
43.              break;
44.           // vérification saisie
```

```
45.           try {
46.              élément = Double.Parse(réponse);
47.              erreur = false;
48.           } catch {
49.              Console.Error.WriteLine("Saisie incorrecte, recommencez");
50.              erreur = true;
51.           }//try-catch
52.           // si pas d'erreur
53.           if (!erreur) {
54.              // un élément de plus dans le tableau
55.              i += 1;
56.              // redimensionnement tableau pour accueillir le nouvel élément
57.              Array.Resize(ref éléments, i);
58.              // insertion nouvel élément
59.              éléments[i - 1] = élément;
60.           }
61.        }//while
62.     }
63.
64.     // méthode générique pour afficher les éléments d'un tableau
65.     static void Affiche<T>(string texte, T[] éléments) {
66.        Console.WriteLine(texte.PadRight(50, '-'));
67.        foreach (T élément in éléments) {
68.           Console.WriteLine(élément);
69.        }
70.     }
71.
72.     // recherche d'un élément dans le tableau
73.     // éléments : tableau de réels
74.     // TypeRecherche : dichotomique ou linéaire
75.     static void Recherche(double[] éléments, TypeRecherche type) {
76.        // Recherche
77.        bool terminé = false;
78.        string réponse = null;
79.        double élément = 0;
80.        bool erreur = false;
81.        int i = 0;
82.        while (!terminé) {
83.           // question
84.           Console.WriteLine("Elément cherché (rien pour arrêter) : ");
85.           // lecture-vérification réponse
86.           réponse = Console.ReadLine().Trim();
87.           // fini ?
88.           if (réponse.Equals(""))
89.              break;
90.           // vérification
91.           try {
92.              élément = Double.Parse(réponse);
93.              erreur = false;
94.           } catch {
95.              Console.WriteLine("Erreur, recommencez...");
96.              erreur = true;
97.           }//try-catch
98.           // si pas d'erreur
99.           if (!erreur) {
100.             // on cherche l'élément dans le tableau
101.             if (type == TypeRecherche.dichotomique)
102.                // recherche dichotomique
103.                i = Array.BinarySearch(éléments, élément);
104.             else
105.                // recherche linéaire
106.                i = Array.IndexOf(éléments, élément);
107.             // Affichage réponse
108.             if (i >= 0)
109.                Console.WriteLine("Trouvé en position " + i);
110.             else
111.                Console.WriteLine("Pas dans le tableau");
112.          }//if
113.       }//while
114.    }
115. }
116.}
```

- lignes 27-62 : la méthode *Saisie* saisit les éléments d'un tableau *éléments* tapés au clavier. Comme on ne peut dimensionner le tableau à priori (on ne connaît pas sa taille finale), on est obligés de le redimensionner à chaque nouvel élément (ligne 57).

Un algorithme plus efficace aurait été d'allouer de la place au tableau par groupe de N éléments. Un tableau n'est cependant pas fait pour être redimensionné . Ce cas là est mieux traité avec une liste (ArrayList, List<T>).

- lignes 75-113 : la méthode *Recherche* permet de rechercher dans le tabeau *éléments*, un élément tapé au clavier. Le mode de recherche est différent selon que le tableau est trié ou non. Pour un tableau non trié, on fait une recherche linéaire avec la méthode *IndexOf* de la ligne 106. Pour un tableau trié, on fait une recherche dichotomique avec la méthode *BinarySearch* de la ligne 103.
- ligne 18 : on trie le tableau *éléments*. On utilise ici, une variante de *Sort* qui n'a qu'un paramètre : le tableau à trier. La relation d'ordre utilisée pour comparer les éléments du tableau est alors celle implicite de ces éléments. Ici, les éléments sont numériques. C'est l'ordre naturel des nombres qui est utilisé.

Les résultats écran sont les suivants :

```
1.  Elément (réel) 0 du tableau (rien pour terminer) : 3,6
2.  Elément (réel) 1 du tableau (rien pour terminer) : 7,4
3.  Elément (réel) 2 du tableau (rien pour terminer) : -1,5
4.  Elément (réel) 3 du tableau (rien pour terminer) : -7
5.  Elément (réel) 4 du tableau (rien pour terminer) :
6.  Tableau non trié----------------------------------
7.  3,6
8.  7,4
9.  -1,5
10. -7
11. Elément cherché (rien pour arrêter) :
12. 7,4
13. Trouvé en position 1
14. Elément cherché (rien pour arrêter) :
15. 0
16. Pas dans le tableau
17. Elément cherché (rien pour arrêter) :
18.
19. Tableau trié--------------------------------------
20. -7
21. -1,5
22. 3,6
23. 7,4
24. Elément cherché (rien pour arrêter) :
25. 7,4
26. Trouvé en position 3
27. Elément cherché (rien pour arrêter) :
28. 0
29. Pas dans le tableau
30. Elément cherché (rien pour arrêter) :
```

5.4 Les collections génériques

Outre le tableau, il existe diverses classes pour stocker des collections d'éléments. Il existe des versions génériques dans l'espace de noms *System.Collections.Generic* et des versions non génériques dans *System.Collections*. Nous présenterons deux collections génériques fréquemment utilisées : la **liste** et le **dictionnaire**.

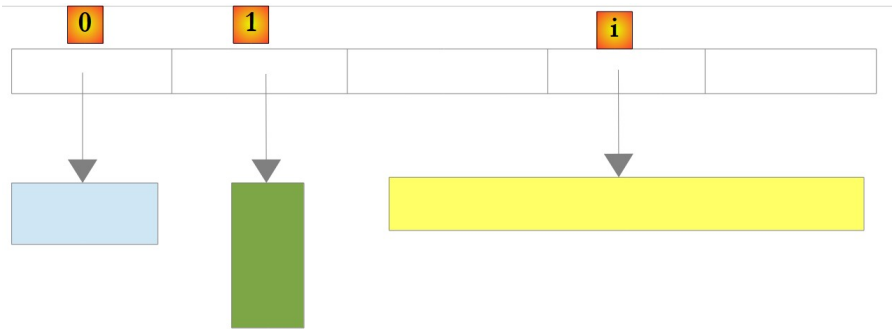
La liste des collections génériques est la suivante :

	Class	Description
	Comparer<T>	Provides a base class for implementations of the IComparer<T> generic interface.
	Dictionary<TKey, TValue>	Represents a collection of keys and values.
	Dictionary<TKey, TValue>.KeyCollection	Represents the collection of keys in a Dictionary<TKey, TValue>. This class cannot be inherited.
	Dictionary<TKey, TValue>.ValueCollection	Represents the collection of values in a Dictionary<TKey, TValue>. This class cannot be inherited.
	EqualityComparer<T>	Provides a base class for implementations of the IEqualityComparer<T> generic interface.
	HashSet<T>	Represents a set of values.
	KeyedByTypeCollection<TItem>	Provides a collection whose items are types that serve as keys.
	KeyNotFoundException	The exception that is thrown when the key specified for accessing an element in a collection does not match any key in the collection.
	LinkedList<T>	Represents a doubly linked list.
	LinkedListNode<T>	Represents a node in a LinkedList<T>. This class cannot be inherited.
	List<T>	Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
	Queue<T>	Represents a first-in, first-out collection of objects.
	SortedDictionary<TKey, TValue>	Represents a collection of key/value pairs that are sorted on the key.
	SortedDictionary<TKey, TValue>.KeyCollection	Represents the collection of keys in a SortedDictionary<TKey, TValue>. This class cannot be inherited.
	SortedDictionary<TKey, TValue>.ValueCollection	Represents the collection of values in a SortedDictionary<TKey, TValue>. This class cannot be inherited
	SortedList<TKey, TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
	Stack<T>	Represents a variable size last-in-first-out (LIFO) collection of instances of the same arbitrary type.
	SynchronizedCollection<T>	Provides a thread-safe collection that contains objects of a type specified by the generic parameter as elements.
	SynchronizedKeyedCollection<K, T>	Provides a thread-safe collection that contains objects of a type specified by a generic parameter and that are grouped by keys.
	SynchronizedReadOnlyCollection<T>	Provides a thread-safe, read-only collection that contains objects of a type specified by the generic parameter as elements.

5.4.1 La classe générique List<T>

La classe *System.Collections.Generic.List<T>* permet d'implémenter des collections d'objets de type T dont la taille varie au cours de l'exécution du programme. Un objet de type *List<T>* se manipule presque comme un tableau. Ainsi l'élément i d'une liste l est-il noté l[i].

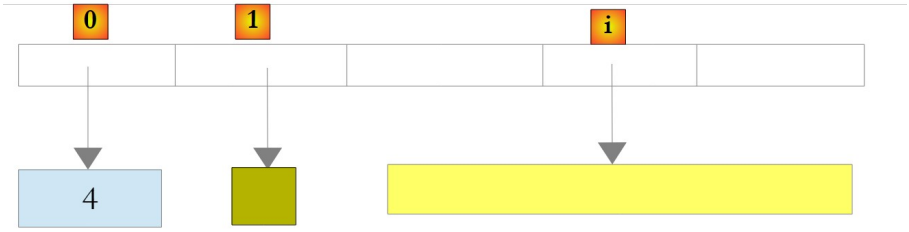
Il existe également un type de liste non générique : *ArrayList* capable de stocker des références sur des objets quelconques. *ArrayList* est fonctionnellement équivalente à *List<Object>*. Un objet *ArrayList* ressemble à ceci :



Ci-dessus, les éléments 0, 1 et i de la liste pointent sur des objets de types différents. Il faut qu'un objet soit d'abord créé avant d'ajouter sa référence à la liste *ArrayList*. Bien qu'un *ArrayList* stocke des références d'objet, il est possible d'y stocker des nombres. Cela se fait par un mécanisme appelé *Boxing* : le nombre est encapsulé dans un objet O de type *Object* et c'est la référence O qui est stocké dans la liste. C'est un mécanisme transparent pour le développeur. On peut ainsi écrire :

```
ArrayList liste=new ArrayList();
liste.Add(4);
```

Cela produira le résultat suivant :

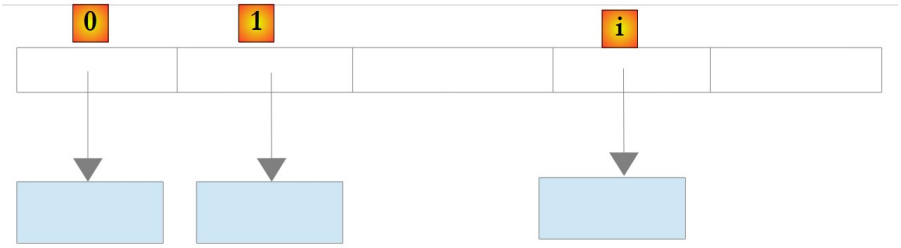


Ci-dessus, le nombre 4 a été encapsulé dans un objet O et la référence O est mémorisée dans la liste. Pour le récupérer, on pourra écrire :

```
        int i = (int)liste[0];
```

L'opération *Object -> int* est appelée *Unboxing*. Si une liste est entièrement composée de types *int*, la déclarer comme *List<int>* améliore les performances. En effet, les nombres de type *int* sont alors stockés dans la liste elle-même et non dans des types *Object* extérieurs à la liste. Les opérations Boxing / Unboxing n'ont plus lieu.

Pour un objet *List<T>* ou *T* est une classe, la liste stocke là encore les références des objets de type *T* :



Voici quelques-unes des propriétés et méthodes des listes génériques :

Propriétés

public int Count {get;}	nombre d'éléments de la liste
public int Capacity {get;}	nombre d'éléments que la liste peut contenir avant d'être redimensionnée. Ce redimensionnement se fait automatiquement. Cette notion de capacité de liste est analogue à celle de capacité décrite pour la classe StringBuilder page 97.

Méthodes

public void Add(T item)	ajoute *item* à la liste
public int BinarySearch<T>(T item)	rend la position de *item* dans la liste s'il s'y trouve sinon un nombre <0
public int BinarySearch<T>(T item, IComparer<T> comparateur)	idem mais le 2ième paramètre permet de comparer deux éléments de la liste. L'interface IComparer<T> a été présentée page 82.
public void Clear()	supprime tous les éléments de la liste
public bool Contains(T item)	rend True si item est dans la liste, False sinon
public void CopyTo(T[] tableau)	copie les éléments de la liste dans *tableau*.

public int IndexOf(T item)	rend la position de *item* dans *tableau* ou -1 si *valeur* n'est pas trouvée.
public void Insert(T item, int index)	insère *item* à la position *index* de la liste
public bool Remove(T item)	supprime *item* de la liste. Rend True si l'opération réussit, False sinon.
public void RemoveAt(int index)	supprime l'élément n° *index* de la liste
public void Sort(IComparer<T> comparateur)	trie la liste selon un ordre défini par *comparateur*. Cette méthode a été présentée page 82.
public void Sort()	trie la liste selon l'ordre défini par le type des éléments de la liste
public T[] ToArray()	rend les éléments de la liste sous forme de tableau

Reprenons l'exemple traité précédemment avec un objet de type *Array* et traitons-le maintenant avec un objet de type *List<T>*. Parce que la liste est un objet proche du tableau, le code change peu. Nous ne présentons que les modifications notables :

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace Chap3 {
5.   class Program {
6.      // type de recherche
7.      enum TypeRecherche { linéaire, dichotomique };
8.
9.      // méthode principale
10.     static void Main(string[] args) {
11.        // lecture des éléments d'une liste tapés au clavier
12.        List<double> éléments;
13.        Saisie(out éléments);
14.        // nombre d'éléments
15.        Console.WriteLine("La liste a {0} éléments et une capacité de {1} éléments",
    éléments.Count, éléments.Capacity);
16.        // affichage liste non triée
17.        Affiche("Liste non triée", éléments);
18.        // Recherche linéaire dans la liste non triée
19.        Recherche(éléments, TypeRecherche.linéaire);
20.        // tri de la liste
21.        éléments.Sort();
22.        // affichage liste triée
23.        Affiche("Liste triée", éléments);
24.        // Recherche dichotomique dans la liste triée
25.        Recherche(éléments, TypeRecherche.dichotomique);
26.     }
27.
28.     // saisie des valeurs de la liste éléments
29.     // éléments : référence sur la liste créée par la méthode
30.     static void Saisie(out List<double> éléments) {
31. ...
32.        // au départ, la liste est vide
33.        éléments = new List<double>();
34.        // boucle de saisie des éléments de la liste
35.        while (!terminé) {
36. ...
37.           // si pas d'erreur
38.           if (!erreur) {
39.              // un élément de plus dans la liste
40.              éléments.Add(élément);
41.           }
42.        }//while
43.     }
44.
45.     // méthode générique pour afficher les éléments d'un objet énumérable
46.     static void Affiche<T>(string texte, IEnumerable<T> éléments) {
47.        Console.WriteLine(texte.PadRight(50, '-'));
48.        foreach (T élément in éléments) {
49.           Console.WriteLine(élément);
50.        }
51.     }
52.
53.     // recherche d'un élément dans la liste
54.     // éléments : liste de réels
55.     // TypeRecherche : dichotomique ou linéaire
56.     static void Recherche(List<double> éléments, TypeRecherche type) {
```

```
57. ...
58.         while (!terminé) {
59. ...
60.            // si pas d'erreur
61.            if (!erreur) {
62.               // on cherche l'élément dans la liste
63.               if (type == TypeRecherche.dichotomique)
64.                  // recherche dichotomique
65.                  i = éléments.BinarySearch(élément);
66.               else
67.                  // recherche linéaire
68.                  i = éléments.IndexOf(élément);
69.               // Affichage réponse
70. ...
71.            }//if
72.         }//while
73.      }
74.   }
75. }
```

- lignes 46-51 : la méthode générique *Affiche<T>* admet deux paramètres :
  - le 1er paramètre est un texte à écrire
  - le 2ième paramètre est un objet implémentant l'interface générique **IEnumerable<T>** :

```
1.  public interface IEnumerable<T>{
2.   IEnumerator GetEnumerator();
3.   IEnumerator<T> GetEnumerator();
4.  }
```

La structure *foreach(T élément in éléments)* de la ligne 48, est valide pour tout objet *éléments* implémentant l'interface *IEnumerable*. Les tableaux (*Array*) et les listes (*List<T>*) implémentent l'interface *IEnumerable<T>*. Aussi la méthode *Affiche* convient-elle aussi bien pour afficher des tableaux que des listes.

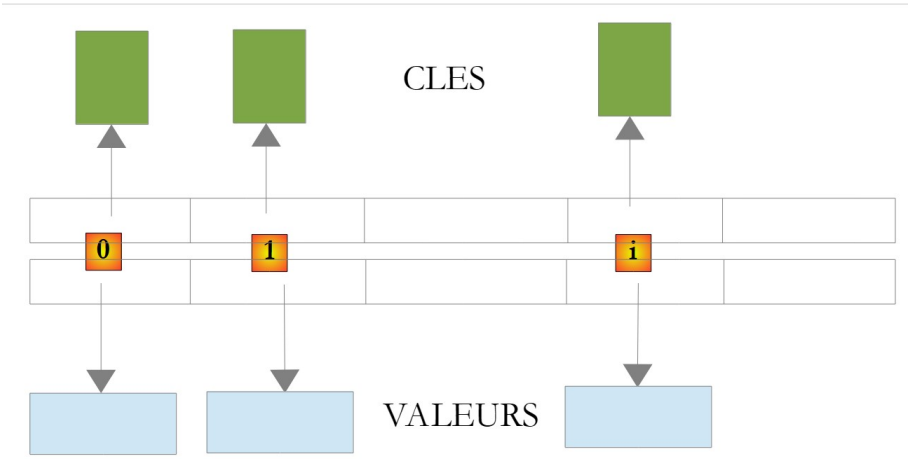
Les résultats d'exécution du programme sont les mêmes que dans l'exemple utilisant la classe *Array*.

5.4.2 La classe Dictionary<TKey,TValue>

La classe *System.Collections.Generic.Dictionary<TKey,TValue>* permet d'implémenter un dictionnaire. On peut voir un dictionnaire comme un tableau à deux colonnes :

clé	valeur
clé1	valeur1
clé2	valeur2
..	...

Dans la classe *Dictionary<TKey,TValue>* les clés sont de type *Tkey*, les valeurs de type *TValue*. Les clés sont uniques, c.a.d. qu'il ne peut y avoir deux clés identiques. Un tel dictionnaire pourrait ressembler à ceci si les types *TKey* et *TValue* désignaient des classes :



La valeur associée à la clé C d'un dictionnaire D est obtenue par la notation D[C]. Cette valeur est en lecture et écriture. Ainsi on peut écrire :

```
1.  TValue v=...;
2.  TKey c=...;
3.  Dictionary<TKey,TValue> D=new Dictionary<TKey,TValue>();
4.  D[c]=v;
5.  v=D[c];
```

Si la clé *c* n'existe pas dans le dictionnaire *D*, la notation *D[c]* lance une exception.

Les méthodes et propriétés principales de la classe ***Dictionary<TKey,TValue>*** sont les suivantes :

Constructeurs

public Dictionary<TKey,TValue>()	constructeur sans paramètres - construit un dictionnaire vide. Il existe plusieurs autres constructeurs.

Propriétés

public int Count {get;}	nombre d'entrées (clé, valeur) dans le dictionnaire
public Dictionary<TKey,TValue>.KeyCollection Keys {get;}	collection des clés du dictionnaire.
public Dictionary<TKey,TValue>.ValueCollection Values {get;}	collection des valeurs du dictionnaire.

Méthodes

public void Add(TKey key, TValue value)	ajoute le couple (key, value) au dictionnaire
public void Clear()	supprime tous les couples du dictionnaire
public bool ContainsKey (TKey key)	rend True si *key* est une clé du dictionnaire, False sinon
public bool ContainsValue (TValue value)	rend True si *value* est une valeur du dictionnaire, False sinon
public void CopyTo(T[] tableau)	copie les éléments de la liste dans *tableau*.
public bool Remove(TKey key)	supprime du dictionnaire le couple de clé key. Rend True si l'opération réussit, False sinon.
public bool TryGetValue(TKey key,out TValue value)	rend dans value, la valeur associée à la clé key si cette dernière existe, sinon rend la valeur par défaut du type TValue (0 pour les nombres, false pour les booléens, null pour les références d'objet)

Considérons le programme exemple suivant :

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace Chap3 {
5.   class Program {
6.      static void Main(string[] args) {
7.         // création d'un dictionnaire <string,int>
8.         string[] liste = { "jean:20", "paul:18", "mélanie:10", "violette:15" };
9.         string[] champs = null;
10.        char[] séparateurs = new char[] { ':' };
11.        Dictionary<string,int> dico = new Dictionary<string,int>();
12.        for (int i = 0; i <liste.Length; i++) {
13.           champs = liste[i].Split(séparateurs);
14.           dico[champs[0]]= int.Parse(champs[1]);
15.        }//for
16.        // nbre d'éléments dans le dictionnaire
17.        Console.WriteLine("Le dictionnaire a " + dico.Count + " éléments");
18.        // liste des clés
19.        Affiche("[Liste des clés]",dico.Keys);
20.        // liste des valeurs
21.        Affiche("[Liste des valeurs]", dico.Values);
22.        // liste des clés & valeurs
```

```
23.         Console.WriteLine("[Liste des clés & valeurs]");
24.         foreach (string clé in dico.Keys) {
25.            Console.WriteLine("clé=" + clé + " valeur=" + dico[clé]);
26.         }
27.         // on supprime la clé "paul"
28.         Console.WriteLine("[Suppression d'une clé]");
29.         dico.Remove("paul");
30.         // liste des clés & valeurs
31.         Console.WriteLine("[Liste des clés & valeurs]");
32.         foreach (string clé in dico.Keys) {
33.            Console.WriteLine("clé=" + clé + " valeur=" + dico[clé]);
34.         }
35.         // recherche dans le dictionnaire
36.         String nomCherché = null;
37.         Console.Write("Nom recherché (rien pour arrêter) : ");
38.         nomCherché = Console.ReadLine().Trim();
39.         int value;
40.         while (!nomCherché.Equals("")) {
41.            dico.TryGetValue(nomCherché, out value);
42.            if (value!=0) {
43.               Console.WriteLine(nomCherché + "," + value);
44.            } else {
45.               Console.WriteLine("Nom " + nomCherché + " inconnu");
46.            }
47.            // recherche suivante
48.            Console.Out.Write("Nom recherché (rien pour arrêter) : ");
49.            nomCherché = Console.ReadLine().Trim();
50.         }//while
51.      }
52.
53.      // méthode générique pour afficher les éléments d'un type énumérable
54.      static void Affiche<T>(string texte, IEnumerable<T> éléments) {
55.         Console.WriteLine(texte.PadRight(50, '-'));
56.         foreach (T élément in éléments) {
57.            Console.WriteLine(élément);
58.         }
59.      }
60.
61.   }
62. }
```

- ligne 8 : un tableau de *string* qui va servir à initialiser le dictionnaire <string,int>
- ligne 11 : le dictionnaire <string,int>
- lignes 12-15 : son initialisation à partir du tableau de *string* de la ligne 8
- ligne 17 : nombre d'entrées du dictionnaire
- ligne 19 : les clés du dictionnaire
- ligne 21 : les valeurs du dictionnaire
- ligne 29 : suppression d'une entrée du dictionnaire
- ligne 41 : recherche d'une clé dans le dictionnaire. Si elle n'existe pas, la méthode *TryGetValue* mettra 0 dans *value*, car *value* est de type numérique. Cette technique n'est utilisable ici que parce qu'on sait que la valeur 0 n'est pas dans le dictionnaire.

Les résultats d'exécution sont les suivants :

```
1.  Le dictionnaire a 4 éléments
2.  [Liste des clés]---------------------------------
3.  jean
4.  paul
5.  mélanie
6.  violette
7.  [Liste des valeurs]-------------------------------
8.  20
9.  18
10. 10
11. 15
12. [Liste des clés & valeurs]
13. clé=jean valeur=20
14. clé=paul valeur=18
15. clé=mélanie valeur=10
16. clé=violette valeur=15
17. [Suppression d'une clé]
18. [Liste des clés & valeurs]
19. clé=jean valeur=20
20. clé=mélanie valeur=10
21. clé=violette valeur=15
```

```
Nom recherché (rien pour arrêter) : violette
violette,15
Nom recherché (rien pour arrêter) : x
Nom x inconnu
```

5.5 Les fichiers texte

5.5.1 La classe StreamReader

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

Constructeurs

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

Propriétés

public bool EndOfStream {get;}	True si le flux a été lu entièrement

Méthodes

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

Voici un exemple :

```
1.  using System;
2.  using System.IO;
3.
4.  namespace Chap3 {
5.   class Program {
6.      static void Main(string[] args) {
7.         // répertoire d'exécution
8.         Console.WriteLine("Répertoire d'exécution : "+Environment.CurrentDirectory);
9.         string ligne = null;
10.        StreamReader fluxInfos = null;
11.        // lecture contenu du fichier infos.txt
12.        try {
13.           // lecture 1
14.           Console.WriteLine("Lecture 1----------------");
15.           using (fluxInfos = new StreamReader("infos.txt")) {
16.              ligne = fluxInfos.ReadLine();
17.              while (ligne != null) {
18.                 Console.WriteLine(ligne);
19.                 ligne = fluxInfos.ReadLine();
20.              }
```

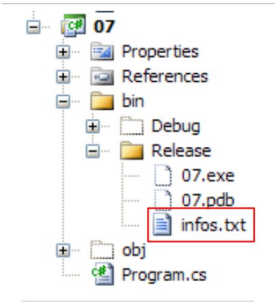
```
21.           }
22.           // lecture 2
23.           Console.WriteLine("Lecture 2----------------");
24.           using (fluxInfos = new StreamReader("infos.txt")) {
25.              Console.WriteLine(fluxInfos.ReadToEnd());
26.           }
27.        } catch (Exception e) {
28.           Console.WriteLine("L'erreur suivante s'est produite : " + e.Message);
29.        }
30.     }
31.  }
32. }
```

- ligne 8 : affiche le nom du répertoire d'exécution
- lignes 12, 27 : un try / catch pour gérer une éventuelle exception.
- ligne 15 : la structure *using flux=new StreamReader(...)* est une facilité pour ne pas avoir à fermer explicitement le flux après son exploitation. Cette fermeture est faite automatiquement dès qu'on sort de la portée du *using*.
- ligne 15 : le fichier lu s'appelle *infos.txt*. Comme c'est un nom relatif, il sera cherché dans le répertoire d'exécution affiché par la ligne 8. S'il n'y est pas, une exception sera lancée et gérée par le try / catch.
- lignes 16-20 : le fichier est lu par lignes successives
- ligne 25 : le fichier est lu d'un seul coup

Le fichier *infos.txt* est le suivant :

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

et placé dans le dossier suivant du projet C# :



On va découvrir que *bin/Release* est le dossier d'exécution lorsque le projet est excécuté par Ctrl-F5.

L'exécution donne les résultats suivants :

```
1.  Répertoire d'exécution : C:\data\2007-2008\c# 2008\poly\Chap3\07\bin\Release
2.  Lecture 1----------------
3.  12620:0:0
4.  13190:0,05:631
5.  15640:0,1:1290,5
6.  Lecture 2----------------
7.  12620:0:0
8.  13190:0,05:631
9.  15640:0,1:1290,5
```

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

```
1.  Répertoire d'exécution : C:\data\2007-2008\c# 2008\poly\Chap3\07\bin\Release
2.  Lecture 1----------------
3.  L'erreur suivante s'est produite : Could not find file 'C:\...\Chap3\07\bin\Release\xx.txt'.
```

5.5.2 La classe StreamWriter

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

Constructeurs

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

Propriétés

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

Méthodes

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

Considérons l'exemple suivant :

```
1.  using System;
2.  using System.IO;
3.
4.  namespace Chap3 {
5.   class Program2 {
6.      static void Main(string[] args) {
7.         // répertoire d'exécution
8.         Console.WriteLine("Répertoire d'exécution : " + Environment.CurrentDirectory);
9.         string ligne = null;                // une ligne de texte
10.        StreamWriter fluxInfos = null;   // le fichier texte
11.        try {
12.           // création du fichier texte
13.           using (fluxInfos = new StreamWriter("infos2.txt")) {
14.              Console.WriteLine("Mode AutoFlush : {0}", fluxInfos.AutoFlush);
15.              // lecture ligne tapée au clavier
16.              Console.Write("ligne (rien pour arrêter) : ");
17.              ligne = Console.ReadLine().Trim();
18.              // boucle tant que la ligne saisie est non vide
19.              while (ligne != "") {
20.                 // écriture ligne dans fichier texte
21.                 fluxInfos.WriteLine(ligne);
22.                 // lecture nouvelle ligne au clavier
23.                 Console.Write("ligne (rien pour arrêter) : ");
24.                 ligne = Console.ReadLine().Trim();
```

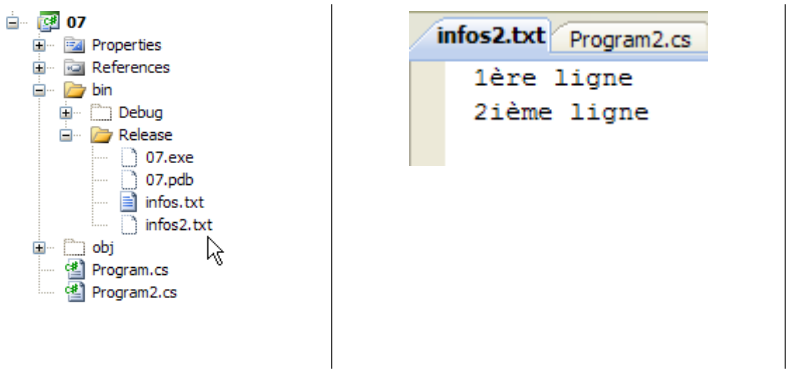
```
25.             }//while
26.           }
27.         } catch (Exception e) {
28.           Console.WriteLine("L'erreur suivante s'est produite : " + e.Message);
29.         }
30.     }
31.  }
32. }
```

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

Les résultats d'exécution sont les suivants :

```
1.  Répertoire d'exécution : C:\data\2007-2008\c# 2008\poly\Chap3\07\bin\Release
2.  Mode AutoFlush : False
3.  ligne (rien pour arrêter) : 1ère ligne
4.  ligne (rien pour arrêter) : 2ième ligne
5.  ligne (rien pour arrêter) :
```

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



5.6 Les fichiers binaires

Les classes **System.IO.BinaryReader** et **System.IO.BinaryWriter** servent à lire et écrire des fichiers binaires.

Considérons l'application suivante :

```
// syntaxe pg texte bin logs
// on lit un fichier texte (texte) et on range son contenu dans un fichier binaire (bin
// le fichier texte a des lignes de la forme nom : age qu'on rangera dans une structure string, int
// (logs) est un fichier texte de logs
```

Le fichier texte a le contenu suivant :

```
1.  paul : 10
2.  helene : 15
3.
4.  jacques : 11
5.  sylvain : 12
6.  xx : -1
7.
8.  xx: yy : zz
9.  xx : yy
```

Le programme est le suivant :

```
1.  using System;
```

```
2.  using System.IO;
3.
4.  // syntaxe pg texte bin logs
5.  // on lit un fichier texte (texte) et on range son contenu dans un fichier binaire (bin)
6.  // le fichier texte a des lignes de la forme nom : age qu'on rangera dans une structure string,
    int
7.  // (logs) est un fichier texte de logs
8.
9.  namespace Chap3 {
10.  class Program {
11.     static void Main(string[] arguments) {
12.        // il faut 3 arguments
13.        if (arguments.Length != 3) {
14.           Console.WriteLine("syntaxe : pg texte binaire log");
15.           Environment.Exit(1);
16.        }//if
17.
18.        // variables
19.        string ligne=null;
20.        string nom=null;
21.        int age=0;
22.        int numLigne = 0;
23.        char[] séparateurs = new char[] { ':' };
24.        string[] champs=null;
25.        StreamReader input = null;
26.        BinaryWriter output = null;
27.        StreamWriter logs = null;
28.        bool erreur = false;
29.        // lecture fichier texte - écriture fichier binaire
30.        try {
31.           // ouverture du fichier texte en lecture
32.           input = new StreamReader(arguments[0]);
33.           // ouverture du fichier binaire en écriture
34.           output = new BinaryWriter(new FileStream(arguments[1], FileMode.Create,
    FileAccess.Write));
35.           // ouverture du fichier des logs en écriture
36.           logs = new StreamWriter(arguments[2]);
37.           // exploitation du fichier texte
38.           while ((ligne = input.ReadLine()) != null) {
39.              // une ligne de plus
40.              numLigne++;
41.              // ligne vide ?
42.              if (ligne.Trim() == "") {
43.                 // on ignore
44.                 continue;
45.              }
46.              // une ligne nom : age
47.              champs = ligne.Split(séparateurs);
48.              // il nous faut 2 champs
49.              if (champs.Length != 2) {
50.                 // on logue l'erreur
51.                 logs.WriteLine("La ligne n° [{0}] du fichier [{1}] a un nombre de champs
    incorrect", numLigne, arguments[0]);
52.                 // ligne suivante
53.                 continue;
54.              }//if
55.              // le 1er champ doit être non vide
56.              erreur = false;
57.              nom = champs[0].Trim();
58.              if (nom == "") {
59.                 // on logue l'erreur
60.                 logs.WriteLine("La ligne n° [{0}] du fichier [{1}] a un nom vide", numLigne,
    arguments[0]);
61.                 erreur = true;
62.              }
63.              // le second champ doit être un entier >=0
64.              if (!int.TryParse(champs[1],out age) || age<0) {
65.                 // on logue l'erreur
66.                 logs.WriteLine("La ligne n° [{0}] du fichier [{1}] a un âge [{2}] incorrect",
    numLigne, arguments[0], champs[1].Trim());
67.                 erreur = true;
68.              }//if
69.              // si pas d'erreur, on écrit les données dans le fichier binaire
70.              if (!erreur) {
71.                 output.Write(nom);
72.                 output.Write(age);
73.              }
74.              // ligne suivante
```

```
75.           }//while
76.        }catch(Exception e){
77.           Console.WriteLine("L'erreur suivante s'est produite : {0}", e.Message);
78.        } finally {
79.           // fermeture des fichiers
80.           if(input!=null) input.Close();
81.           if(output!=null) output.Close();
82.           if(logs!=null) logs.Close();
83.        }
84.     }
85.  }
86. }
```

Attardons-nous sur les opérations concernant la classe *BinaryWriter* :

- ligne 34 : 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

- lignes 70-73 : les opérations d'écriture

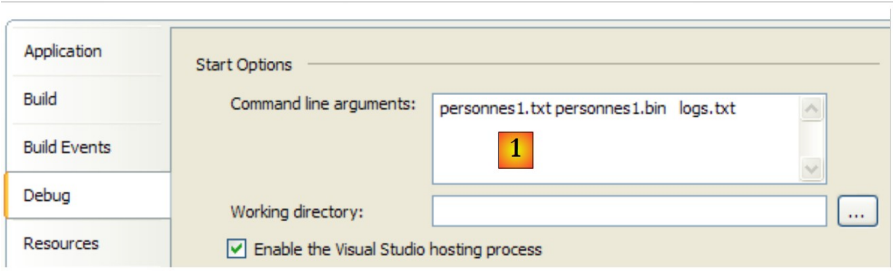
```
                    // on écrit les données dans le fichier binaire
                    output.Write(nom);
                    output.Write(age);
```

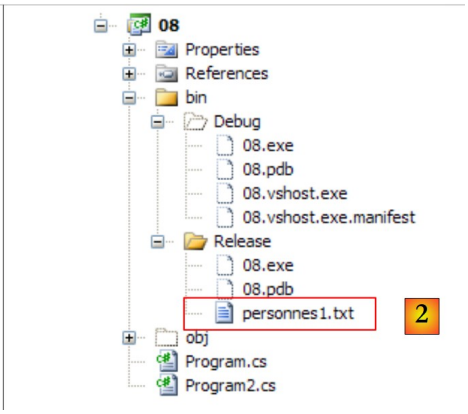
La classe *BinaryWriter* dispose de différentes méthodes *Write* surchargées pour écrire les différents types de données simples

- ligne 81 : l'opération de fermeture du flux

```
                output.Close();
```

Les trois arguments de la méthode *Main* sont donnés au projet (via ses propriétés) [1] et le fichier texte à exploiter est placé dans le dossier *bin/Release* [2] :

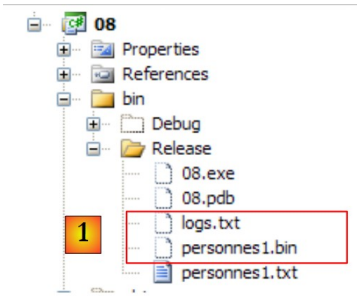




Avec le fichier [personnes1.txt] suivant :

```
1.  paul : 10
2.  helene : 15
3.
4.  jacques : 11
5.  sylvain : 12
6.  xx : -1
7.
8.  xx: yy : zz
9.  xx : yy
```

les résultats de l'exécution sont les suivants :



- en [1], le fichier binaire [personnes1.bin] créé ainsi que le fichier de logs [logs.txt]. Celui-ci a le contenu suivant :

```
1.  La ligne n° [6] du fichier [personnes1.txt] a un âge [-1] incorrect
2.  La ligne n° [8] du fichier [personnes1.txt] a un nombre de champs incorrect
3.  La ligne n° [9] du fichier [personnes1.txt] a un âge [yy] incorrect
```

Le contenu du fichier binaire [personnes1.bin] va nous être donné par le programme qui suit. Celui-ci accepte également trois arguments :

```
// syntaxe pg bin texte logs
// 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
// logs est un fichier texte de logs
```

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 :

```
1.  using System;
2.  using System.IO;
3.
4.  // syntaxe pg bin texte logs
5.  // on lit un fichier binaire bin et on range son contenu dans un fichier texte (texte)
6.  // le fichier binaire a une structure string, int
7.  // le fichier texte a des lignes de la forme nom : age
8.  // logs est un fichier texte de logs
9.
10. namespace Chap3 {
11.  class Program2 {
12.     static void Main(string[] arguments) {
13.        // il faut 3 arguments
14.        if (arguments.Length != 3) {
15.           Console.WriteLine("syntaxe : pg binaire texte log");
16.           Environment.Exit(1);
17.        }//if
18.
19.        // variables
20.        string nom = null;
21.        int age = 0;
22.        int numPersonne = 1;
23.        BinaryReader input = null;
24.        StreamWriter output = null;
25.        StreamWriter logs = null;
26.        bool fini;
27.        // lecture fichier binaire - écriture fichier texte
28.        try {
29.           // ouverture du fichier binaire en lecture
30.           input = new BinaryReader(new FileStream(arguments[0], FileMode.Open, FileAccess.Read));
31.           // ouverture du fichier texte en écriture
32.           output = new StreamWriter(arguments[1]);
33.           // ouverture du fichier des logs en écriture
34.           logs = new StreamWriter(arguments[2]);
35.           // exploitation du fichier binaire
36.           fini = false;
37.           while (!fini) {
38.              try {
```

```
39.                 // lecture nom
40.                 nom = input.ReadString().Trim();
41.                 // lecture age
42.                 age = input.ReadInt32();
43.                 // écriture dans fichier texte
44.                 output.WriteLine(nom + ":" + age);
45.                 // personne suivante
46.                 numPersonne++;
47.              } catch (EndOfStreamException) {
48.                 fini = true;
49.              } catch (Exception e) {
50.                 logs.WriteLine("L'erreur suivante s'est produite à la lecture de la personne n°
    {0} : {1}", numPersonne, e.Message);
51.              }
52.           }//while
53.        } catch (Exception e) {
54.           Console.WriteLine("L'erreur suivante s'est produite : {0}", e.Message);
55.        } finally {
56.           // fermeture des fichiers
57.           if (input != null)
58.              input.Close();
59.           if (output != null)
60.              output.Close();
61.           if (logs != null)
62.              logs.Close();
63.        }
64.     }
65.  }
66. }
```

Attardons-nous sur les opérations concernant la classe *BinaryReader* :

- ligne 30 : 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

- lignes 40, 42 : les opérations de lecture

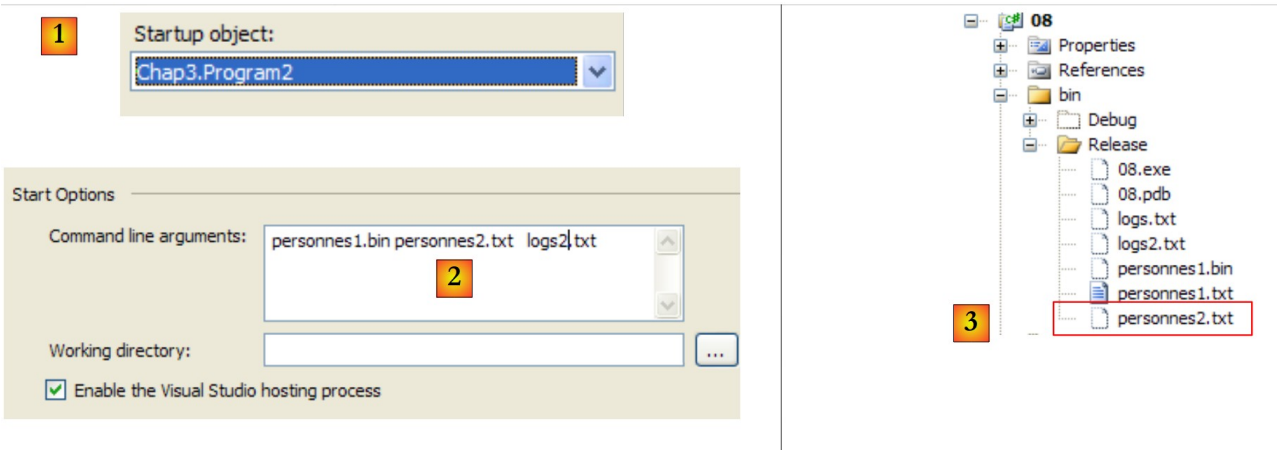
```
nom=input.ReadString().Trim();
age=input.ReadInt32();
```

  La classe *BinaryReader* dispose de différentes méthodes *ReadXX* pour lire les différents types de données simples

- ligne 60 : l'opération de fermeture du flux

```
              input.Close();
```

Si on exécute les deux programmes à la chaîne transformant *personnes1.txt* en *personnes1.bin* puis *personnes1.bin* en *personnes2.txt2* on obtient les résultats suivants :



- en [1], le projet est configuré pour exécuter la 2ième application
- en [2], les arguments passés à *Main*
- en [3], les fichiers produits par l'exécution de l'application.

Le contenu de [personnes2.txt] est le suivant :

```
1. paul:10
2. helene:15
3. jacques:11
4. sylvain:12
```

5.7 Les expressions régulières

La classe **System.Text.RegularExpressions.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 :

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 "fooooot".
{*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 "fooooot". "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 "fooooooot" 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

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*	\b*juste*\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*	\b*secret*\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).

5.7.1 Vérifier qu'une chaîne correspond à un modèle donné

Un objet de type Regex se construit de la façon suivante :

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 :

```
1.  using System;
2.  using System.Text.RegularExpressions;
3.
4.  namespace Chap3 {
5.   class Program {
6.      static void Main(string[] args) {
7.         // une expression régulière modèle
8.         string modèle1 = @"^\s*\d+\s*$";
9.         Regex regex1 = new Regex(modèle1);
10.        // comparer un exemplaire au modèle
11.        string exemplaire1 = "  123  ";
12.        if (regex1.IsMatch(exemplaire1)) {
13.           Console.WriteLine("[{0}] correspond au modèle [{1}]", exemplaire1, modèle1);
14.        } else {
15.           Console.WriteLine("[{0}] ne correspond pas au modèle [{1}]", exemplaire1, modèle1);
16.        }//if
17.        string exemplaire2 = "  123a  ";
18.        if (regex1.IsMatch(exemplaire2)) {
19.           Console.WriteLine("[{0}] correspond au modèle [{1}]", exemplaire2, modèle1);
20.        } else {
21.           Console.WriteLine("[{0}] ne correspond pas au modèle [{1}]", exemplaire2, modèle1);
22.        }//if
23.     }
24.
25.  }
26. }
```

et les résultats d'exécution :

```
1.  [  123  ] correspond au modèle [^\s*\d+\s*$]
2.  [  123a  ] ne correspond pas au modèle [^\s*\d+\s*$]
```

5.7.2 Trouver toutes les occurrences d'un modèle dans une chaîne

La méthode **Matches** permet de récupérer les éléments d'une chaîne correspondant à un modèle :

public MatchCollection Matches(string input)	rend la collection des éléments de la chaîne *input* correspondant au modèle

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**. La classe *Match* a diverses propriétés dont les suivantes :

- **Value** : la valeur de l'objet *Match,* donc un élément correspondant au modèle
- **Index** : la position où l'élément a été trouvé dans la chaîne explorée

Examinons l'exemple suivant :

```
1.  using System;
2.  using System.Text.RegularExpressions;
3.
4.  namespace Chap3 {
5.   class Program2 {
6.      static void Main(string[] args) {
7.         // plusieurs occurrences du modèle dans l'exemplaire
8.         string modèle2 = @"\d+";
9.         Regex regex2 = new Regex(modèle2);
10.        string exemplaire3 = "  123  456  789 ";
11.        MatchCollection résultats = regex2.Matches(exemplaire3);
12.        Console.WriteLine("Modèle=[{0}],exemplaire=[{1}]", modèle2, exemplaire3);
13.        Console.WriteLine("Il y a {0} occurrences du modèle dans l'exemplaire ", résultats.Count);
14.        for (int i = 0; i < résultats.Count; i++) {
15.           Console.WriteLine("[{0}] trouvé en position {1}", résultats[i].Value,
    résultats[i].Index);
16.        }//for
17.     }
18.  }
19. }
```

- ligne 8 : le modèle recherché est une suite de chiffres
- ligne 10 : la chaîne dans laquelle on recherche ce modèle
- ligne 11 : on récupère tous les éléments de *exemplaire3* vérifiant le modèle *modèle2*
- lignes 14-16 : on les affiche

Les résultats de l'exécution du programme sont les suivants :

```
1.  Modèle=[\d+],exemplaire=[  123  456  789 ]
2.  Il y a 3 occurrences du modèle dans l'exemplaire
3.  [123] trouvé en position 2
4.  [456] trouvé en position 7
5.  [789] trouvé en position 12
```

5.7.3 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 :

```
1.  using System;
2.  using System.Text.RegularExpressions;
3.
4.  namespace Chap3 {
5.   class Program3 {
6.      static void Main(string[] args) {
7.         // capture d'éléments dans le modèle
8.         string modèle3 = @"(\d\d):(\d\d):(\d\d)";
9.         Regex regex3 = new Regex(modèle3);
10.        string exemplaire4 = "Il est 18:05:49";
11.        // vérification modèle
12.        Match résultat = regex3.Match(exemplaire4);
13.        if (résultat.Success) {
14.           // l'exemplaire correspond au modèle
15.           Console.WriteLine("L'exemplaire [{0}] correspond au modèle [{1}]",exemplaire4,modèle3);
16.           // on affiche les groupes de parenthèses
17.           for (int i = 0; i < résultat.Groups.Count; i++) {
18.              Console.WriteLine("groupes[{0}]=[{1}] trouvé en position {2}",i,
    résultat.Groups[i].Value,résultat.Groups[i].Index);
19.           }//for
20.        } else {
21.           // l'exemplaire ne correspond pas au modèle
22.           Console.WriteLine("L'exemplaire[{0}] ne correspond pas au modèle [{1}]", exemplaire4,
    modèle3);
23.        }
24.     }
25.  }
26. }
```

L'exécution de ce programme produit les résultats suivants :

```
1.  L'exemplaire [Il est 18:05:49] correspond au modèle [(\d\d):(\d\d):(\d\d)]
2.  groupes[0]=[18:05:49] trouvé en position 7
3.  groupes[1]=[18] trouvé en position 7
4.  groupes[2]=[05] trouvé en position 10
5.  groupes[3]=[49] trouvé en position 13
```

La nouveauté se trouve dans les lignes 12-19 :

- ligne 12 : 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** (ligne 13) : indique s'il y a eu correspondance
  - **Groups** (lignes 17, 18) : 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** (ligne 18) : la valeur de l'objet *Group* qui est l'élément correspondant au contenu d'une parenthèse
- **Index** (ligne 18) : la position où l'élément a été trouvé dans la chaîne explorée

5.7.4 Un programme d'apprentissage

Trouver l'expression régulière qui 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 si la chaîne correspond ou non au modèle.

```
1.  using System;
2.  using System.Text.RegularExpressions;
3.
4.  namespace Chap3 {
5.   class Program4 {
6.      static void Main(string[] args) {
7.         // données
8.         string modèle, chaine;
9.         Regex regex = null;
10.        MatchCollection résultats;
11.        // on demande à l'utilisateur les modèles et les exemplaires à comparer à celui-ci
12.        while (true) {
13.           // on demande le modèle
14.           Console.Write("Tapez le modèle à tester ou rien pour arrêter :");
15.           modèle = Console.In.ReadLine();
16.           // fini ?
17.           if (modèle.Trim() == "")
18.              break;
19.           // on crée l'expression régulière
20.           try {
21.              regex = new Regex(modèle);
22.           } catch (Exception ex) {
23.              Console.WriteLine("Erreur : " + ex.Message);
24.              continue;
25.           }
26.           // on demande à l'utilisateur les exemplaires à comparer au modèle
27.           while (true) {
28.              Console.Write("Tapez la chaîne à comparer au modèle [{0}] ou rien pour arrêter :",
    modèle);
29.              chaine = Console.ReadLine();
30.              // fini ?
31.              if (chaine.Trim() == "")
32.                 break;
33.              // on fait la comparaison
34.              résultats = regex.Matches(chaine);
35.              // succès ?
36.              if (résultats.Count == 0) {
37.                 Console.WriteLine("Je n'ai pas trouvé de correspondances");
38.                 continue;
39.              }//if
40.              // on affiche les éléments correspondant au modèle
41.              for (int i = 0; i < résultats.Count; i++) {
42.                 Console.WriteLine("J'ai trouvé la correspondance [{0}] en position [{1}]",
    résultats[i].Value, résultats[i].Index);
43.                 // des sous-éléments
44.                 if (résultats[i].Groups.Count != 1) {
45.                    for (int j = 1; j < résultats[i].Groups.Count; j++) {
46.                       Console.WriteLine("\tsous-élément [{0}] en position [{1}]",
    résultats[i].Groups[j].Value, résultats[i].Groups[j].Index);
47.                    }
48.                 }
49.              }
50.           }
51.        }
52.     }
53.  }
54. }
```

Voici un exemple d'exécution :

```
1.  Tapez le modèle à tester ou rien pour arrêter :\d+
```

```
2.  Tapez la chaîne à comparer au modèle [\d+] ou rien pour arrêter :123 456 789
3.  J'ai trouvé la correspondance [123] en position [0]
4.  J'ai trouvé la correspondance [456] en position [4]
5.  J'ai trouvé la correspondance [789] en position [8]
6.  Tapez la chaîne à comparer au modèle [\d+] ou rien pour arrêter :
7.  Tapez le modèle à tester ou rien pour arrêter :(\d{2}):(\d\d)
8.  Tapez la chaîne à comparer au modèle [(\d{2}):(\d\d)] ou rien pour arrêter :14:15 abcd 17:18 xyzt
9.  J'ai trouvé la correspondance [14:15] en position [0]
10.         sous-élément [14] en position [0]
11.         sous-élément [15] en position [3]
12. J'ai trouvé la correspondance [17:18] en position [11]
13.         sous-élément [17] en position [11]
14.         sous-élément [18] en position [14]
15. Tapez la chaîne à comparer au modèle [(\d{2}):(\d\d)] ou rien pour arrêter :
16. Tapez le modèle à tester ou rien pour arrêter :^\s*\d+\s*$
17. Tapez la chaîne à comparer au modèle [^\s*\d+\s*$] ou rien pour arrêter :   1456
18. J'ai trouvé la correspondance [   1456] en position [0]
19. Tapez la chaîne à comparer au modèle [^\s*\d+\s*$] ou rien pour arrêter :
20. Tapez le modèle à tester ou rien pour arrêter :^\s*(\d+)\s*$
21. Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou rien pour arrêter :1456
22. J'ai trouvé la correspondance [1456] en position [0]
23.         sous-élément [1456] en position [0]
24. Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou rien pour arrêter :abcd 1456
25. Je n'ai pas trouvé de correspondances
26. Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou rien pour arrêter :
27. Tapez le modèle à tester ou rien pour arrêter :
```

5.7.5 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

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 qu'on ait 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 :

```
1.  using System;
2.  using System.Text.RegularExpressions;
3.
4.  namespace Chap3 {
5.   class Program5 {
6.      static void Main(string[] args) {
7.         // une ligne
8.         string ligne = "abc  , def  , ghi";
9.         // un modèle
10.        Regex modèle = new Regex(@"\s*,\s*");
11.        // décomposition de ligne en champs
12.        string[] champs = modèle.Split(ligne);
13.        // affichage
14.        for (int i = 0; i < champs.Length; i++) {
15.           Console.WriteLine("champs[{0}]=[{1}]", i, champs[i]);
16.        }
17.     }
18.  }
19. }
```

Les résultats d'exécution :

```
champs[0]=[abc]
champs[1]=[def]
champs[2]=[ghi]
```

5.8 Application exemple - V3

Nous reprenons l'application étudiée aux paragraphes 3.6 page 32 (version 1) et 4.10 page 85 (version 2).

Dans la dernière version étudiée, le calcul de l'impôt se faisait dans la classe abstraite *AbstractImpot* :

```
29. namespace Chap2 {
30.  abstract class AbstractImpot : IImpot {
31.
32.     // les tranches d'impôt nécessaires au calcul de l'impôt
33.     // proviennent d'une source extérieure
34.
35.     protected TrancheImpot[] tranchesImpot;
36.
37.     // calcul de l'impôt
38.     public int calculer(bool marié, int nbEnfants, int salaire) {
39.        // calcul du nombre de parts
40.        decimal nbParts;
41.        if (marié) nbParts = (decimal)nbEnfants / 2 + 2;
42.        else nbParts = (decimal)nbEnfants / 2 + 1;
43.        if (nbEnfants >= 3) nbParts += 0.5M;
44.        // calcul revenu imposable & Quotient familial
45.        decimal revenu = 0.72M * salaire;
46.        decimal QF = revenu / nbParts;
47.        // calcul de l'impôt
48.        tranchesImpot[tranchesImpot.Length - 1].Limite = QF + 1;
49.        int i = 0;
50.        while (QF > tranchesImpot[i].Limite) i++;
51.        // retour résultat
52.        return (int)(revenu * tranchesImpot[i].CoeffR - nbParts * tranchesImpot[i].CoeffN);
53.     }//calculer
54.  }//classe
55.
56. }
```

La méthode *calculer* de la ligne 38 utilise le tableau *tranchesImpot* de la ligne 35, tableau non initialisé par la classe *AbstractImpot*. C'est pourquoi elle est abstraite et doit être dérivée pour être utile. Cette initialisation était faite par la classe dérivée *HardwiredImpot* :

```
21. using System;
22.
23. namespace Chap2 {
24.  class HardwiredImpot : AbstractImpot {
25.
26.     // tableaux de données nécessaires au calcul de l'impôt
27.     decimal[] limites = { 4962M, 8382M, 14753M, 23888M, 38868M, 47932M, 0M };
28.     decimal[] coeffR = { 0M, 0.068M, 0.191M, 0.283M, 0.374M, 0.426M, 0.481M };
29.     decimal[] coeffN = { 0M, 291.09M, 1322.92M, 2668.39M, 4846.98M, 6883.66M, 9505.54M };
30.
31.     public HardwiredImpot() {
32.           // création du tableau des tranches d'impôt
33.        tranchesImpot = new TrancheImpot[limites.Length];
34.           // remplissage
35.        for (int i = 0; i < tranchesImpot.Length; i++) {
36.           tranchesImpot[i] = new TrancheImpot { Limite = limites[i], CoeffR = coeffR[i], CoeffN =
    coeffN[i] };
37.           }
38.     }
39.  }// classe
40. }// namespace
```

Ci-dessus, les données nécessaires au calcul de l'impôt étaient placées en "dur" dans le code de la classe. La nouvelle version de l'exemple les place dans un fichier texte :

```
4962:0:0
8382:0,068:291,09
14753:0,191:1322,92
```

```
23888:0,283:2668,39
38868:0,374:4846,98
47932:0,426:6883,66
0:0,481:9505,54
```

L'exploitation de ce fichier pouvant produire des exceptions, nous créons une classe spéciale pour gérer ces dernières :

```
1.  using System;
2.
3.  namespace Chap3 {
4.   class FileImpotException : Exception {
5.      // codes d'erreur
6.      [Flags]
7.      public enum CodeErreurs { Acces = 1, Ligne = 2, Champ1 = 4, Champ2 = 8, Champ3 = 16 };
8.
9.      // code d'erreur
10.     public CodeErreurs Code { get; set; }
11.
12.     // constructeurs
13.     public FileImpotException() {
14.     }
15.     public FileImpotException(string message)
16.        : base(message) {
17.     }
18.     public FileImpotException(string message, Exception e)
19.        : base(message,e) {
20.     }
21.  }
22. }
```

- ligne 4 : la classe *FileImpotException* dérive de la classe *Exception*. Elle servira à mémoriser toute erreur survenant lors de l'exploitation du fichier texte des données.
- ligne 7 : une énumération représentant des codes d'erreur :
  - *Acces* : erreur d'accès au fichier texte des données
  - *Ligne* : ligne n'ayant pas les trois champs attendus
  - *Champ1* : le champ n° 1 est erroné
  - *Champ2* : le champ n° 2 est erroné
  - *Champ3* : le champ n° 3 est erroné

  Certaines de ces erreurs peuvent se combiner (*Champ1*, *Champ2*, *Champ3*). Aussi l'énumération *CodeErreurs* a-t-elle été annotée avec l'attribut [Flags] qui implique que les différentes valeurs de l'énumération doivent être des puissances de 2. Une erreur sur les champs 1 et 2 se traduira alors par le code d'erreur *Champ1* | *Champ2*.
- ligne 10 : la propriété automatique *Code* mémorisera le code de l'erreur.
- lignes 15 : un constructeur permettant de construire un objet *FileImpotException* en lui passant comme paramètre un message d'erreur.
- lignes 18 : un constructeur permettant de construire un objet *FileImpotException* en lui passant comme paramètres un message d'erreur et l'exception à l'origine de l'erreur.

La classe qui initialise le tableau *tranchesImpot* de la classe *AbstractImpot* est désormais la suivante :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.IO;
4.  using System.Text.RegularExpressions;
5.
6.  namespace Chap3 {
7.   class FileImpot : AbstractImpot {
8.
9.      public FileImpot(string fileName) {
10.        // données
11.        List<TrancheImpot> listTranchesImpot = new List<TrancheImpot>();
12.        int numLigne = 1;
13.        // exception
14.        FileImpotException fe = null;
15.        // lecture contenu du fichier fileName, ligne par ligne
16.        Regex pattern = new Regex(@"s*:\s*");
17.        // au départ pas d'erreur
18.        FileImpotException.CodeErreurs code = 0;
19.        try {
20.           using (StreamReader input = new StreamReader(fileName)) {
21.              while (!input.EndOfStream && code == 0) {
22.                 // ligne courante
23.                 string ligne = input.ReadLine().Trim();
```

```
24.                 // on ignore les lignes vides
25.                 if (ligne == "") continue;
26.                 // ligne décomposée en trois champs séparés par :
27.                 string[] champsLigne = pattern.Split(ligne);
28.                 // a-t-on 3 champs ?
29.                 if (champsLigne.Length != 3) {
30.                    code = FileImpotException.CodeErreurs.Ligne;
31.                 }
32.                 // conversions des 3 champs
33.                 decimal limite = 0, coeffR = 0, coeffN = 0;
34.                 if (code == 0) {
35.                    if (!Decimal.TryParse(champsLigne[0], out limite)) code =
    FileImpotException.CodeErreurs.Champ1;
36.                    if (!Decimal.TryParse(champsLigne[1], out coeffR)) code |=
    FileImpotException.CodeErreurs.Champ2;
37.                    if (!Decimal.TryParse(champsLigne[2], out coeffN)) code |=
    FileImpotException.CodeErreurs.Champ3; ;
38.                 }
39.                 // erreur ?
40.                 if (code != 0) {
41.                    // on note l'erreur
42.                    fe = new FileImpotException(String.Format("Ligne n° {0} incorrecte", numLigne))
    { Code = code };
43.                 } else {
44.                    // on mémorise la nouvelle tranche d'impôt
45.                    listTranchesImpot.Add(new TrancheImpot() { Limite = limite, CoeffR = coeffR,
    CoeffN = coeffN });
46.                    // ligne suivante
47.                    numLigne++;
48.                 }
49.              }
50.           }
51.           // on transfère la liste listImpot dans le tableau tranchesImpot
52.           if (code == 0) {
53.              // on transfère la liste listImpot dans le tableau tranchesImpot
54.              tranchesImpot = listTranchesImpot.ToArray();
55.           }
56.        } catch (Exception e) {
57.           // on note l'erreur
58.           fe= new FileImpotException(String.Format("Erreur lors de la lecture du fichier {0}",
    fileName), e) { Code = FileImpotException.CodeErreurs.Acces };
59.        }
60.        // erreur à signaler ?
61.        if (fe != null) throw fe;
62.     }
63.  }
64. }
```

- ligne 7 : la classe *FileImpot* dérive de la classe *AbstractImpot* comme le faisait dans la version 2 la classe *HardwiredImpot*.
- ligne 9 : le constructeur de la classe *FileImpot* a pour rôle d'initialiser le champ *trancheImpot* de sa classe de base *AbstractImpot*. Il admet pour paramètre, le nom du fichier texte contenant les données.
- ligne 11 : le champ *tranchesImpot* de la classe de base *AbstractImpot* est un tableau qui a être rempli avec les données du fichier *filename* passé en paramètre. La lecture d'un fichier texte est séquentielle. On ne connaît le nombre de lignes qu'après avoir lu la totalité du fichier. Aussi ne peut-on dimensionner le tableau *tranchesImpot*. On mémorisera momentanément les données dans la liste générique *listTranchesImpot*.

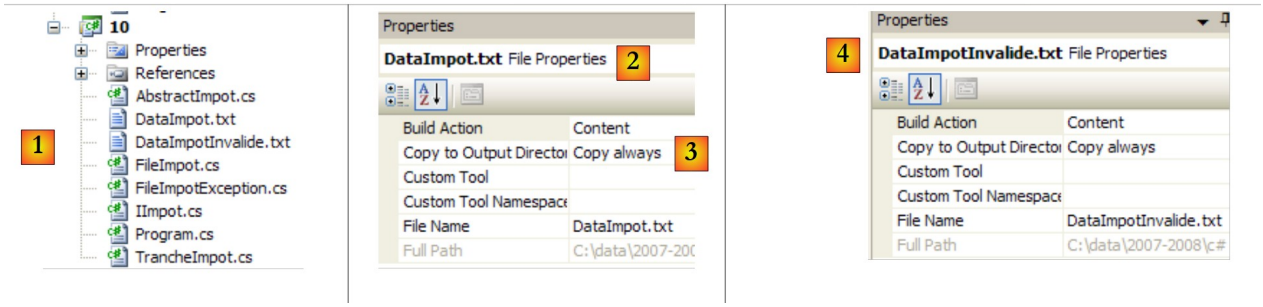
On rappelle que le type *TrancheImpot* est une structure :

```
namespace Chap3 {
  // une tranche d'impôt
  struct TrancheImpot {
     public decimal Limite { get; set; }
     public decimal CoeffR { get; set; }
     public decimal CoeffN { get; set; }
  }
}
```

- ligne 14 : *fe* de type *FileImpotException* sert à encapsuler une éventuelle erreur d'exploitation du fichier texte.
- ligne 16 : l'expression régulière du séparateur de champs dans une ligne *champ1:champ2:champ3* du fichier texte. Les champs sont séparés par le caractère : précédé et suivi d'un nombre quelconque d'espaces.
- ligne 18 : le code de l'erreur en cas d'erreur
- ligne 20 : exploitation du fichier texte avec un *StreamReader*
- ligne 21 : on boucle tant qu'il reste une ligne à lire et qu'il n'y a pas eu d'erreur

- ligne 27 : la ligne lue est divisée en champs grâce à l'expression régulière de la ligne 16
- lignes 29-31 : on vérifie que la ligne a bien trois champs - on note une éventuelle erreur
- lignes 33-38 : conversion des trois chaînes en trois nombres décimaux - on note les éventuelles erreurs
- lignes 40-43 : s'il y a eu erreur, une exception de type *FileImpotException* est créée.
- lignes 44-47 : s'il n'y a pas eu d'erreur, on passe à la lecture de la ligne suivante du fichier texte après avoir mémorisé les données issues de la ligne courante.
- lignes 52-55 : à la sortie de la bouche *while*, les données de la liste générique *listTranchesImpot* sont recopiées dans le tableau *tranchesImpot* de la classe de base *AbstractImpot*. On rappelle que tel était le but du constructeur.
- lignes 56-59 : gestion d'une éventuelle exception. Celle-ci est encapsulée dans un objet de type *FileImpotException*.
- ligne 61 : si l'exception *fe* de la ligne 18 a été initialisée, alors elle est lancée.

L'ensemble du projet C# est le suivant :



- en [1] : l'ensemble du projet
- en [2,3] : les propriétés du fichier [DataImpot.txt] [2]. La propriété [Copy to Output Directory] [3] est mise à **always**. Ceci fait que le fichier [DataImpot.txt] sera copié dans le dossier *bin/Release* (mode Release) ou *bin/Debug* (mode Debug) à chaque exécution. C'est là qu'il est cherché par l'exécutable.
- en [4] : on fait de même avec le fichier [DataImpotInvalide.txt].

Le contenu de [DataImpot.txt] est le suivant :

```
4962:0:0
8382:0,068:291,09
14753:0,191:1322,92
23888:0,283:2668,39
38868:0,374:4846,98
47932:0,426:6883,66
0:0,481:9505,54
```

Le contenu de [DataImpotInvalide.txt] est le suivant :

```
a:b:c
```

Le programme de test [Program.cs] n'a pas changé : c'est celui de la version 2 page 87, à la différence près suivante :

```
1.  using System;
2.
3.  namespace Chap3 {
4.   class Program {
5.      static void Main() {
6.  ...
7.         // création d'un objet IImpot
8.         IImpot impot = null;
9.         try {
10.           // création d'un objet IImpot
11.           impot = new FileImpot("DataImpot.txt");
12.        } catch (FileImpotException e) {
13.           // affichage erreur
14.           string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
15.           Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
16.           // arrêt programme
17.           Environment.Exit(1);
```

```
18.         }
19.
20.         // boucle infinie
21.         while (true) {
22. ...
23.         }//while
24.     }
25.  }
26. }
```

- ligne 8 : objet *impot* du type de l'interface *IImpot*
- ligne 11 : instanciation de l'objet *impot* avec un objet de type *FileImpot*. Celle-ci peut générer une exception qui est gérée par le try / catch des lignes 9 / 12 / 18.

Voici des exemples d'exécution :

Avec le fichier [*DataImpot.txt*]

```
1.  Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :o
    2 60000
2.  Impot=4282 euros
3.  Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :
```

Avec un fichier [*xx*] inexistant

```
1.  L'erreur suivante s'est produite : [Code=Acces,Message=Erreur lors de la lecture du fichier xx,
    Exception d'origine : Could not find file 'C:\data\2007-
    2008\c#2008\poly\Chap3\10\bin\Release\xx'.]
```

Avec le fichier [*DataImpotInvalide.txt*]

```
1.  L'erreur suivante s'est produite : [Code=Champ1, Champ2, Champ3,Message=Ligne n° 1 incorrecte]
```

6 Architectures 3 couches

6.1 Introduction

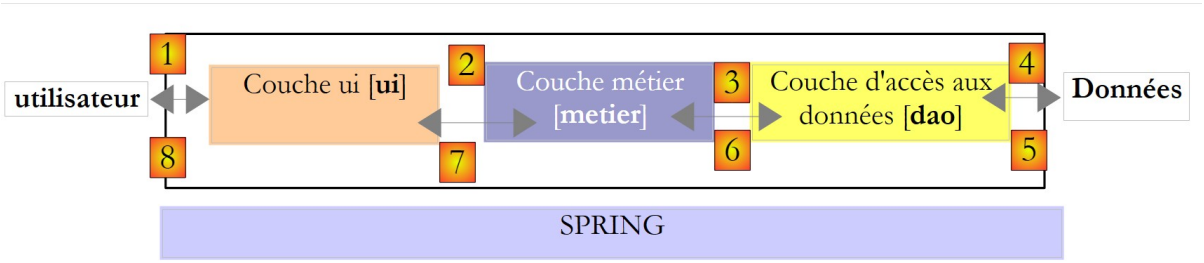
Revenons sur la dernière version de l'application de calcul d'impôt :

```
1.  using System;
2.
3.  namespace Chap3 {
4.   class Program {
5.      static void Main() {
6.         // programme interactif de calcul d'Impot
7.         // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
8.         // le programme affiche alors l'Impot à payer
9.  ...
10.
11.        // création d'un objet IImpot
12.        IImpot impot = null;
13.        try {
14.           // création d'un objet IImpot
15.           impot = new FileImpot("DataImpotInvalide.txt");
16.        } catch (FileImpotException e) {
17.           // affichage erreur
18. ...
19.           // arrêt programme
20.           Environment.Exit(1);
21.        }
22.        // boucle infinie
23.        while (true) {
24.           // on demande les paramètres du calcul de l'impôt
25.           Console.Write("Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire
    ou rien pour arrêter :");
26.           string paramètres = Console.ReadLine().Trim();
27. ...
28.           // les paramètres sont corrects - on calcule l'Impot
29.           Console.WriteLine("Impot=" + impot.calculer(marié == "o", nbEnfants, salaire) + "
    euros");
30.           // contribuable suivant
31.        }//while
32.     }
33.  }
34. }
```

La solution précédente inclut des traitements classiques en programmation :

1. la récupération de données mémorisées dans des fichiers, bases de données, ... lignes 12-21
2. le dialogue avec l'utilisateur, lignes 26 (saisies) et 29 (affichages)
3. l'utilisation d'un algorithme métier, ligne 29

La pratique a montré qu'isoler ces différents traitements dans des classes séparées améliorait la maintenabilité des applications. L'architecture d'une application ainsi structurée est la suivante :



On appelle cette architecture, "**architecture trois tiers**", traduction de l'anglais "three tier architecture". Le terme "trois tiers" désigne normalement une architecture où chaque tier est sur une machine différente. Lorsque les tiers sont sur une même machine, l'architecture devient une architecture "**trois couches**".

- la couche [metier] est celle qui contient les règles métier de l'application. Pour notre application de calcul d'impôt, ce sont les règles qui permettent de calculer l'impôt d'un contribuable. Cette couche a besoin de données pour travailler :
  - les tranches d'impôt, données qui changent chaque année
  - le nombre d'enfants, le statut marital et le salaire annuel du contribuable

  Dans le schéma ci-dessus, les données peuvent provenir de deux endroits :

  - la couche d'accès aux données ou [dao] (DAO = Data Access Object) pour les données déjà enregistrées dans des fichiers ou bases de données. Ce pourrait être le cas ici des tranches d'impôt comme il a été fait dans la version précédente de l'application.
  - la couche d'interface avec l'utilisateur ou [ui] (UI = User Interface) pour les données saisies par l'utilisateur ou affichées à l'utilisateur. Ce pourrait être le cas ici du nombre d'enfants, du statut marital et du salaire annuel du contribuable

- de façon générale, la couche [dao] s'occupe de l'accès aux données persistantes (fichiers, bases de données) ou non persistantes (réseau, capteurs, ...).
- la couche [ui] elle, s'occupe des interactions avec l'utilisateur s'il y en a un.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces.

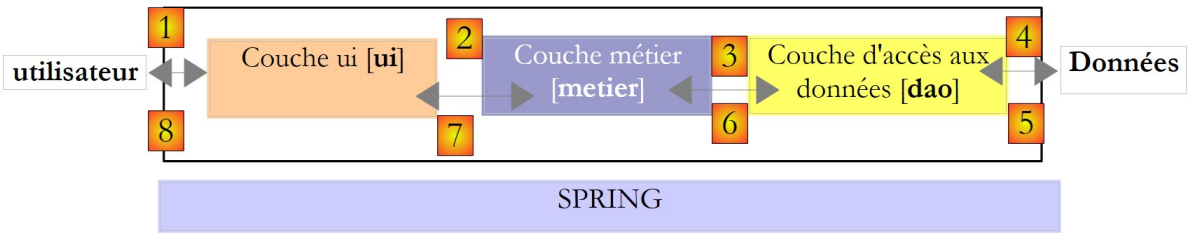
Nous allons reprendre l'application [Impots] déjà étudiée à plusieurs reprises pour lui donner une architecture 3 couches. Pour cela, nous allons étudier les couches [ui, metier, dao] les unes après les autres, en commençant par la couche [dao], couche qui s'occupe des données persistantes.

Auparavant, il nous faut définir les interfaces des différentes couches de l'application [Impots].

6.2 Les interfaces de l'application [Impots]

Rappelons qu'une interface définit un ensemble de signatures de méthodes. Les classes implémentant l'interface donnent un contenu à ces méthodes.

Revenons à l'architecture 3 couches de notre application :

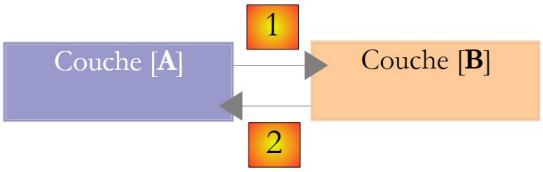


Dans ce type d'architecture, c'est souvent l'utilisateur qui prend les initiatives. Il fait une demande en [1] et reçoit une réponse en [8]. On appelle cela le cycle demande - réponse. Prenons l'exemple du calcul de l'impôt d'un contribuable. Celui-ci va nécessiter plusieurs étapes :

(a) la couche [ui] va devoir demander à l'utilisateur son nombre d'enfants, son statut marital et son salaire annuel. C'est l'opération [1] ci-dessus.
(b) ceci fait, la couche [ui] va demander à la couche métier de faire le calcul de l'impôt. Pour cela elle va lui transmettre les données qu'elle a reçues de l'utilisateur. C'est l'opération [2].
(c) la couche [metier] a besoin de certaines informations pour mener à bien son travail : les tranches d'impôt. Elle va demander ces informations à la couche [dao] avec le chemin [3, 4, 5, 6]. [3] est la demande initiale et [6] la réponse à cette demande.
(d) ayant toutes les données dont elle avait besoin, la couche [metier] calcule l'impôt.
(e) la couche [metier] peut maintenant répondre à la demande de la couche [ui] faite en (b). C'est le chemin [7].

(f) la couche [ui] va mettre en forme ces résultats puis les présenter à l'utilisateur. C'est le chemin [8].
(g) on pourrait imaginer que l'utilisateur fait des simulations d'impôt et qu'il veuille mémoriser celles-ci. Il utilisera le chemin [1-8] pour le faire.

On voit dans cette description qu'une couche est amenée à utiliser les ressources de la couche qui est à sa droite, jamais de celle qui est à sa gauche. Considérons deux couches contigües :



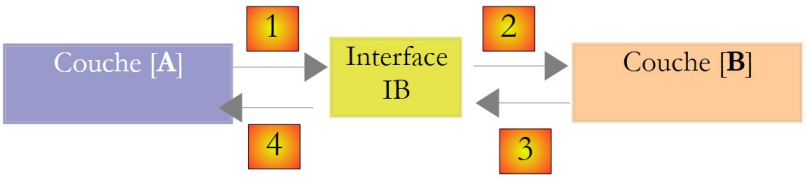
La couche [A] fait des demandes à la couche [B]. Dans les cas les plus simples, une couche est implémentée par une unique classe. Une application évolue au cours du temps. Ainsi la couche [B] peut avoir des classes d'implémentation différentes [B1, B2, ...]. Si la couche [B] est la couche [dao], celle-ci peut avoir une première implémentation [B1] qui va chercher des données dans un fichier. Quelques années plus tard, on peut vouloir mettre les données dans une base de données. On va alors construire une seconde classe d'implémentation [B2]. Si dans l'application initiale, la couche [A] travaillait directement avec la classe [B1] on est obligés de réécrire partiellement le code de la couche [A]. Supposons par exemple qu'on ait écrit dans la couche [A] quelque chose comme suit :

```
1. B1 b1=new B1(...);
2. ..
3. b1.getData(...);
```

- ligne 1 : une instance de la classe [B1] est créée
- ligne 3 : des données sont demandées à cette instance

Si on suppose, que la nouvelle classe d'implémentation [B2] utilise des méthodes de même signature que celle de la classe [B1], il faudra changer tous les [B1] en [B2]. Ca, c'est le cas très favorable et assez improbable si on n'a pas prêté attention à ces signatures de méthodes. Dans la pratique, il est fréquent que les classes [B1] et [B2] n'aient pas les mêmes signatures de méthodes et que donc une bonne partie de la couche [A] doive être totalement réécrite.

On peut améliorer les choses si on met une interface entre les couches [A] et [B]. Cela signifie qu'on fige dans une interface les signatures des méthodes présentées par la couche [B] à la couche [A]. Le schéma précédent devient alors le suivant :



La couche [A] ne s'adresse désormais plus directement à la couche [B] mais à son interface [IB]. Ainsi dans le code de la couche [A], la classe d'implémentation [Bi] de la couche [B] n'apparaît qu'une fois, au moment de l'implémentation de l'interface [IB]. Ceci fait, c'est l'interface [IB] et non sa classe d'implémentation qui est utilisée dans le code. Le code précédent devient celui-ci :

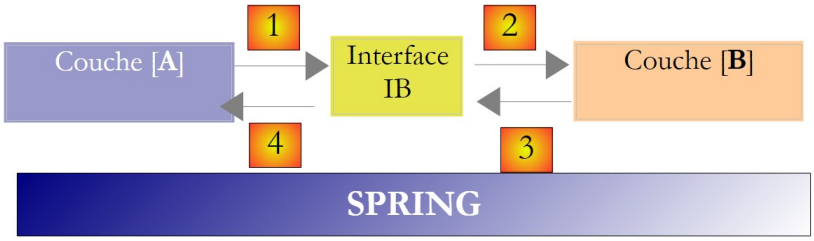
```
1. IB ib=new B1(...);
2. ..
3. ib.getData(...);
```

- ligne 1 : une instance [ib] implémentant l'interface [IB] est créée par instanciation de la classe [B1]
- ligne 3 : des données sont demandées à l'instance [ib]

Désormais si on remplace l'implémentation [B1] de la couche [B] par une implémentation [B2], et que ces deux implémentations respectent la même interface [IB], alors seule la ligne 1 de la couche [A] doit être modifiée et aucune autre. C'est un grand avantage qui à lui seul justifie l'usage systématique des interfaces entre deux couches.

On peut aller encore plus loin et rendre la couche [A] totalement indépendante de la couche [B]. Dans le code ci-dessus, la ligne 1 pose problème parce qu'elle référence en dur la classe [B1]. L'idéal serait que la couche [A] puisse disposer d'une implémentation de l'interface [IB] sans avoir à nommer de classe. Ce serait cohérent avec notre schéma ci-dessus. On y voit que la couche [A] s'adresse à l'interface [IB] et on ne voit pas pourquoi elle aurait besoin de connaître le nom de la classe qui implémente cette interface. Ce détail n'est pas utile à la couche [A].

Le framework Spring (http://www.springframework.org) permet d'obtenir ce résultat. L'architecture précédente évolue de la façon suivante :



La couche transversale [Spring] va permettre à une couche d'obtenir par configuration une référence sur la couche située à sa droite sans avoir à connaître le nom de la classe d'implémentation de la couche. Ce nom sera dans les fichiers de configuration et non dans le code C#. Le code C# de la couche [A] prend alors la forme suivante :

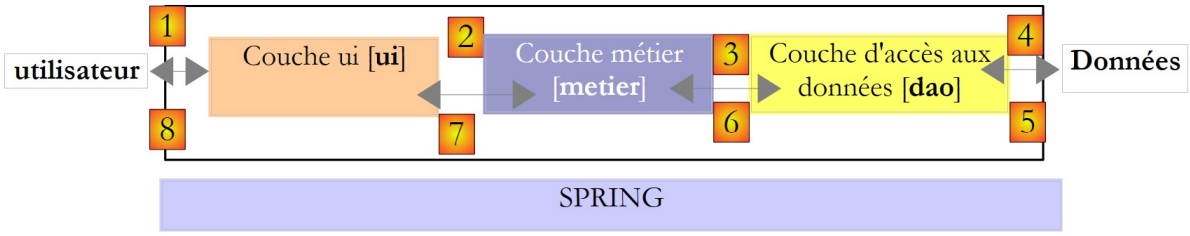
```
1.  IB ib; // initialisé par Spring
2.  ..
3.  ib.getData(...);
```

- ligne 1 : une instance [ib] implémentant l'interface [IB] de la couche [B]. Cette instance est créée par Spring sur la base d'informations trouvées dans un fichier de configuration. Spring va s'occuper de créer :
  - l'instance [b] implémentant la couche [B]
  - l'instance [a] implémentant la couche [A]. Cette instance sera initialisée. Le champ [ib] ci-dessus recevra pour valeur la référence [b] de l'objet implémentant la couche [B]
- ligne 3 : des données sont demandées à l'instance [ib]

On voit maintenant que, la classe d'implémentation [B1] de la couche B n'apparaît nulle part dans le code de la couche [A]. Lorsque l'implémentation [B1] sera remplacée par une nouvelle implémentation [B2], rien ne changera dans le code de la classe [A]. On changera simplement les fichiers de configuration de Spring pour instancier [B2] au lieu de [B1].

Le couple **Spring** et **interfaces C#** apporte une amélioration décisive à la maintenance d'applications en rendant les couches de celles-ci étanches entre elles. C'est cette solution que nous utiliserons pour une nouvelle version de l'application [Impots].

Revenons à l'architecture trois couches de notre application :



Dans les cas simples, on peut partir de la couche [metier] pour découvrir les interfaces de l'application. Pour travailler, elle a besoin de données :

- déjà disponibles dans des fichiers, bases de données ou via le réseau. Elles sont fournies par la couche [dao].
- pas encore disponibles. Elles sont alors fournies par la couche [ui] qui les obtient auprès de l'utilisateur de l'application.

Quelle interface doit offrir la couche [dao] à la couche [metier] ? Quelles sont les interactions possibles entre ces deux couches ? La couche [dao] doit fournir les données suivantes à la couche [metier] :

- les tranches d'impôt

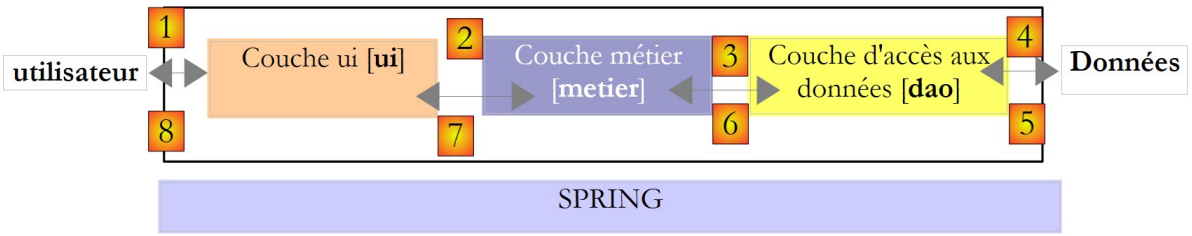
Dans notre application, la couche [dao] exploite des données existantes mais n'en crée pas de nouvelles. Une définition de l'interface de la couche [dao] pourrait être la suivante :

```
1.  using Entites;
2.
3.  namespace Dao {
4.   public interface IImpotDao {
5.      // les tranches d'impôt
6.      TrancheImpot[] TranchesImpot{get;}
7.   }
8.  }
```

- ligne 3 : la couche [dao] sera placée dans l'espace de noms [Dao]
- ligne 6 : l'interface *IImpotDao* définit la propriété *TranchesImpot* qui fournira les tranches d'impôt à la couche [métier].
- ligne 1 : importe l'espace de noms dans lequel est définie la structure *TrancheImpot* :

```
1.  namespace Entites {
2.   // une tranche d'impôt
3.   public struct TrancheImpot {
4.      public decimal Limite { get; set; }
5.      public decimal CoeffR { get; set; }
6.      public decimal CoeffN { get; set; }
7.   }
8.  }
```

Revenons à l'architecture trois couches de notre application :



Quelle interface la couche [metier] doit-elle présenter à la couche [ui] ? Rappelons les interactions entre ces deux couches :

(a) la couche [ui] demandee à l'utilisateur son nombre d'enfants, son statut marital et son salaire annuel. C'est l'opération [1] ci-dessus.
(b) ceci fait, la couche [ui] va demander à la couche métier de faire le calcul des sièges. Pour cela elle va lui transmettre les données qu'elle a reçues de l'utilisateur. C'est l'opération [2].

Une définition de l'interface de la couche [metier] pourrait être la suivante :

```
1.  namespace Metier {
2.   interface IImpotMetier {
3.      int CalculerImpot(bool marié, int nbEnfants, int salaire);
4.   }
5.  }
```

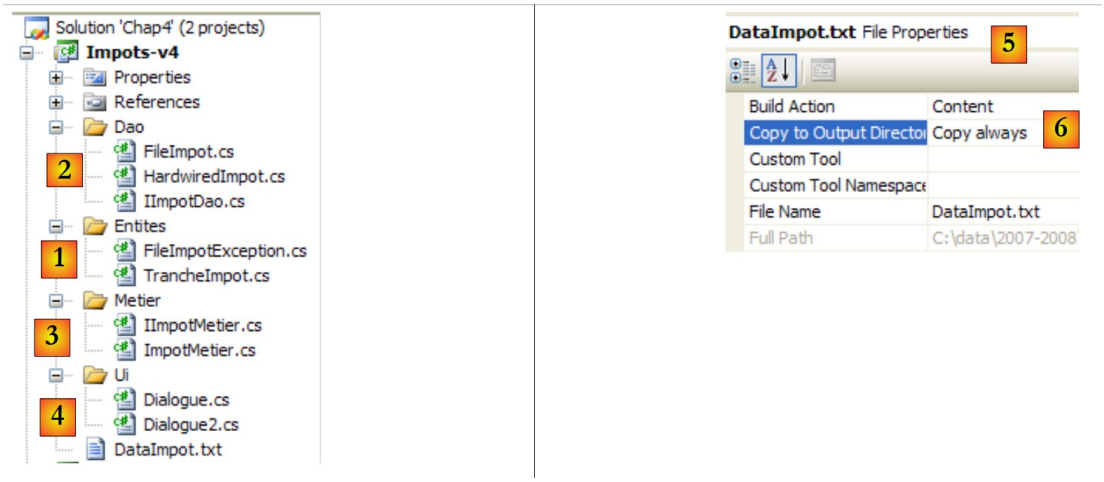
- ligne 1 : on mettra tout ce qui concerne la couche [metier] dans l'espace de noms [Metier].
- ligne 2 : l'interface *IImpotMetier* ne définit qu'une méthode : celles qui permet de calculer l'impôt d'un contribuable à partir de son état marital, son nombre d'enfants et son salaire annuel.

Nous étudions une première implémentation de cette architecture en couches.

6.3 Application exemple - version 4

6.3.1 Le projet Visual Studio

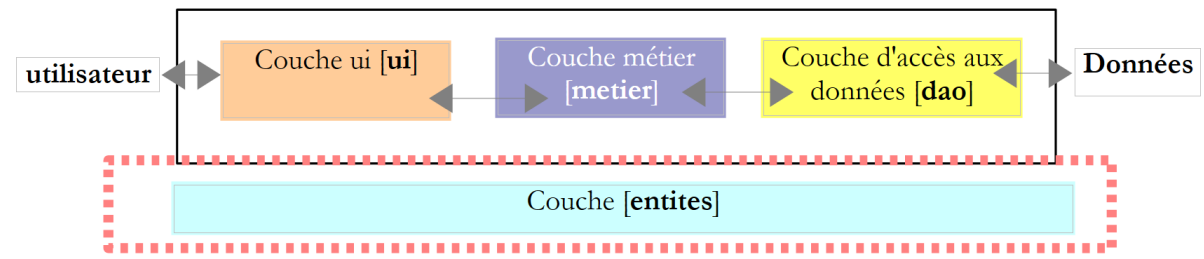
Le projet Visual Studio sera le suivant :



- [1] : le dossier [Entites] contient les objets transversaux aux couches [ui, metier, dao] : la structure *TrancheImpot*, l'exception *FileImpotException*.
- [2] : le dossier [Dao] contient les classes et interfaces de la couche [dao]. Nous utiliserons deux implémentations de l'interface *IImpotDao* : la classe *HardwiredImpot* étudiée au paragraphe 4.10 page 85 et *FileImpot* étudiée au paragraphe 5.8 page 123.
- [3] : le dossier [Metier] contient les classes et interfaces de la couche [metier]
- [4] : le dossier [Ui] contient les classes de la couche [ui]
- [5] : le fichier [DataImpot.txt] contient les tranches d'impôt utilisées par l'implémentation *FileImpot* de la couche [dao]. Il est configuré [6] pour être automatiquement recopié dans le dossier d'exécution du projet.

6.3.2 Les entités de l'application

Revenons sur l'architecture 3 couches de notre application :



Nous appelons *entités* les classes transversales aux couches. C'est le cas en général des classes et structures qui encapsulent des données de la couche [dao]. Ces entités remontent en général jusqu'à la couche [ui].

Les entités de l'application sont les suivantes :

La structure *TrancheImpot*

```
1.  namespace Entites {
2.   // une tranche d'impôt
```

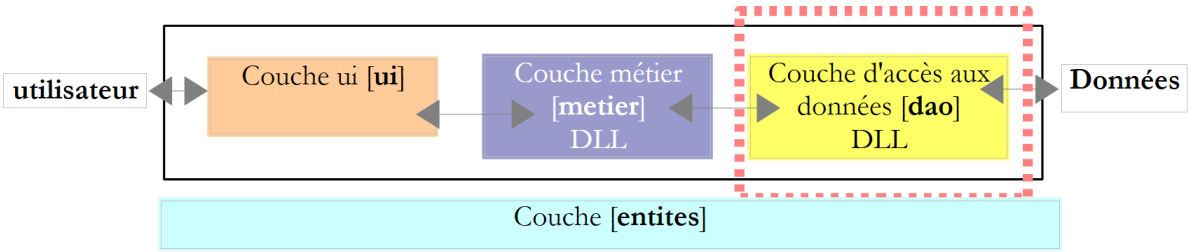
```
3.   public struct TrancheImpot {
4.      public decimal Limite { get; set; }
5.      public decimal CoeffR { get; set; }
6.      public decimal CoeffN { get; set; }
7.   }
8.  }
```

L'exception *FileImpotException*

```
1.  using System;
2.
3.  namespace Entites {
4.   public class FileImpotException : Exception {
5.      // codes d'erreur
6.      [Flags]
7.      public enum CodeErreurs { Acces = 1, Ligne = 2, Champ1 = 4, Champ2 = 8, Champ3 = 16 };
8.
9.      // code d'erreur
10.     public CodeErreurs Code { get; set; }
11.
12.     // constructeurs
13.     public FileImpotException() {
14.     }
15.     public FileImpotException(string message)
16.        : base(message) {
17.     }
18.     public FileImpotException(string message, Exception e)
19.        : base(message, e) {
20.     }
21.  }
22. }
```

Note : la classe *FileImpotException* n'est utile que si la couche [dao] est implémentée par la classe *FileImpot*.

6.3.3 La couche [dao]



Rappelons l'interface de la couche [dao] :

```
1.  using Entites;
2.
3.  namespace Dao {
4.   public interface IImpotDao {
5.      // les tranches d'impôt
6.      TrancheImpot[] TranchesImpot{get;}
7.   }
8.  }
```

Nous implémenterons cette interface de deux façons différentes.

Tout d'abord avec la classe *HardwiredImpot* étudiée au paragraphe 4.10 page 85 :

```
1.  using System;
2.  using Entites;
3.
4.  namespace Dao {
5.   public class HardwiredImpot : IImpotDao {
6.
7.      // tableaux de données nécessaires au calcul de l'impôt
8.      decimal[] limites = { 4962M, 8382M, 14753M, 23888M, 38868M, 47932M, 0M };
```

```
9.      decimal[] coeffR = { 0M, 0.068M, 0.191M, 0.283M, 0.374M, 0.426M, 0.481M };
10.     decimal[] coeffN = { 0M, 291.09M, 1322.92M, 2668.39M, 4846.98M, 6883.66M, 9505.54M };
11.     // tranches d'impôt
12.     public TrancheImpot[] TranchesImpot { get; private set; }
13.
14.     // constructeur
15.     public HardwiredImpot() {
16.        // création du tableau des tranches d'impôt
17.        TranchesImpot = new TrancheImpot[limites.Length];
18.        // remplissage
19.        for (int i = 0; i < TranchesImpot.Length; i++) {
20.           TranchesImpot[i] = new TrancheImpot { Limite = limites[i], CoeffR = coeffR[i], CoeffN =
    coeffN[i] };
21.        }
22.     }
23.  }// classe
24. }// namespace
```

- ligne 5 : la classe *HardwiredImpot* implémente l'interface *IImpotDao*
- ligne 12 : implémentation de la propriété *TranchesImpot* de l'interface *IImpotDao*. Cette propriété est une propriété automatique. Elle implémente la méthode *get* de la propriété *TranchesImpot* de l'interface *IImpotDao*. On a de plus déclaré une méthode *set* privée donc interne à la classe afin que le constructeur des lignes 15-22 puisse initialiser le tableau des tranches d'impôt.

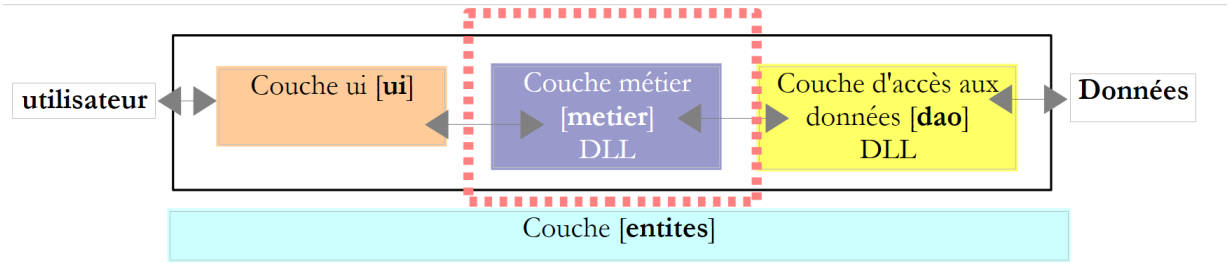
L'interface *IImpotDao* sera également implémentée par la classe *FileImpot* étudiée au paragraphe 5.8 page 123 :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.IO;
4.  using System.Text.RegularExpressions;
5.  using Entites;
6.
7.  namespace Dao {
8.   class FileImpot : IImpotDao {
9.
10.     // fichier des données
11.     public string FileName { get; set; }
12.
13.     // tranches d'impôt
14.     public TrancheImpot[] TranchesImpot { get; private set; }
15.
16.     // constructeur
17.     public FileImpot(string fileName) {
18.        // on mémorise le nom du fichier
19.        FileName = fileName;
20.        // données
21.        List<TrancheImpot> listTranchesImpot = new List<TrancheImpot>();
22.        int numLigne = 1;
23.        // exception
24.        FileImpotException fe = null;
25.        // lecture contenu du fichier fileName, ligne par ligne
26.        Regex pattern = new Regex(@"s*:\s*");
27.        // au départ pas d'erreur
28.        FileImpotException.CodeErreurs code = 0;
29.        try {
30.           using (StreamReader input = new StreamReader(FileName)) {
31.              while (!input.EndOfStream && code == 0) {
32.                 // ligne courante
33.                 string ligne = input.ReadLine().Trim();
34.                 // on ignore les lignes vides
35.                 if (ligne == "")
36.                    continue;
37.                 // ligne décomposée en trois champs séparés par :
38.                 string[] champsLigne = pattern.Split(ligne);
39.                 // a-t-on 3 champs ?
40.                 if (champsLigne.Length != 3) {
41.                    code = FileImpotException.CodeErreurs.Ligne;
42.                 }
43.                 // conversions des 3 champs
44.                 decimal limite = 0, coeffR = 0, coeffN = 0;
45.                 if (code == 0) {
46.                    if (!Decimal.TryParse(champsLigne[0], out limite))
47.                       code = FileImpotException.CodeErreurs.Champ1;
48.                    if (!Decimal.TryParse(champsLigne[1], out coeffR))
49.                       code |= FileImpotException.CodeErreurs.Champ2;
```

```
50.                     if (!Decimal.TryParse(champsLigne[2], out coeffN))
51.                        code |= FileImpotException.CodeErreurs.Champ3;
52.                     ;
53.                  }
54.                  // erreur ?
55.                  if (code != 0) {
56.                     // on note l'erreur
57.                     fe = new FileImpotException(String.Format("Ligne n° {0} incorrecte", numLigne))
    { Code = code };
58.                  } else {
59.                     // on mémorise la nouvelle tranche d'impôt
60.                     listTranchesImpot.Add(new TrancheImpot() { Limite = limite, CoeffR = coeffR,
    CoeffN = coeffN });
61.                     // ligne suivante
62.                     numLigne++;
63.                  }
64.               }
65.            }
66.         } catch (Exception e) {
67.            // on note l'erreur
68.            fe = new FileImpotException(String.Format("Erreur lors de la lecture du fichier {0}",
    FileName), e) { Code = FileImpotException.CodeErreurs.Acces };
69.         }
70.         // erreur à signaler ?
71.         if (fe != null) {
72.            // on lance l'exception
73.            throw fe;
74.         } else {
75.            // on rend la liste listImpot dans le tableau tranchesImpot
76.            TranchesImpot = listTranchesImpot.ToArray();
77.         }
78.      }
79.   }
80. }
```

- ce code a déjà été étudié au paragraphe 5.8 page 123.
- ligne 14 : la méthode *TranchesImpot* de l'interface *IImpotDao*
- ligne 76 : initialisation des tranches d'impôt dans le constructeur de la classe, à partir du fichier dont le contructeur a reçu le nom ligne 17.

6.3.4 La couche [metier]



Rappelons l'interface de cette couche :

```
1.  namespace Metier {
2.   public interface IImpotMetier {
3.      int CalculerImpot(bool marié, int nbEnfants, int salaire);
4.   }
5.  }
```

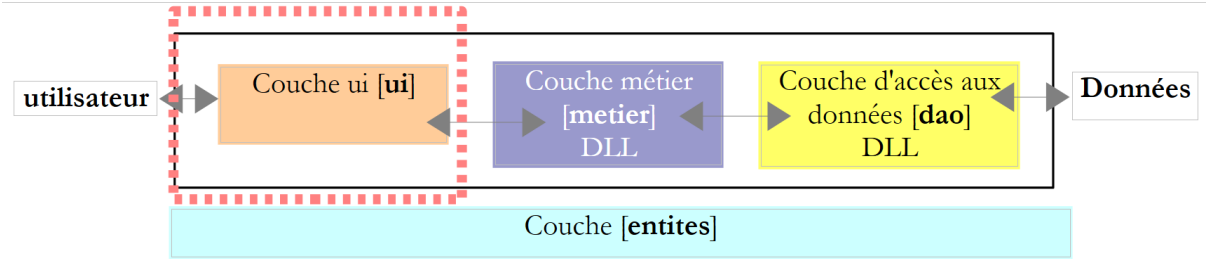
L'implémentation *ImpotMetier* de cette interface est la suivante :

```
1.  using Entites;
2.  using Dao;
3.
4.  namespace Metier {
5.   public class ImpotMetier : IImpotMetier {
```

```
6.
7.      // couche [dao]
8.      private IImpotDao Dao { get; set; }
9.
10.     // les tranches d'impôt
11.     private TrancheImpot[] tranchesImpot;
12.
13.     // constructeur
14.     public ImpotMetier(IImpotDao dao) {
15.        // mémorisation
16.        Dao = dao;
17.        // tranches d'impôt
18.        tranchesImpot = dao.TranchesImpot;
19.     }
20.
21.     // calcul de l'impôt
22.     public int CalculerImpot(bool marié, int nbEnfants, int salaire) {
23.        // calcul du nombre de parts
24.        decimal nbParts;
25.        if (marié)
26.           nbParts = (decimal)nbEnfants / 2 + 2;
27.        else
28.           nbParts = (decimal)nbEnfants / 2 + 1;
29.        if (nbEnfants >= 3)
30.           nbParts += 0.5M;
31.        // calcul revenu imposable & Quotient familial
32.        decimal revenu = 0.72M * salaire;
33.        decimal QF = revenu / nbParts;
34.        // calcul de l'impôt
35.        tranchesImpot[tranchesImpot.Length - 1].Limite = QF + 1;
36.        int i = 0;
37.        while (QF > tranchesImpot[i].Limite)
38.           i++;
39.        // retour résultat
40.        return (int)(revenu * tranchesImpot[i].CoeffR - nbParts * tranchesImpot[i].CoeffN);
41.     }//calculer
42.  }//classe
43.
44. }
```

- ligne 5 : la classe [Metier] implémente l'interface [IImpotMetier].
- lignes 14-19 : la couche [metier] doit collaborer avec la couche [dao]. Elle doit donc avoir une référence sur l'objet implémentant l'interface *IImpotDao*. C'est pourquoi cette reférence est-elle passée en paramètre au constructeur.
- ligne 16 : la référence sur la couche [dao] est mémorisé dans le champ privé de la ligne 8
- ligne 18 : à partir de cette référence, le constructeur demande le tableau des tranches d'impôt et en mémorise une référence dans la propriété privée de la ligne 8.
- lignes 22-41 : implémentation de la méthode *CalculerImpot* de l'interface *IImpotMetier*. Cette implémentation utilise le tableau des tranches d'impôt initialisé par le constructeur.

6.3.5 La couche [ui]



Les classes de dialogue avec l'utilisateur des versions 2 et 3 étaient très proches. Celle de la version 2 était la suivante :

```
1.  using System;
2.
3.  namespace Chap2 {
```

```
4.   public class Program {
5.      static void Main() {
6.  ...
7.
8.         // création d'un objet IImpot
9.         IImpot impot = new HardwiredImpot();
10.
11.        // boucle infinie
12.        while (true) {
13. ...
14.        }//while
15.     }
16.  }
17. }
```

et celle de la version 3 :

```
1.  using System;
2.
3.  namespace Chap3 {
4.   public class Program {
5.      static void Main() {
6.  ...
7.
8.         // création d'un objet IImpot
9.         IImpot impot = null;
10.        try {
11.           // création d'un objet IImpot
12.           impot = new FileImpot("DataImpotInvalide.txt");
13.        } catch (FileImpotException e) {
14.           // affichage erreur
15.           string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
16.           Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
17.           // arrêt programme
18.           Environment.Exit(1);
19.        }
20.        // boucle infinie
21.        while (true) {
22. ...
23.        }//while
24.     }
25.  }
26. }
```

Seule change la façon d'instancier l'objet de type *IImpot* qui permet le calcul de l'impôt. Cet objet correspond ici à notre couche [métier].

Pour une implémentation [dao] avec la classe *HardwiredImpot*, la classe de dialogue est la suivante :

```
1.  using System;
2.  using Metier;
3.  using Dao;
4.  using Entites;
5.
6.  namespace Ui {
7.   public class Dialogue2 {
8.      static void Main() {
9.  ...
10.
11.        // on crée les couches [metier et dao]
12.        IImpotMetier metier = new ImpotMetier(new HardwiredImpot());
13.
14.        // boucle infinie
15.        while (true) {
16. ...
17.           // les paramètres sont corrects - on calcule l'Impot
18.           Console.WriteLine("Impot=" + metier.CalculerImpot(marié == "o", nbEnfants, salaire) + "
    euros");
19.           // contribuable suivant
20.        }//while
21.     }
22.  }
```

```
23. }
```

- ligne 12 : instanciation des couches [dao] et [metier]. On rappelle que la couche [metier] a besoin de la couche [dao].
- ligne 18 : utilisation de la couche [metier] pour calculer l'impôt

Pour une implémentation [dao] avec la classe *FileImpot*, la classe de dialogue est la suivante :

```
1.  using System;
2.  using Metier;
3.  using Dao;
4.  using Entites;
5.
6.  namespace Ui {
7.   public class Dialogue {
8.      static void Main() {
9.  ...
10.         // on crée les couches [metier et dao]
11.         IImpotMetier metier = null;
12.         try {
13.          // création couche [metier]
14.            metier = new ImpotMetier(new FileImpot("DataImpot.txt"));
15.         } catch (FileImpotException e) {
16.            // affichage erreur
17.            string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
18.            Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
19.            // arrêt programme
20.            Environment.Exit(1);
21.         }
22.         // boucle infinie
23.         while (true) {
24. ...
25.            // les paramètres sont corrects - on calcule l'Impot
26.            Console.WriteLine("Impot=" + metier.CalculerImpot(marié == "o", nbEnfants, salaire) + "
    euros");
27.            // contribuable suivant
28.         }//while
29.      }
30.  }
31. }
```

- ligne 11-21 : instanciation des couches [dao] et [metier]. L'instanciation de la couche [dao] pouvant lancer une exception, celle-ci est gérée
- ligne 26 : utilisation de la couche [metier] pour calculer l'impôt, comme dans la version précédente

6.3.6 Conclusion

L'architecture en couches et l'utilisation d'interfaces a amené une certaine souplesse à notre application. Celle-ci apparaît notamment dans la façon dont la couche [ui] instancie les couches [dao] et [métier] :

```
1.         // on crée les couches [metier et dao]
2.  IImpotMetier metier = new ImpotMetier(new HardwiredImpot());
```

dans un cas et :

```
1.  // on crée les couches [metier et dao]
2.           IImpotMetier metier = null;
3.           try {
4.          // création couche [metier]
5.              metier = new ImpotMetier(new FileImpot("DataImpot.txt"));
6.           } catch (FileImpotException e) {
7.              // affichage erreur
8.              string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
9.              Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
10.             // arrêt programme
11.             Environment.Exit(1);
12.          }
```

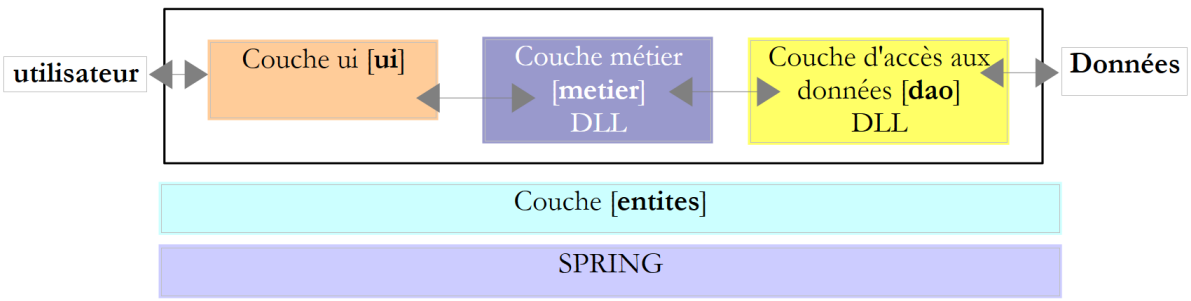
dans l'autre. Si on excepte la gestion de l'exception dans le cas 2, l'instanciation des couches [dao] et [metier] est similaire dans les deux applications. Une fois les couches [dao] et [metier] instanciées, le code de la couche [ui] est identique dans les deux cas. Ceci est du au fait que la couche [métier] est manipulée via son interface *IImpotMetier* et non via la classe d'implémentation de celle-ci. Changer la couche [metier] ou la couche [dao] de l'application sans changer leurs interfaces reviendra toujours à changer les seules lignes précédentes dans la couche [ui].

Un autre exemple de souplesse amenée par cette architecture est celui de l'implémentation de la couche [métier] :

```
1.  using Entites;
2.  using Dao;
3.
4.  namespace Metier {
5.   public class ImpotMetier : IImpotMetier {
6.
7.      // couche [dao]
8.      private IImpotDao Dao { get; set; }
9.
10.     // les tranches d'impôt
11.     private TrancheImpot[] tranchesImpot;
12.
13.     // constructeur
14.     public ImpotMetier(IImpotDao dao) {
15.        // mémorisation
16.        Dao = dao;
17.        // tranches d'impôt
18.        tranchesImpot = dao.TranchesImpot;
19.     }
20.
21.     // calcul de l'impôt
22.     public int CalculerImpot(bool marié, int nbEnfants, int salaire) {
23. ...
24.     }//calculer
25.  }//classe
26.
27. }
```

Ligne 14, on voit que la couche [métier] est construite à partir d'une référence sur l'interface de la couche [dao]. Changer l'implémentation de cette dernière a donc un impact zéro sur la couche [métier]. C'est pour cela, que notre unique implémentation de la couche [métier] a pu travailler sans modifications avec deux implémentations différentes de la couche [dao].

6.4 Application exemple - version 5



Cette nouvelle version reprend la précédente en y apportant les modifications suivantes :

- les couches [métier] et [dao] sont chacune encapsulées dans une DLL et testée avec le framework de tests unitaires NUnit.
- l'intégration des couches est assurée par le framework Spring

Dans les grands projets, plusieurs développeurs travaillent sur le même projet. Les architectures en couches facilitent ce mode de travail : parce que les couches communiquent entre-elles avec des interfaces bien définies, un développeurs travaillant sur une couche n'a pas à se préoccuper du travail des autres développeurs sur les autres couches. Il suffit juste que tout le monde respecte les interfaces.

Ci-dessus, le développeur de la couche [métier] aura besoin au moment des tests de sa couche d'une implémentation de la couche [dao]. Tant que celle-ci n'est pas terminée, il peut utiliser une implémentation factice de la couche [dao] tant qu'elle respecte l'interface *IImpotDao*. C'est là également un avantage de l'architecture en couches : un retard dans la couche [dao] n'empêche pas les tests de la couche [métier]. L'implémentation factice de la couche [dao] a également l'avantage d'être bien souvent plus facile à mettre en oeuvre que la véritable couche [dao] qui peut nécessiter de lancer un SGBD, d'avoir des connexions réseau, ...

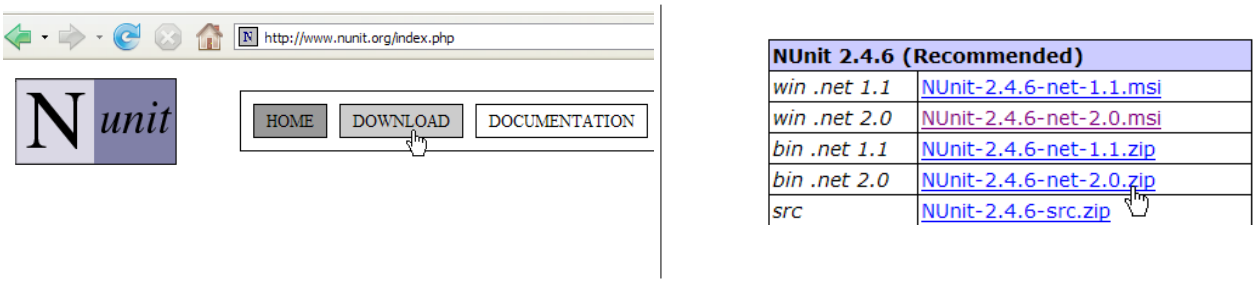
Lorsque la couche [dao] est terminée et testée, elle sera fournie aux développeurs de la couche [métier] sous la forme d'une DLL plutôt que de code source. Au final, l'application est souvent délivrée sous la forme d'un exécutable .exe (celui de la couche [ui]) et de bibliothèques de classes .dll (les autres couches).

6.4.1 NUnit

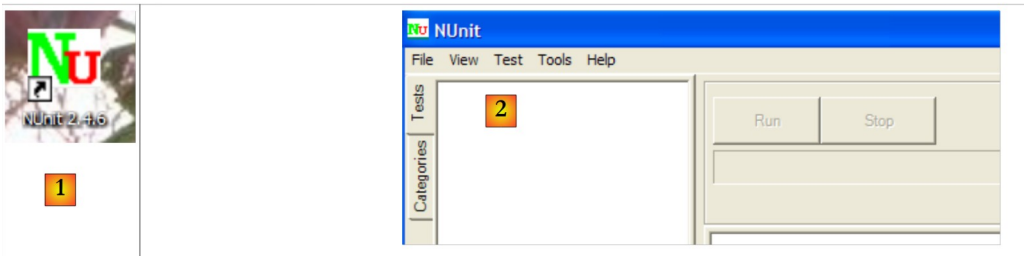
Les tests faits jusqu'à maintenant pour nos diverses applications reposaient sur une vérification visuelle. On vérifiait qu'on obtenait à l'écran ce qui était attendu. C'est une méthode inutilisable lorsqu'il y a de nombreux tests à faire. L'être humain est en effet sujet à la fatigue et sa capacité à vérifier des tests s'émousse au fil de la journée. Les tests doivent alors être automatisés et viser à ne nécessiter aucune intervention humaine.

Une application évolue au fil du temps. A chaque évolution, on doit vérifier que l'application ne "régresse" pas, c.a.d. qu'elle continue à passer les tests de bon fonctionnement qui avaient été faits lors de son écriture initiale. On appelle ces tests, des tests de "non régression". Une application un peu importante peut nécessiter des centaines de tests. On teste en effet chaque méthode de chaque classe de l'application. On appelle cela des **tests unitaires**. Ceux-ci peuvent mobiliser beaucoup de développeurs s'ils n'ont pas été automatisés.

Des outils ont été développés pour automatiser les tests. L'un d'eux s'appelle **NUnit**. Il est disponible sur le site [http://www.nunit.org] :



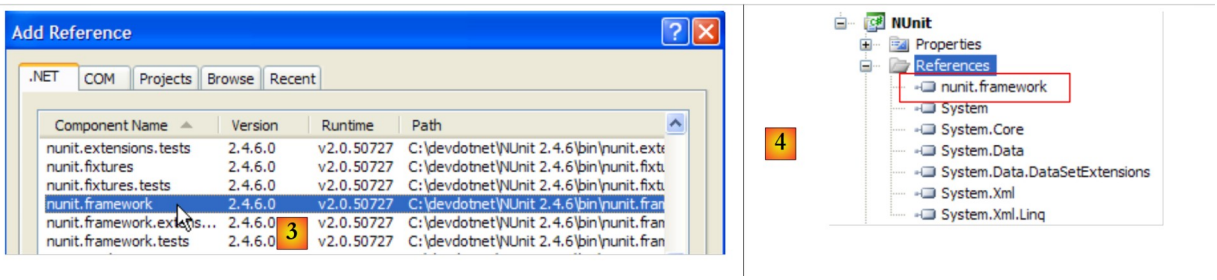
C'est la version 2.4.6 ci-dessus qui a été utilisée pour ce document (mars 2008). L'installation place une icône [1] sur le bureau :



Un double-clic sur l'icône [1] lance l'interface graphique de NUnit [2]. Celle-ci n'aide en rien à l'automatisation des tests puisque de nouveau nous sommes ramenés à une vérification visuelle : le testeur vérifie les résultats des tests affichés dans l'interface graphique. Néanmoins les tests peuvent être également exécutés par des outils batch et leurs résultats enregistrés dans des fichiers XML. C'est cette méthode qui est utilisée par les équipes de développement : les tests sont lancés la nuit et les développeurs ont le résultat le lendemain matin.

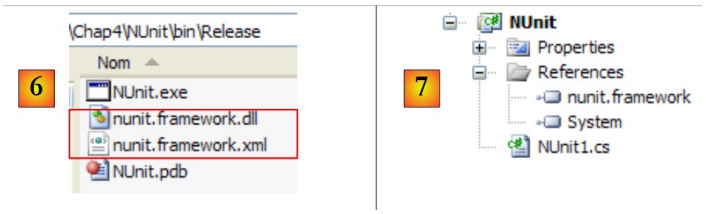
Examinons avec un exemple le principe des tests NUnit. Tout d'abord, créons un nouveau projet C# de type *Console Application* :

En [1], on voit les *références* du projet. Ces références sont des DLL contenant des classes et interfaces utilisées par le projet. Celles présentées en [1] sont incluses par défaut dans chaque nouveau projet C#. Pour pouvoir utiliser les classes et interfaces du framework NUnit, il nous faut ajouter [2] une nouvelle référence au projet.



Dans l'onglet .NET ci-dessus, nous choisissons le composant [nunit.framework]. Les composants [nunit.*] ci-dessus ne sont pas des composants présents par défaut dans l'environnement .NET. Ils y ont été amenés par l'installation précédente du framework NUnit. Une fois l'ajout de la référence validée, celle-ci apparaît [4] dans la liste des références du projet.

Avant génération de l'application, le dossier [bin/Release] du projet est vide. Après génération (F6), on peut constater que le dossier [bin/Release] n'est plus vide :



En [6], on voit la présence de la DLL [nunit.framework.dll]. C'est l'ajout de la référence [nunit.framework] qui a provoqué la copie de cette DLL dans le dossier d'exécution. Celui-ci est en effet l'un des dossiers qui seront explorés par le CLR (Common Language Runtime) .NET pour trouver les classes et interfaces référencées par le projet.

Construisons une première classe de test NUnit. Pour cela, nous supprimons la classe [Program.cs] générée par défaut puis nous ajoutons une nouvelle classe [Nunit1.cs] au projet. Nous supprimons également les références inutiles [7].

La classe de test *NUnit1* sera la suivante :

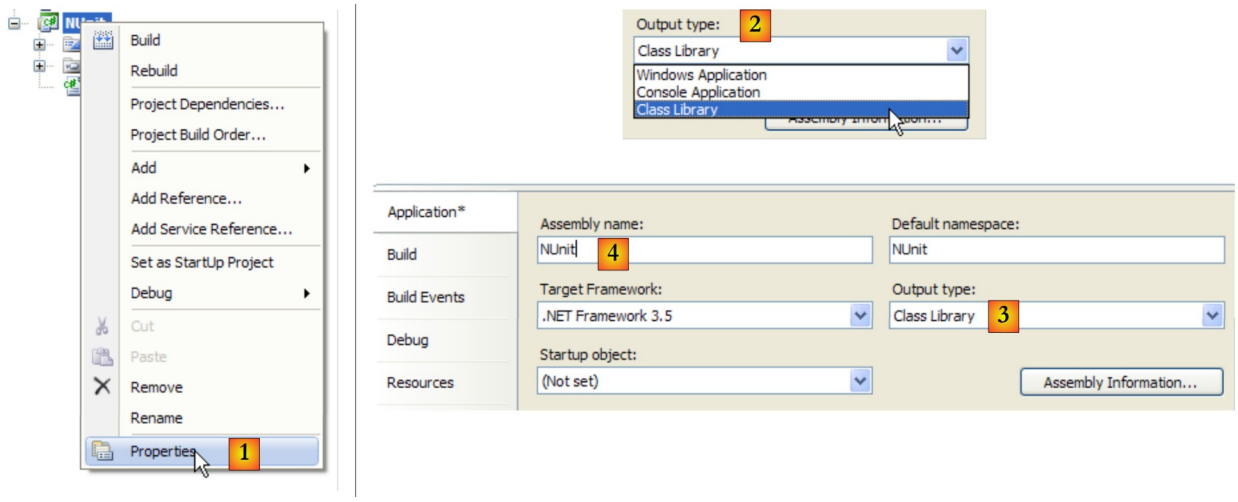
```
1.  using System;
2.  using NUnit.Framework;
3.
4.  namespace NUnit {
5.   [TestFixture]
6.   public class NUnit1 {
7.      public NUnit1() {
8.         Console.WriteLine("constructeur");
9.      }
10.     [SetUp]
11.     public void avant() {
```

```
12.         Console.WriteLine("Setup");
13.     }
14.     [TearDown]
15.     public void après() {
16.         Console.WriteLine("TearDown");
17.     }
18.     [Test]
19.     public void t1() {
20.         Console.WriteLine("test1");
21.         Assert.AreEqual(1, 1);
22.     }
23.     [Test]
24.     public void t2() {
25.         Console.WriteLine("test2");
26.         Assert.AreEqual(1, 2, "1 n'est pas égal à 2");
27.     }
28.  }
29. }
```

- ligne 6 : la classe *NUnit1* doit être publique. Le mot clé **public** n'est pas généré par défaut par Visual Studio. Il faut le rajouter.
- ligne 5 : l'attribut [*TestFixture*] est un attribut *NUnit*. Il indique que la classe est une classe de test.
- lignes 7-9 : le constructeur. Il n'est utilisé ici que pour écrire un message à l'écran. On veut voir quand il est exécuté.
- ligne 10 : l'attribut [*SetUp*] définit une méthode exécutée **avant** chaque test unitaire.
- ligne 14 : l'attribut [*TearDown*] définit une méthode exécuté **après** chaque test unitaire.
- ligne 18 : l'attribut [*Test*] définit une méthode de test. Pour chaque méthode annotée avec l'attribut [*Test*], la méthode annotée [*SetUp*] sera exécutée avant le test et la méthode annotée [*TearDown*] sera exécutée après le test.
- ligne 21 : l'une des méthodes [*Assert.**] définies par le framework NUnit. On trouve les méthodes [Assert] suivantes :
  - **[***Assert.AreEqual(expression1, expression2)***]** : vérifie que les valeurs des deux expressions sont égales. De nombreux types d'expression sont acceptés (int, string, float, double, decimal, ...). Si les deux expressions ne sont pas égales, alors une exception est lancée.
  - **[***Assert.AreEqual(réel1, réel2, delta)***]** : vérifie que deux réels sont égaux à **delta** près, c.a.d abs(réel1-réel2)<=delta. On pourra écrire par exemple [*Assert.AreEqual(réel1, réel2, 1E-6)*] pour vérifier que deux valeurs sont égales à 10^{-6} près.
  - **[***Assert.AreEqual(expression1, expression2, message)***]** et [*Assert.AreEqual(réel1, réel2, delta, message)*] sont des variantes permettant de préciser le message d'erreur à associer à l'exception lancée lorsque la méthode [*Assert.AreEqual*] échoue.
  - **[***Assert.IsNotNull(object)***]** et [*Assert.IsNotNull(object, message)*] : vérifie que o**bject** n'est pas égal à **null**.
  - **[***Assert.IsNull(object)***]** et [*Assert.IsNull(object, message)*] : vérifie que o**bject** est égal à **null**.
  - **[***Assert.IsTrue(expression)***]** et [*Assert.IsTrue(expression, message)*] : vérifie que **expression** est égale à **true**.
  - **[***Assert.IsFalse(expression)***]** et [*Assert.IsFalse(expression, message)*] : vérifie que **expression** est égale à **false**.
  - **[***Assert.AreSame(object1, object2)***]** et [*Assert.AreSame(object1, object2, message)*] : vérifie que les références **object1** et **object2** désignent le même objet.
  - **[***Assert.AreNotSame(object1, object2)***]** et [*Assert.AreNotSame(object1, object2, message)*] : vérifie que les références **object1** et **object2** ne désignent pas le même objet.

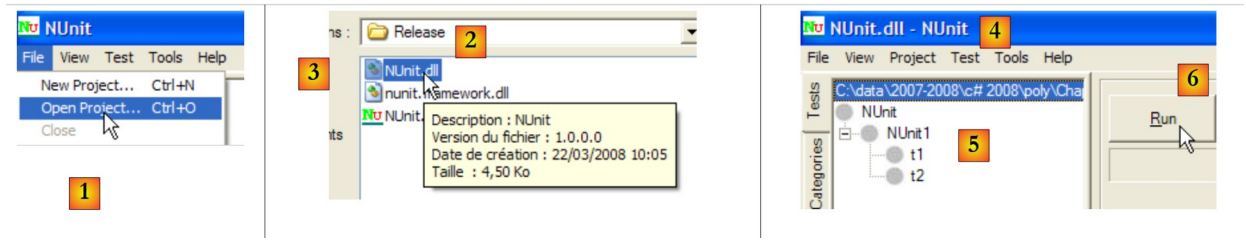
- ligne 21 : l'assertion doit réussir
- ligne 26 : l'assertion doit échouer

Configurons le projet afin que sa génération produise une DLL plutôt qu'un exécutable .exe :

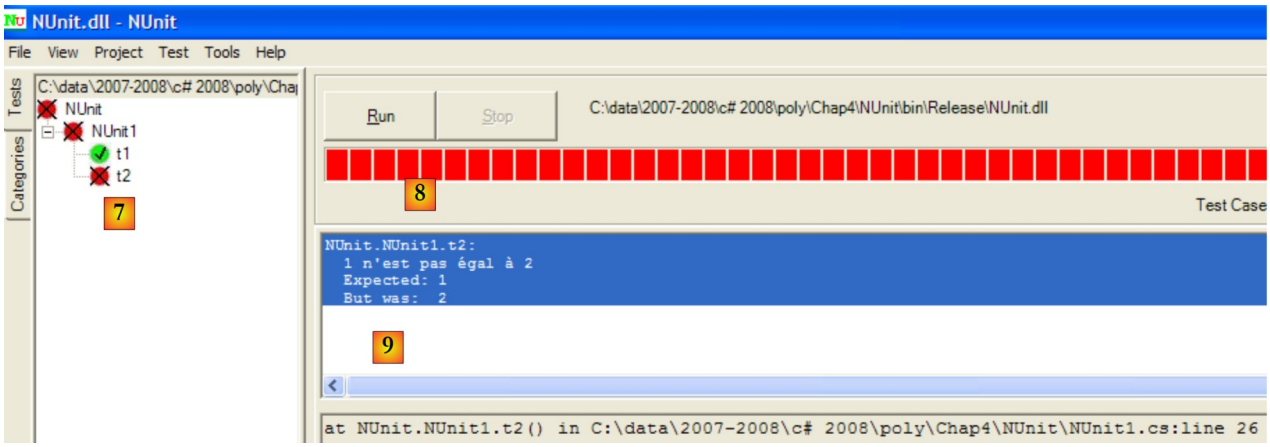


- en [1] : propriétés du projet
- en [2, 3] : comme type de projet, on choisit [Class Library] (Bibliothèque de classes)
- en [4] : la génération du projet produira une DLL (assembly) appelée [Nunit.dll]

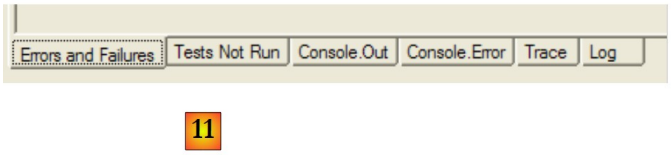
Utilisons maintenant, NUnit pour exécuter la classe de test :

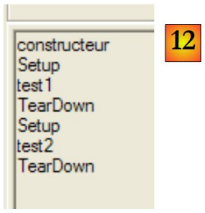


- en [1] : ouverture d'un projet NUnit
- en [2, 3] : on charge la DLL bin/Release/Nunit.dll produite par la génération du projet C#
- en [4] : la DLL a été chargée
- en [5] : l'arbre des tests
- en [6] : on les exécute



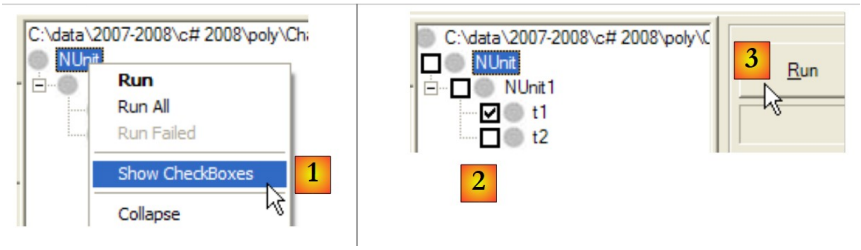
- en [7] : les résultats : t1 a réussi, t2 a échoué
- en [8] : une barre rouge indique l'échec global de la classe de tests
- en [9] : le message d'erreur lié au test raté





- en [11] : les différents onglets de la fenêtre des résultats
- en [12] : l'onglet [Console.Out]. On y voit que :
  - le constructeur n'a été exécuté qu'une fois
  - la méthode [SetUp] a été exécutée avant chacun des deux tests
  - la méthode [TearDown] a été exécutée après chacun des deux tests

Il est possible de préciser les méthodes à tester :



- en [1] : on demande l'affichage d'une case à cocher à côté de chaque test
- en [2] : on coche les tests à exécuter
- en [3] : on les exécute

Pour corriger les erreurs, il suffit de corriger le projet C# et de le régénérer. NUnit détecte que la DLL qu'il teste a été changée et charge la nouvelle automatiquement. Il suffit alors de relancer les tests.

Considérons la nouvelle classe de test suivante :

```
1.  using System;
2.  using NUnit.Framework;
3.
4.  namespace NUnit {
5.   [TestFixture]
6.   public class NUnit2 : AssertionHelper {
7.      public NUnit2() {
8.         Console.WriteLine("constructeur");
9.      }
10.     [SetUp]
11.     public void avant() {
12.        Console.WriteLine("Setup");
13.     }
14.     [TearDown]
15.     public void après() {
16.        Console.WriteLine("TearDown");
17.     }
18.     [Test]
19.     public void t1() {
20.        Console.WriteLine("test1");
21.        Expect(1, EqualTo(1));
22.     }
23.     [Test]
24.     public void t2() {
25.        Console.WriteLine("test2");
26.        Expect(1, EqualTo(2), "1 n'est pas égal à 2");
27.     }
28.  }
29. }
```

A partir de la version 2.4 de NUnit, une nouvelle syntaxe est devenue disponible, celles des lignes 21 et 26. Pour cela, la classe de test doit dériver de la classe **AssertionHelper** (ligne 6).

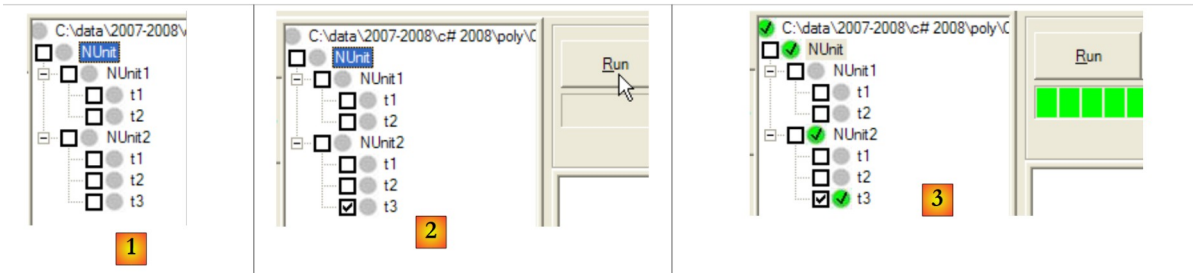
La correspondance (non exhaustive) entre ancienne et nouvelle syntaxe est la suivante :

`Assert.AreEqual(expression1, expression2, message)`	`Expect(expression1,EqualTo(expression2),message)`
`Assert.AreEqual(réel1, réel2, delta, message)`	`Expect(expression1,EqualTo(expression2).Within(delta),message)`
`Assert.AreSame(objet1, objet2, message)`	`Expect(objet1,SameAs(objet2),message)`
`Assert.AreNotSame(objet1, objet2, message)`	`Expect(objet1,Not.SameAs(objet2),message)`
`Assert.IsNull(objet,message)`	`Expect(objet,Null,message)`
`Assert.IsNotNull(objet,message)`	`Expect(objet,Not.Null,message)`
`Assert.IsTrue(expression,message)`	`Expect(expression,True,message)`
`Assert.IsFalse(expression,message)`	`Expect(expression,False,message)`

Ajoutons le test suivant à la classe NUnit2 :

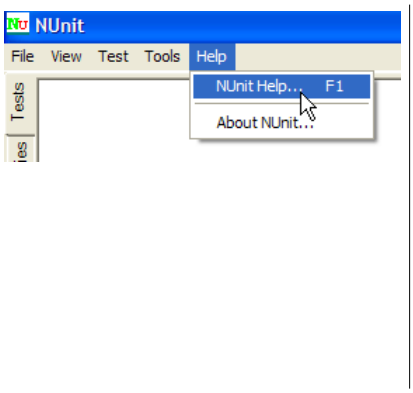
```
1.         [Test]
2.         public void t3() {
3.            bool vrai = true, faux = false;
4.            Expect(vrai, True);
5.            Expect(faux, False);
6.            Object obj1 = new Object(), obj2 = null, obj3=obj1;
7.            Expect(obj1, Not.Null);
8.            Expect(obj2, Null);
9.            Expect(obj3, SameAs(obj1));
10.           double d1 = 4.1, d2 = 6.4, d3 = d1;
11.           Expect(d1, EqualTo(d3).Within(1e-6));
12.           Expect(d1, Not.EqualTo(d2));
13. }
```

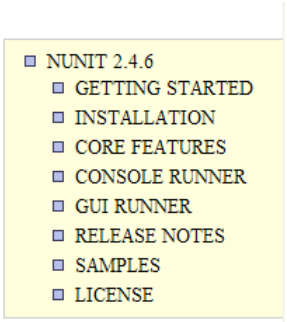
Si on génère (F6) la nouvelle DLL du projet C#, le projet NUnit devient le suivant :



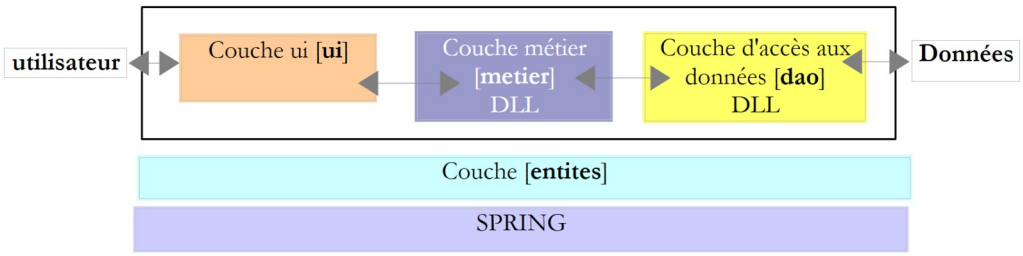
- en [1] : la nouvelle classe de test [NUnit2] a été automatiquement détectée
- en [2] : on exécute le test t3 de NUnit2
- en [3] : le test t3 a été réussi

Pour approfondir NUnit, on lira l'aide de NUnit :

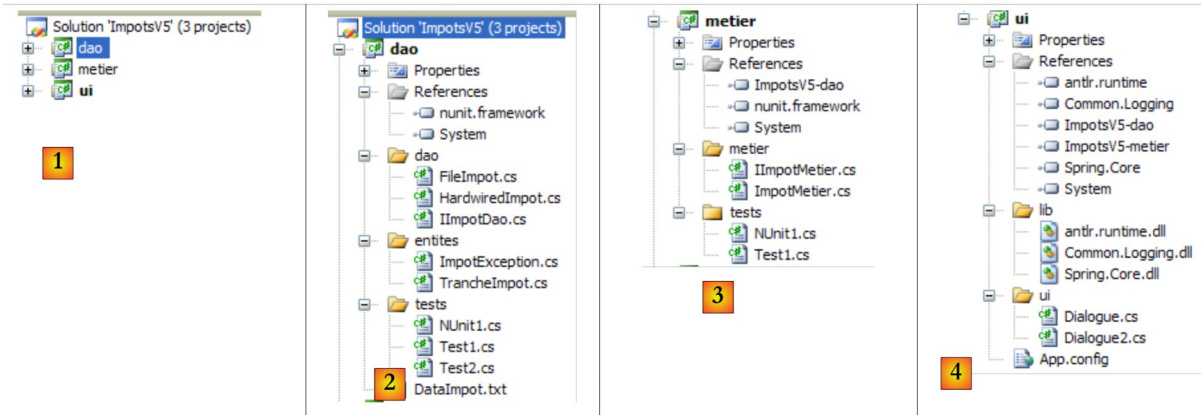




6.4.2 La solution Visual Studio

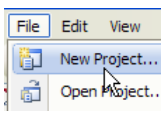


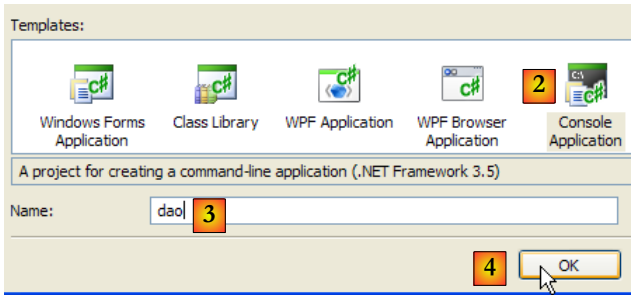
Nous allons construire progressivement la solution Visual Studio suivante :

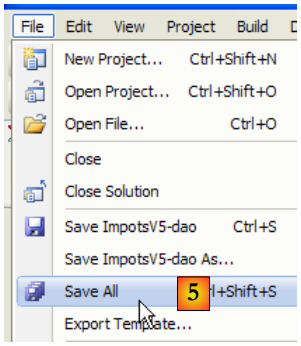


- en [1] : la solution *ImpotsV5* est formée de trois projets, un pour chacune des trois couches de l'application
- en [2] : le projet [dao] de la couche [dao]
- en [3] : le projet [metier] de la couche [metier]
- en [4] : le projet [ui] de la couche [ui]

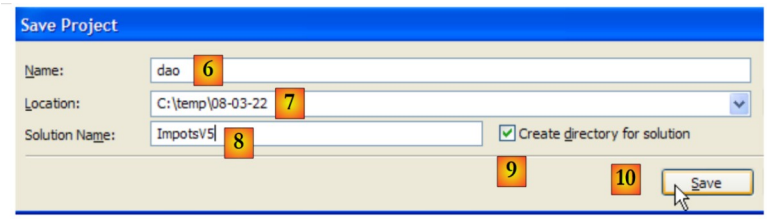
La solution *ImpotsV5* peut être construite de la façon suivante :

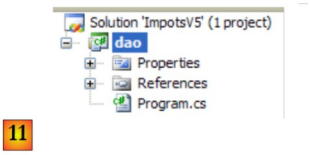




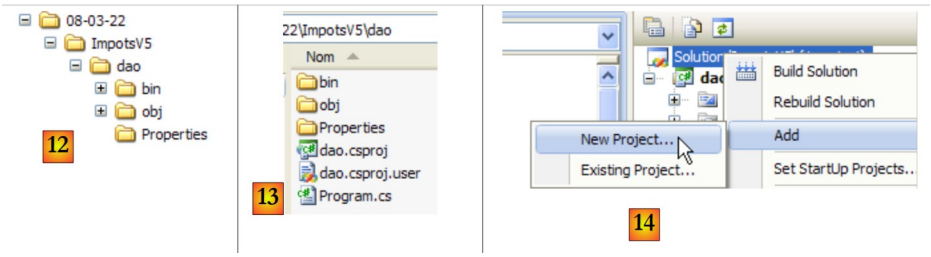


- en [1] : créer un nouveau projet
- en [2] : choisir une application console
- en [3] : appeler le projet [dao]
- en [4] : créer le projet
- en [5] : une fois le projet créé, le sauvegarder

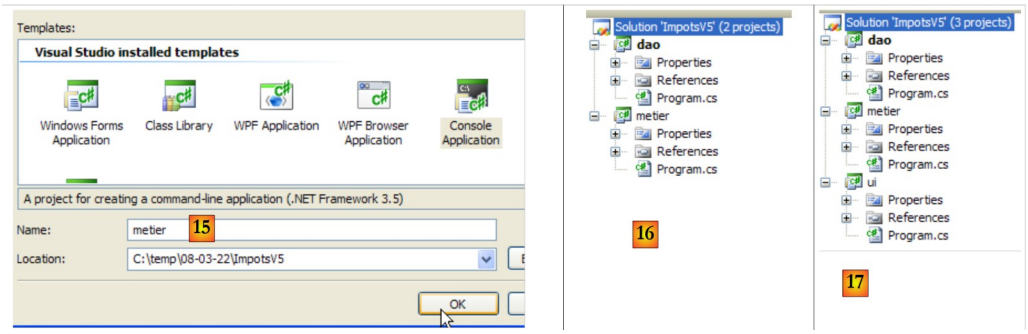




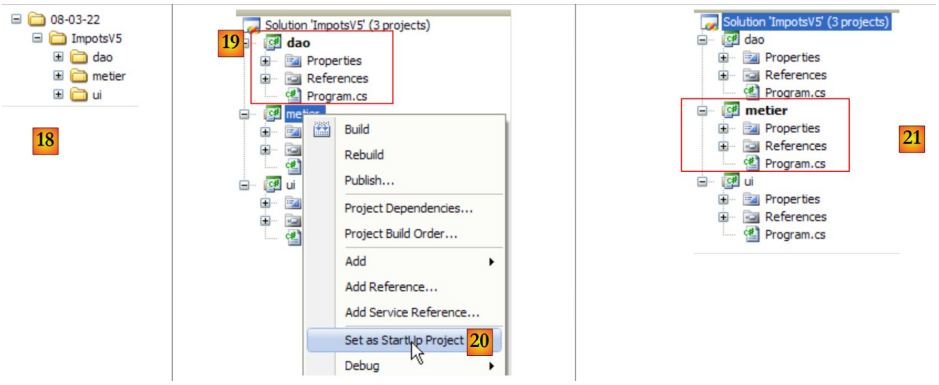
- en [6] : garder le nom [dao] pour le projet
- en [7] : préciser un dossier pour enregistrer le projet et sa solution
- en [8] : donner un nom à la solution
- en [9] : indiquer que la solution doit avoir son propre dossier
- en [10] : enregistrer le projet et sa solution
- en [11] : le projet [dao] dans sa solution *ImpotsV5*



- en [12] : le dossier de la solution *ImpotsV5*. Il contient le dossier [dao] du dossier [dao].
- en [13] : le contenu du dossier [dao]
- en [14] : on ajoute un nouveau projet à la solution *ImpotsV5*

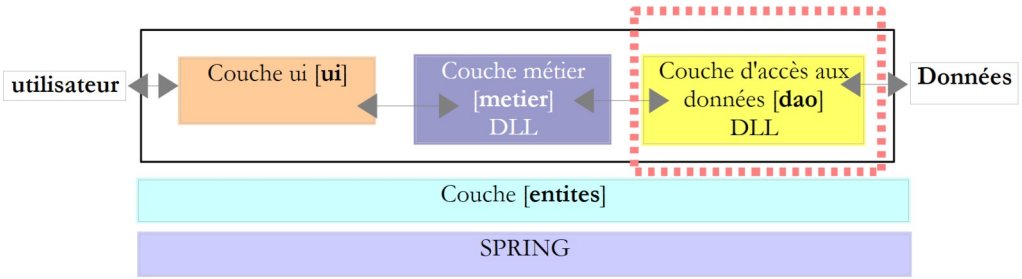


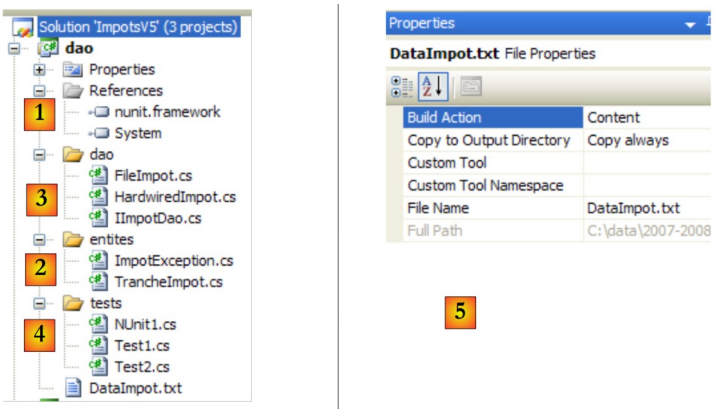
- en [15] : le nouveau projet s'appelle [metier]
- en [16] : la solution avec ses deux projets
- en [17] : la solution, une fois qu'on lui a ajouté le 3ième projet [ui]



- en [18] : le dossier de la solution et les dossiers des trois projets
- lorsqu'on exécute une solution par (Ctrl+F5), c'est le **projet actif** qui est exécuté. Il en est de même lorsqu'on génère (F6) la solution. Le nom du **projet actif** est en gras [19] dans la solution.
- en [20] : pour changer le projet actif de la solution
- en [21] : le projet [metier] est désormais le projet actif de la solution

6.4.3 La couche [dao]





Les références du projet (cf [1] dans le projet)

On ajoute la référence [nunit.framework] nécessaire aux tests [NUnit]

Les entités (cf [2] dans le projet)

La classe [TrancheImpot] est celle des versions précédentes. La classe [FileImpotException] de la version précédente est renommée en [ImpotException] pour la rendre plus générique et ne pas la lier à une couche [dao] particulière :

```
1.  using System;
2.
3.  namespace Entites {
4.   public class ImpotException : Exception {
5.
6.      // code d'erreur
7.      public int Code { get; set; }
8.
9.      // constructeurs
10.     public ImpotException() {
11.     }
12.     public ImpotException(string message)
13.        : base(message) {
14.     }
15.     public ImpotException(string message, Exception e)
16.        : base(message, e) {
17.     }
18.  }
19. }
```

La couche [dao] (cf [3] dans le projet)

L'interface [IImpotDao] est celle de la version précédente. Il en est de même pour la classe [HardwiredImpot]. La classe [FileImpot] évolue pour tenir compte du changement de l'exception [FileImpotException] en [ImpotException] :

```
1.  ...
2.
3.  namespace Dao {
4.   public class FileImpot : IImpotDao {
5.
6.      // codes d'erreur
7.      [Flags]
8.      public enum CodeErreurs { Acces = 1, Ligne = 2, Champ1 = 4, Champ2 = 8, Champ3 = 16 };
9.
10. ...
11.
12.     // constructeur
13.     public FileImpot(string fileName) {
14.        // on mémorise le nom du fichier
15.        FileName = fileName;
16. ...
17.        // au départ pas d'erreur
18.        CodeErreurs code = 0;
```

```
19.         try {
20.            using (StreamReader input = new StreamReader(FileName)) {
21.               while (!input.EndOfStream && code == 0) {
22. ...
23.                  // erreur ?
24.                  if (code != 0) {
25.                     // on note l'erreur
26.                     fe = new ImpotException(String.Format("Ligne n° {0} incorrecte", numLigne))
    { Code = (int)code };
27.                  } else {
28. ...
29.                  }
30.               }
31.            }
32.         } catch (Exception e) {
33.            // on note l'erreur
34.            fe = new ImpotException(String.Format("Erreur lors de la lecture du fichier {0}",
    FileName), e) { Code = (int)CodeErreurs.Acces };
35.         }
36.         // erreur à signaler ?
37. ...
38.      }
39.   }
40. }
```

- ligne 8 : les codes d'erreurs auparavant dans la classe [FileImpotException] ont migré dans la classe [FileImpot]. Ce sont en effet des codes d'erreur spécifiques à cette implémentation de l'interface [IImpotDao].
- lignes 26 et 34 : pour encapsuler une erreur, c'est la classe [ImpotException] qui est utilisée et non plus la classe [FileImpotException].

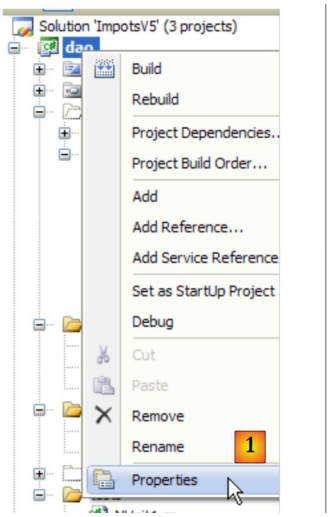
Le test [Test1] (cf [4] dans le projet)

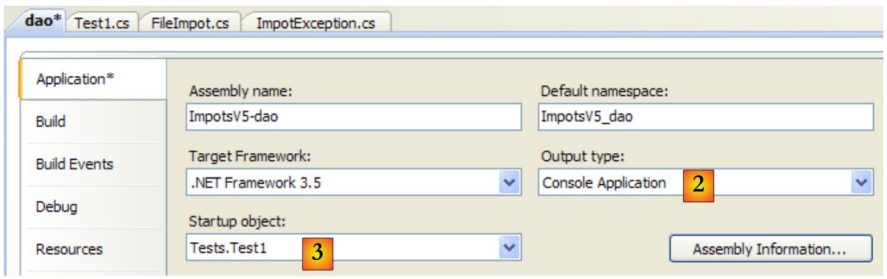
La classe [Test1] se contente d'afficher les tranches d'impôt à l'écran :

```
1.  using System;
2.  using Dao;
3.  using Entites;
4.
5.  namespace Tests {
6.   class Test1 {
7.      static void Main() {
8.
9.         // on crée la couche [dao]
10.        IImpotDao dao = null;
11.        try {
12.           // création couche [dao]
13.           dao = new FileImpot("DataImpot.txt");
14.        } catch (ImpotException e) {
15.           // affichage erreur
16.           string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
17.           Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
18.           // arrêt programme
19.           Environment.Exit(1);
20.        }
21.        // on affiche les tranches d'impôt
22.        TrancheImpot[] tranchesImpot = dao.TranchesImpot;
23.        foreach (TrancheImpot t in tranchesImpot) {
24.           Console.WriteLine("{0}:{1}:{2}", t.Limite, t.CoeffR, t.CoeffN);
25.        }
26.     }
27.  }
28. }
```

- ligne 13 : la couche [dao] est implémentée par la classe [FileImpot]
- ligne 14 : on gère l'exception de type [ImpotException] qui peut survenir.

Le fichier [DataImpot.txt] nécessaire aux tests est recopié automatiquement dans le dossier d'exécution du projet (cf [5] dans le projet). Le projet [dao] va avoir plusieurs classes contenant une méthode [Main]. Il faut indiquer alors explicitement la classe à exécuter lorsque l'utilisateur demande l'exécution du project par Ctrl-F5 :





- en [1] : accéder aux propriétés du projet
- en [2] : préciser que c'est une application console
- en [3] : préciser la classe à exécuter

L'exécution de la classe [Test1] précédente donne les résultats suivants :

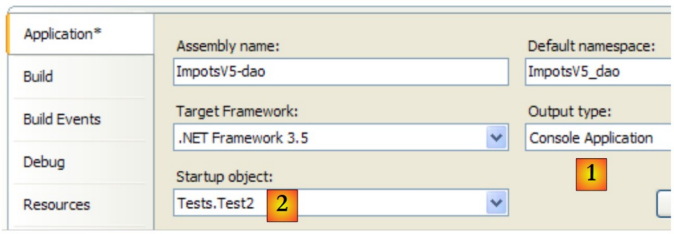
```
4962:0:0
8382:0,068:291,09
14753:0,191:1322,92
23888:0,283:2668,39
38868:0,374:4846,98
47932:0,426:6883,66
0:0,481:9505,54
```

Le test [Test2] (cf [4] dans le projet)

La classe [Test2] fait la même chose que la classe [Test1] en implémentant la couche [dao] avec la classe [HardwiredImpot]. La ligne 13 de [Test1] est remplacée par la suivante :

```
            dao = new HardwiredImpot();
```

Le projet est modifié pour exécuter désormais la classe [Test2] :



Les résultats écran sont les mêmes que précédemment.

Le test NUnit [NUnit1] (cf [4] dans le projet)

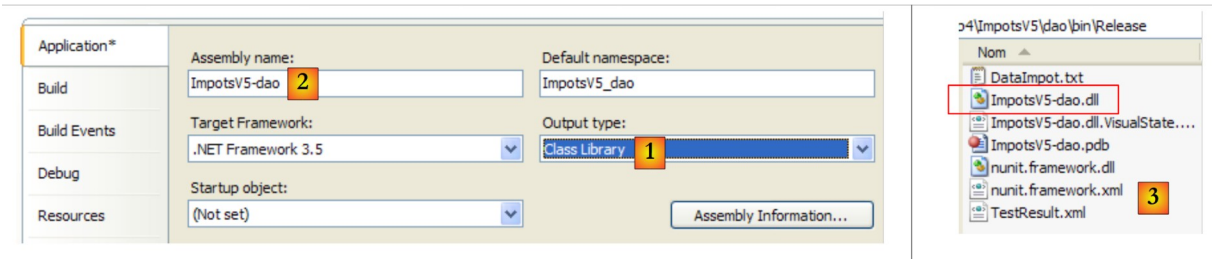
Le test unitaire [NUnit1] est le suivant :

```
1.  using System;
2.  using Dao;
3.  using Entites;
4.  using NUnit.Framework;
5.
```

```
6.  namespace Tests {
7.   [TestFixture]
8.   public class NUnit1 : AssertionHelper{
9.      // couche [dao] à tester
10.     private IImpotDao dao;
11.
12.     // constructeur
13.     public NUnit1() {
14.        // initialisation couche [dao]
15.        dao = new FileImpot("DataImpot.txt");
16.     }
17.
18.     // test
19.     [Test]
20.     public void ShowTranchesImpot(){
21.        // on affiche les tranches d'impôt
22.        TrancheImpot[] tranchesImpot = dao.TranchesImpot;
23.        foreach (TrancheImpot t in tranchesImpot) {
24.           Console.WriteLine("{0}:{1}:{2}", t.Limite, t.CoeffR, t.CoeffN);
25.        }
26.        // qqs tests
27.        Expect(tranchesImpot.Length,EqualTo(7));
28.        Expect(tranchesImpot[2].Limite,EqualTo(14753));
29.        Expect(tranchesImpot[2].CoeffR, EqualTo(0.191));
30.        Expect(tranchesImpot[2].CoeffN, EqualTo(1322.92));
31.     }
32.  }
33. }
```

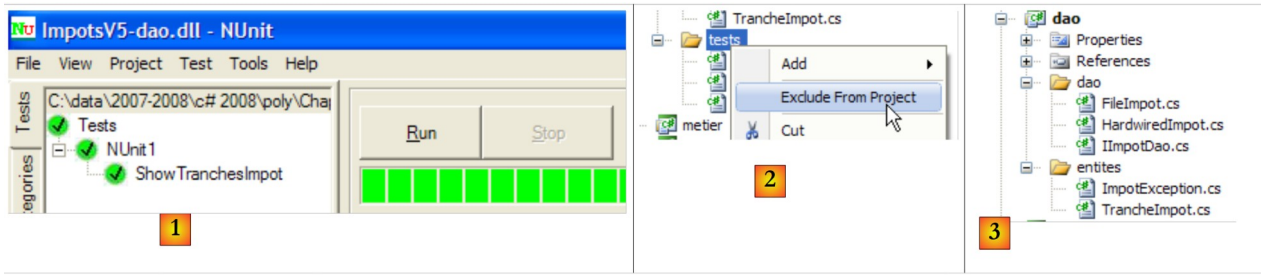
- la classe de test dérive de la classe [AssertionHelper], ce qui permet l'utilisation de la méthode statique *Expect* (lignes 27-30).
- ligne 10 : une référence sur la couche [dao]
- lignes 13-16 : le constructeur instancie la couche [dao] avec la classe [FileImpot]
- lignes 19-20 : la méthode de test
- ligne 22 : on récupère le tableau des tranches d'impôt auprès de la couche [dao]
- lignes 23-25 : on les affiche comme précédemment. Cet affichage n'aurait pas lieu d'être dans un test unitaire réel. Ici, cet affichage a un souci pédagogique.
- lignes 27 : on vérifie qu'il y a bien 7 tranches d'impôt
- lignes 28-30 : on vérifie les valeurs de la tranche d'impôt n° 2

Pour exécuter ce test unitaire, le projet doit être de type [Class Library] :



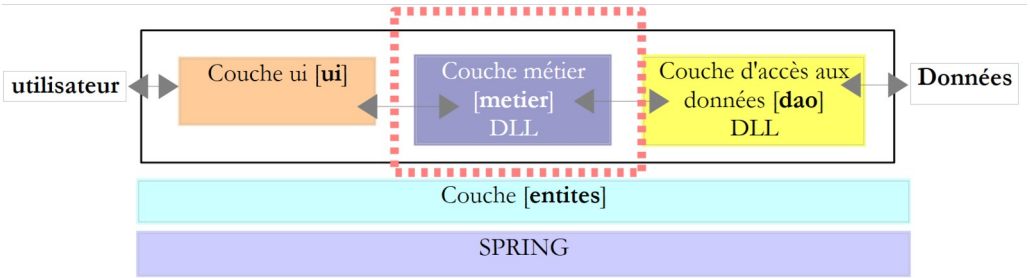
- en [1] : la nature du projet a été changée
- en [2] : la DLL générée s'appellera [ImpotsV5-dao.dll]
- en [3] : après génération (F6) du projet, le dossier [dao/bin/Release] contient la DLL [ImpotsV5-dao.dll]

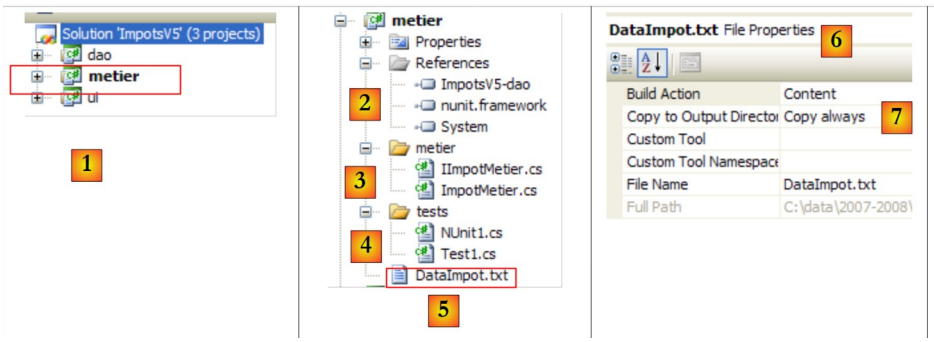
La DLL [ImpotsV5-dao.dll] est ensuite chargée dans le framework *NUnit* et exécutée :



- en [1] : les tests ont été réussis. Nous considérons désormais la couche [dao] opérationnelle. Sa DLL contient toutes les classes du projet dont les classes de test. Celles-ci sont inutiles. Nous reconstruisons la DLL afin d'en exclure les classes de tests.
- en [2] : le dossier [tests] est exclu du projet
- en [3] : le nouveau projet. Celui-ci est régénéré par F6 afin de générer une nouvelle DLL.

6.4.4 La couche [metier]

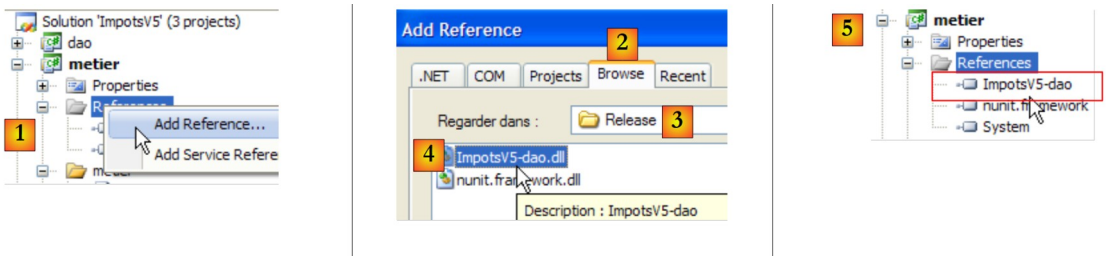




- en [1], le projet [metier] est devenu le projet actif de la solution
- en [2] : les références du projet
- en [3] : la couche [metier]
- en [4] : les classes de test
- en [5] : le fichier [DataImpot.txt] des tranches d'impôt configuré [6] pour être recopié automatiquement dans le dossier d'exécution du projet [7]

Les références du projet (cf [2] dans le projet)

Comme pour le projet [dao], on ajoute la référence [nunit.framework] nécessaire aux tests [NUnit]. La couche [metier] a besoin de la couche [dao]. Il lui faut donc une référence sur la DLL de cette couche. On procède ainsi :



- en [1] : on ajoute une nouvelle référence aux références du projet [metier]
- en [2] : on sélectionne l'onglet [Browse]
- en [3] : on sélectionne le dossier [dao/bin/Release]
- en [4] : on sélectionne la DLL [ImpotsV5-dao.dll] générée dans le projet [dao]
- en [5] : la nouvelle référence

La couche [metier] (cf [3] dans le projet)

L'interface [IImpotMetier] est celle de la version précédente. Il en est de même pour la classe [ImpotMetier].

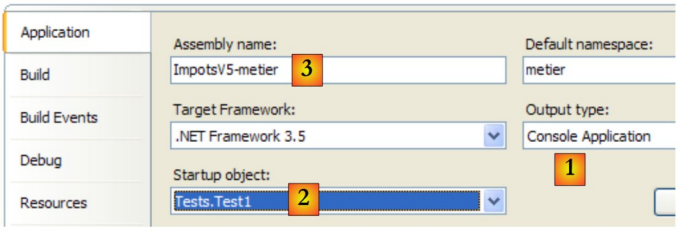
Le test [Test1] (cf [4] dans le projet)

La classe [Test1] se contente de faire quelques calculs de salaire :

```
1.  using System;
2.  using Dao;
3.  using Entites;
4.  using Metier;
5.
6.  namespace Tests {
7.   class Test1 {
8.      static void Main() {
9.
10.        // on crée la couche [metier]
11.        IImpotMetier metier = null;
12.        try {
13.           // création couche [metier]
14.           metier = new ImpotMetier(new FileImpot("DataImpot.txt"));
15.        } catch (ImpotException e) {
16.           // affichage erreur
17.           string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
18.           Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
19.           // arrêt programme
20.           Environment.Exit(1);
21.        }
22.        // on calcule qqs impots
23.        Console.WriteLine(String.Format("Impot(true,2,60000)={0} euros", metier.CalculerImpot(true,
    2, 60000)));
24.        Console.WriteLine(String.Format("Impot(false,3,60000)={0} euros",
    metier.CalculerImpot(false, 3, 60000)));
25.        Console.WriteLine(String.Format("Impot(false,3,60000)={0} euros",
    metier.CalculerImpot(false, 3, 6000)));
26.        Console.WriteLine(String.Format("Impot(false,3,60000)={0} euros",
    metier.CalculerImpot(false, 3, 600000)));
27.     }
28.  }
29. }
```

- ligne 14 : création des couches [metier] et [dao]. La couche [dao] est implémentée avec la classe [FileImpot]
- lignes 12-21 : gestion d'une éventuelle exception de type [ImpotException]
- lignes 23-26 : appels répétés de l'unique méthode *CalculerImpot* de l'interface [IImpotMetier].

Le projet [metier] est configuré comme suit :



- [1] : le projet est de type application console
- [2] : la classe exécutée est la classe [Test1]
- [3] : la génération du projet produira l'exécutable [ImpotsV5-metier.exe]

L'exécution du projet donne les résultats suivants :

```
1.  Impot(true,2,60000)=4282 euros
2.  Impot(false,3,60000)=4282 euros
3.  Impot(false,3,60000)=0 euros
4.  Impot(false,3,60000)=179275 euros
```

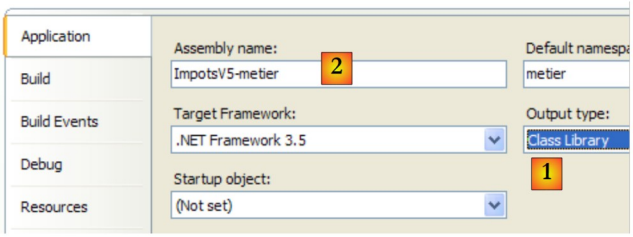
Le test [NUnit1] (cf [4] dans le projet)

La classe de tests unitaires [NUnit1] reprend les quatre calculs précédents et en vérifie le résultat :

```
1.  using Dao;
2.  using Metier;
3.  using NUnit.Framework;
4.
5.  namespace Tests {
6.   [TestFixture]
7.   public class NUnit1:AssertionHelper {
8.      // couche [metier] à tester
9.      private IImpotMetier metier;
10.
11.     // constructeur
12.     public NUnit1() {
13.        // initialisation couche [metier]
14.        metier = new ImpotMetier(new FileImpot("DataImpot.txt"));
15.     }
16.
17.     // test
18.     [Test]
19.     public void CalculsImpot(){
20.        // on affiche les tranches d'impôt
21.        Expect(metier.CalculerImpot(true, 2, 60000), EqualTo(4282));
22.        Expect(metier.CalculerImpot(false, 3, 60000), EqualTo(4282));
23.        Expect(metier.CalculerImpot(false, 3, 6000), EqualTo(0));
24.        Expect(metier.CalculerImpot(false, 3, 600000), EqualTo(179275));
25.     }
26.  }
27. }
```

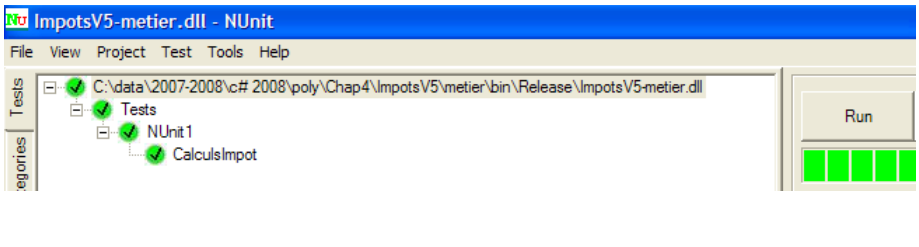
- ligne 14 : création des couches [metier] et [dao]. La couche [dao] est implémentée avec la classe [FileImpot]
- lignes 21-24 : appels répétés de l'unique méthode *CalculerImpot* de l'interface [IImpotMetier] avec vérification des résultats.

Le projet [metier] est maintenant configuré comme suit :

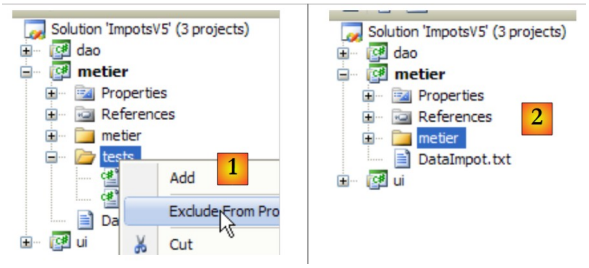


- [1] : le projet est de type "bibliothèque de classes"
- [2] : la génération du projet produira la DLL [ImpotsV5-metier.dll]

Le projet est généré (F6). Puis la DLL [ImpotsV5-metier.dll] générée est chargée dans *NUnit* et testée :

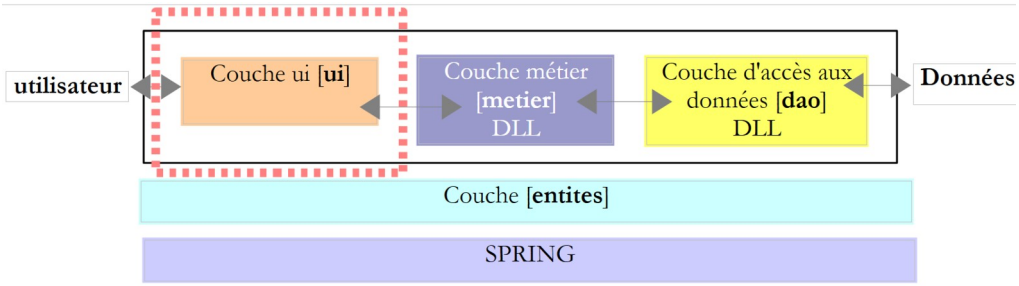


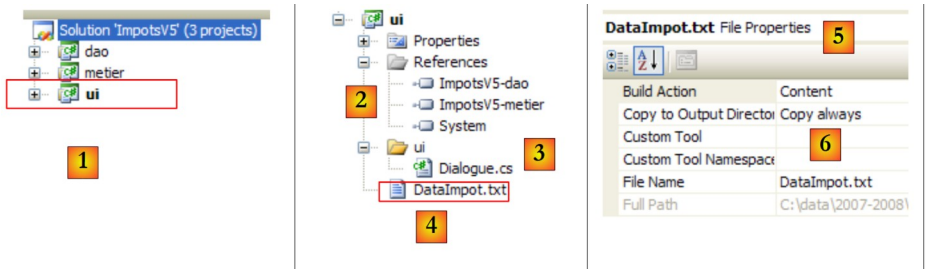
Ci-dessus, les tests ont été réussis. Nous considérons désormais la couche [metier] opérationnelle. Sa DLL contient toutes les classes du projet dont les classes de test. Celles-ci sont inutiles. Nous reconstruisons la DLL afin d'en exclure les classes de tests.



- en [1] : le dossier [tests] est exclu du projet
- en [2] : le nouveau projet. Celui-ci est régénéré par F6 afin de générer une nouvelle DLL.

6.4.5 La couche [ui]





- en [1], le projet [ui] est devenu le projet actif de la solution
- en [2] : les références du projet
- en [3] : la couche [ui]
- en [4] : le fichier [DataImpot.txt] des tranches d'impôt, configuré [5] pour être recopié automatiquement dans le dossier d'exécution du projet [6]

Les références du projet (cf [2] dans le projet)

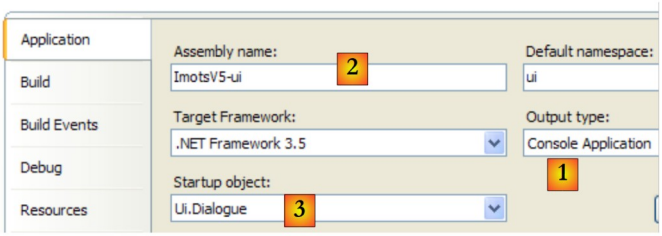
La couche [ui] a besoin des couches [metier] et [dao] pour mener à bien ses calculs d'impôt. Il lui faut donc une référence sur les DLL de ces deux couches. On procède comme il a été montré pour la couche [metier]

La classe principale [Dialogue.cs] (cf [3] dans le projet)

La classe [Dialogue.cs] est celle de la version précédente.

Tests

Le projet [ui] est configuré comme suit :



- [1] : le projet est de type "application console"
- [2] : la génération du projet produira l'exécutable [ImpotsV5-ui.exe]
- [3] : la classe qui sera exécutée

Un exemple d'exécution (Ctrl+F5) est le suivant :

```
Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :o 2 60000
Impot=4282 euros
```

6.4.6 La couche [Spring]

Revenons au code dans [Dialogue.cs] qui crée les couches [dao] et [metier] :

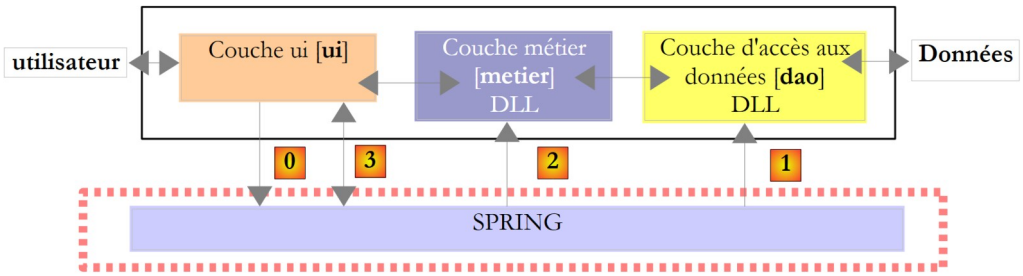
```
1.  // on crée les couches [metier et dao]
2.          IImpotMetier metier = null;
3.          try {
4.         // création couche [metier]
5.             metier = new ImpotMetier(new FileImpot("DataImpot.txt"));
6.          } catch (ImpotException e) {
```

```
7.              // affichage erreur
8.  ...
9.              // arrêt programme
10.             Environment.Exit(1);
11.         }
```

La ligne 5 crée les couches [dao] et [metier] en nommant explicitement les classes d'implémentation des deux couches : FileImpot pour la couche [dao], ImpotMetier pour la couche [metier]. Si l'implémentation d'une des couches est faite avec une nouvelle classe, la ligne 5 sera changée. Par exemple :

```
            metier = new ImpotMetier(new HardwiredImpot());
```

En-dehors de ce changement, rien ne changera dans l'application du fait que chaque couche communique avec la suivante selon une interface. Tant que cette dernière ne change pas, la communication entre couches ne change pas non plus. Le framework **Spring** nous permet d'aller un peu plus loin dans l'indépendance des couches en nous permettant d'externaliser dans un fichier de configuration le nom des classes implémentant les différentes couches. Changer l'implémentation d'une couche revient alors à changer un fichier de configuration. Il n'y a aucun impact sur le code de l'application.

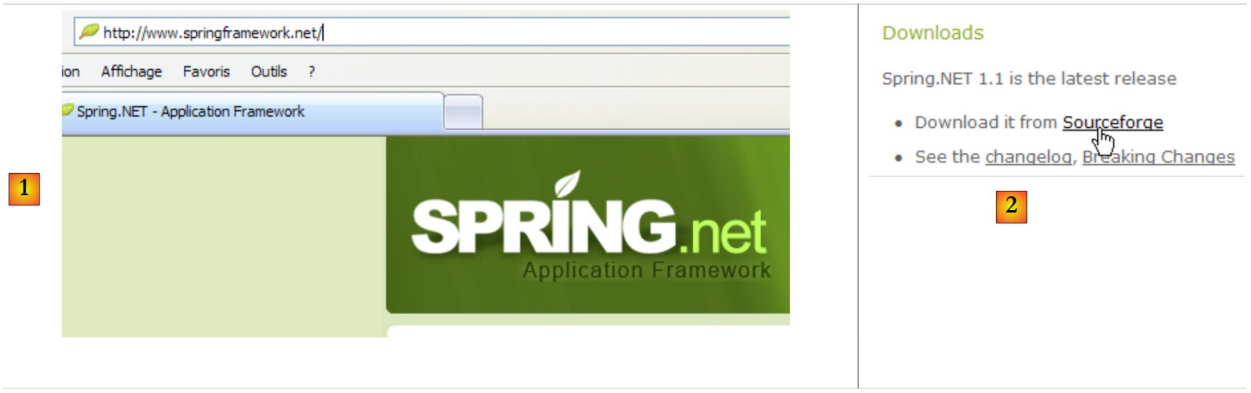


Ci-dessus, la couche [ui] va demander [0] à Spring d'instancier les couches [dao] [1] et [metier] [2] d'après les informations contenues dans un fichier de configuration. La couche [ui] demandera ensuite à Spring [3], une référence sur la couche [metier] :

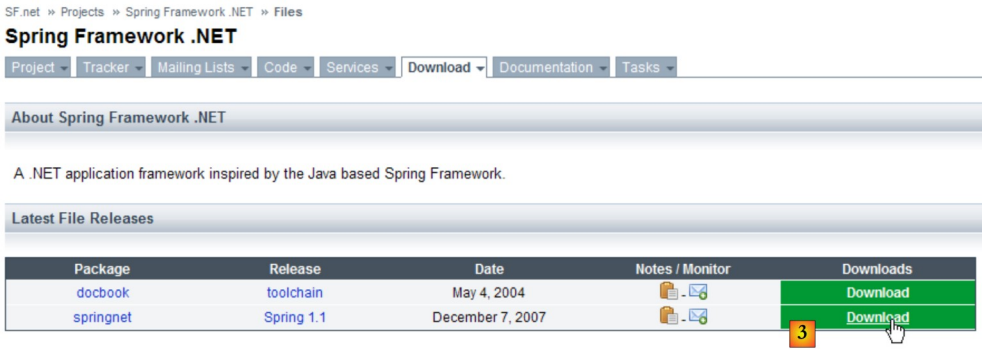
```
1.          // on crée les couches [metier et dao]
2.          IImpotMetier metier = null;
3.          try {
4.              // contexte Spring
5.              IApplicationContext ctx = ContextRegistry.GetContext();
6.              // on demande une référence sur la couche [metier]
7.              metier = (IImpotMetier)ctx.GetObject("metier");
8.          } catch (Exception e1) {
9.  ...
10. }
```

- ligne 5 : instanciation des couches [dao] et [metier] par Spring
- ligne 7 : on récupère une référence sur la couche [metier]. On notera que la couche [ui] a eu cette référence sans donner le nom de la classe implémentant la couche [metier].

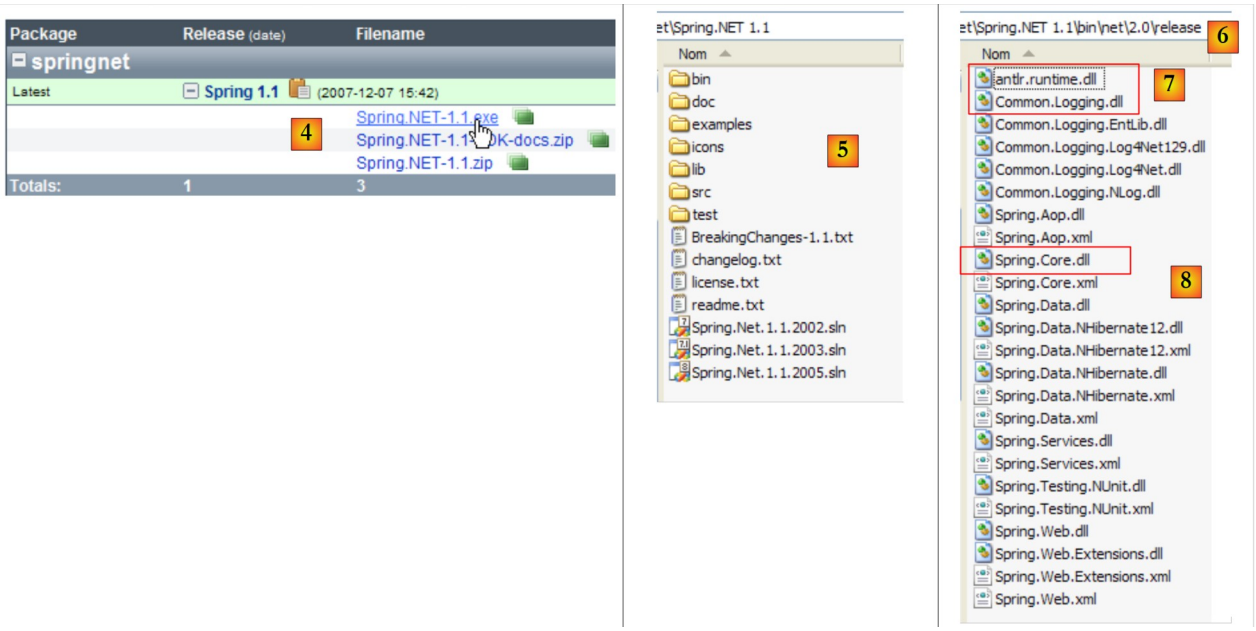
Le framework Spring existe en deux versions : Java et .NET. La version .NET est disponible à l'url (mars 2008) [http://www.springframework.net/] :



- en [1] : le site de [Spring.net]
- en [2] : la page des téléchargements



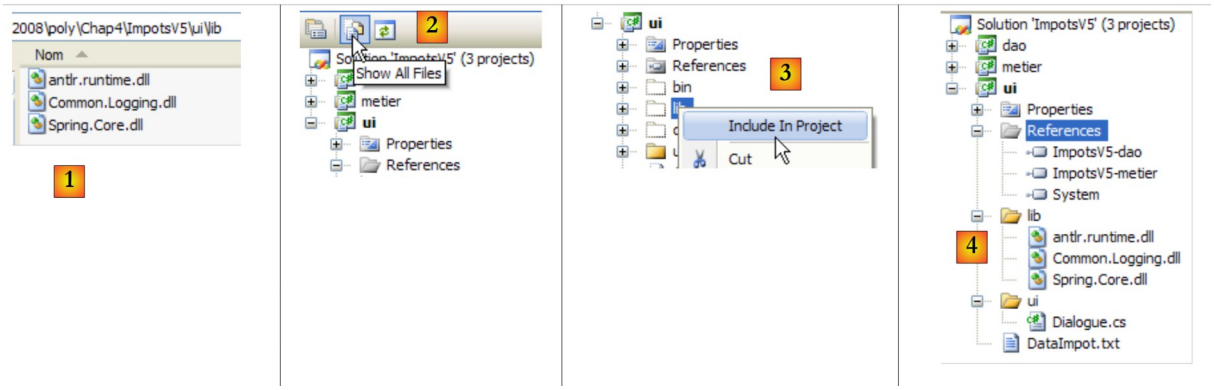
- en [3] : télécharger Spring 1.1 (mars 2008)



- en [4] : télécharger la version .exe puis l'installer
- en [5] : le dossier généré par l'installation

- en [6] : le dossier [bin/net/2.0/release] contient les DLL de Spring pour les projets Visual Studio .NET 2.0 ou supérieur. Spring est un framework riche. L'aspect de Spring que nous allons utiliser ici pour gérer l'intégration des couches dans une application s'appelle **IoC** : *Inversion of Contro*l ou encore **DI** : *Dependence Injection*. Spring apporte des bibliothèques pour l'accès aux bases de données avec *NHibernate*, la génération et l'exploitation de services web, d'applications web, ...
- les DLL nécessaires pour gérer l'intégration des couches dans une application sont les DLL [7] et [8].

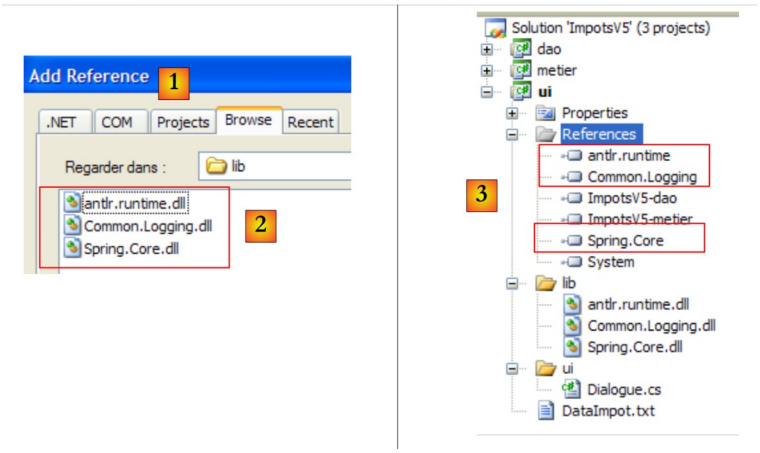
Nous stockons ces trois DLL dans un dossier [lib] de notre projet :



- [1] : les trois DLL sont placées dans le dossier [lib] avec l'explorateur windows
- [2] : dans le projet [ui], on fait afficher tous les fichiers
- [3] : le dossier [ui/lib] est désormais visible. On l'inclut dans le projet
- [4] : le dossier [ui/lib] fait partie du projet

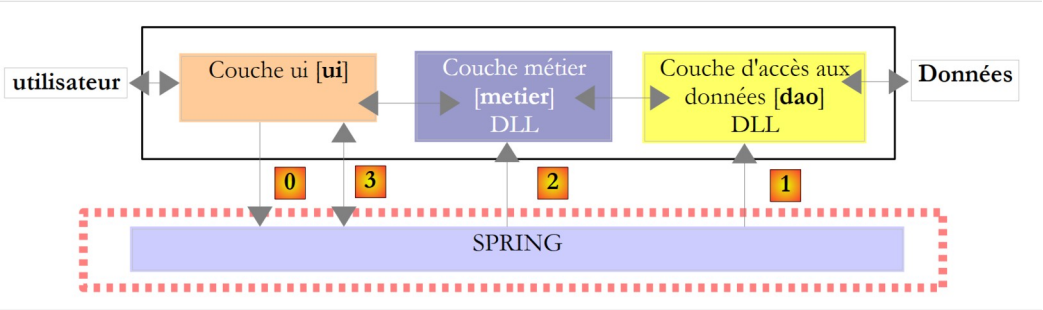
L'opération de création du dossier [lib] n'est nullement indispensable. Les références pouvaient être créées directement sur les trois DLL du dossier [bin/net/2.0/release] de [Spring.net]. La création du dossier [lib] permet cependant de développer l'application sur un poste ne disposant pas de [Spring.net] la rendant ainsi moins dépendante de l'environnement de développement disponible.

Nous ajoutons au projet [ui] des références sur les trois nouvelles DLL :



- [1] : on crée des références sur les trois DLL du dossier [lib] [2]
- [3] : les trois DLL font partie des références du projet

Revenons à une vue d'ensemble de l'architecture de l'application :



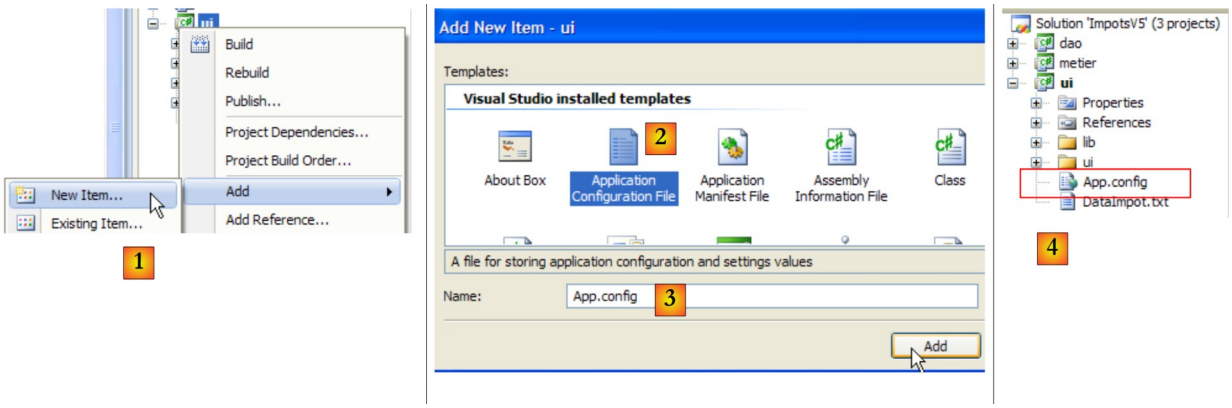
Ci-dessus, la couche [ui] va demander [0] à Spring d'instancier les couches [dao] [1] et [metier] [2] d'après les informations contenues dans un fichier de configuration. La couche [ui] demandera ensuite à Spring [3], une référence sur la couche [metier]. Cela se traduira dans la couche [ui] par le code suivant :

```
1.          // on crée les couches [metier et dao]
2.          IImpotMetier metier = null;
3.          try {
4.             // contexte Spring
5.             IApplicationContext ctx = ContextRegistry.GetContext();
6.             // on demande une référence sur la couche [metier]
7.             metier = (IImpotMetier)ctx.GetObject("metier");
8.          } catch (Exception e1) {
9.  ...
10. }
```

- ligne 5 : instanciation des couches [dao] et [metier] par Spring
- ligne 7 : on récupère une référence sur la couche [metier].

La ligne [5] ci-dessus exploite le fichier de configuration [App.config] du projet Visual Studio. Dans un projet C#, ce fichier sert à configurer l'application. [App.config] n'est donc pas une notion Spring mais une notion Visual Studio que Spring exploite. Spring sait exploiter d'autres fichiers de configuration que [App.config]. La solution présentée ici n'est donc pas la seule disponible.

Créons le fichier [App.config] avec l'assistant Visual Studio :



- en [1] : ajout d'un nouvel élément au projet
- en [2] : sélectionner "Application Configuration File"
- en [3] : [App.config] est le nom par défaut de ce fichier de configuration
- en [4] : le fichier [App.config] a été ajouté au projet

Le contenu du fichier [App.config] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.  </configuration>
```

[App.config] est un fichier XML. La configuration du projet se fait entre les balises <configuration>. La configuration nécessaire à *Spring* est la suivante :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.   </configSections>
10.
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <object name="dao" type="Dao.FileImpot, ImpotsV5-dao">
17.           <constructor-arg index="0" value="DataImpot.txt"/>
18.        </object>
19.        <object name="metier" type="Metier.ImpotMetier, ImpotsV5-metier">
20.           <constructor-arg index="0" ref="dao"/>
21.        </object>
22.     </objects>
23.  </spring>
24. </configuration>
```

- lignes 11-23 : la section délimitée par la balise <spring> est appelée le groupe de sections <spring>. On peut créer autant de groupes de sections que l'on veut dans [App.config].
- un groupe de sections a des sections : c'est le cas ici :
  - lignes 12-14 : la section <spring/context>
  - lignes 15-22 : la section <spring/objects>
- lignes 4-9 : la région <configSections> définit la liste des gestionnaires (handlers) des groupes de sections présents dans [App.config].
- lignes 5-8 : définit la liste des gestionnaires des sections du groupe <spring> (**name**="spring").
- ligne 6 : le gestionnaire de la section <context> du groupe <spring> :
  - **name** : nom de la section gérée
  - **type** : nom de la classe gérant la section sous la forme *NomClasse, NomDLL*.
  - la section <context> du groupe <spring> est gérée par la classe [Spring.Context.Support.ContextHandler] qui sera trouvée dans la DLL [Spring.Core.dll]
- ligne 7 : le gestionnaire de la section <objects> du groupe <spring>

Les lignes 4-9 sont standard dans un fichier [App.config] avec Spring. On se contente de les recopier d'un projet à l'autre.

- lignes 12-14 : définit la section <spring/context>.
- ligne 13 : la balise <**resource**> a pour but d'indiquer où se trouve le fichier définissant les classes que Spring doit instancier. Celles-ci peuvent être dans [App.config] comme ici mais elles peuvent être également dans un fichier de configuration autre. La localisation de ces classes est faite dans l'attribut **uri** de la balise <resource> :
  - **<resource uri="config://spring/objects>** indique que la liste des classes à instancier se trouve dans le fichier [App.config] (**config:**), dans la section **//spring/objects**, c.a.d. dans la balise <objects> de la balise <spring>.
  - <**resource uri="file://spring-config.xml"**> indiquerait que la liste des classes à instancier se trouve dans le fichier [spring-config.xml]. Celui-ci devrait être placé dans les dossiers d'exécution (bin/Release ou bin/Debug) du projet. Le plus simple est de le placer, comme il a été fait pour le fichier [DataImpot.txt], à la racine du projet avec la propriété [Copy to output directory=always].

  Les lignes 12-14 sont standard dans un fichier [App.config] avec Spring. On se contente de les recopier d'un projet à l'autre.
- lignes 15-22 : définissent les classes à instancier. C'est dans cette partie que se fait la configuration spécifique d'une application. La balise <**objects**> délimite la section de définition des classes à instancier.
- lignes 16-18 : définissent la classe à instancier pour la couche [dao]
  - ligne 16 : chaque objet instancié par Spring fait l'objet d'une balise <**object**>. Celle-ci a un attribut **name** qui est le nom de l'objet instancié. C'est via celui-ci que l'application demande à Spring une référence : "donne-moi une référence sur l'objet qui s'appelle dao". L'attribut **type** définit la classe à instancier sous la forme *NomClasse, NomDLL*. Ainsi la ligne 16 définit un objet appelé "**dao**", instance de la classe "**Dao.FileImpot**" qui se trouve dans la DLL "**ImpotsV5-dao.dll**". On notera qu'on donne le nom complet de la classe (espace de noms inclus) et que le suffixe .dll n'est pas précisé dans le nom de la DLL.

  Une classe peut être instanciée de deux façons avec Spring :

1. via un constructeur particulier auquel on passe des paramètres : c'est ce qui est fait dans les lignes 16-18.
2. via le constructeur par défaut sans paramètres. L'objet est alors initialisé via ses **propriétés publiques** : la balise <object> a alors des sous-balises <**property**> pour initialiser ces différentes propriétés. Nous n'avons pas d'exemple de ce cas ici.

- ligne 16 : la classe instanciée est la classe *FileImpot*. Celle-ci a le constructeur suivant :

```
public FileImpot(string fileName);
```

Les paramètres du constructeur sont définis à l'aide de balises **<constructor-arg>**.

- ligne 17 : définit le 1er et seul paramètre du constructeur. L'attribut **index** est le n° du paramètre du constructeur, l'attribut **value** sa valeur : **<constructor-arg index="i" value="valuei"/>**

- lignes 19-21 : définissent la classe à instancier pour la couche [metier] : la classe [Metier.ImpotMetier] qui se trouve dans la DLL [ImpotsV5-metier.dll].
  - ligne 19 : la classe instanciée est la classe *ImpotMetier*. Celle-ci a le constructeur suivant :

```
public ImpotMetier(IImpotDao dao);
```

  - ligne 20 : définit le 1er et seul paramètre du constructeur. Ci-dessus, le paramètre *dao* du constructeur est une référence d'objet. Dans ce cas, dans la balise <constructor-arg> on utilise l'attribut **ref** au lieu de l'attribut **value** qui a été utilisé pour la couche [dao] : **<constructor-arg index="i" ref="refi"/>**. Dans le constructeur ci-dessus, le paramètre *dao* représente une instance sur la couche [dao]. Cette instance a été définie par les lignes 16-18 du fichier de configuration. Ainsi dans la ligne 20 :

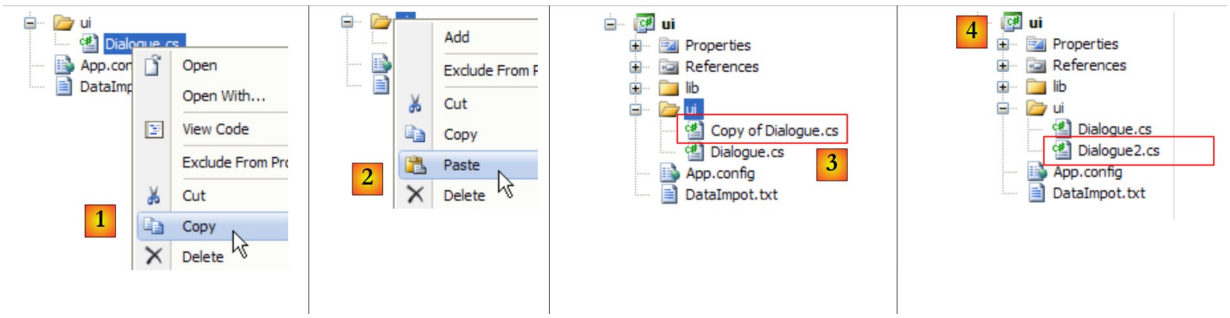
```
<constructor-arg index="0" ref="dao"/>
```

ref="dao" représente l'objet Spring "dao" défini par les lignes 16-18.

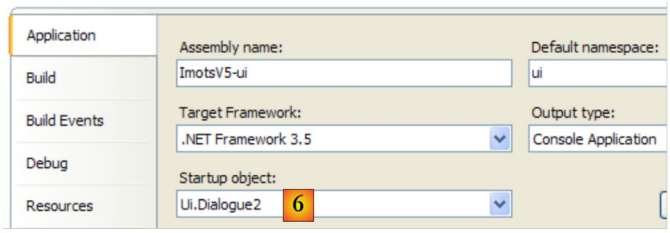
Pour résumer, le fichier [App.config] :

- instancie la couche [dao] avec la classe *FileImpot* qui reçoit pour paramètre *DataImpot.txt* (ligne 16-18). L'objet résultant est appelé "dao"
- instancie la couche [metier] avec la classe *ImpotMetier* qui reçoit pour paramètre l'objet "dao" précédent (lignes 19-21).

Il ne nous reste plus qu'à utiliser ce fichier de configuration Spring dans la couche [ui]. Pour cela, nous dupliquons la classe [Dialogue.cs] en [Dialogue2.cs] et nous faisons de cette dernière la classe principale du projet [ui] :



- en [1] : copie de [Dialogue.cs]
- en [2] : collage
- en [3] : la copie de [Dialogue.cs]
- en [4] : renommée [Dialogue2.cs]



- en [6] : on fait de [Dialogue2.cs] la classe principale du projet [ui].

Le code suivant de [Dialogue.cs] :

```
1.          // on crée les couches [metier et dao]
2.          IImpotMetier metier = null;
3.          try {
4.         // création couche [metier]
5.             metier = new ImpotMetier(new FileImpot("DataImpot.txt"));
6.          } catch (ImpotException e) {
7.             // affichage erreur
8.             string msg = e.InnerException == null ? null : String.Format(", Exception
    d'origine : {0}", e.InnerException.Message);
9.             Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
10.            // arrêt programme
11.            Environment.Exit(1);
12.         }
13.         // boucle infinie
14.         while (true) {
15. ...
```

devient le suivant dans [Dialogue2.cs] :

```
1.        // on crée les couches [metier et dao]
2.        IApplicationContext ctx = null;
3.        try {
4.           // contexte Spring
5.           ctx = ContextRegistry.GetContext();
6.        } catch (Exception e1) {
7.           // affichage erreur
8.           Console.WriteLine("Chaîne des exceptions : \n{0}", "".PadLeft(40, '-'));
9.           Exception e = e1;
10.          while (e != null) {
11.             Console.WriteLine("{0}: {1}", e.GetType().FullName, e.Message);
12.             Console.WriteLine("".PadLeft(40, '-'));
13.             e = e.InnerException;
14.          }
15.          // arrêt programme
16.          Environment.Exit(1);
17.       }
18.       // on demande une référence sur la couche [metier]
19.       IImpotMetier metier = (IImpotMetier)ctx.GetObject("metier");
20.       // boucle infinie
21.       while (true) {
22. ..................................
```

- ligne 2 : **IApplicationContext** donne accès à l'ensemble des objets instanciés par Spring. On appelle cet objet, le contexte Spring de l'application ou plus simplement le contexte de l'application. Pour l'instant, ce contexte n'a pas été initialisé. C'est le try / catch qui suit qui le fait.
- ligne 5 : la configuration de Spring dans [App.config] est lue et exploitée. Après cette opération, s'il n'y a pas eu d'exception, tous les objets de la section <objects> **ont été instanciés** :
  - l'objet Spring "dao" est une instance sur la couche [dao]
  - l'objet Spring "metier" est une instance sur la couche [metier]

- ligne 19 : la classe [Dialogue2.cs] a besoin d'une référence sur la couche [metier]. Celle-ci est demandée au contexte de l'application. L'objet **IApplicationContext** donne accès aux objets Spring via leur nom (attribut **name** de la balise **<object>** de la configuration Spring). La référence rendue est une référence sur le type générique *Object*. On est amenés à transtyper la référence rendue dans le bon type, ici le type de l'interface de la couche [metier] : *IImpotMetier*.

Si tout s'est bien passé, après la ligne 19, [Dialogue2.cs] a une référence sur la couche [metier]. Le code des lignes 21 et au-delà est celui de la classe [Dialogue.cs] déjà étudiée.

- lignes 6-17 : gestion de l'exception qui survient lorsque l'exploitation du fichier de configuration de Spring ne peut être menée à son terme. Il peut y avoir diverses raisons à cela : syntaxe incorrecte du fichier de configuration lui-même ou bien impossibilité à instancier l'un des objets configurés. Dans notre exemple, ce dernier cas se produirait si le fichier *DataImpot.txt* de la ligne 17 de [App.config] n'était pas trouvé dans le dossier d'exécution du projet.

  L'exception qui remonte ligne 6 est une chaîne d'exceptions où chaque exception a deux propriétés :
  - **Message** : le message d'erreur liée à l'exception
  - **InnerException** : l'exception précédente dans la chaîne des exceptions

  La boucle des lignes 10-14 fait afficher toutes les exceptions de la chaîne sous la forme : classe de l'exception et message associé.

Lorsqu'on exécute le projet [ui] avec un fichier de configuration valide, on obtient les résultats habituels :

```
Paramètres du calcul de l'Impot au format : Marié (o/n) NbEnfants Salaire ou rien pour arrêter :o 2 60000
Impot=4282 euros
```

Lorsqu'on exécute le projet [ui] avec un fichier [DataImpotInexistant.txt] inexistant,

```
        <object name="dao" type="Dao.FileImpot, ImpotsV5-dao">
           <constructor-arg index="0" value="DataImpotInexistant.txt"/>
        </object>
```

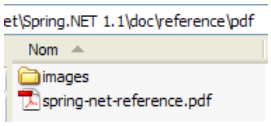
on obtient les résultats suivants :

```
1.  Chaîne des exceptions :
2.  -------------------------------------
3.  System.Configuration.ConfigurationErrorsException: Error creating context 'spring.root': Could not
    find file 'C:\data\2007-2008\c# 2008\poly\Chap4\ImpotsV5\ui\bin\Release\DataImpotInexistant.txt'.
4.  -------------------------------------
5.  Spring.Util.FatalReflectionException: Cannot instantiate Type
    [Spring.Context.Support.XmlApplicationContext] using ctor [Void .ctor(System.String, Boolean,
    System.String[])] : 'Exception has been thrown by the target of an invocation.'
6.  -------------------------------------
7.  System.Reflection.TargetInvocationException: Exception has been thrown by the target of an
    invocation.
8.  -------------------------------------
9.  Spring.Objects.Factory.ObjectCreationException: Error creating object with name'dao' defined in
    'config [spring/objects]' : Initialization of object failed : Cannot instantiate Type
    [Dao.FileImpot] using ctor [Void .ctor(System.String)] :'Exception has been thrown by the target
    of an invocation.'
10. -------------------------------------
11. Spring.Util.FatalReflectionException: Cannot instantiate Type [Dao.FileImpot] using ctor
    [Void .ctor(System.String)] : 'Exception has been thrown by the targetof an invocation.'
12. -------------------------------------
13. System.Reflection.TargetInvocationException: Exception has been thrown by the target of an
    invocation.
14. -------------------------------------
15. Entites.ImpotException: Erreur lors de la lecture du fichier DataImpotInexistant.txt
16. -------------------------------------
17. System.IO.FileNotFoundException: Could not find file 'C:\data\2007-2008\c#
    2008\poly\Chap4\ImpotsV5\ui\bin\Release\DataImpotInexistant.txt'.
```

- ligne 17 : l'exception originelle de type [FileNotFoundException]
- ligne 15 : la couche [dao] encapsule cette exception dans un type [Entites.ImpotException]
- ligne 9 : l'exception lancée par Spring parce qu'il n'a pas réussi à instancier l'objet nommé "dao". Dans le processus de création de cet objet, deux autres exceptions sont intervenues auparavant : celles des lignes 11 et 13.
- parce que l'objet "dao" n'a pu être créé, le contexte de l'application n'a pu être créé. C'est le sens de l'exception ligne 5. Auparavant, une autre exception, celle de la ligne 7 s'était produite.
- ligne 3 : l'exception de plus haut niveau, la dernière de la chaîne : une erreur de configuration est signalée.

De tout cela, on retiendra que c'est l'exception la plus profonde, ici celle de la ligne 17 qui est souvent la plus significative. On notera cependant que Spring a conservé le message d'erreur de la ligne 17 pour le remonter à l'exception de plus haut niveau ligne 3 afin d'avoir la cause originelle de l'erreur au niveau le plus haut.

Spring mérite à lui tout seul un livre. Nous n'avons fait ici qu'effleurer le sujet. On pourra l'approfondir avec le document [spring-net-reference.pdf] qu'on trouve dans le dossier d'installation de Spring :



On pourra lire également [http://tahe.developpez.com/dotnet/springioc], un tutoriel Spring présenté dans un contexte VB.NET.

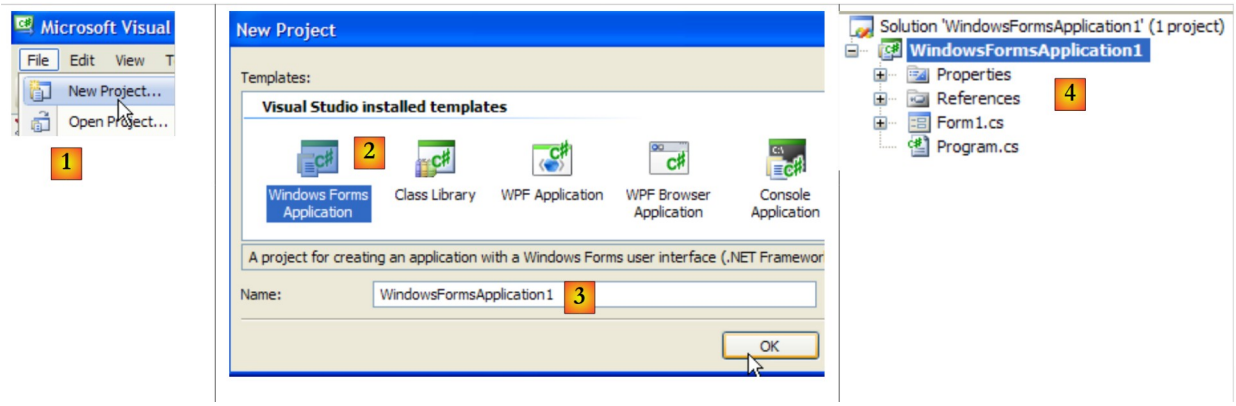
7 Interfaces graphiques avec C# et VS.NET

7.1 Les bases des interfaces graphiques

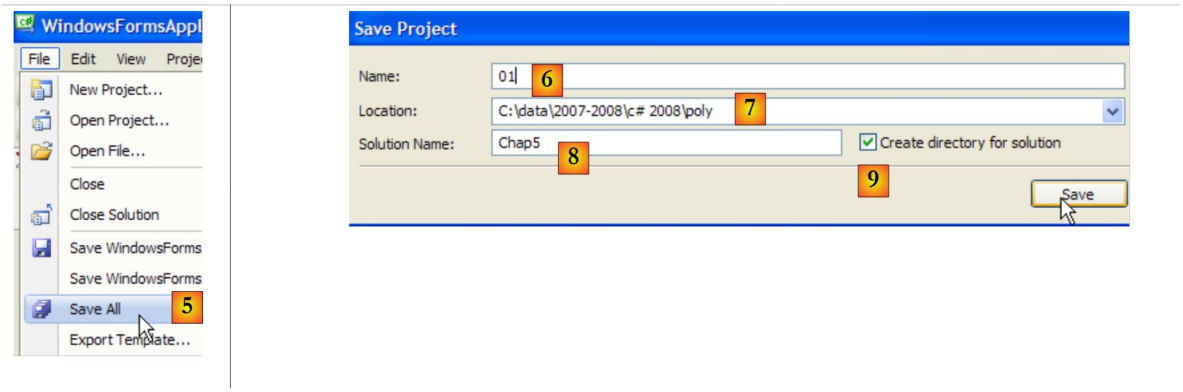
Nous nous proposons ici de donner les premiers éléments pour construire des interfaces graphiques et gérer leurs événements.

7.1.1 Un premier projet

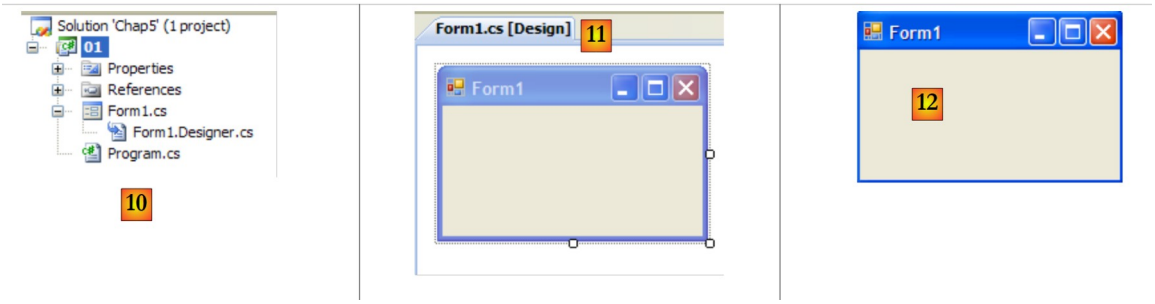
Construisons un premier projet de type "Application windows" :



- [1] : créer un nouveau projet
- [2] : de type Application Windows
- [3] : le nom du projet importe peu pour le moment
- [4] : le projet créé



- [5] : on sauvegarde la solution courante
- [6] : nom du projet
- [7] : dossier de la solution
- [8] : nom de la solution
- [9] : un dossier sera créé pour la solution [Chap5]. Les projets de celle-ci seront dans des sous-dossiers.



- [10] : le projet [01] dans la solution [Chap5] :
  - [Program.cs] est la classe principale du projet
  - [Form1.cs] est le fichier source qui va gérer le comportement de la fenêtre [11]
  - [Form1.Designer.cs] est le fichier source qui va encapsuler l'information sur les composants de la fenêtre [11]
- [11] : le fichier [Form1.cs] en mode "conception" (design)
- [12] : l'application générée peut être exécutée par (Ctrl-F5). La fenêtre [Form1] s'affiche. On peut la déplacer, la redimensionner et la fermer. On a donc les éléments de base d'une fenêtre graphique.

La classe principale [Program.cs] est la suivante :

```
1.  using System;
2.  using System.Windows.Forms;
3.
4.  namespace Chap5 {
5.   static class Program {
6.      /// <summary>
7.      /// The main entry point for the application.
8.      /// </summary>
9.      [STAThread]
10.     static void Main() {
11.        Application.EnableVisualStyles();
12.        Application.SetCompatibleTextRenderingDefault(false);
13.        Application.Run(new Form1());
14.     }
15.  }
16. }
```

- ligne 2 : les applications avec formulaires utilisent l'espace de noms *System.Windows.Forms*.
- ligne 4 : l'espace de noms initial a été renommé en *Chap5*.
- ligne 10 : à l'exécution du projet (Ctrl-F5), la méthode [Main] est exécutée.
- lignes 11-13 : la classe *Application* appartient à l'espace de noms *System.Windows.Forms*. Elle contient des méthodes statiques pour lancer / arrêter les applications graphiques windows.
- ligne 11 : facultative - permet de donner différents styles visuels aux contrôles déposés sur un formulaire
- ligne 12 : facultative - fixe le moteur de rendu des textes des contrôles : GDI+ (true), GDI (false)
- ligne 13 : la seule ligne indispensable de la méthode [Main] : instancie la classe [Form1] qui est la classe du formulaire et lui demande de s'exécuter.

Le fichier source [Form1.cs] est le suivant :

```
1.  using System;
2.  using System.Windows.Forms;
3.
4.  namespace Chap5 {
5.   public partial class Form1 : Form {
6.      public Form1() {
7.         InitializeComponent();
8.      }
9.   }
10. }
```

- ligne 5 : la classe *Form1* dérive de la classe [System.Windows.Forms.Form] qui est la classe mère de toutes les fenêtres. Le mot clé **partial** indique que la classe est partielle et qu'elle peut être complétée par d'autres fichiers source. C'est le cas ici, où la classe *Form1* est répartie dans deux fichiers :
  - [Form1.cs] : dans lequel on trouvera le comportement du formulaire, notamment ses gestionnaires d'événements
  - [Form1.Designer.cs] : dans lequel on trouvera les composants du formulaire et leurs propriétés. Ce fichier a la particularité d'être régénéré à chaque fois que l'utilisateur modifie la fenêtre en mode [conception].

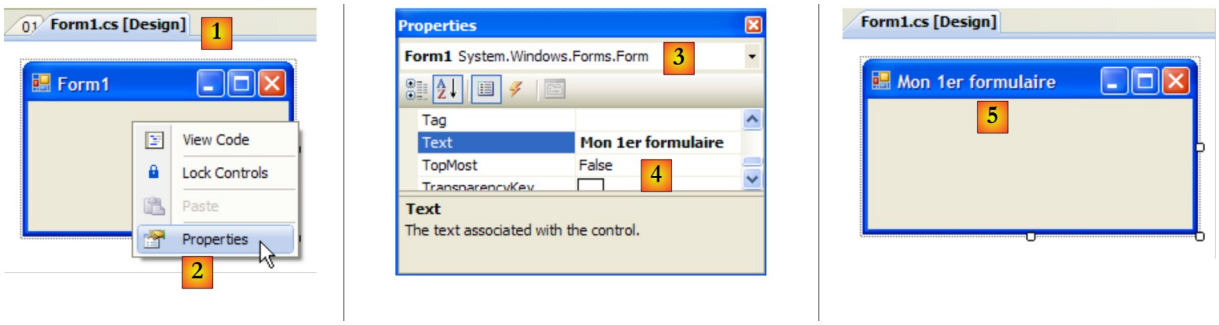
- lignes 6-8 : le constructeur de la classe *Form1*
- ligne 7 : fait appel à la méthode *InitializeComponent*. On voit que cette méthode n'est pas présente dans [Form1.cs]. On la trouve dans [Form1.Designer.cs].

Le fichier source [Form1.Designer.cs] est le suivant :

```
1.  namespace Chap5 {
2.   partial class Form1 {
3.      /// <summary>
4.      /// Required designer variable.
5.      /// </summary>
6.      private System.ComponentModel.IContainer components = null;
7.
8.      /// <summary>
9.      /// Clean up any resources being used.
10.     /// </summary>
11.     /// <param name="disposing">true if managed resources should be disposed; otherwise,
    false.</param>
12.     protected override void Dispose(bool disposing) {
13.        if (disposing && (components != null)) {
14.           components.Dispose();
15.        }
16.        base.Dispose(disposing);
17.     }
18.
19.     #region Windows Form Designer generated code
20.
21.     /// <summary>
22.     /// Required method for Designer support - do not modify
23.     /// the contents of this method with the code editor.
24.     /// </summary>
25.     private void InitializeComponent() {
26.        this.SuspendLayout();
27.        //
28.        // Form1
29.        //
30.        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
31.        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
32.        this.ClientSize = new System.Drawing.Size(196, 98);
33.        this.Name = "Form1";
34.        this.Text = "Form1";
35.        this.ResumeLayout(false);
36.
37.     }
38.
39.     #endregion
40.
41.  }
42. }
```

- ligne 2 : il s'agit toujours de la classe *Form1*. On notera qu'il n'est plus besoin de répéter qu'elle dérive de la classe *Form*.
- lignes 25-37 : la méthode *InitializeComponent* appelée par le constructeur de la classe [Form1]. Cette méthode va créer et initialiser tous les composants du formulaire. Elle est régénérée à chaque changement de celui-ci en mode [conception]. Une section, appelée *région*, est créée pour la délimiter lignes 19-39. Le développeur ne doit pas ajouter de code dans cette région : il sera écrasé à la régénération suivante.

Il est plus simple dans un premier temps de ne pas s'intéresser au code de [Form1.Designer.cs]. Il est généré automatiquement et est la traduction en langage C# des choix que le développeur fait en mode [conception]. Prenons un premier exemple :



- [1] : sélectionner le mode [conception] en double-cliquant sur le fichier [Form1.cs]
- [2] : cliquer droit sur le formulaire et choisir [Properties]
- [3] : la fenêtre des propriétés de [Form1]
- [4] : la propriété [Text] représente le titre de la fenêtre
- [5] : le changement de la propriété [Text] est pris en compte en mode [conception] ainsi que dans le code source [Form1.Designer.cs] :

```
1.        private void InitializeComponent() {
2.           this.SuspendLayout();
3.  ...
4.           this.Text = "Mon 1er formulaire";
5.  ...
6.  }
```

7.1.2 Un second projet

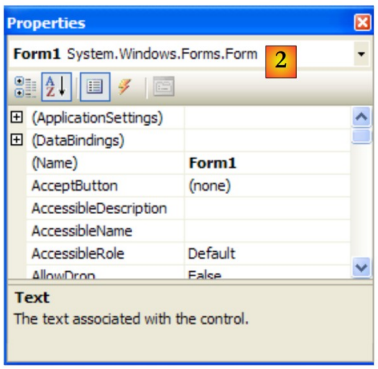
7.1.2.1 Le formulaire

Nous commençons un nouveau projet appelé *02*. Pour cela nous suivons la procédure explicitée précédemment pour créer un projet. La fenêtre à créer est la suivante :

Les composants du formulaire sont les suivants :

n°	nom	type	rôle
1	labelSaisie	Label	un libellé
2	textBoxSaisie	TextBox	une zone de saisie
3	buttonAfficher	Button	pour afficher dans une boîte de dialogue le contenu de la zone de saisie textBoxSaisie

On pourra procéder comme suit pour construire cette fenêtre :



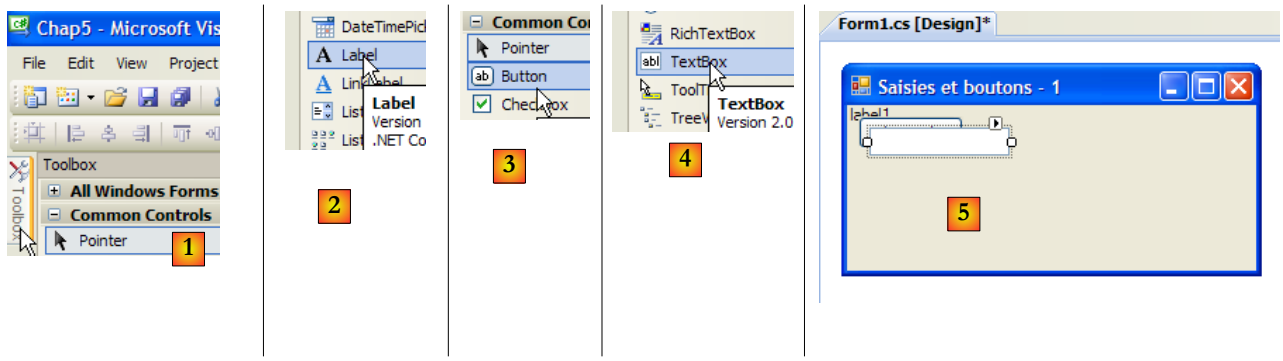
- [1] : cliquer droit sur le formulaire en-dehors de tout composant et choisir l'option [Properties]
- [2] : la feuille de propriétés de la fenêtre apparaît dans le coin inférieur droit de Visual studio

Parmi les propriétés du formulaire 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

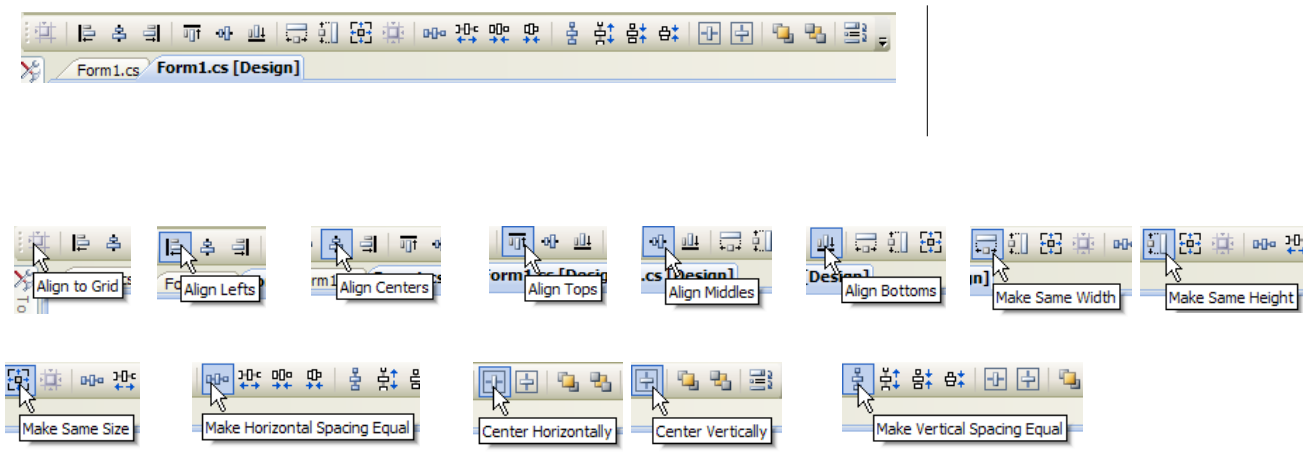
Ici, nous fixons les propriétés *Text* et *Name* :

`Text`	Saisies et boutons - 1
`Name`	frmSaisiesBoutons



- [1] : choisir la boîte à outils [Common Controls] parmi les boîtes à outils proposées par Visual Studio
- [2, 3, 4] : double-cliquer successivement sur les composants [Label], [Button] et [TextBox]
- [5] : les trois composants sont sur le formulaire

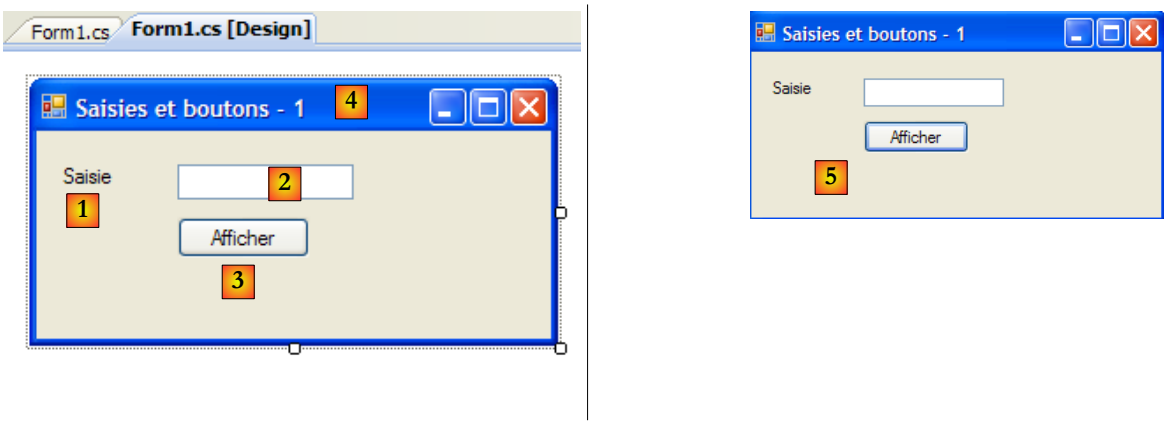
Pour aligner et dimensionner correctement les composants, on peut utiliser les éléments de la barre d'outils :



Le principe du formatage est le suivant :

1. sélectionnez les différents composants à formater ensemble (touche Ctrl appuyée pendant les différents clics sélectionnant les composants)
2. sélectionnez le type de formatage désiré :
   - les options *Align* permettent d'aligner des composants par le haut, le bas, le côté gauche ou droit, le milieu
   - les options *Make Same Size* permettent que des composants aient la même hauteur ou la même largeur
   - l'option *Horizontal Spacing* permet d'aligner horizontalement des composants avec des intervalles entre eux de même largeur. Idem pour l'option *Vertical Spacing* pour aligner verticalement.
   - l'option *Center* permet de centrer un composant horizontalement (*Horizontally*) ou verticalement (*Vertically*) dans la fenêtre

Une fois placés les composants nous fixons leurs propriétés. Pour cela, cliquer droit sur le composant et prendre l'option *Properties* :



- [1] : sélectionner le composant pour avoir sa fenêtre de propriétés. Dans celle-ci, modifier les propriétés suivantes : **name** : *labelSaisie*, **text** : *Saisie*
- [2] : procéder de même : **name** : *textBoxSaisie*, **text** : ne rien mettre
- [3] : **name** : *buttonAfficher*, **text** : *Afficher*
- [4] : la fenêtre elle-même : **name** : *frmSaisiesBoutons*, **text** : *Saisies et boutons - 1*
- [5] : exécuter (Ctrl-F5) le projet pour avoir un premier aperçu de la fenêtre en action.

Ce qui a été fait en mode [conception] a été traduit dans le code de [Form1.Designer.cs] :

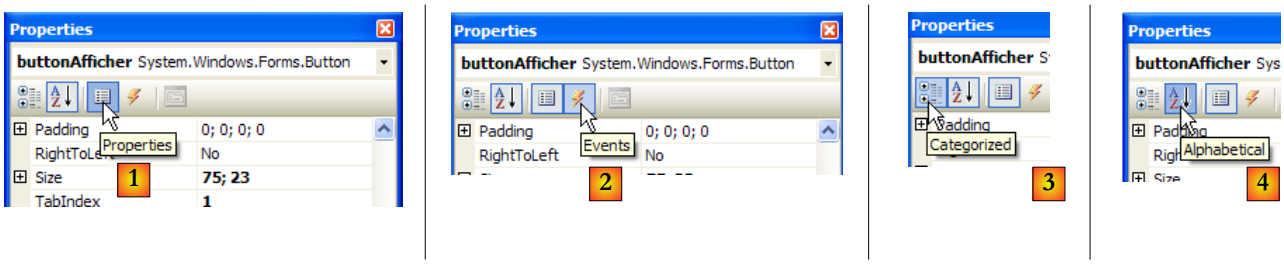
```
1.  namespace Chap5 {
2.   partial class frmSaisiesBoutons {
3.  ...
4.      private System.ComponentModel.IContainer components = null;
5.  ...
6.      private void InitializeComponent() {
7.          this.labelSaisie = new System.Windows.Forms.Label();
8.          this.buttonAfficher = new System.Windows.Forms.Button();
9.          this.textBoxSaisie = new System.Windows.Forms.TextBox();
10.         this.SuspendLayout();
11.         //
12.         // labelSaisie
13.         //
14.         this.labelSaisie.AutoSize = true;
15.         this.labelSaisie.Location = new System.Drawing.Point(12, 19);
16.         this.labelSaisie.Name = "labelSaisie";
17.         this.labelSaisie.Size = new System.Drawing.Size(35, 13);
18.         this.labelSaisie.TabIndex = 0;
19.         this.labelSaisie.Text = "Saisie";
20.         //
21.         // buttonAfficher
22.         //
23.         this.buttonAfficher.Location = new System.Drawing.Point(80, 49);
24.         this.buttonAfficher.Name = "buttonAfficher";
25.         this.buttonAfficher.Size = new System.Drawing.Size(75, 23);
26.         this.buttonAfficher.TabIndex = 1;
27.         this.buttonAfficher.Text = "Afficher";
28.         this.buttonAfficher.UseVisualStyleBackColor = true;
29.         this.buttonAfficher.Click += new System.EventHandler(this.buttonAfficher_Click);
30.         //
31.         // textBoxSaisie
32.         //
33.         this.textBoxSaisie.Location = new System.Drawing.Point(80, 19);
34.         this.textBoxSaisie.Name = "textBoxSaisie";
35.         this.textBoxSaisie.Size = new System.Drawing.Size(100, 20);
36.         this.textBoxSaisie.TabIndex = 2;
37.         //
38.         // frmSaisiesBoutons
39.         //
40.         this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
41.         this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
42.         this.ClientSize = new System.Drawing.Size(292, 118);
```

```
43.         this.Controls.Add(this.textBoxSaisie);
44.         this.Controls.Add(this.buttonAfficher);
45.         this.Controls.Add(this.labelSaisie);
46.         this.Name = "frmSaisiesBoutons";
47.         this.Text = "Saisies et boutons - 1";
48.         this.ResumeLayout(false);
49.         this.PerformLayout();
50.
51.     }
52.
53.     private System.Windows.Forms.Label labelSaisie;
54.     private System.Windows.Forms.Button buttonAfficher;
55.     private System.Windows.Forms.TextBox textBoxSaisie;
56.
57.  }
58. }
```

- lignes 53-55 : les trois composants ont donné naissance à trois champs privés de la classe [Form1]. On notera que les noms de ces champs sont les noms donnés aux composants en mode [conception]. C'est le cas également du formulaire ligne 2 qui est la classe elle-même.
- lignes 7-9 : les trois objets de type [Label], [TextBox] et [Button] sont créés. C'est à travers eux que les composants visuels sont gérés.
- lignes 14-19 : configuration du label *labelSaisie*
- lignes 23-29 : configuration du bouton *buttonAfficher*
- lignes 33-36 : configuration du champ de saisie *textBoxSaisie*
- lignes 40-47 : configuration du formulaire *frmSaisiesBoutons*. On notera, lignes 43-45, la façon d'ajouter des composants au formulaire.

Ce code est compréhensible. Il est ainsi possible de construire des formulaires par code sans utiliser le mode [conception]. De nombreux exemples de ceci sont donnés dans la documentation MSDN de Visual Studio. Maîtriser ce code permet de créer des formulaires en cours d'exécution : par exemple, créer à la volée un formulaire permettant la mise à jour d'une table de base de données, la structure de cette table n'étant découverte qu'à l'exécution.

Il nous reste à écrire la procédure de gestion d'un clic sur le bouton *Afficher*. Sélectionner le bouton pour avoir accès à sa fenêtre de propriétés. Celle-ci a plusieurs onglets :

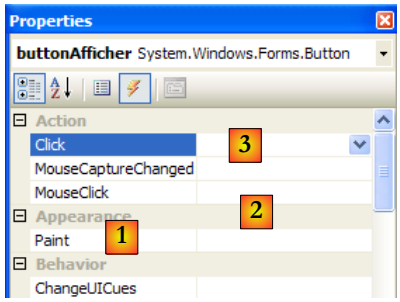


- [1] : liste des propriétés par ordre alphabétique
- [2] : événements liés au contrôle

Les propriétés et événements d'un contrôle sont accessibles par catégories ou par ordre alphabétique :

- [3] : Propriétés ou événements par catégorie
- [4] : Propriétés ou événements par ordre alphabétique

L'onglet *Events* en mode *Catégories* pour le bouton *buttonAfficher* est le suivant :



- [1] : la colonne de gauche de la fenêtre liste les événements possibles sur le bouton. Un clic sur un bouton correspond à l'événement *Click*.
- [2] : la colonne de droite contient le nom de la procédure appelée lorsque l'événement correspondant se produit.
- [3] : si on double-clique 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 *buttonAfficher* :

```
1.  using System;
2.  using System.Windows.Forms;
3.
4.  namespace Chap5 {
5.   public partial class frmSaisiesBoutons : Form {
6.      public frmSaisiesBoutons() {
7.         InitializeComponent();
8.      }
9.
10.     private void buttonAfficher_Click(object sender, EventArgs e) {
11.
12.     }
13.  }
14. }
```

- lignes 10-12 : le squelette du gestionnaire de l'événement *Click* sur le bouton nommé *buttonAfficher*. On notera les points suivants :
  - la méthode est nommée selon le schéma **nomDuComposant_NomEvénement**
  - la méthode est privée. Elle reçoit deux paramètres :
  - *sender* : est l'objet qui a provoqué l'événement. Si la procédure est exécutée à la suite d'un clic sur le bouton *buttonAfficher*, *sender* sera égal à *buttonAfficher*. On peut imaginer que la procédure *buttonAfficher_Click* soit exécutée à partir d'une autre procédure. Celle-ci aurait alors tout loisir de mettre comme premier paramètre, l'objet *sender* de son choix.
  - *EventArgs* : un objet qui contient des informations sur l'événement. Pour un événement *Click*, il ne contient rien. Pour un événement ayant trait aux déplacements de la souris, on y trouvera les coordonnées (X,Y) de la souris.
  - nous n'utiliserons aucun de ces paramètres ici.

Ecrire un gestionnaire d'événement consiste à compléter le squelette de code précédent. Ici, nous voulons présenter une boîte de dialogue avec dedans, le contenu du champ *textBoxSaisie* s'il est non vide [1], un message d'erreur sinon [2] :

Le code réalisant cela pourrait-être le suivant :

```
1.        private void buttonAfficher_Click(object sender, EventArgs e) {
2.           // on affiche le texte qui a été saisi dans le TextBox textboxSaisie
3.           string texte = textBoxSaisie.Text.Trim();
4.           if (texte.Length != 0) {
5.              MessageBox.Show("Texte saisi= " + texte, "Vérification de la saisie",
    MessageBoxButtons.OK, MessageBoxIcon.Information);
6.           } else {
7.              MessageBox.Show("Saissez un texte...", "Vérification de la saisie",
    MessageBoxButtons.OK, MessageBoxIcon.Error);
8.  }
```

La classe *MessageBox* sert à afficher des messages dans une fenêtre. Nous avons utilisé ici la méthode *Show* suivante :

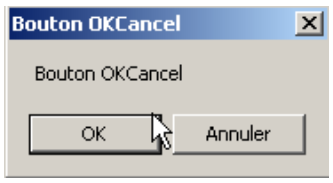
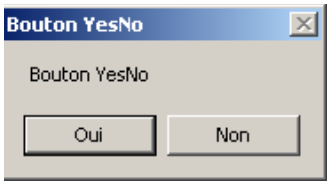
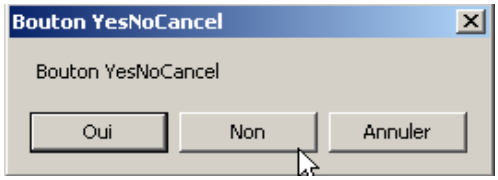
```
public static DialogResult Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon
icon);
```

avec

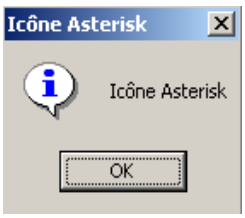
`text`	le message à afficher
`caption`	le titre de la fenêtre
`buttons`	les boutons présents dans la fenêtre
`icon`	l'icone présente dans la fenêtre

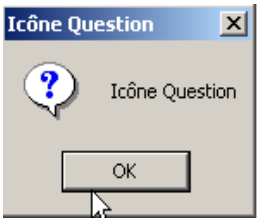
Le paramètre *buttons* peut prendre ses valeurs parmi les constantes suivantes (préfixées par *MessageBoxButtons* comme montré ligne 7) ci-dessus :

constante	boutons
AbortRetryIgnore	
OK	

constante	boutons
OKCancel	
RetryCancel	
YesNo	
YesNoCancel	

Le paramètre *icon* peut prendre ses valeurs parmi les constantes suivantes (préfixées par *MessageBoxIcon* comme montré ligne 10) ci-dessus :

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 :

Members

	Member name	Description
	Abort	The dialog box return value is **Abort** (usually sent from a button labeled Abort).
	Cancel	The dialog box return value is **Cancel** (usually sent from a button labeled Cancel).
	Ignore	The dialog box return value is **Ignore** (usually sent from a button labeled Ignore).
	No	The dialog box return value is **No** (usually sent from a button labeled No).
	None	**Nothing** is returned from the dialog box. This means that the modal dialog continues running.
	OK	The dialog box return value is **OK** (usually sent from a button labeled OK).
	Retry	The dialog box return value is **Retry** (usually sent from a button labeled Retry).
	Yes	The dialog box return value is **Yes** (usually sent from a button labeled Yes).

Pour savoir sur quel bouton a appuyé l'utilisateur pour fermer la fenêtre de type *MessageBox* on écrira :

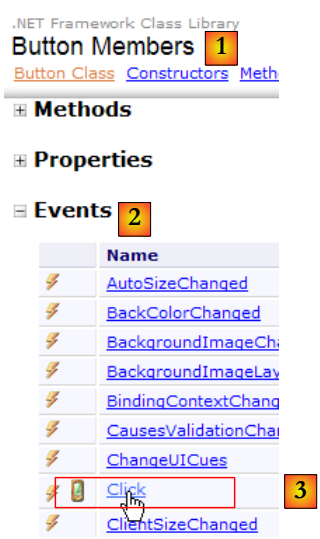
```
1.  DialogResult res=MessageBox.Show(..);
2.  if (res==DialogResult.Yes){ // il a appuyé sur le bouton oui...}
```

7.1.2.2 Le code lié à la gestion des événements

Outre la fonction *buttonAfficher_Click* que nous avons écrite, Visual studio a généré dans la méthode *InitializeComponents* de [Form1.Designer.cs] qui crée et initialise les composants du formulaire, la ligne suivante :

```
        this.buttonAfficher.Click += new System.EventHandler(this.buttonAfficher_Click);
```

Click est un **événement** de la classe *Button* [1, 2, 3] :



- [5] : la déclaration de l'événement [Control.Click] [4]. Ainsi on voit que l'événement **Click** n'est pas propre à la classe [Button]. Il appartient à la classe [Control], classe parente de la classe [Button].
  - *EventHandler* est un prototype (un modèle) de méthode appelé **delegate**. Nous y allons y revenir.
  - *event* est un mot clé qui restreint les fonctionnalités du *delegate* **EventHandler** : un objet *delegate* a des fonctionnalités plus riches qu'un objet *event*.

Le *delegate* **EventHandler** est défini comme suit :

Le *delegate* **EventHandler** désigne un modèle de méthode :
- ayant pour 1er paramètre un type **Object**
- ayant pour 2ième paramètre un type **EventArgs**
- ne rendant aucun résultat

C'est le cas de la méthode de gestion du clic sur le bouton *buttonAfficher* qui a été générée par Visual Studio :

```
    private void buttonAfficher_Click(object sender, EventArgs e);
```

Ainsi la méthode *buttonAfficher_Click* correspond au prototype défini par le type *EventHandler*. Pour créer un objet de type *EventHandler*, on procède comme suit :

```
EventHandler evtHandler=new EventHandler(méthode correspondant au prototype  défini par le type
EventHandler);
```

Puisque la méthode *buttonAfficher_Click* correspond au prototype défini par le type *EventHandler*, on pourra écrire :

```
EventHandler evtHandler=new EventHandler(buttonAfficher_Click);
```

Une variable de type *delegate* est en fait une liste de références sur des méthodes du type du *delegate*. Pour ajouter une nouvelle méthode *M* à la variable *evtHandler* ci-dessus, on utilisera la syntaxe :

```
evtHandler+=new EvtHandler(M);
```

La notation += peut être utilisée même si *evtHandler* est une liste vide.

Revenons à la ligne de [InitializeComponent] qui ajoute un gestionnaire d'événement à l'événement *Click* de l'objet *buttonAfficher* :

```
        this.buttonAfficher.Click += new System.EventHandler(this.buttonAfficher_Click);
```

Cette instruction ajoute une méthode de type *EventHandler* à la liste des méthodes du champ *buttonAfficher.Click*. Ces méthodes seront appelées à chaque fois que l'événement *Click* sur le composant *buttonAfficher* sera détecté. Il n'y en a souvent qu'une. On l'appelle le "gestionnaire de l'événement".

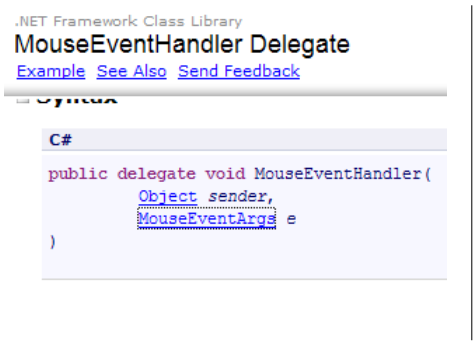
Revenons sur la signature de *EventHandler* :

```
    private delegate void EventHandler(object sender, EventArgs e);
```

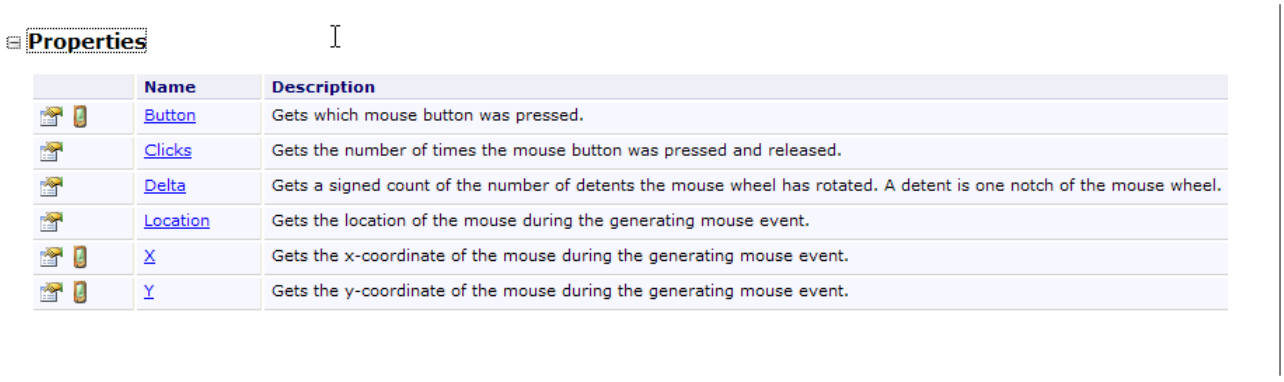
Le second paramètre du *delegate* est un objet de type *EventArgs* ou d'une classe dérivée. Le type *EventArgs* est très général et n'apporte en fait aucune information sur l'événement qui s'est produit. Pour un clic sur un bouton, c'est suffisant. Pour un déplacement de souris sur un formulaire, on aurait un événement *MouseMove* de la classe [Form] défini par :

```
public event MouseEventHandler MouseMove;
```

Le *delegate MouseEventHandler* est défini comme :



C'est une fonction déléguée (delegate) de signature *void f (object, MouseEventArgs)*. La classe *MouseEventArgs* est elle définie par :



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.

7.1.2.3 *Conclusion*

Des deux projets étudiés, nous pouvons conclure qu'une fois l'interface graphique construite avec Visual studio, le travail du développeur consiste principalement à écrire les gestionnaires des événements qu'il veut gérer pour cette interface graphique. Du code est généré automatiquement par Visual Studio. Ce code, qui peut être complexe, peut être ignoré en première approche. Ultérieurement, son étude peut permettre une meilleure compréhension de la création et de la gestion des formulaires.

7.2 Les composants de base

Nous présentons maintenant diverses applications mettant en jeu les composants les plus courants afin de découvrir les principales méthodes et propriétés de ceux-ci. Pour chaque application, nous présentons l'interface graphique et le code intéressant, principalement celui des gestionnaires d'événements.

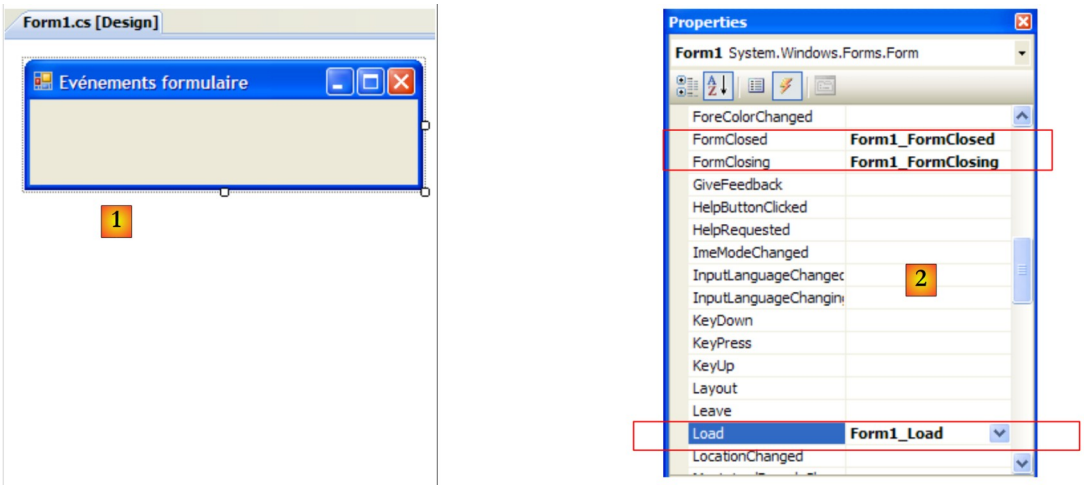
7.2.1 Formulaire Form

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 :



- [1] : le formulaire
- [2] : les trois événements traités

Le code de [Form1.cs] est le suivant :

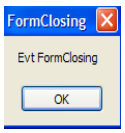
```
1.  using System;
2.  using System.Windows.Forms;
3.
4.  namespace Chap5 {
5.   public partial class Form1 : Form {
6.      public Form1() {
7.         InitializeComponent();
8.      }
9.
10.     private void Form1_Load(object sender, EventArgs e) {
11.        // chargement initial du formulaire
12.        MessageBox.Show("Evt Load", "Load");
```

```
13.     }
14.
15.     private void Form1_FormClosing(object sender, FormClosingEventArgs e) {
16.         // le formulaire est en train de se fermer
17.         MessageBox.Show("Evt FormClosing", "FormClosing");
18.         // on demande confirmation
19.         DialogResult réponse = MessageBox.Show("Voulez-vous vraiment quitter l'application",
    "Closing", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
20.         if (réponse == DialogResult.No)
21.             e.Cancel = true;
22.     }
23.
24.     private void Form1_FormClosed(object sender, FormClosedEventArgs e) {
25.         // le formulaire va être fermé
26.         MessageBox.Show("Evt FormClosed", "FormClosed");
27.     }
28.  }
29. }
```

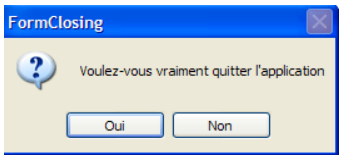
Nous utilisons la fonction *MessageBox* pour être averti des différents événements.

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

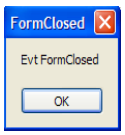
ligne 15 : L'événement *FormClosing* va se produire lorsque l'utilisateur ferme la fenêtre.



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



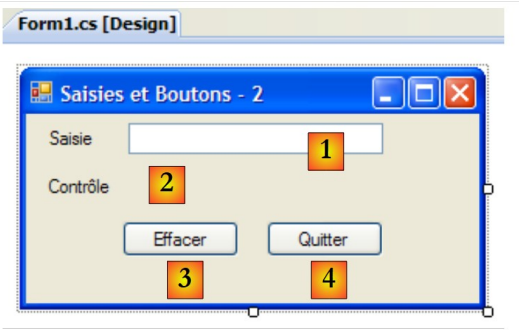
ligne 20 : S'il répond Non, nous fixons la propriété *Cancel* de l'événement *CancelEventArgs e* que la méthode a reçu en paramètre. Si nous mettons cette propriété à *False*, la fermeture de la fenêtre est abandonnée, sinon elle se poursuit. L'événement *FormClosed* va alors se produire :



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

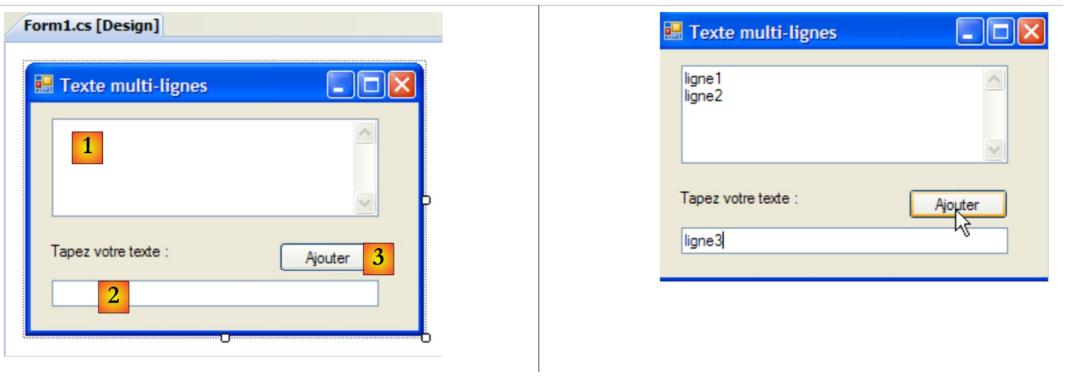
Le code de cette application est le suivant :

```
1.  using System.Windows.Forms;
2.
3.  namespace Chap5 {
4.   public partial class Form1 : Form {
5.      public Form1() {
6.         InitializeComponent();
7.      }
8.
9.      private void textBoxSaisie_TextChanged(object sender, System.EventArgs e) {
10.        // le contenu du TextBox a changé - on le copie dans le Label labelControle
11.        labelControle.Text = textBoxSaisie.Text;
12.     }
13.
14.     private void buttonEffacer_Click(object sender, System.EventArgs e) {
15.        // on efface le contenu de la boîte de saisie
16.        textBoxSaisie.Text = "";
17.     }
18.
19.     private void buttonQuitter_Click(object sender, System.EventArgs e) {
20.        // clic sur bouton Quitter - on quitte l'application
21.        Application.Exit();
22.     }
23.
24.     private void Form1_Shown(object sender, System.EventArgs e) {
25.        // on met le focus sur le champ de saisie
26.        textBoxSaisie.Focus();
27.     }
28.  }
29. }
```

- ligne 24 : l'événement *[Form].Shown* a lieu lorsque le formulaire est affiché
- ligne 26 : on met alors le focus (pour une saisie) sur le composant *textBoxSaisie*.
- ligne 9 : l'événement *[TextBox].TextChanged* se produit à chaque fois que le contenu d'un composant *TextBox* change
- ligne 11 : on recopie le contenu du composant [TextBox] dans le composant [Label]
- ligne 14 : gère le clic sur le bouton [Effacer]
- ligne 16 : on met la chaîne vide dans le composant [TextBox]
- ligne 19 : gère le clic sur le bouton [Quitter]
- ligne 21 : pour arrêter l'application en cours d'exécution. On se rappelle que l'objet *Application* sert à lancer l'application dans la méthode [Main] de [Form1.cs] :

```
1.        static void Main() {
2.           Application.EnableVisualStyles();
3.           Application.SetCompatibleTextRenderingDefault(false);
4.           Application.Run(new Form1());
5.  }
```

L'exemple suivant utilise un *TextBox* multilignes :



La liste des contrôles est la suivante :

n°	type	nom	rôle
1	TextBox	textBoxLignes	champ de saisie multilignes Multiline=true, ScrollBars=Both, AcceptReturn=True, AcceptTab=True
2	TextBox	textBoxLigne	champ de saisie monoligne
3	Button	buttonAjouter	Ajoute le contenu de 2 à 1

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

`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

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

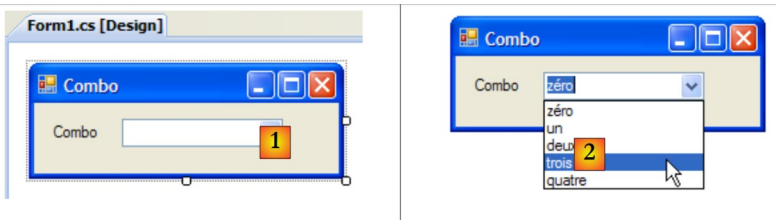
Le code de l'application est le suivant :

```
1.  using System.Windows.Forms;
2.  using System;
3.
4.  namespace Chap5 {
5.   public partial class Form1 : Form {
6.      public Form1() {
7.         InitializeComponent();
8.      }
9.
10.     private void buttonAjouter_Click(object sender, System.EventArgs e) {
11.        // ajout du contenu de textBoxLigne à celui de textBoxLignes
12.        textBoxLignes.Text += textBoxLigne.Text+Environment.NewLine;
13.        textBoxLigne.Text = "";
14.     }
15.
16.     private void Form1_Shown(object sender, EventArgs e) {
17.        // on met le focus sur le champ de saisie
18.        textBoxLigne.Focus();
19.     }
20.  }
21. }
```

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

7.2.3 Listes déroulantes ComboBox

Nous créons le formulaire suivant :



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

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 *DropDownStyle* :

`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*.

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.

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 élément 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
`C.SelectedIndex`	index de l'élément sélectionné
`C.SelectedItem`	élément sélectionné
`C.SelectedItem.Text`	texte affiché de l'élément sélectionné
`C.Text`	texte affiché de l'élément sélectionné

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

L'élément *Item* 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. Le texte sélectionné peut être obtenu de diverses façons : *C.SelectedItem.Text*, *C.Text*

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.

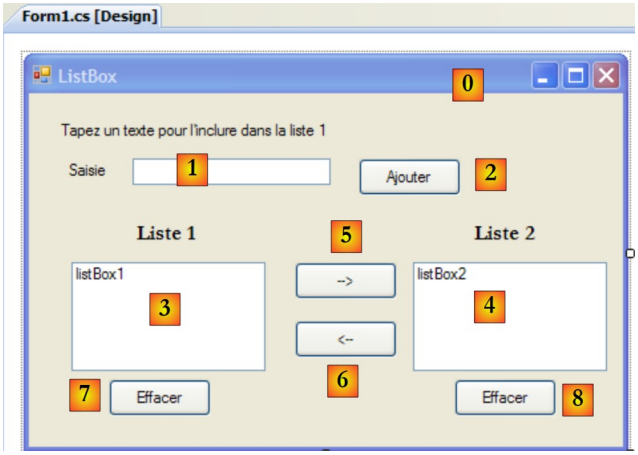
Le code de l'application est le suivant :

```
1.  using System.Windows.Forms;
2.
3.  namespace Chap5 {
4.   public partial class Form1 : Form {
5.      private int previousSelectedIndex=0;
6.
7.      public Form1() {
8.         InitializeComponent();
9.         // remplissage combo
10.        comboBoxNombres.Items.AddRange(new string[] { "zéro", "un", "deux", "trois", "quatre" });
11.        // sélection élément n° 0
12.        comboBoxNombres.SelectedIndex = 0;
13.     }
14.
15.     private void comboBoxNombres_SelectedIndexChanged(object sender, System.EventArgs e) {
16.        int newSelectedIndex = comboBoxNombres.SelectedIndex;
17.        if (newSelectedIndex != previousSelectedIndex) {
18.           // l'élément sélectionné à changé - on l'affiche
19.           MessageBox.Show(string.Format("Elément sélectionné : ({0},{1})", comboBoxNombres.Text,
    newSelectedIndex), "Combo", MessageBoxButtons.OK, MessageBoxIcon.Information);
20.           // on note le nouvel index
21.           previousSelectedIndex = newSelectedIndex;
22.        }
23.     }
24.  }
25. }
```

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

7.2.4 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 *FormBorderStyle*=FixedSingle (cadre non redimensionable)
1	TextBox	textBoxSaisie	champ de saisie
2	Button	buttonAjouter	bouton permettant d'ajouter le contenu du champ de saisie [1] dans la liste [3]
3	ListBox	listBox1	liste 1 *SelectionMode*=MultiExtended :
4	ListBox	listBox2	liste 2 *SelectionMode*=MultiSimple :
5	Button	button1vers2	transfère les éléments sélectionnés de liste 1 vers liste 2
6	Button	button2vers1	fait l'inverse
7	Button	buttonEffacer1	vide la liste 1
8	Button	buttonEffacer2	vide la liste 2

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

`One`	un seul élément peut être sélectionné
`MultiExtended`	multi-sélection possible : maintenir appuyée la touche SHIFT et cliquer sur un élément étend la sélection de l'élément précédemment sélectionné à l'élément courant.
`MultiSimple`	multi-sélection possible : un élément est sélectionné / désélectionné par un clic de souris ou par appui sur la barre d'espace.

Fonctionnement de l'application

- L'utilisateur tape du texte dans le champ 1. Il l'ajoute à la liste 1 avec le bouton *Ajouter* (2). Le champ de saisie (1) est alors vidé et l'utilisateur peut ajouter un nouvel élément.
- Il peut transférer des éléments d'une liste à l'autre en sélectionnant l'élément à transférer dans l'une des listes et en choisissant le bouton de transfert adéquat 5 ou 6. L'élément transféré est ajouté à la fin de la liste de destination et enlevé de la liste source.
- Il peut double-cliquer sur un élément de la liste 1. Cet élément est alors transféré dans la boîte de saisie pour modification et enlevé de la liste 1.

Les boutons sont allumés ou éteints selon les règles suivantes :

- le bouton *Ajouter* n'est allumé que s'il y a un texte non vide dans le champ de saisie
- le bouton [5] de transfert de la liste 1 vers la liste 2 n'est allumé que s'il y a un élément sélectionné dans la liste 1
- le bouton [6] de transfert de la liste 2 vers la liste 1 n'est allumé que s'il y a un élément sélectionné dans la liste 2
- les boutons [7] et [8] d'effacement des listes 1 et 2 ne sont allumés que si la liste à effacer contient des éléments.

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

```
1.          public Form1() {
2.             InitializeComponent();
3.             // --- initialisations complémentaires ---
4.             // on inhibe un certain nombre de boutons
5.             buttonAjouter.Enabled = false;
6.             button1vers2.Enabled = false;
7.             button2vers1.Enabled = false;
8.             buttonEffacer1.Enabled = false;
9.             buttonEffacer2.Enabled = false;
10. }
```

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 :

```
1.      private void textBoxSaisie_TextChanged(object sender, System.EventArgs e) {
2.         // le contenu de textBoxSaisie a changé
3.         // le bouton Ajouter n'est allumé que si la saisie est non vide
4.         buttonAjouter.Enabled = textBoxSaisie.Text.Trim() != "";
5.      }
6.
```

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

```
1.          private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e) {
2.             // un élément a été sélectionné
3.             // on allume le bouton de transfert 1 vers 2
4.             button1vers2.Enabled = true;
5.          }
6.
7.          private void listBox2_SelectedIndexChanged(object sender, System.EventArgs e) {
8.             // un élément a été sélectionné
9.             // on allume le bouton de transfert 2 vers 1
10.            button2vers1.Enabled = true;
11. }
```

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

```
1.          private void buttonAjouter_Click(object sender, System.EventArgs e) {
2.             // ajout d'un nouvel élément à la liste 1
3.             listBox1.Items.Add(textBoxSaisie.Text.Trim());
4.             // raz de la saisie
5.             textBoxSaisie.Text = "";
6.             // Liste 1 n'est pas vide
7.             buttonEffacer1.Enabled = true;
8.             // retour du focus sur la boîte de saisie
9.             textBoxSaisie.Focus();
10. }
```

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* :

```
1.          private void buttonEffacer1_Click(object sender, System.EventArgs e) {
2.             // on efface la liste 1
3.             listBox1.Items.Clear();
4.             // bouton Effacer
5.             buttonEffacer1.Enabled = false;
6.          }
7.
8.          private void buttonEffacer2_Click(object sender, System.EventArgs e) {
9.             // on efface la liste 2
10.            listBox2.Items.Clear();
11.            // bouton Effacer
12.            buttonEffacer2.Enabled = false;
13. }
```

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

```
1.      private void button1vers2_Click(object sender, System.EventArgs e) {
2.         // transfert de l'élément sélectionné dans Liste 1 dans Liste 2
3.         transfert(listBox1, button1vers2, buttonEffacer1, listBox2, button2vers1, buttonEffacer2);
4.      }
5.
6.      private void button2vers1_Click(object sender, System.EventArgs e) {
```

```
7.          // transfert de l'élément sélectionné dans Liste 2 dans Liste 1
8.          transfert(listBox2, button2vers1, buttonEffacer2, listBox1, button1vers2, buttonEffacer1);
9.      }
10.
```

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

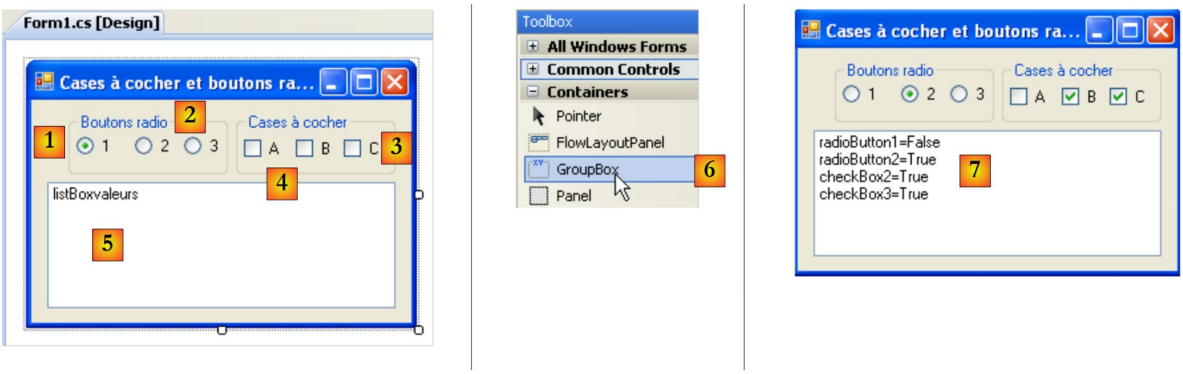
```
(a)       // transfert
(b)       private void transfert(ListBox l1, Button button1vers2, Button buttonEffacer1, ListBox l2,
    Button button2vers1, Button buttonEffacer2) {
(c)          // transfert dans la liste l2 des éléments sélectionnés de la liste l1
(d)          for (int i = l1.SelectedIndices.Count - 1; i >= 0; i--) {
(e)             // index de l'élément sélectionné
(f)             int index = l1.SelectedIndices[i];
(g)             // ajout dans l2
(h)             l2.Items.Add(l1.Items[index]);
(i)             // suppression dans l1
(j)             l1.Items.RemoveAt(index);
(k)          }
(l)          // boutons Effacer
(m)          buttonEffacer2.Enabled = l2.Items.Count != 0;
(n)          buttonEffacer1.Enabled = l1.Items.Count != 0;
(o)          // boutons de transfert
(p)          button1vers2.Enabled = false;
(q) }
```

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

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

7.2.5 Cases à cocher CheckBox, boutons radio ButtonRadio

Nous nous proposons d'écrire l'application suivante :

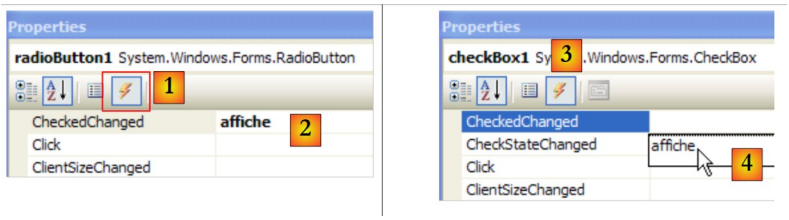


Les composants de la fenêtre sont les suivants :

n°	type	nom	rôle
1	GroupBox cf [6]	groupBox1	un conteneur de composants. On peut y déposer d'autres composants. *Text*=Boutons radio
2	RadioButton	radioButton1 radioButton2 radioButton3	3 boutons radio - *radioButton1* a la propriété *Checked*=True et la propriété *Text*=1 - *radioButton2* a la propriété *Text*=2 - *radioButton3* a la propriété *Text*=3 Des boutons radio présents dans un même conteneur, ici le *GroupBox*, sont **exclusifs** l'un de l'autre : seul l'un d'entre-eux est allumé.
3	GroupBox	groupBox2	
4	CheckBox	checkBox1 checkBox2 checkBox3	3 cases à cocher. *chechBox1* a la propriété *Checked*=True et la propriété *Text*=A - *chechBox2* a la propriété *Text*=B - *chechBox3* a la propriété *Text*=C
5	ListBox	listBoxValeurs	une liste qui affiche les valeurs des boutons radio et des cases à cocher dès qu'un changement intervient.
6			montre où trouver le conteneur *GroupBox*

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 *Checked* qui à **vrai** signifie que le contrôle est coché. Nous n'utiliserons ici qu'une seule méthode pour traiter les six événements *CheckChanged*, la méthode *affiche*. Pour faire en sorte que les six événements *CheckChanged* soient gérés par la même méthode *affiche*, on pourra procéder comme suit :

Sélectionnons le composant *radioButton1* et cliquons droit dessus pour avoir accès à ses propriétés :



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

```
1.  private void affiche(object sender, EventArgs e) {
2.          }
```

La méthode *affiche* est une méthode de type *EventHandler*.

Pour les cinq autres composants, on procède de même. Sélectionnons par exemple l'option *CheckBox1* et ses événements [3]. En face de l'événement *Click*, on a une liste déroulante [4] dans laquelle sont présentes les méthodes existantes pouvant traiter cet événement. Ici on n'a que la méthode *affiche*. On la sélectionne. On répète ce processus pour tous les autres composants.

Dans la méthode *InitializeComponent* du code a été généré. La méthode *affiche* a été déclarée comme gestionnaire des six événements *CheckedChanged* de la façon suivante :

```
1.  this.radioButton1.CheckedChanged += new System.EventHandler(this.affiche);
2.  this.radioButton2.CheckedChanged += new System.EventHandler(this.affiche);
3.  this.radioButton3.CheckedChanged += new System.EventHandler(this.affiche);
4.  this.checkBox1.CheckedChanged += new System.EventHandler(this.affiche);
5.  this.checkBox2.CheckedChanged += new System.EventHandler(this.affiche);
6.  this.checkBox3.CheckedChanged += new System.EventHandler(this.affiche);
```

La méthode *affiche* est complétée comme suit :

```
1.        private void affiche(object sender, System.EventArgs e) {
2.           // affiche l'état du bouton radio ou de la case à cocher
3.           // est-ce un checkbox ?
4.           if (sender is CheckBox) {
5.              CheckBox chk = (CheckBox)sender;
6.              listBoxvaleurs.Items.Add(chk.Name + "=" + chk.Checked);
7.           }
8.           // est-ce un radiobutton ?
9.           if (sender is RadioButton) {
10.             RadioButton rdb = (RadioButton)sender;
11.             listBoxvaleurs.Items.Add(rdb.Name + "=" + rdb.Checked);
12.          }
13. }
```

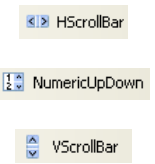
La syntaxe

```
        if (sender is CheckBox) {
```

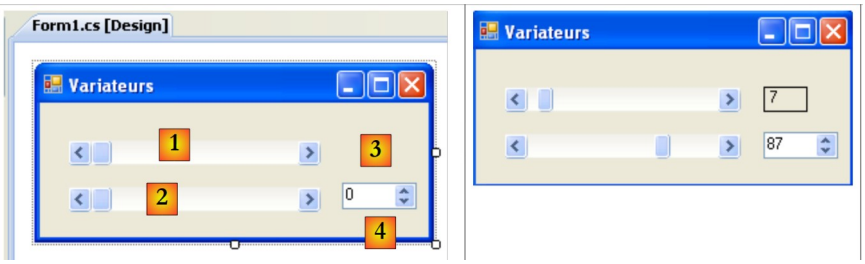
permet de vérifier que l'objet *sender* est de type *CheckBox*. Cela nous permet ensuite de faire un transtypage vers le type exact de *sender*. La méthode *affiche* écrit dans la liste *listBoxValeurs* le nom du composant à l'origine de l'événement et la valeur de sa propriété *Checked*. A l'exécution [7], 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".

7.2.6 Variateurs ScrollBar

Il existe plusieurs types de variateur : le variateur horizontal (H*ScrollBar*), le variateur vertical (V*ScrollBar*), 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	Label	labelValeurHS1	affiche la valeur du variateur horizontal
4	NumericUpDown	numericUpDown2	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 vaut 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 vaut 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**.

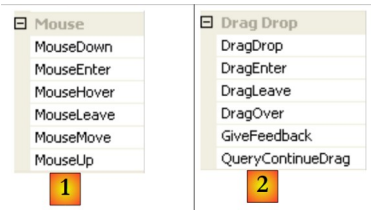
Un composant **NumericUpDown** est proche du variateur : il a lui aussi les propriétés *Minimum*, *Maximum* et *Value*, par défaut 0, 100, 0. Mais ici, la propriété *Value* est affichée dans une boîte de saisie faisant partie intégrante du contrôle. L'utilisateur peut lui même modifier cette valeur sauf si on a mis la propriété *ReadOnly* du contrôle à vrai. La valeur de l'incrément est fixée 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 de l'application est le suivant :

```
1.  using System.Windows.Forms;
2.
3.  namespace Chap5 {
4.   public partial class Form1 : Form {
5.      public Form1() {
6.         InitializeComponent();
7.         // on fixe les caractéristiques du variateur 1
8.         hScrollBar1.Value = 7;
9.         hScrollBar1.Minimum = 1;
10.        hScrollBar1.Maximum = 130;
11.        hScrollBar1.LargeChange = 11;
12.        hScrollBar1.SmallChange = 1;
13.        // on donne au variateur 2 les mêmes caractéristiques qu'au variateur 1
14.        hScrollBar2.Value = hScrollBar1.Value;
15.        hScrollBar2.Minimum = hScrollBar1.Minimum;
16.        hScrollBar2.Maximum = hScrollBar1.Maximum;
17.        hScrollBar2.LargeChange = hScrollBar1.LargeChange;
18.        hScrollBar2.SmallChange = hScrollBar1.SmallChange;
19.        // idem pour l'incrémenteur
20.        numericUpDown2.Value = hScrollBar1.Value;
21.        numericUpDown2.Minimum = hScrollBar1.Minimum;
22.        numericUpDown2.Maximum = hScrollBar1.Maximum;
23.        numericUpDown2.Increment = hScrollBar1.SmallChange;
24.
25.        // on donne au Label la valeur du variateur 1
26.        labelValeurHS1.Text = hScrollBar1.Value.ToString();
27.     }
28.
29.     private void hScrollBar1_Scroll(object sender, ScrollEventArgs e) {
30.        // changement de valeur du variateur 1
31.        // on répercute sa valeur sur le variateur 2 et sur le label
32.        hScrollBar2.Value = hScrollBar1.Value;
33.        labelValeurHS1.Text = hScrollBar1.Value.ToString();
34.     }
35.
36.     private void numericUpDown2_ValueChanged(object sender, System.EventArgs e) {
37.        // l'incrémenteur a changé de valeur
38.        // on fixe la valeur du variateur 2
39.        hScrollBar2.Value = (int)numericUpDown2.Value;
40.     }
41.  }
42. }
```

7.3 Événements souris

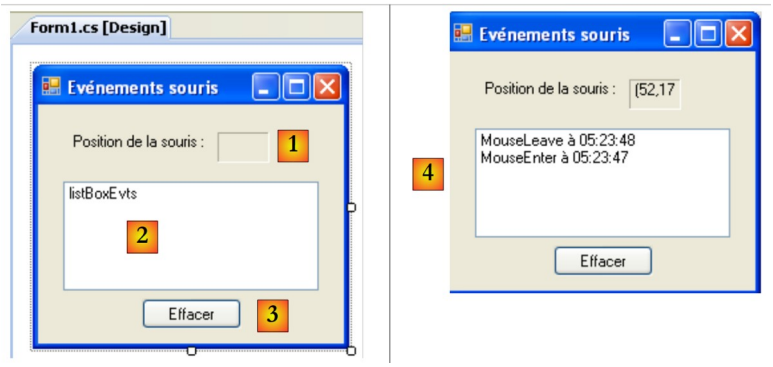
Lorsqu'on dessine dans un conteneur, il est important de connaître la position de la souris pour, par exemple, afficher un point lors d'un clic. Les déplacements de la souris provoquent des événements dans le conteneur dans lequel elle se déplace.



- [1] : les événements survenant lors d'un déplacement de la souris sur le formulaire ou sur un contrôle
- [2] : les événements survenant lors d'un glisser / lâcher (Drag'nDrop)

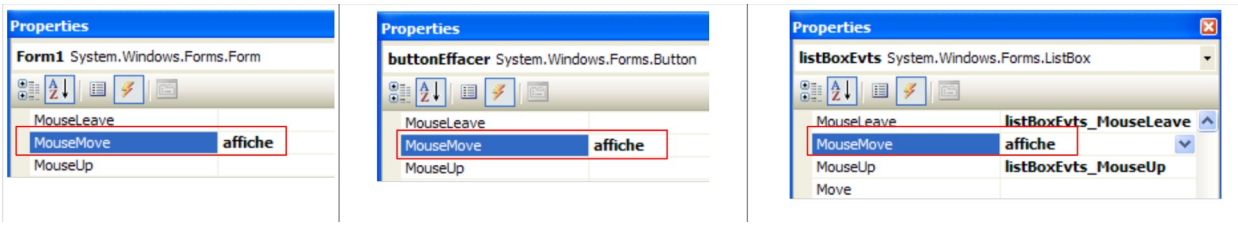
`MouseEnter`	la souris vient d'entrer dans le domaine du contrôle
`MouseLeave`	la souris vient de quitter le domaine du contrôle
`MouseMove`	la souris bouge dans le domaine du contrôle
`MouseDown`	Pression sur le bouton gauche de la souris
`MouseUp`	Relâchement du bouton gauche de la souris
`DragDrop`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

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



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

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

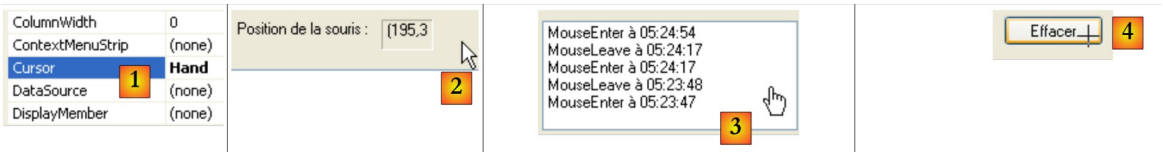


Le code de la procédure *affiche* est le suivant :

```
1.          private void affiche(object sender, MouseEventArgs e) {
2.              // mvt souris - on affiche les coordonnées (X,Y) de celle-ci
```

```
3.            labelPositionSouris.Text = "(" + e.X + "," + e.Y + ")";
4.    }
```

A chaque fois que la souris entre dans le domaine d'un contrôle son système de coordonnées change. Son origine (0,0) est le coin supérieur gauche du contrôle sur lequel elle se trouve. Ainsi à l'exécution, lorsqu'on passe la souris du formulaire au bouton, on voit clairement le changement de coordonnées. Afin de mieux voir ces changements de domaine de la souris, on peut utiliser la propriété *Cursor* [1] 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 [2], **Hand** pour la liste 2 [3] et à **Cross** pour le bouton 3 [4].

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 :

```
1.        private void listBoxEvts_MouseEnter(object sender, System.EventArgs e) {
2.            // on signale l'évt
3.            listBoxEvts.Items.Insert(0, string.Format("MouseEnter à {0:hh:mm:ss}",DateTime.Now));
4.        }
5.
6.        private void listBoxEvts_MouseLeave(object sender, EventArgs e) {
7.            // on signale l'évt
8.            listBoxEvts.Items.Insert(0, string.Format("MouseLeave à {0:hh:mm:ss}", DateTime.Now));
9.    }
```

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

```
1.        private void listBoxEvts_MouseDown(object sender, MouseEventArgs e) {
2.            // on signale l'évt
3.            listBoxEvts.Items.Insert(0, string.Format("MouseDown à {0:hh:mm:ss}", DateTime.Now));
4.        }
5.
6.        private void listBoxEvts_MouseUp(object sender, MouseEventArgs e) {
7.            // on signale l'évt
8.            listBoxEvts.Items.Insert(0, string.Format("MouseUp à {0:hh:mm:ss}", DateTime.Now));
9.    }
```

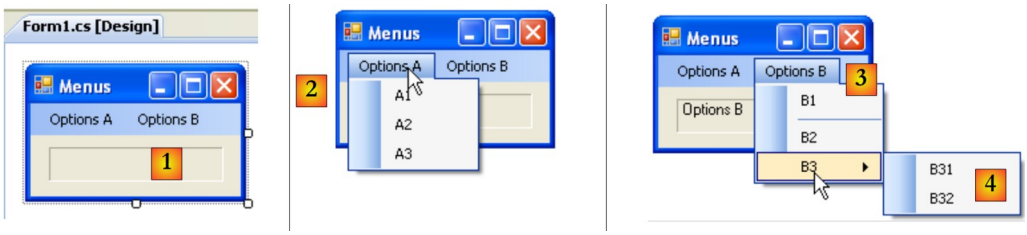
- lignes 3 et 8 : les messages sont placés en 1ère position dans le *ListBox* afin que les événements les plus récents soient les premiers dans la liste.

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

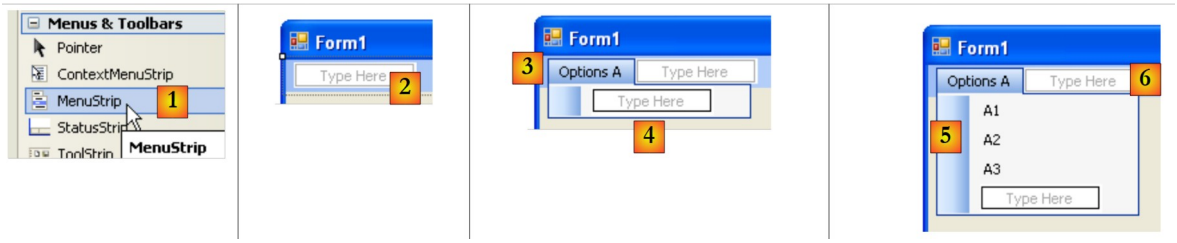
```
1.        private void buttonEffacer_Click(object sender, EventArgs e) {
2.            listBoxEvts.Items.Clear();
3.    }
```

7.4 Créer une fenêtre avec menu

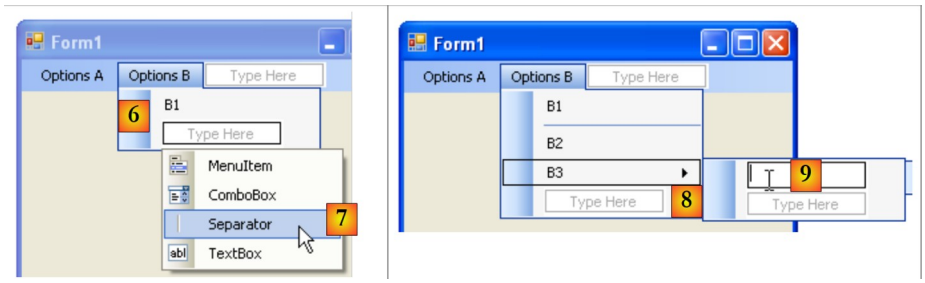
Voyons maintenant comment créer une fenêtre avec menu. Nous allons créer la fenêtre suivante :



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

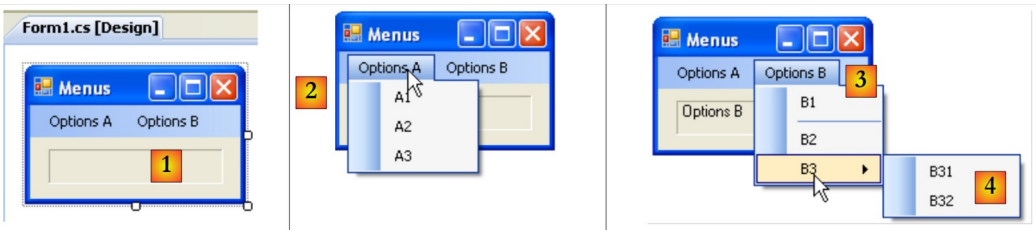


- [1] : choix du composant [MenuStrip]
- [2] : on a alors un menu qui s'installe sur le formulaire avec des cases vides intitulées "Type Here". Il suffit d'y indiquer les différentes options du menu.
- [3] : le libellé "Options A" a été tapé. On passe au libellé [4].
- [5] : les libellés des options A ont été saisis. On passe au libellé [6]



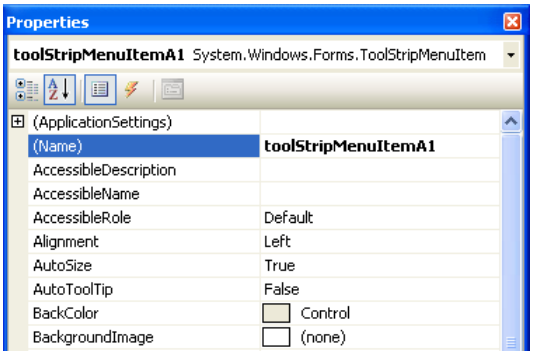
- [6] : les premières options B
- [7] : sous B1, on met un séparateur. Celui-ci est disponible dans un combo associé au texte "Type Here"
- [8] : pour faire un sous-menu, utiliser la flèche [8] et taper le sous-menu dans [9]

Il reste à nommer le différents composants du formulaire :



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

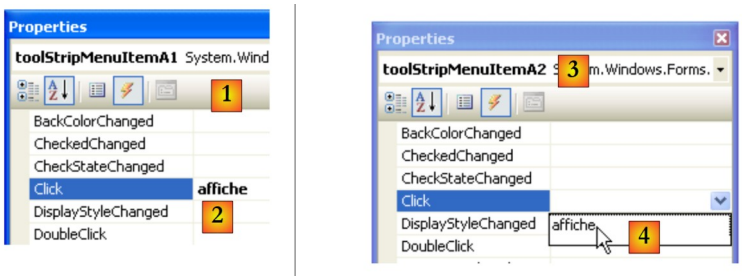
Les options de menu sont des contrôles comme les autres composants visuels et ont des propriétés et événements. Par exemple les propriétés de l'option de menu *A1* sont les suivantes :



Deux propriétés sont utilisées dans notre exemple :

`Name`	le nom du contrôle menu
`Text`	le libellé de l'option de menu

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



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

```
3.  private void affiche(object sender, EventArgs e) {
4.          }
```

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

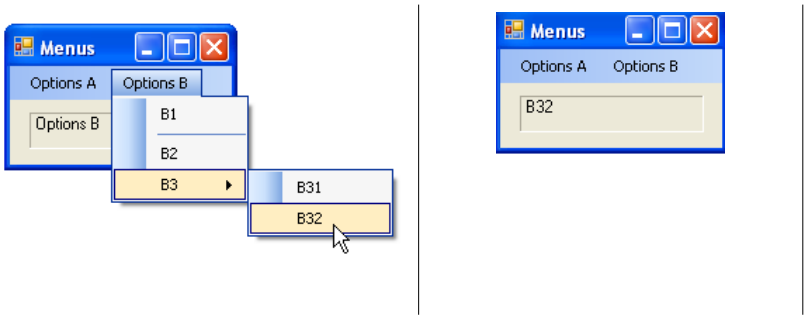
```
1.  private void affiche(object sender, EventArgs e) {
2.          // affiche dans le TextBox le nom du sous-menu choisi
3.          labelStatut.Text = ((ToolStripMenuItem)sender).Text;
```

```
4.  }
```

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

Pour toutes les options de menu, on fixe le gestionnaire du clic à la méthode *affiche* [3,4].

Exécutons l'application et sélectionnons un élément de menu :

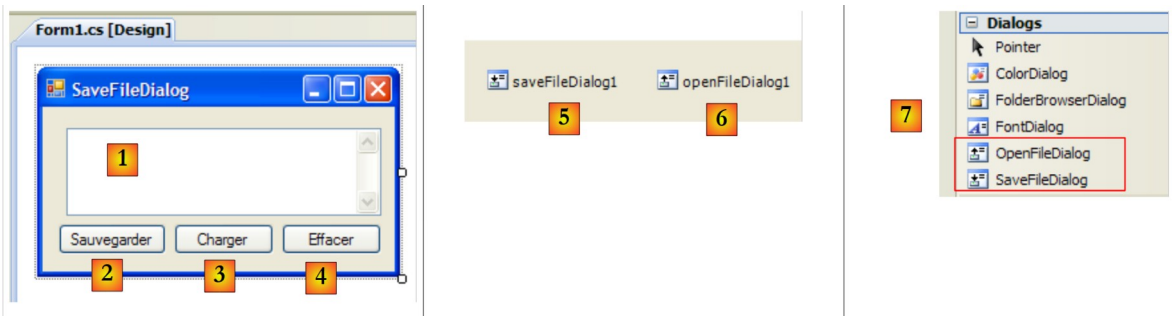


7.5 Composants non visuels

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.

7.5.1 Boîtes de dialogue OpenFileDialog et SaveFileDialog

Nous allons construire l'application suivante :



Les contrôles sont les suivants :

N°	type	nom	rôle
1	TextBox	TextBoxLignes	texte tapé par l'utilisateur ou chargé à partir d'un fichier *MultiLine*=True, *ScrollBars*=Both, *AccepReturn*=True, *AcceptTab*=True
2	Button	buttonSauvegarder	permet de sauvegarder le texte de [1] dans un fichier texte
3	Button	buttonCharger	permet de charger le contenu d'un fichier texte dans [1]
4	Button	buttonEffacer	efface le contenu de [1]
5	SaveFileDialog	saveFileDialog1	composant permettant de choisir le nom et l'emplacement du fichier de sauvegarde de [1]. Ce composant est pris dans la barre d'outils [7] et simplement déposé sur le formulaire. Il est alors enregistré mais il n'occupe pas de place sur le formulaire. C'est un composant non visuel.
6	OpenFileDialog	openFileDialog1	composant permettant de choisir le fichier à charger dans [1].

Le code associé au bouton *Effacer* est simple :

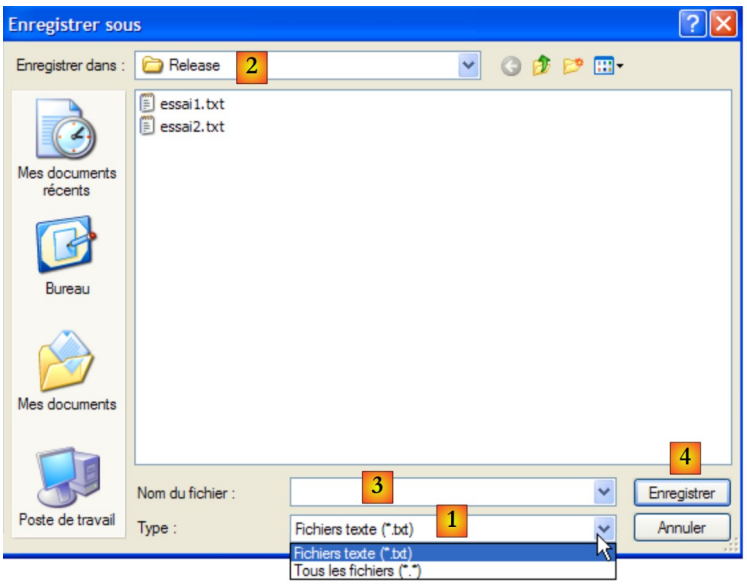
```
1.        private void buttonEffacer_Click(object sender, EventArgs e) {
```

```
2.          // on met la chaîne vide dans le TexBox
3.          textBoxLignes.Text = "";
4.  }
```

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

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

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



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

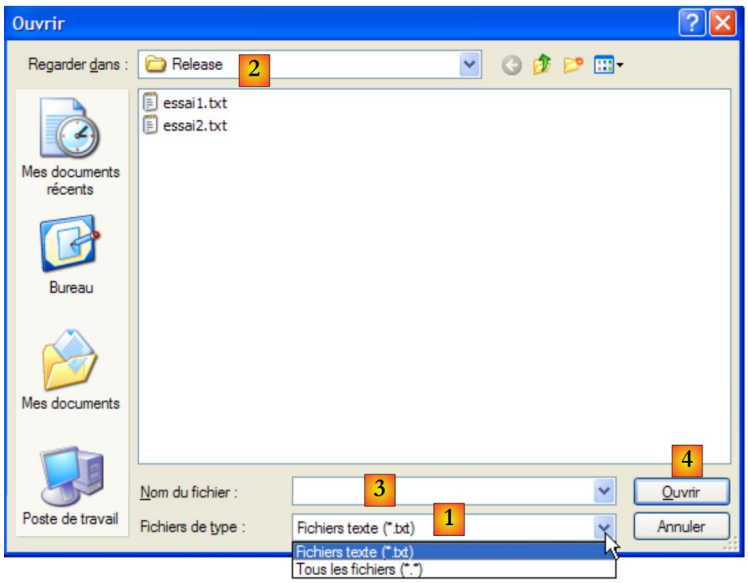
La procédure de sauvegarde peut s'écrire ainsi :

```
1.  private void buttonSauvegarder_Click(object sender, System.EventArgs e) {
2.          // on sauvegarde la boîte de saisie dans un fichier texte
3.          // on paramètre la boîte de dialogue savefileDialog1
4.          saveFileDialog1.InitialDirectory = Application.ExecutablePath;
5.          saveFileDialog1.Filter = "Fichiers texte (*.txt)|*.txt|Tous les fichiers (*.*)|*.*";
6.          saveFileDialog1.FilterIndex = 0;
7.          // on affiche la boîte de dialogue et on récupère son résultat
8.          if (saveFileDialog1.ShowDialog() == DialogResult.OK) {
9.             // on récupère le nom du fichier
10.            string nomFichier = saveFileDialog1.FileName;
11.            StreamWriter fichier = null;
12.            try {
13.               // on ouvre le fichier en écriture
14.               fichier = new StreamWriter(nomFichier);
15.               // on écrit le texte dedans
16.               fichier.Write(textBoxLignes.Text);
```

```
17.             } catch (Exception ex) {
18.                // problème
19.                MessageBox.Show("Problème à l'écriture du fichier (" +
20.                ex.Message + ")", "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Error);
21.                return;
22.             } finally {
23.                // on ferme le fichier
24.                if (fichier != null) {
25.                   fichier.Dispose();
26.                }
27.             }
28.          }
29.       }
```

- ligne 4 : on fixe le dossier initial (*InitialDirectory*) au dossier (*Application.ExecutablePath*) qui contient l'exécutable de l'application.
- ligne 5 : on fixe les types de fichiers à présenter. On notera la syntaxe des filtres : *filtre1*|*filtre2*|..|*filtren* avec *filtrei*= *Texte*|*modèle de fichier*. Ici l'utilisateur aura le choix entre les fichiers **.txt* et **.**.
- ligne 6 : on fixe le type de fichier à présenter en premier à l'utilisateur. Ici l'index 0 désigne les fichiers *.txt.
- ligne 8 : la boîte de dialogue est affichée et son résultat récupéré. Pendant que la boîte de dialogue est affichée, l'utilisateur n'a plus accès au formulaire principal (boîte de dialogue dite modale). L'utilisateur fixe le nom du fichier à sauvegarder et quitte la boîte soit par le bouton *Enregistrer*, soit par le bouton *Annuler,* soit en fermant la boîte. Le résultat de la méthode *ShowDialog* est *DialogResult.OK* uniquement si l'utilisateur a utilisé le bouton *Enregistrer* pour quitter la boîte de dialogue.
- Ceci fait, le nom du fichier à créer est maintenant dans la propriété *FileName* de l'objet *saveFileDialog1*. On est alors ramené à la création classique d'un fichier texte. On y écrit le contenu du *TextBox* : *textBoxLignes.Text* tout en gérant les exceptions qui peuvent se produire.

La classe **OpenFileDialog** est très proche de la classe *SaveFileDialog*. On utilisera les mêmes méthodes et propriétés que précédemment. La méthode *ShowDialog* affiche une boîte de dialogue analogue à la 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 **Ouvrir/Annuler**. Si le bouton *Ouvrir* est utilisé, la fonction *ShowDialog* rend le résultat **DialogResult.OK**

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

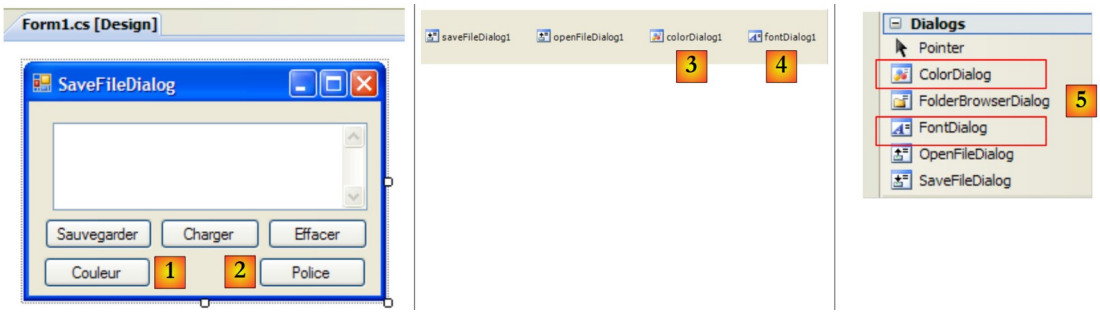
```
1.  private void buttonCharger_Click(object sender, EventArgs e) {
2.          // on charge un fichier texte dans la boîte de saisie
3.          // on paramètre la boîte de dialogue openfileDialog1
4.          openFileDialog1.InitialDirectory = Application.ExecutablePath;
5.          openFileDialog1.Filter = "Fichiers texte (*.txt)|*.txt|Tous les fichiers (*.*)|*.*";
6.          openFileDialog1.FilterIndex = 0;
7.          // on affiche la boîte de dialogue et on récupère son résultat
```

```
8.          if (openFileDialog1.ShowDialog() == DialogResult.OK) {
9.             // on récupère le nom du fichier
10.            string nomFichier = openFileDialog1.FileName;
11.            StreamReader fichier = null;
12.            try {
13.               // on ouvre le fichier en lecture
14.               fichier = new StreamReader(nomFichier);
15.               // on lit tout le fichier et on le met dans le TextBox
16.               textBoxLignes.Text = fichier.ReadToEnd();
17.            } catch (Exception ex) {
18.               // problème
19.               MessageBox.Show("Problème à la lecture du fichier (" +
20.               ex.Message + ")", "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Error);
21.               return;
22.            } finally {
23.               // on ferme le fichier
24.               if (fichier != null) {
25.                  fichier.Dispose();
26.               }
27.            }//finally
28.         }//if
29.      }
```

- ligne 4 : on fixe le dossier initial (*InitialDirectory*) au dossier (*Application.ExecutablePath*) qui contient l'exécutable de l'application.
- ligne 5 : on fixe les types de fichiers à présenter. On notera la syntaxe des filtres : *filtre1*|*filtre2*|..|*filtren* avec *filtrei*= *Texte*|*modèle de fichier*. Ici l'utilisateur aura le choix entre les fichiers **.txt* et **.**.
- ligne 6 : on fixe le type de fichier à présenter en premier à l'utilisateur. Ici l'index 0 désigne les fichiers *.txt.
- ligne 8 : la boîte de dialogue est affichée et son résultat récupéré. Pendant que la boîte de dialogue est affichée, l'utilisateur n'a plus accès au formulaire principal (boîte de dialogue dite modale). L'utilisateur fixe le nom du fichier à sauvegarder et quitte la boîte soit par le bouton *Ouvrir*, soit par le bouton *Annuler*, soit en fermant la boîte. Le résultat de la méthode *ShowDialog* est *DialogResult.OK* uniquement si l'utilisateur a utilisé le bouton *Enregistrer* pour quitter la boîte de dialogue.
- Ceci fait, le nom du fichier à 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, ligne 16, la méthode qui permet de lire la totalité d'un fichier.

7.5.2 Boîtes de dialogue FontColor et ColorDialog

Nous continuons l'exemple précédent en y ajoutant deux nouveaux boutons et deux nouveaux contrôles non visuels :

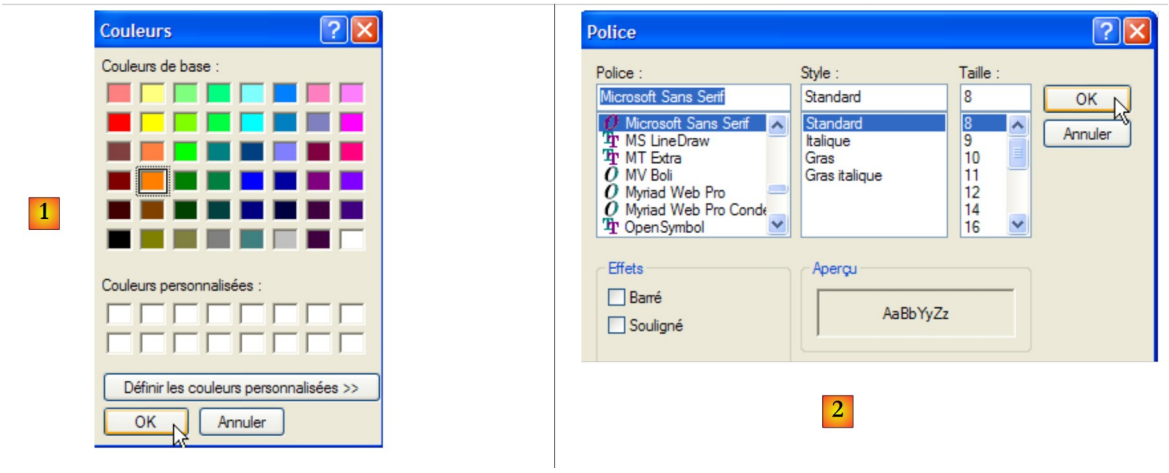


N°	type	nom	rôle
1	Button	buttonCouleur	pour fixer la couleur des caractères du TextBox
2	Button	buttonPolice	pour fixer la police de caractères du TextBox
3	ColorDialog	colorDialog1	le composant qui permet la sélection d'une couleur - pris dans la boîte à outils [5].
4	FontDialog	colorDialog1	le composant qui permet la sélection d'une police de caractères - pris dans la boîte à outils [5].

Les classes *FontDialog* et *ColorDialog* ont une méthode *ShowDialog* analogue à la méthode *ShowDialog* des classes *OpenFileDialog* et *SaveFileDialog*.

6

7



La méthode *ShowDialog* de la classe *ColorDialog* permet de choisir une couleur [1]. Celle de la classe *FontDialog* permet de choisir une police de caractères [2] :

- [1] : 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.
- [2] : 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.

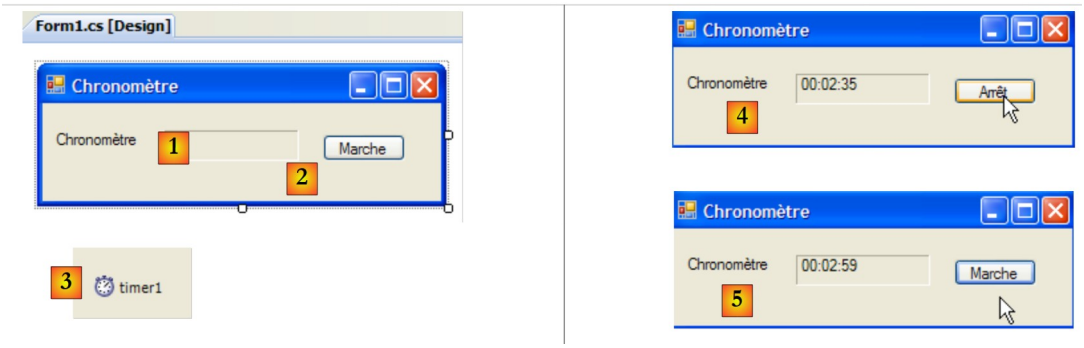
Nous avons désormais les éléments pour traiter les clics sur les boutons *Couleur* et *Police* :

```
1.        private void buttonCouleur_Click(object sender, EventArgs e) {// choix d'une couleur de
    texte
2.           if (colorDialog1.ShowDialog() == DialogResult.OK) {
3.              // on change la propriété Forecolor du TextBox
4.              textBoxLignes.ForeColor = colorDialog1.Color;
5.           }//if
6.        }
7.
8.        private void buttonPolice_Click(object sender, EventArgs e) {
9.           // choix d'une police de caractères
10.          if (fontDialog1.ShowDialog() == DialogResult.OK) {
11.             // on change la propriété Font du TextBox
12.             textBoxLignes.Font = fontDialog1.Font;
13. }
```

- ligne [4] : la propriété [ForeColor] d'un composant *TextBox* désigne la couleur de type [Color] des caractères du *TextBox*. Ici cette couleur est celle choisie par l'utilisateur dans la boîte de dialogue de type [ColorDialog].
- ligne [12] : la propriété [Font] d'un composant *TextBox* désigne la police de caractères de type [Font] des caractères du *TextBox*. Ici cette police est celle choisie par l'utilisateur dans la boîte de dialogue de type [FontDialog].

7.5.3 Timer

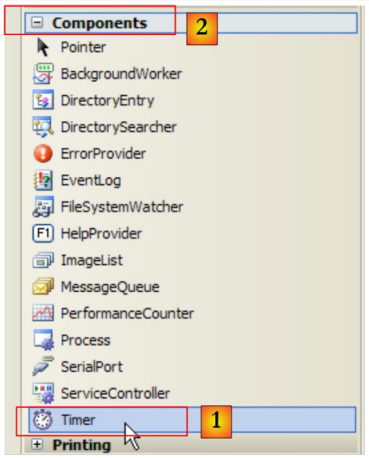
Nous nous proposons ici d'écrire l'application suivante :



n°	Type	Nom	Rôle
1	Label	labelChrono	affiche un chronomètre
2	Button	buttonArretMarche	bouton Arrêt/Marche du chronomètre
3	Timer	timer1	composant émettant ici un événement toutes les secondes

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

Pour changer toutes les secondes le contenu du Label *LabelChrono*, il nous faut un composant qui génère un événement toutes les secondes, événement qu'on pourra intercepter pour mettre à jour l'affichage du chronomètre. Ce composant c'est le *Timer* [1] disponible dans la boîte à outils *Components* [2] :



Les propriétés du composant Timer utilisées ici seront les suivantes :

`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 *buttonArretMarche_Click* suivante :

```
1.  using System;
2.  using System.Windows.Forms;
3.
4.  namespace Chap5 {
5.   public partial class Form1 : Form {
6.      public Form1() {
7.         InitializeComponent();
8.      }
9.
10.     // variable d'instance
11.     private DateTime début = DateTime.Now;
12. ...
```

```
13.     private void buttonArretMarche_Click(object sender, EventArgs e) {
14.        // arrêt ou marche ?
15.        if (buttonArretMarche.Text == "Marche") {
16.           // on note l'heure de début
17.           début = DateTime.Now;
18.           // on l'affiche
19.           labelChrono.Text = "00:00:00";
20.           // on lance le timer
21.           timer1.Enabled = true;
22.           // on change le libellé du bouton
23.           buttonArretMarche.Text = "Arrêt";
24.           // fin
25.           return;
26.        }//
27.        if (buttonArretMarche.Text == "Arrêt") {
28.           // arrêt du timer
29.           timer1.Enabled = false;
30.           // on change le libellé du bouton
31.           buttonArretMarche.Text = "Marche";
32.           // fin
33.           return;
34.        }
35.     }
36.
37.  }
38. }
```

- ligne 13 : la procédure qui traite le clic sur le bouton Arrêt/Marche.
- ligne 15 : le libellé du bouton Arrêt/Marche est soit "Arrêt" soit "Marche". On est donc obligé de faire un test sur ce libellé pour savoir quoi faire.
- ligne 17 : dans le cas de "Marche", on note l'heure de début dans une variable *début* qui est une variable globale (ligne 11) de l'objet formulaire
- ligne 19 : initialise le contenu du label *LabelChrono*
- ligne 21 : le timer est lancé (Enabled=true)
- ligne 23 : libellé du bouton passe à "Arrêt".
- ligne 27 : dans le cas de "Arrêt"
- ligne 29 : on arrête le timer (Enabled=false)
- ligne 31 : on passe le libellé du bouton à "Marche".

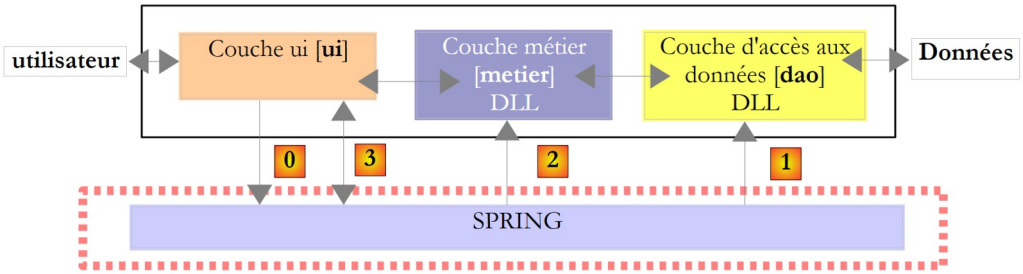
Il nous reste à traiter l'événement *Tick* sur l'objet *timer1*, événement qui se produit toutes les secondes :

```
1.  private void timer1_Tick(object sender, EventArgs e) {
2.           // une seconde s'est écoulée
3.           DateTime maintenant = DateTime.Now;
4.           TimeSpan durée = maintenant - début;
5.           // on met à jour le chronomètre
6.           labelChrono.Text = durée.Hours.ToString("d2") + ":" + durée.Minutes.ToString("d2") + ":" +
    durée.Seconds.ToString("d2");
7.        }
```

- ligne 3 : on note l'heure du moment
- ligne 4 : on calcule le temps écoulé depuis l'heure de lancement du chronomètre. On obtient un objet de type *TimeSpan* qui représente une durée dans le temps.
- ligne 6 : celle-ci doit être affichée dans le chronomètre sous la forme *hh:mm:ss*. Pour cela nous utilisons les propriétés *Hours*, *Minutes*, *Seconds* de l'objet *TimeSPan* qui représentent respectivement les heures, minutes, secondes de la durée que nous affichons au format *ToString("d2")* pour avoir un affichage sur 2 chiffres.

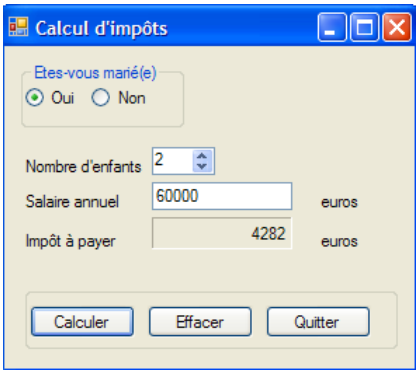
7.6 Application exemple - version 6

On reprend l'application exemple IMPOTS. La dernière version a été étudiée au paragraphe 6.4, page 140. C'était l'application à trois couches suivante :



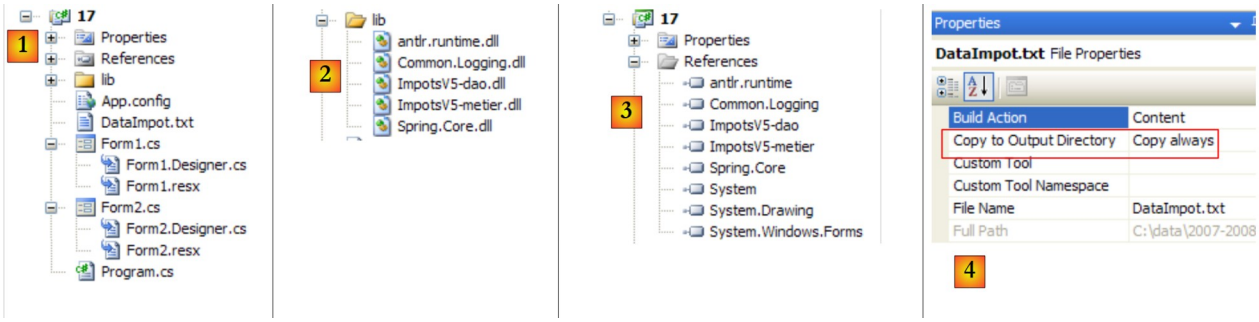
- les couches [metier] et [dao] étaient encapsulées dans des DLL
- la couche [ui] était une couche [console]
- l'instanciation des couches et leur intégration dans l'application étaient assurées par Spring.

Dans cette nouvelle version, la couche [ui] sera assurée par l'interface graphique suivante :



7.6.1 La solution Visual Studio

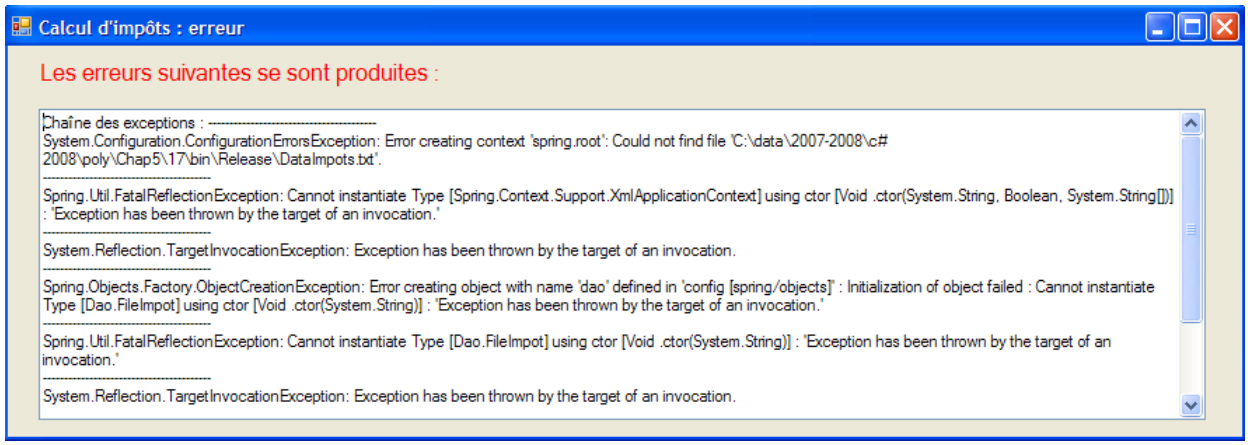
La solution Visual Studio est composée des éléments suivants :



- [1] : le projet est constitué des éléments suivants :
  - [Program.cs] : la classe qui lance l'application
  - [Form1.cs] : la classe d'un 1er formulaire
  - [Form2] : la classe d'un 2ième formulaire
  - [lib] détaillé dans [2] : on y a mis toutes les DLL nécessaires au projet :
    - [ImpotsV5-dao.dll] : la DLL de la couche [dao] générée au paragraphe 6.4.3, page 153
    - [ImpotsV5-metier.dll] : la DLL de la couche [dao] générée au paragraphe 6.4.4, page 157
    - [Spring.Core.dll], [Common.Logging.dll], [antlr.runtime.dll] : les DLL de Spring déjà utilisées dans la version précédente (cf paragraphe 6.4.6, page 158).

- [references] détaillé dans [3] : les références du projet. On a ajouté une référence pour chacune des DLL du dossier [lib]
- [App.config] : le fichier de configuration du projet. Il est identique à celui de la version précédente décrit au paragraphe 6.4.6, page 163.
- [DataImpot.txt] : le fichier des tranches d'impôt configuré pour être recopié automatiquement dans le dossier d'exécution du projet [4]

Le formulaire [Form1] est le formulaire de saisies des paramètres du calcul de l'impôt [A] déjà présenté plus haut. Le formulaire [Form2] [B] sert à afficher un message d'erreur :



7.6.2 La classe [Program.cs]

La classe [Program.cs] lance l'application. Son code est le suivant :

```
1.  using System;
2.  using System.Windows.Forms;
3.  using Spring.Context;
4.  using Spring.Context.Support;
5.  using Metier;
6.  using System.Text;
7.
8.  namespace Chap5 {
9.   static class Program {
10.     /// <summary>
11.     /// The main entry point for the application.
12.     /// </summary>
13.     [STAThread]
14.     static void Main() {
15.        // code généré par Vs
16.        Application.EnableVisualStyles();
17.        Application.SetCompatibleTextRenderingDefault(false);
18.
19.        // --------------- Code développeur
20.        // instanciations couches [metier] et [dao]
21.        IApplicationContext ctx = null;
22.        Exception ex = null;
23.        IImpotMetier metier = null;
24.        try {
25.           // contexte Spring
26.           ctx = ContextRegistry.GetContext();
27.           // on demande une référence sur la couche [metier]
28.           metier = (IImpotMetier)ctx.GetObject("metier");
29.        } catch (Exception e1) {
30.           // mémorisation exception
31.           ex = e1;
32.        }
33.        // formulaire à afficher
34.        Form form = null;
35.        // y-a-t-il eu une exception ?
36.        if (ex != null) {
37.           // oui - on crée le message d'erreur à afficher
```

```
38.           StringBuilder msgErreur = new StringBuilder(String.Format("Chaîne des exceptions : {0}
    {1}", "".PadLeft(40, '-'), Environment.NewLine));
39.           Exception e = ex;
40.           while (e != null) {
41.              msgErreur.Append(String.Format("{0}: {1}{2}", e.GetType().FullName, e.Message,
    Environment.NewLine));
42.              msgErreur.Append(String.Format("{0}{1}", "".PadLeft(40, '-'), Environment.NewLine));
43.              e = e.InnerException;
44.           }
45.           // création fenêtre d'erreur à laquelle on passe le message d'erreur à afficher
46.           Form2 form2 = new Form2();
47.           form2.MsgErreur = msgErreur.ToString();
48.           // ce sera la fenêtre à afficher
49.           form = form2;
50.        } else {
51.           // tout s'est bien passé
52.           // création interface graphique [Form1] à laquelle on passe la référence sur la couche
    [metier]
53.           Form1 form1 = new Form1();
54.           form1.Metier = metier;
55.           // ce sera la fenêtre à afficher
56.           form = form1;
57.        }
58.        // affichage fenêtre
59.        Application.Run(form);
60.     }
61.  }
62. }
```

Le code généré par Visual Studio a été complété à partir de la ligne 19. L'application exploite le fichier [App.config] suivant :

```
(a) <?xml version="1.0" encoding="utf-8" ?>
(b) <configuration>
(c)
(d)  <configSections>
(e)     <sectionGroup name="spring">
(f)        <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
(g)        <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
(h)     </sectionGroup>
(i)  </configSections>
(j)
(k)  <spring>
(l)     <context>
(m)        <resource uri="config://spring/objects" />
(n)     </context>
(o)     <objects xmlns="http://www.springframework.net">
(p)        <object name="dao" type="Dao.FileImpot, ImpotsV5-dao">
(q)           <constructor-arg index="0" value="DataImpot.txt"/>
(r)        </object>
(s)        <object name="metier" type="Metier.ImpotMetier, ImpotsV5-metier">
(t)           <constructor-arg index="0" ref="dao"/>
(u)        </object>
(v)     </objects>
(w)  </spring>
(x) </configuration>
```

- lignes 24-32 : exploitation du fichier [App.config] précédent pour instancier les couches [metier] et [dao]
- ligne 26 : exploitation du fichier [App.config]
- ligne 28 : récupération d'une référence sur la couche [metier]
- ligne 31 : mémorisation de l'éventuelle exception
- ligne 34 : la référence *form* désignera le formulaire à afficher (*form1* ou *form2*)
- lignes 36-50 : s'il y a eu exception, on se prépare à afficher un formulaire de type [Form2]
- lignes 38-44 : on construit le message d'erreur à afficher. Il est constitué de la concaténation des messages d'erreurs des différentes exceptions présentes dans la chaîne des exceptions.
- ligne 46 : un formulaire de type [Form2] est créé.
- ligne 47 : nous le verrons ultérieurement, ce formulaire a une propriété publique *MsgErreur* qui est le message d'erreur à afficher :

```
    public string MsgErreur { private get; set; }
```

  On renseigne cette propriété.
- ligne 49 : la référence *form* qui désigne la fenêtre à afficher est initialisée. On notera le polymorphisme à l'oeuvre. *form2* n'est pas de type [Form] mais de type [Form2], un type dérivé de [Form].

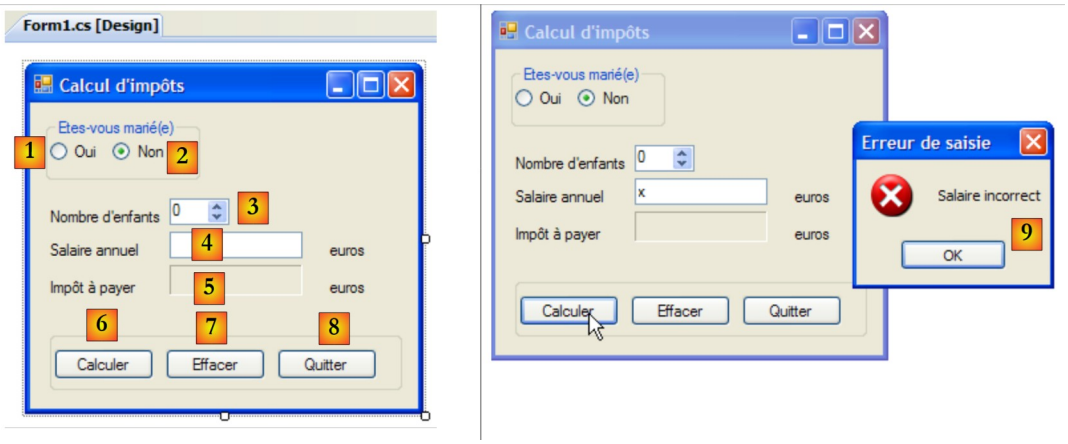
- lignes 50-57 : il n'y a pas eu d'exception. On se prépare à afficher un formulaire de type [Form1].
- ligne 53 : un formulaire de type [Form1] est créé.
- ligne 54 : nous le verrons ultérieurement, ce formulaire a une propriété publique *Metier* qui est une référence sur la couche [metier] :

```
            public IImpotMetier Metier { private get; set; }
```

  On renseigne cette propriété.
- ligne 56 : la référence *form* qui désigne la fenêtre à afficher est initialisée. On notera de nouveau le polymorphisme à l'oeuvre. *form1* n'est pas de type [Form] mais de type [Form1], un type dérivé de [Form].
- ligne 59 : la fenêtre référencée par *form* est affichée.

7.6.3 Le formulaire [Form1]

En mode [conception] le formulaire [Form1] est le suivant :



Les contrôles sont les suivants

n°	type	nom	rôle
0	GroupBox	groupBox1	*Text*=Etes-vous marié(e)
1	RadioButton	radioButtonOui	coché si marié
2	RadioButton	radioButtonNon	coché si pas marié *Checked*=True
3	NumericUpDown	numericUpDownEnfants	nombre d'enfants du contribuable *Minimum*=0, *Maximum*=20, *Increment*=1
4	TextBox	textSalaire	salaire annuel du contribuable en euros
5	Label	labelImpot	montant de l'impôt à payer *BorderStyle*=Fixed3D
6	Button	buttonCalculer	lance le calcul de l'impôt
7	Button	buttonEffacer	remet le formulaire dans l'état qu'il avait lors du chargement
8	Button	buttonQuitter	pour quitter l'application

Règles de fonctionnement du formulaire

- 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 [9]

Le code de la classe est le suivant :

```
1.  using System.Windows.Forms;
2.  using Metier;
3.  using System;
4.
5.  namespace Chap5 {
6.   public partial class Form1 : Form {
7.      // couche [métier]
```

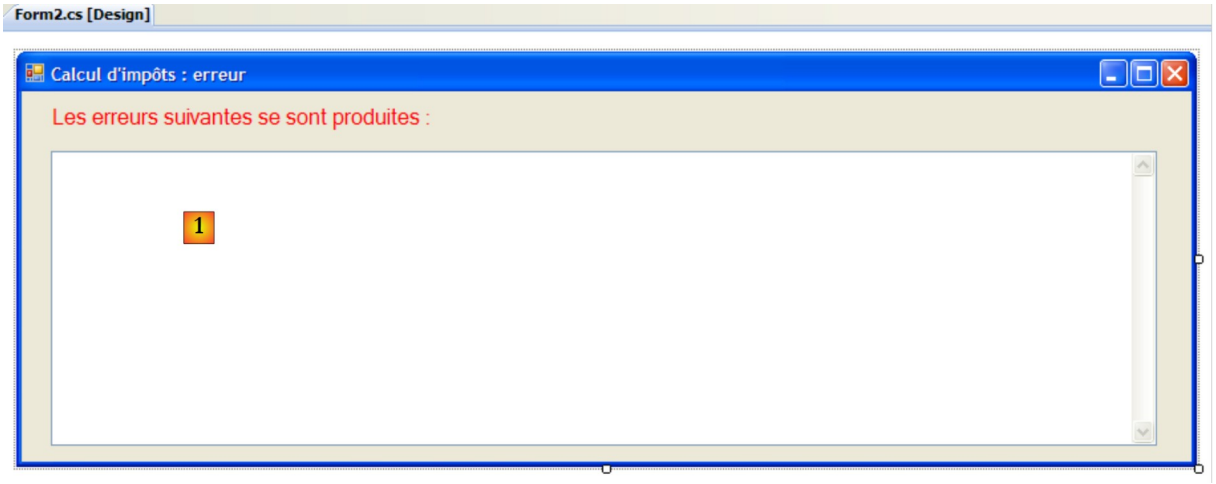
```
8.      public IImpotMetier Metier { private get; set; }
9.
10.     public Form1() {
11.        InitializeComponent();
12.     }
13.
14.     private void buttonCalculer_Click(object sender, System.EventArgs e) {
15.        // le salaire est-il correct
16.        int salaire;
17.        bool ok=int.TryParse(textSalaire.Text.Trim(), out salaire);
18.        if (! ok  || salaire < 0) {
19.           // msg d'erreur
20.           MessageBox.Show("Salaire incorrect", "Erreur de saisie", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
21.           // retour sur le champ erroné
22.           textSalaire.Focus();
23.           // sélection du texte du champ de saisie
24.           textSalaire.SelectAll();
25.           // retour à l'interface de saisie
26.           return;
27.        }
28.        // le salaire est correct - on peut calculer l'impôt
29.        labelImpot.Text = Metier.CalculerImpot(radioButtonOui.Checked,
    (int)numericUpDownEnfants.Value, salaire).ToString();
30.     }
31.
32.     private void buttonQuitter_Click(object sender, System.EventArgs e) {
33.        Environment.Exit(0);
34.     }
35.
36.     private void buttonEffacer_Click(object sender, System.EventArgs e) {
37.        // raz formulaire
38.        labelImpot.Text = "";
39.        numericUpDownEnfants.Value = 0;
40.        textSalaire.Text = "";
41.        radioButtonNon.Checked = true;
42.     }
43.
44.     private void textSalaire_TextChanged(object sender, EventArgs e) {
45.        // état bouton [Calculer]
46.        buttonCalculer.Enabled=textSalaire.Text.Trim()!="";
47.     }
48.
49.  }
50. }
```

Nous ne commentons que les parties importantes :

- ligne [8] : la propriété publique *Metier* qui permet à la classe de lancement [Program.cs] d'injecter dans [Form1] une référence sur la couche [metier].
- ligne [14] : la procédure de calcul de l'impôt
- lignes 15-27 : vérification de la validité du salaire (un nombre entier >=0).
- ligne 29 : calcul de l'impôt à l'aide de la méthode [CalculerImpot] de la couche [metier]. On notera la simplicité de cette opération obtenue grâce à l'encapsulation de la couche [metier] dans une DLL.

7.6.4 Le formulaire [Form2]

En mode [conception] le formulaire [Form2] est le suivant :



Les contrôles sont les suivants

n°	type	nom	rôle
1	TextBox	textBoxErreur	*Multiline*=True, *Scrollbars*=Both

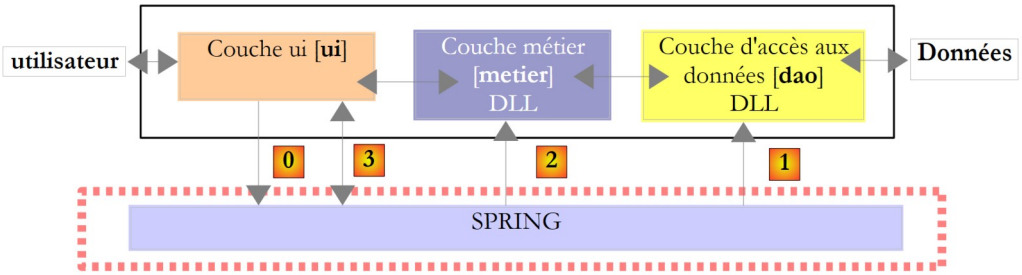
Le code de la classe est le suivant :

```
1.  using System.Windows.Forms;
2.
3.  namespace Chap5 {
4.   public partial class Form2 : Form {
5.      // msg d'erreur
6.      public string MsgErreur { private get; set; }
7.
8.      public Form2() {
9.         InitializeComponent();
10.     }
11.
12.     private void Form2_Load(object sender, System.EventArgs e) {
13.        // on affiche le msg d'erreur
14.        textBoxErreur.Text = MsgErreur;
15.        // on désélectionne tout le texte
16.        textBoxErreur.Select(0, 0);
17.     }
18.  }
19. }
```

- ligne 6 : la propriété publique *MsgErreur* qui permet à la classe de lancement [Program.cs] d'injecter dans [Form2] le message d'erreur à afficher. Ce message est affiché lors du traitement de l'événement *Load*, lignes 12-16.
- ligne 14 : le message d'erreur est mis dans le *TextBox*
- ligne 16 : on enlève la sélection qui a lieu lors de l'opération précédente. *[TextBox].Select(début,longueur)* sélectionne (mise en surbrillance) *longueur* caractères à partir du caractère n° *début*. *[TextBox].Select(0,0)* revient à désélectionner tout texte.

7.6.5 Conclusion

Revenons sur l'architecture trois couches utilisée :



Cette architecture nous a permis de substituer une implémentation graphique à l'implémentation console de la couche [ui] existante, sans rien changer aux couches [metier] et [dao]. Nous avons pu nous concentrer sur la couche [ui] sans nous inquiéter des impacts possibles sur les autres couches. C'est là le principal intérêt des architectures 3 couches. Nous en verrons un autre exemple plus loin, lorsque la couche [dao] qui exploite actuellement les données d'un fichier texte, sera remplacée par une couche [dao] exploitant les données d'une base de données. Nous verrons que cela se fera sans impact sur les couches [ui] et [metier].

8 Evénements utilisateur

Nous avons dans le chapitre précédent abordé la notion d'événements liés à des composants de formulaire. Nous voyons maintenant comment créer des événements dans nos propres classes.

8.1 Objets delegate prédéfinis

La notion d'objet *delegate* a été rencontrée dans le chapitre précédent mais elle avait été alors survolée. Lorsque nous avions regardé comment les gestionnaires des événements des composants d'un formulaire étaient déclarés, nous avions rencontré du code similaire au suivant :

```
        this.buttonAfficher.Click += new System.EventHandler(this.buttonAfficher_Click);
```

où *buttonAfficher* était un composant de type [Button]. Cette classe a un champ **Click** défini comme suit :

- [1] : la classe [Button]
- [2] : ses événements
- [3,4] : l'événement *Click*
- [5] : la déclaration de l'événement [Control.Click] [4].
  - *EventHandler* est un prototype (un modèle) de méthode appelé **delegate**.
  - *event* est un mot clé qui restreint les fonctionnalités du *delegate* **EventHandler** : un objet *delegate* a des fonctionnalités plus riches qu'un objet *event*.

Le *delegate* **EventHandler** est défini comme suit :

Le *delegate* **EventHandler** désigne un modèle de méthode :

- ayant pour 1er paramètre un type **Object**
- ayant pour 2ième paramètre un type **EventArgs**
- ne rendant aucun résultat

Une méthode correspondant au modèle défini par *EventHandler* pourrait être la suivante :

```
    private void buttonAfficher_Click(object sender, EventArgs e);
```

Pour créer un objet de type *EventHandler*, on procède comme suit :

```
EventHandler evtHandler=new EventHandler(méthode correspondant au prototype  défini par le type EventHandler);
```

On pourra ainsi écrire :

```
EventHandler evtHandler=new EventHandler(buttonAfficher_Click);
```

Une variable de type *delegate* est en fait une liste de références sur des méthodes correspondant au modèle du *delegate*. Pour ajouter une nouvelle méthode *M* à la variable *evtHandler* ci-dessus, on utilise la syntaxe :

```
evtHandler+=new EvtHandler(M);
```

La notation += peut être utilisée même si *evtHandler* est une liste vide.

L'instruction :

```
        this.buttonAfficher.Click += new System.EventHandler(this.buttonAfficher_Click);
```

ajoute une méthode de type *EventHandler* à la liste des méthodes de l'événement *buttonAfficher.Click*. Lorsque l'événement *Click* sur le composant buttonAfficher se produit, VB exécute l'instruction :

```
        buttonAfficher.Click(source, evt);
```

où :
- *source* est le composant de type *object* à l'origine de l'événement
- *evt* de type *EventArgs* et ne contient pas d'information

Toutes les méthodes de signature *void M(object,EventArgs)* qui ont été associées à l'événement *Click* par :

```
        this.buttonAfficher.Click += new System.EventHandler(M);
```

seront appelées avec les paramètres (*source*, *evt*) transmis par VB.

8.2 Définir des objets delegate

L'instruction

```
 public delegate int Opération(int n1, int n2);
```

définit un type appelé *Opération* qui représente 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 méthode *fi* dans la variable *op* se fait par *op=new Opération(fi)* ou plus simplement par *op=fi*. Pour ajouter une méthode *fj* à la liste des fonctions déjà enregistrées, on écrit *op+= fj*. Pour enlever une méthode *fk* déjà enregistrée on écrit *op-=fk*. Si dans notre exemple on écrit *n=op(n1,n2)*, l'ensemble des méthodes enregistrées dans la variable *op* seront exécutées avec les paramètres *n1* et *n2*. Le résultat *n* récupéré sera celui de la dernière méthode exécutée. Il n'est pas possible d'obtenir les résultats produits par l'ensemble des méthodes. Pour cette raison, si on enregistre une liste de méthodes dans une fonction déléguée, celles-ci rendent le plus souvent un résultat de type *void*.

Considérons l'exemple suivant :

```
1.  using System;
2.  namespace Chap6 {
3.   class Class1 {
4.      // définition d'un prototype de fonction
5.      // accepte 2 entiers en paramètre et rend un entier
6.      public delegate int Opération(int n1, int n2);
7.
8.      // deux méthodes d'instance correspondant au prototype
9.      public int Ajouter(int n1, int n2) {
10.        Console.WriteLine("Ajouter(" + n1 + "," + n2 + ")");
11.        return n1 + n2;
12.     }//ajouter
13.
14.     public int Soustraire(int n1, int n2) {
15.        Console.WriteLine("Soustraire(" + n1 + "," + n2 + ")");
16.        return n1 - n2;
17.     }//soustraire
18.
19.     // une méthode statique correspondant au prototype
20.     public static int Augmenter(int n1, int n2) {
21.        Console.WriteLine("Augmenter(" + n1 + "," + n2 + ")");
22.        return n1 + 2 * n2;
23.     }//augmenter
24.
25.     static void Main(string[] args) {
26.
27.        // on définit un objet de type opération pour y enregistrer des fonctions
28.        // on enregistre la fonction statique augmenter
29.        Opération op = Augmenter;
30.        // on exécute le délégué
31.        int n = op(4, 7);
32.        Console.WriteLine("n=" + n);
33.
34.        // création d'un objet c1 de type class1
35.        Class1 c1 = new Class1();
36.        // on enregistre dans le délégué la méthode ajouter de c1
37.        op = c1.Ajouter;
38.        // exécution de l'objet délégué
39.        n = op(2, 3);
40.        Console.WriteLine("n=" + n);
41.        // on enregistre dans le délégué la méthode soustraire de c1
42.        op = c1.Soustraire;
43.        n = op(2, 3);
44.        Console.WriteLine("n=" + n);
45.        //enregistrement de deux fonctions dans le délégué
46.        op = c1.Ajouter;
47.        op += c1.Soustraire;
48.        // exécution de l'objet délégué
49.        op(0, 0);
50.        // on retire une fonction du délégué
51.        op -= c1.Soustraire;
52.        // on exécute le délégué
53.        op(1, 1);
54.     }
55.  }
56. }
```

- ligne 3 : définit une classe *Class1*.
- ligne 6 : définition du *delegate Opération* : un prototype de méthodes acceptant deux paramètres de type *int* et rendant un résultat de type *int*
- lignes 9-12 : la méthode d'instance *Ajouter* a la signature du *delegate Opération*.
- lignes 14-17 : la méthode d'instance S*oustraire* a la signature du *delegate Opération*.
- lignes 20-23 : la méthode **de classe** A*ugmenter* a la signature du *delegate Opération*.
- ligne 25 : la méthode *Main* exécutée
- ligne 20 : la variable *op* est de type *delegate Opération*. Elle contiendra une liste de méthodes ayant la signature du type *delegate Opération*. On lui affecte une première référence de méthode, celle sur la méthode statique *Class1.Augmenter*.
- ligne 31 : le *delegate op* est exécuté : ce sont toutes les méthodes référencées par *op* qui vont être exécutées. Elles le seront avec les paramètres passés au *delegate op*. Ici, seule la méthode statique *Class1.Augmenter* va être exécutée.
- ligne 35 : une instance *c1* de la classe *Class1* est créée.
- ligne 37 : la méthode d'instance *c1.Ajouter* est affectée au *delegate op*. *Augmenter* était une méthode statique, *Ajouter* est une méthode d'instance. On a voulu montrer que cela n'avait pas d'importance.
- ligne 39 : le *delegate op* est exécuté : la méthode *Ajouter* va être exécutée avec les paramètres passés au *delegate op*.
- ligne 42 : on refait de même avec la méthode d'instance *Soustraire*.
- lignes 46-47 : on met les méthodes *Ajouter* et *Soustraire* dans le *delegate op*.

- ligne 49 : le *delegate op* est exécuté : les deux méthodes *Ajouter* et *Soustraire* vont être exécutées avec les paramètres passés au *delegate op*.
- ligne 51 : la méthode *Soustraire* est enlevée du *delegate op*.
- ligne 53 : le *delegate op* est exécuté : la méthode restante *Ajouter* va être exécutée.

Les résultats de l'exécution sont les suivants :

```
1.  Augmenter(4,7)
2.  n=18
3.  Ajouter(2,3)
4.  n=5
5.  Soustraire(2,3)
6.  n=-1
7.  Ajouter(0,0)
8.  Soustraire(0,0)
9.  Ajouter(1,1)
```

8.3 Delegates ou interfaces ?

Les notions de *delegates* et d'interfaces peuvent sembler assez proches et on peut se demander quelles sont exactement les différences entre ces deux notions. Prenons l'exemple suivant proche d'un exemple déjà étudié :

```
1.  using System;
2.  namespace Chap6 {
3.   class Program1 {
4.      // définition d'un prototype de fonction
5.      // accepte 2 entiers en paramètre et rend un entier
6.      public delegate int Opération(int n1, int n2);
7.
8.      // deux méthodes d'instance correspondant au prototype
9.      public static int Ajouter(int n1, int n2) {
10.        Console.WriteLine("Ajouter(" + n1 + "," + n2 + ")");
11.        return n1 + n2;
12.     }//ajouter
13.
14.     public static int Soustraire(int n1, int n2) {
15.        Console.WriteLine("Soustraire(" + n1 + "," + n2 + ")");
16.        return n1 - n2;
17.     }//soustraire
18.
19.     // Exécution d'un délégué
20.     public static int Execute(Opération op, int n1, int n2){
21.        return op(n1, n2);
22.     }
23.
24.     static void Main(string[] args) {
25.        // exécution du délégué Ajouter
26.        Console.WriteLine(Execute(Ajouter, 2, 3));
27.        // exécution du délégué Soustraire
28.        Console.WriteLine(Execute(Soustraire, 2, 3));
29.        // exécution d'un délégué multicast
30.        Opération op = Ajouter;
31.        op += Soustraire;
32.        Console.WriteLine(Execute(op, 2, 3));
33.        // on retire une fonction du délégué
34.        op -= Soustraire;
35.        // on exécute le délégué
36.        Console.WriteLine(Execute(op, 2, 3));
37.     }
38.  }
39. }
```

Ligne 20, la méthode *Execute* attend une référence sur un objet du type *delegate Opération* défini ligne 6. Cela permet de passer à la méthode *Execute*, différentes méthodes (lignes 26, 28, 32 et 36). Cette propriété de polymorphisme peut être obtenue également avec une interface :

```
1.  using System;
2.
3.  namespace Chap6 {
4.
5.   // interface IOperation
6.   public interface IOperation {
7.      int operation(int n1, int n2);
8.   }
9.
```

```
10.  // classe Ajouter
11.  public class Ajouter : IOperation {
12.     public int operation(int n1, int n2) {
13.        Console.WriteLine("Ajouter(" + n1 + "," + n2 + ")");
14.        return n1 + n2;
15.     }
16.  }
17.
18.  // classe Soustraire
19.  public class Soustraire : IOperation {
20.     public int operation(int n1, int n2) {
21.        Console.WriteLine("Soustraire(" + n1 + "," + n2 + ")");
22.        return n1 - n2;
23.     }
24.  }
25.
26.  // classe de test
27.  public static class Program2 {
28.     // Exécution de la méthode unique de l'interface IOperation
29.     public static int Execute(IOperation op, int n1, int n2) {
30.        return op.operation(n1, n2);
31.     }
32.
33.     public static void Main() {
34.        // exécution du délégué Ajouter
35.        Console.WriteLine(Execute(new Ajouter(), 2, 3));
36.        // exécution du délégué Soustraire
37.        Console.WriteLine(Execute(new Soustraire(), 2, 3));
38.     }
39.  }
40. }
```

- lignes 6-8 : l'interface [IOperation] définit une méthode *operation*.
- lignes 11-16 et 19-24 : les classes [Ajouter] et [Soustraire] implémentent l'interface [IOperation].
- lignes 29-31 : la méthode *Execute* dont le 1er paramètre est du type de l'interface *IOperation*. La méthode *Execute* va recevoir successivement comme 1er paramètre, une instance de la classe *Ajouter* puis une instance de la classe *Soustraire*.

On retrouve bien l'aspect polymorphique qu'avait le paramètre de type *delegate* de l'exemple précédent. Les deux exemples montrent en même temps des différences entre ces deux notions.

Les types *delegate* et *interface* sont interchangeables

- si l'interface n'a qu'une méthode. En effet, le type *delegate* est une enveloppe pour une unique méthode alors que l'interface peut, elle, définir plusieurs méthodes.
- si l'aspect multicast du *delegate* n'est pas utilisé. Cette notion de multicast n'existe en effet pas dans l'interface.

Si ces deux conditions sont vérifiées, alors on a le choix entre les deux signatures suivantes pour la méthode *Execute* :

```
1.  int Execute(IOperation op, int n1, int n2)
2.  int Execute(Opération op, int n1, int n2)
```

La seconde qui utilise le *delegate* peut se montrer plus souple d'utilisation. En effet dans la première signature, le premier paramètre de la méthode doit implémenter l'interface *IOperation*. Cela oblige à créer une classe pour y définir la méthode appelée à être passée en premier paramètre à la méthode *Execute*. Dans la seconde signature, toute méthode existante ayant la bonne signature fait l'affaire. Il n'y a pas de construction supplémentaire à faire.

8.4 Gestion d'événements

Les objets *delegate* peuvent servir à définir des événements. Une classe *C1* peut définir un événement *evt* de la façon suivante :

- un type *delegate* est défini dans ou en-dehors de la classe C1 :

```
delegate TResult Evt(T1 param1, T2 param2, ...);
```

- la classe *C1* définit un champ de type *delegate Evt* :

```
public Evt Evt1;
```

- lorsque une instance *c1* de la classe *C1* voudra signaler un événement, elle exécutera son *delegate Evt1* en lui passant les paramètres définis par le *delegate Evt*. Toutes les méthodes enregistrées dans le *delegate Evt1* seront alors exécutées avec ces paramètres. On peut dire qu'elles ont été averties de l'événement *Evt1*.

- si un objet c*2* utilisant un objet c*1* veut être averti de l'occurrence de l'événement *Evt1* sur l'objet c*1*, il enregistrera l'une de ses méthodes c*2.M* dans l'objet délégué c*1.Evt1* de l'objet c*1*. Ainsi sa méthode c*2.M* sera exécutée à chaque fois que l'événement E*vt1* se produira sur l'objet c*1*. Il pourra également de désinscrire lorsqu'il ne voudra plus être averti de l'événement.
- comme l'objet délégué c*1.Evt1* peut enregistrer plusieurs méthodes, différents objets c*i* pourront s'enregistrer auprès du délégué c*1.Evt1* pour être prévenus de l'événement *Evt1* sur c*1*.

Dans ce scénario, on a :
- une classe qui signale un événement
- des classes qui sont averties de cet événement. On dit qu'elles souscrivent à l'événement ou s'abonnent à l'événement.
- un type *delegate* qui définit la signature des méthodes qui seront averties de l'événement

Le framework .NET définit :

- une signature standard du *delegate* d'un événement

```
public delegate void MyEventHandler(object source, EventArgs evtInfo);
```

  - *source* : l'objet qui a signalé l'événement
  - *evtInfo* : un objet de type *EventArgs* ou dérivé qui apporte des informations sur l'événement
  - le nom du *delegate* doit être terminé par *EventHandler*

- une façon standard pour déclarer un événement de type *MyEventHandler* dans une classe :

```
1.  public Class C1{
2.   public event MyEventHandler Evt1;
3.  ...
4.  }
```

  Le champ E*vt1* est de type *delegate*. Le mot clé *event* est là pour restreindre les opérations qu'on peut faire sur lui :
  - de l'extérieur de la classe *C1*, seules les opérations += et -= sont possibles. Cela empêche la suppression (par erreur du développeur par exemple) des méthodes abonnées à l'événement. On peut simplement s'abonner (+=) ou se désabonner (-=) de l'événement.
  - seule une instance de type *C1* peut exécuter l'appel *Evt1(source,evtInfo)* qui déclenche l'exécution des méthodes abonnées à l'événement *Evt1*.

Le framework .NET fournit une méthode générique satisfaisant à la signature du *delegate* d'un événement :

```
public delegate void EventHandler<TEventArgs>(object source, TEventArgs evtInfo) where TEventArgs : EventArgs
```

- le *delegate EventHandler* utilise le type générique *TEventArgs* qui est le type de son 2ième paramètre
- le type *TEventArgs* doit dériver du type *EventsArgs* (*where TEventArgs : EventArgs*)

Avec ce *delegate* générique, la déclaration d'un événement *X* dans la classe C suivra le schéma conseillé suivant :
- définir un type *XEventArgs* dérivé de *EventArgs* pour encapsuler les informations sur l'événement *X*
- définir dans la classe C un événement de type *EventHandler*<*XEventArgs*>.
- définir dans la classe C une méthode protégée
  ```
  protected void OnXHandler(XEventArgs e);
  ```
  destinée à "publier" l'événement X aux abonnés.

Considérons l'exemple suivant :
- une classe *Emetteur* encapsule une température. Cette température est observée. Lorsque cette température dépasse un certain seuil, un événement doit être lancé. Nous appellerons cet événement *TemperatureTropHaute*. Les informations sur cet événement seront encapsulées dans un type *TemperatureTropHauteEventArgs*.
- une classe *Souscripteur* s'abonne à l'événement précédent. Lorsqu'elle est avertie de l'événement, elle affiche un message sur la console.
- un programme console crée un émetteur et deux abonnés. Il saisit les températures au clavier et les enregistre dans une instance *Emetteur*. Si celle-ci est trop haute, l'instance *Emetteur* publie l'événement *TemperatureTropHaute*.

Pour se conformer à la méthode conseillée de gestion des événements, nous définissons tout d'abord le type *TemperatureTropHauteEventArgs* pour encapsuler les informations sur l'événement :

```
1.  using System;
2.
3.  namespace Chap6 {
4.   public class TemperatureTropHauteEventArgs:EventArgs {
5.      // température lors de l'évt
6.      public decimal Temperature { get; set; }
```

```
7.      // constructeurs
8.      public TemperatureTropHauteEventArgs() {
9.      }
10.     public TemperatureTropHauteEventArgs(decimal temperature) {
11.        Temperature = temperature;
12.     }
13.  }
14. }
```

- ligne 6 : l'information encapsulée par la classe *TemperatureTropHauteEventArgs* est la température qui a provoqué l'événement *TemperatureTropHaute*.

La classe *Emetteur* est la suivante :

```
1.  using System;
2.
3.  namespace Chap6 {
4.   public class Emetteur {
5.      static decimal SEUIL = 19;
6.
7.      // température observée
8.      private decimal temperature;
9.      // nom de la source
10.     public string Nom { get; set; }
11.     // évt signalé
12.     public event EventHandler<TemperatureTropHauteEventArgs> TemperatureTropHaute;
13.
14.     // lecture / écriture température
15.     public decimal Temperature {
16.        get {
17.           return temperature;
18.        }
19.        set {
20.           temperature = value;
21.           if (temperature > SEUIL) {
22.              // on signale l'événement aux abonnés
23.              OnTemperatureTropHaute(new TemperatureTropHauteEventArgs(temperature));
24.           }
25.        }
26.     }
27.
28.     // signalement d'un évt
29.     protected virtual void OnTemperatureTropHaute(TemperatureTropHauteEventArgs evt) {
30.        // émission de l'évt TemperatureTropHaute aux abonnés
31.        TemperatureTropHaute(this, evt);
32.     }
33.  }
34. }
```

- ligne 5 : le seuil de température au-delà duquel l'événement *TemperatureTropHaute* sera publié.
- ligne 10 : l'émetteur a un nom pour être identifié
- ligne 12 : l'événement *TemperatureTropHaute*.
- lignes 15-26 : la méthode *get* qui rend la température et la méthode *set* qui l'enregistre. C'est la méthode *set* qui fait publier l'événement *TemperatureTropHaute* si la température à enregistrer dépasse le seuil de la ligne 5. Elle fait publier l'événement par la méthode *OnTemperatureTropHauteHandler* de la ligne 29 en lui passant pour paramètre un objet *TemperatureTropHauteEventArgs* dans lequel on a enregistré la température qui a dépassé le seuil.
- lignes 29-32 : l'événement *TemperatureTropHaute* est publié avec pour 1er paramètre l'émetteur lui-même et pour second paramètre l'objet *TemperatureTropHauteEventArgs* reçu en paramètre.

La classe *Souscripteur* qui va s'abonner à l'événement *TemperatureTropHaute* est la suivante :

```
1.  using System;
2.
3.  namespace Chap6 {
4.   public class Souscripteur {
5.      // nom
6.      public string Nom { get; set; }
7.
8.      // gestionnaire de l'évt TemperatureTropHaute
9.      public void EvtTemperatureTropHaute(object source, TemperatureTropHauteEventArgs e) {
10.        // affichage console opérateur
11.        Console.WriteLine("Souscripteur [{0}] : la source [{1}] a signalé une température trop
    haute : [{2}]", Nom, ((Emetteur)source).Nom, e.Temperature);
12.     }
13.  }
14. }
```

- ligne 6 : chaque souscripteur est identifié par un nom.
- lignes 9-12 : la méthode qui sera associée à l'événement *TemperatureTropHaute*. Elle a la signature du type *delegate EventHandler<TEventArgs>* qu'un gestionnaire d'événement doit avoir. La méthode affiche sur la console : le nom du souscripteur qui affiche le message, le nom de l'émetteur qui a signalé l'événement, la température qui a déclenché ce dernier.
- l'abonnement à l'événement *TemperatureTropHaute* d'un objet *Emetteur* n'est pas fait dans la classe *Souscripteur*. Il sera fait par une classe externe.

Le programme [Program.cs] lie tous ces éléments entre-eux :

```
1.  using System;
2.  namespace Chap6 {
3.   class Program {
4.      static void Main(string[] args) {
5.         // création d'un émetteur d'evts
6.         Emetteur e1 = new Emetteur() { Nom = "e" };
7.         // création d'un tableau de 2 souscripteurs
8.         Souscripteur[] souscripteurs = new Souscripteur[2];
9.         for (int i = 0; i < souscripteurs.Length; i++) {
10.           // création souscripteur
11.           souscripteurs[i] = new Souscripteur() { Nom = "s" + i };
12.           // on l'abonne à l'évt TemperatureTropHaute de e1
13.           e1.TemperatureTropHaute += souscripteurs[i].EvtTemperatureTropHaute;
14.        }
15.        // on lit les températures au clavier
16.        decimal temperature;
17.        Console.Write("Température (rien pour arrêter) : ");
18.        string saisie = Console.ReadLine().Trim();
19.        // tant que la ligne saisie est non vide
20.        while (saisie != "") {
21.           // la saisie est-elle un nombre décimal ?
22.           if (decimal.TryParse(saisie, out temperature)) {
23.              // température correcte - on l'enregistre
24.              e1.Temperature = temperature;
25.           } else {
26.              // on signale l'erreur
27.              Console.WriteLine("Température incorrecte");
28.           }
29.           // nouvelle saisie
30.           Console.Write("Température (rien pour arrêter) : ");
31.           saisie = Console.ReadLine().Trim();
32.        }//while
33.     }
34.  }
35. }
```

- ligne 6 : création de l'émetteur
- lignes 8-14 : création de deux souscripteurs qu'on abonne à l'événement *TemperatureTropHaute* de l'émetteur.
- lignes 20-32 : boucle de saisie des températures au clavier
- ligne 24 : si la température saisie est correcte, elle est transmise à l'objet *Emetteur* e1 qui déclenchera l'événement *TemperatureTropHaute* si la température est supérieure à 19 ° C.

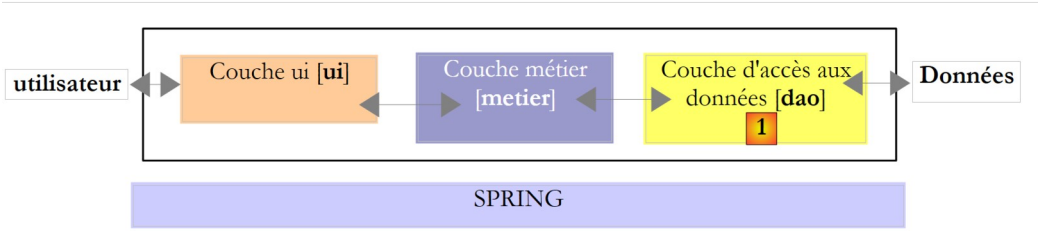
Les résultats de l'exécution sont les suivants :

```
1.  Température (rien pour arrêter) : 17
2.  Température (rien pour arrêter) : 21
3.  Souscripteur [s0] : la source [e] a signalé une température trop haute : [21]
4.  Souscripteur [s1] : la source [e] a signalé une température trop haute : [21]
```

9 Accès aux bases de données

9.1 Connecteur ADO.NET

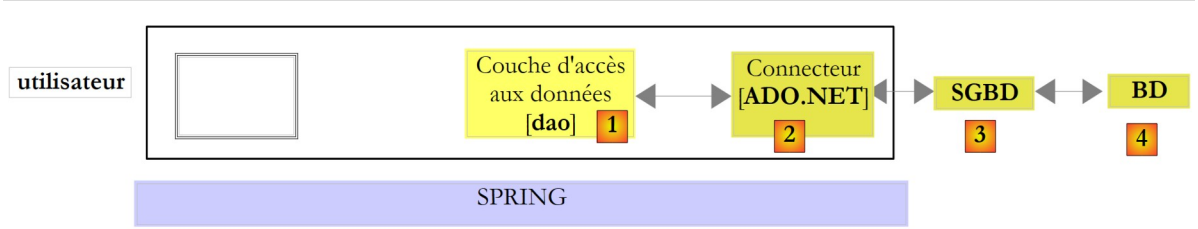
Reprenons l'architecture en couches utilisée à diverses reprises



Dans les exemples étudiés, la couche [dao] a pour l'instant exploité deux types de sources de données :

- des données placées en dur dans le code
- des données provenant de fichiers texte

Nous étudions dans ce chapitre le cas où les données proviennent d'une base de données. L'architecture 3 couches évolue alors vers une architecture multi-couches. Il en existe diverses. Nous allons étudier les concepts de base avec la suivante :



Dans le schéma ci-dessus, la couche [dao] [1] dialogue avec le SGBD [3] au travers d'une bibliothèque de classes propre au SGBD utilisé et livrée avec lui. Cette couche implémente des fonctionnalités standard réunies sous le vocable ADO (Active X Data Objects). On appelle une telle couche, un **provider** (fournisseur d'accès à une base de données ici) ou encore **connecteur**. La plupart des SGBD disposent désormais d'un connecteur ADO.NET, ce qui n'était pas le cas aux débuts de la plate-forme .NET. Les connecteurs .NET n'offrent pas une interface standard à la couche [dao], aussi celle-ci a-t-elle dans son code le nom des classes du connecteur. Si on change de SGBD, on change de connecteur et de classes et il faut alors changer la couche [dao]. C'est à la fois une architecture **performante** parce le connecteur .NET ayant été écrit pour un SGBD particulier sait utiliser au mieux celui-ci et **rigide** car changer de SGBD implique de changer la couche [dao]. Ce deuxième argument est à relativiser : les entreprises ne changent pas de SGBD très souvent. Par ailleurs, nous verrons ultérieurement que depuis la version 2.0 de .NET, il existe un connecteur générique qui amène de la souplesse sans sacrifier la performance.

9.2 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

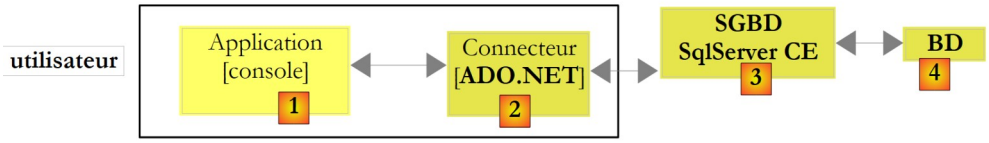
5. lorsque le travail est fini, ouvre une connexion, envoie les données modifiées à la source de données pour qu'elle les prenne en compte, ferme la connexion

Nous n'étudions ici que le mode connecté.

9.3 Les concepts de base de l'exploitation d'une base de données

Nous allons exposer les principaux concepts d'utilisation d'une base de données avec une base de données SQL Server Compact 3.5. Ce SGBD est livré avec Visual Studio Express. C'est un SGBD léger qui ne sait gérer qu'un utilisateur à la fois. Il est cependant suffisant pour introduire la programmation avec les bases de données. Ultérieurement, nous présenterons d'autres SGBD.

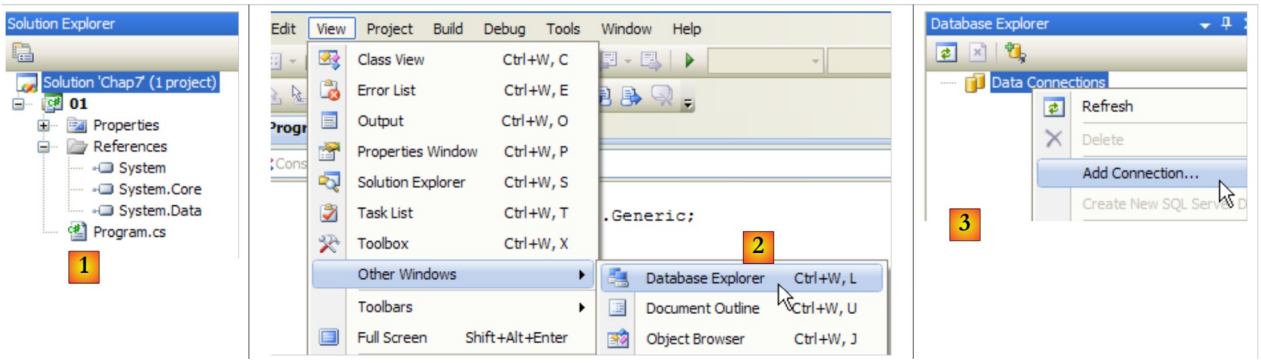
L'architecture utilisée sera la suivante :



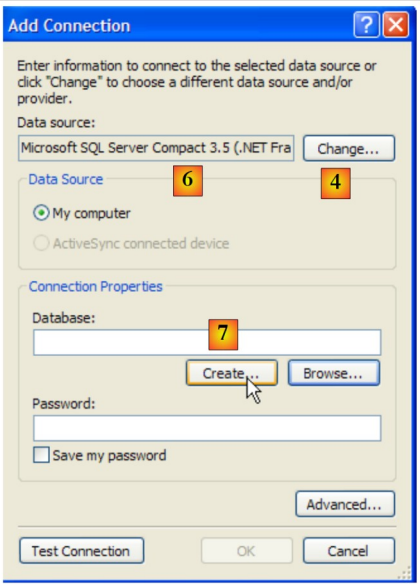
Une application console [1] exploitera une base de données de type SqlServer Compact [3,4] via le connecteur Ado.Net de ce SGBD [2].

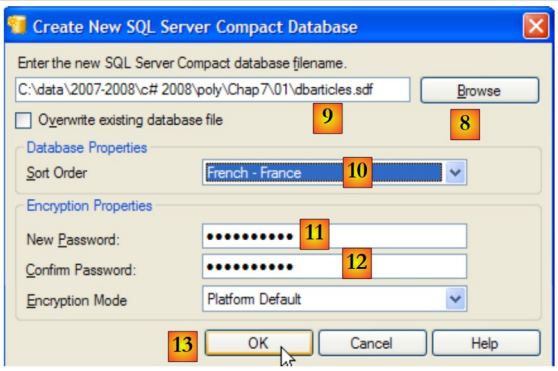
9.3.1 La base de données exemple

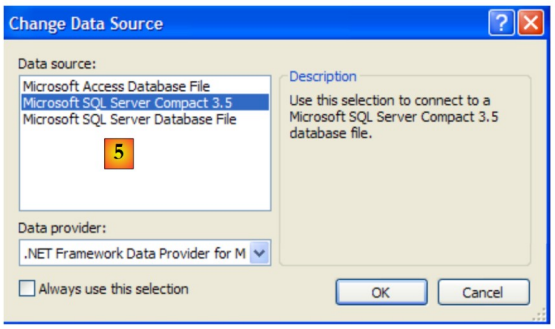
Nous allons construire la base de données directement dans Visual studio Express. Pour cela, nous créons un nouveau projet de type console.



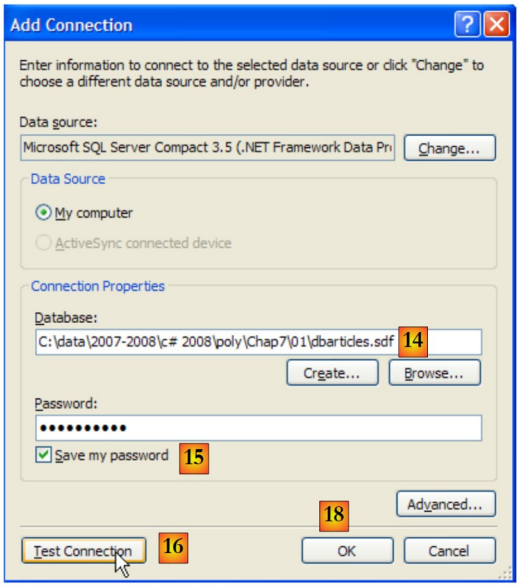
- [1] : le projet
- [2] : on ouvre une vue "Explorateur de bases de données"
- [3] : on crée une nouvelle connexion

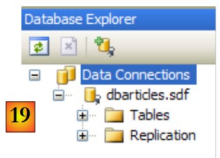


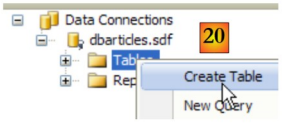




- [4] : on sélectionne le type du SGBD
- [5,6] : on choisit le SGBD SQL Server Compact
- [7] : on crée la base de données
- [8] : une base de données SQL Server Compact est encapsulée dans un unique fichier de suffixe *.sdf*. On indique où la créer, ici dans le dossier du projet C#.
- [9] : on a donné le nom [dbarticles.sdf] à la nouvelle base
- [10] : on sélectionne la langue française. Cela a une conséquence sur les opérations de tri.
- [11,12] : la base de données peut être protégée par un mot de passe. Ici "*dbarticles*".
- [13] : on valide la page de renseignements. La base va être physiquement créée :



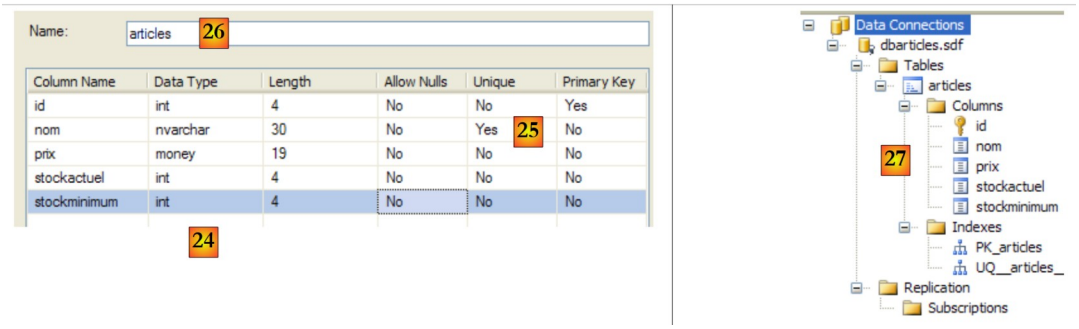




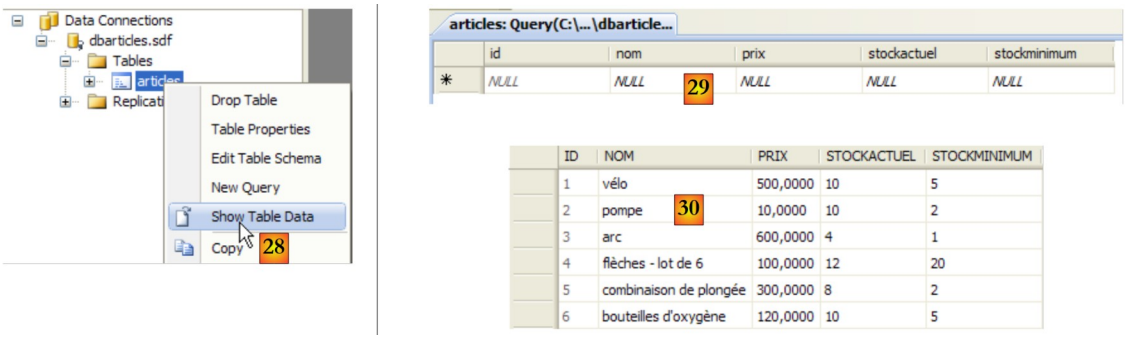
- [14] : le nom de la base qui vient d'être créée
- [15] : on coche l'option "Save my password" afin de ne pas avoir à le retaper à chaque fois
- [16] : on vérifie la connexion

- [17] : tout va bien
- [18] : on valide la page d'informations
- [19] : la connexion apparaît dans l'explorateur de bases de données
- [20] : pour l'instant la base est sans tables. On en crée une. Un article aura les champs suivants :
  - *id* : un identifiant unique - clé primaire
  - *nom* : nom de l'article - unique
  - *prix* : prix de l'article
  - *stockactuel* : son stock actuel
  - *stockminimum* : le stock minimum en-deça duquel il faut réapprovisionner l'article

- [21] : le champ [id] est de type entier et est clé primaire [22] de la table.
- [23] : cette clé primaire est de type *Identity*. Cette notion propre aux SGBD SQL Server indique que la clé primaire sera générée par le SGBD lui-même. Ici la clé primaire sera un nombre entier commençant à 1 et incrémenté de 1 à chaque nouvelle clé.

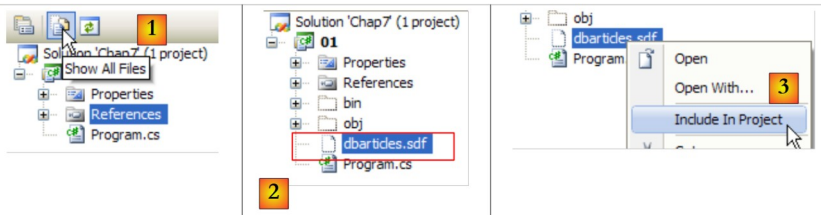


- [24] : les autres champs sont créés. On notera que le champ [nom] a une contrainte d'unicité [25].
- [26] : on donne un nom à la table
- [27] : après avoir validé la structure de la table, celle-ci apparaît dans la base.

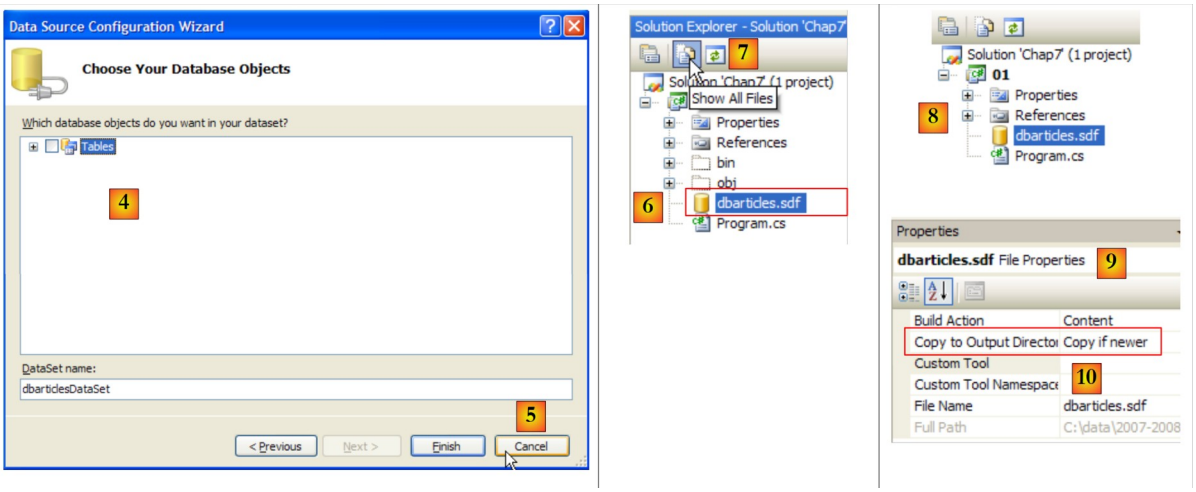


- [28] : on demande à voir le contenu de la table
- [29] : elle est vide pour l'instant
- [30] : on la remplit avec quelques données. Une ligne est validée dès qu'on passe à la saisie de la ligne suivante. Le champ [id] n'est pas saisi : il est généré automatiquement lorsque la ligne est validée.

Il nous reste à configurer le projet pour que cette base qui est actuellement à la racine du projet soit recopiée automatiquement dans le dossier d'exécution du projet :



- [1] : on demande à voir tous les fichiers
- [2] : la base [dbarticles.sdf] apparaît
- [3] : on l'inclut dans le projet



- [4] : l'opération d'ajout d'une source de données dans un projet lance un assistant dont nous n'avons pas besoin ici [5].
- [6] : la base fait maintenant partie du projet. On revient en mode normal [7].
- [8] : le projet avec sa base
- [9] : dans les propriétés de la base, on peut voir [10] que celle-ci sera automatiquement recopiée dans le dossier d'exécution du projet. C'est là que le programme que nous allons écrire, ira la chercher.

Maintenant que nous avons une base de données disponible, nous allons pouvoir l'exploiter. Auparavant, nous faisons quelques rappels SQL.

9.3.2 Les quatre commandes de base du langage SQL

SQL (Structured Language Query) est un langage, partiellement normalisé, d'interrogation et de mise à jour des bases de données. Tous les SGBD respectent la partie normalisée de SQL mais ajoutent au langage des extensions propriétaires qui exploitent certaines particularités du SGBD. Nous en avons déjà rencontré deux exemples : la génération automatique des clés primaires et les types autorisés pour les colonnes d'une table sont souvent dépendants du SGBD.

Les quatre commandes de base du langage SQL que nous présentons sont normalisées et acceptées par tous les SGBD :

```
select col1, col2,...
from table1, table2,...
where condition
order by expression
...
```

La requête qui permet d'obtenir les données contenues dans une base. 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. Ce résultat est une table.

```
insert into
table(col1,col2, ...)
values (val1,val2, ...)
```

Insère une ligne dans **table**. (**col1**, **col2**, ...) précise les colonnes de la ligne à initialiser avec les valeurs (**val1**, **val2**, ...).

```
update table
set col1=val1, col2=val2
```

Met à jour les lignes de **table** vérifiant **condition** (toutes les lignes si pas de **where**). Pour ces lignes, la colonne **coli** reçoit la valeur **vali**

`where condition`	
`delete from table where condition`	Supprime toutes les lignes de **table** vérifiant **condition**

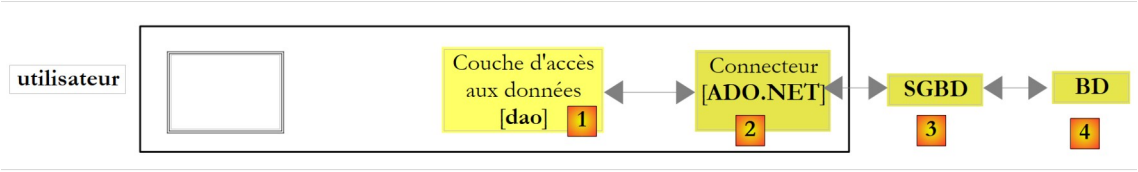
Nous allons écrire une application console permettant d'émettre des ordres SQL sur la base [dbarticles] que nous avons créée précédemment. Voici un exemple d'exécution. Le lecteur est invité à comprendre les ordres SQL émis et leurs résultats.

```
1.  Chaîne de connexion à la base : [Data Source=|DataDirectory|
    \dbarticles.sdf;Password=dbarticles;Persist Security Info=True]
2.
3.  Requête SQL (rien pour arrêter) : select id,nom,prix,stockactuel,stockminimum from articles
4.
5.  -----------------------------------
6.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
7.  -----------------------------------
8.
9.  1 vélo 500 10 5
10. 2 pompe 10 10 2
11. 3 arc 600 4 1
12. 4 flèches - lot de 6 100 12 20
13. 5 combinaison de plongée 300 8 2
14. 6 bouteilles d'oxygène 120 10 5
15.
16. Requête SQL (rien pour arrêter) : insert into articles(nom,prix,stockactuel,stockminimum)
    values('x',100,10,1)
17. Il y a eu 1 ligne(s) modifiée(s)
18.
19. Requête SQL (rien pour arrêter) : select id,nom,prix,stockactuel,stockminimum from articles
20.
21. -----------------------------------
22. ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
23. -----------------------------------
24.
25. 1 vélo 500 10 5
26. ...
27. 6 bouteilles d'oxygène 120 10 5
28. 9 x 100 10 1
29.
30. Requête SQL (rien pour arrêter) : update articles set prix=prix*1.1 where id=9
31. Il y a eu 1 ligne(s) modifiée(s)
32.
33. Requête SQL (rien pour arrêter) : select id,nom,prix,stockactuel,stockminimum from articles
34.
35. -----------------------------------
36. ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
37. -----------------------------------
38.
39. 1 vélo 500 10 5
40. ...
41. 6 bouteilles d'oxygène 120 10 5
42. 9 x 110 10 1
43.
44. Requête SQL (rien pour arrêter) : delete from articles where id=9
45. Il y a eu 1 ligne(s) modifiée(s)
46.
47. Requête SQL (rien pour arrêter) : select id,nom,prix,stockactuel,stockminimum from articles
48.
49. -----------------------------------
50. ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
51. -----------------------------------
52.
53. 1 vélo 500 10 5
54. ...
55. 6 bouteilles d'oxygène 120 10 5
```

- ligne 1 : la chaîne dite de connexion : elle contient tous les paramètres permettant de se connecter à la base de données.
- ligne 3 : on demande le contenu de la table [articles]
- ligne 16 : on insère une nouvelle ligne. On notera que le champ *id* n'est pas initialisé dans cette opération car c'est le SGBD qui va générer la valeur de ce champ.
- ligne 19 : vérification. Ligne 28, la ligne a bien été ajoutée.
- ligne 30 : on augmente de 10% le prix de l'article qui vient d'être ajouté.
- ligne 33 : on vérifie
- ligne 42 : l'augmentation du prix a bien eu lieu
- ligne 44 : on supprime l'article qu'on a ajouté précédemment
- ligne 47 : on vérifie
- lignes 53-55 : l'article n'est plus là.

9.3.3 Les interfaces de base d'ADO.NET pour le mode connecté

Revenons au schéma d'une application exploitant une base de données au travers d'un connecteur ADO.NET :



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

Trois interfaces ADO.NET sont principalement concernées par ces opérations :

- **IDbConnection** qui encapsule les propriétés et méthodes de la connexion.
- **IDbCommand** qui encapsule les propriétés et méthodes de la commande SQL exécutée.
- **IDataReader** qui encapsule les propriétés et méthodes du résultat d'un ordre SQL Select.

L'interface IDbConnection

Sert à gérer la connexion avec la base de données. Les méthodes **M** et propriétés **P** de cette interface que nous utiliserons seront les suivantes :

Nom	Type	Rôle
`ConnectionString`	P	**chaîne de connexion** à la base. Elle précise tous les paramètres nécessaires à l'établissement de la connexion avec une base précise.
`Open`	M	ouvre la connexion avec la base définie par *ConnectionString*
`Close`	M	ferme la connexion
`BeginTransaction`	M	démarre une transaction.
`State`	P	état de la connexion : *ConnectionState.Closed*, *ConnectionState.Open*, *ConnectionState.Connecting*, *ConnectionState.Executing*, *ConnectionState.Fetching*, *ConnectionState.Broken*

Si *Connection* est une classe implémentant l'interface *IDbConnection*, l'ouverture de la connexion peut se faire comme suit :

```
1.  IDbConnection connexion=new Connection();
2.  connexion.ConnectionString=...;
3.  connexion.Open();
```

L'interface IDbCommand

Sert à exécuter un ordre SQL ou une procédure stockée. Les méthodes **M** et propriétés **P** de cette interface que nous utiliserons seront les suivantes :

Nom	Type	Rôle
`CommandType`	P	indique ce qu'il faut exécuter - prend ses valeurs dans une énumération : - *CommandType.Text* : exécute l'ordre SQL défini dans la propriété *CommandText*. C'est la valeur par défaut. - *CommandType.StoredProcedure* : exécute une procédure stockée dans la base
`CommandText`	P	- le texte de l'ordre SQL à exécuter si *CommandType= CommandType.Text* - le nom de la procédure stockée à exécuter si *CommandType= CommandType.StoredProcedure*
`Connection`	P	la connexion *IDbConnection* à utiliser pour exécuter l'ordre SQL
`Transaction`	P	la transaction *IDbTransaction* dans laquelle exécuter l'ordre SQL
`Parameters`	P	la liste des paramètres d'un ordre SQL paramétré. L'ordre *update articles set prix=prix*1.1 where id=@id* a le paramètre @id.

Nom	Type	Rôle
ExecuteReader	M	pour exécuter un ordre SQL *Select*. On obtient un objet *IDataReader* représentant le résultat du *Select*.
ExecuteNonQuery	M	pour exécuter un ordre SQL *Update, Insert, Delete*. On obtient le nombre de lignes affectées par l'opération (mises à jour, insérées, détruites).
ExecuteScalar	M	pour exécuter un ordre SQL *Select* ne rendant qu'un unique résultat comme dans : *select count(*) from articles*.
CreateParameter	M	pour créer les paramètres *IDbParameter* d'un ordre SQL paramétré.
Prepare	M	permet d'optimiser l'exécution d'une requête paramétrée lorsqu'elle est exécutée de multiples fois avec des paramètres différents.

Si *Command* est une classe implémentant l'interface *IDbCommand*, l'exécution d'un ordre SQL sans transaction aura la forme suivante :

```
1.  // ouverture connexion
2.  IDbConnection connexion=...
3.  connexion.Open();
4.  // préparation commande
5.  IDbCommand commande=new Command();
6.  commande.Connection=connexion;
7.  // exécution ordre select
8.  commande.CommandText="select ...";
9.  IDbDataReader reader=commande.ExecuteReader();
10. ...
11. // exécution ordre update, insert, delete
12. commande.CommandText="insert ...";
13. int nbLignesInsérées=commande.ExecuteNonQuery();
14. ...
15. // fermeture connexion
16. connexion.Close();
```

L'interface IDataReader

Sert à encapsuler les résultats d'un ordre SQL *Select*. Un objet *IDataReader* représente une table avec des lignes et des colonnes, qu'on exploite séquentiellement : d'abord la 1ère ligne, puis la seconde, ... Les méthodes **M** et propriétés **P** de cette interface que nous utiliserons seront les suivantes :

Nom	Type	Rôle
FieldCount	P	le nombre de colonnes de la table *IDataReader*
GetName	M	*GetName(i)* rend le nom de la colonne n° i de la table *IDataReader*.
Item	P	*Item[i]* représente la colonne n° i de la ligne courante de la table *IDataReader*.
Read	M	passe à la ligne suivante de la table *IDataReader*. Rend le booléen *True* si la lecture a pu se faire, *False* sinon.
Close	M	ferme la table *IDataReader*.
GetBoolean	M	*GetBoolean*(i) : rend la valeur booléenne de la colonne n° i de la ligne courante de la table *IDataReader*. Les autres méthodes analogues sont les suivantes : *GetDateTime*, *GetDecimal*, *GetDouble*, *GetFloat*, *GetInt16*, *GetInt32*, *GetInt64*, *GetString*.
Getvalue	M	*Getvalue*(i) : rend la valeur de la colonne n° i de la ligne courante de la table *IDataReader* en tant que type *object*.
IsDBNull	M	*IsDBNull(i)* rend *True* si la colonne n° i de la ligne courante de la table *IDataReader* n'a pas de valeur ce qui est symbolisé par la valeur SQL NULL.

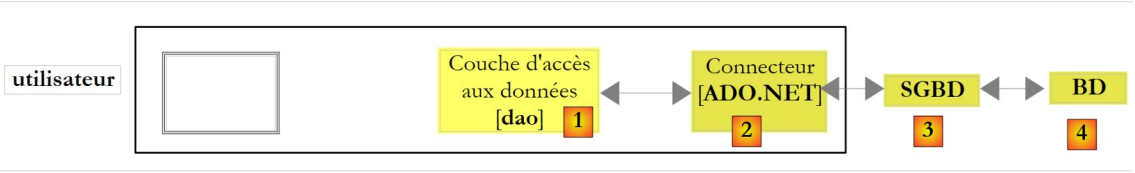
L'exploitation d'un objet *IDataReader* ressemble souvent à ce qui suit :

```
1.  // ouverture connexion
2.  IDbConnection connexion=...
3.  connexion.Open();
4.  // préparation commande
5.  IDbCommand commande=new Command();
6.  commande.Connection=connexion;
7.  // exécution ordre select
8.  commande.CommandText="select ...";
9.  IDataReader reader=commande.ExecuteReader();
```

```
10. // exploitation résultats
11. while(reader.Read()){
12.  // exploiter ligne courante
13.     ...
14. }
15. // fermeture reader
16. reader.Close();
17. // fermeture connexion
18. connexion.Close();
```

9.3.4 La gestion des erreurs

Revenons sur l'architecture d'une application avec base de données :



La couche [dao] peut rencontrer de nombreuses erreurs lors de l'exploitation de la base de données. Celles-ci vont être remontées en tant qu'exceptions lancées par le connecteur ADO.NET. Le code de la couche [dao] doit les gérer. Toute opération avec la base de données doit se faire dans un try / catch / finally pour intercepter et gérer une éventuelle exception et libérer les ressources qui doivent l'être. Ainsi le code vu plus haut pour exploiter le résultat d'un ordre *Select* devient le suivant :

```
1.  // initialisation connexion
2.  IDbConnection connexion=...
3.  // exploitation connexion
4.  try{
5.   // ouverture
6.   connexion.Open();
7.   // préparation commande
8.   IDbCommand commande=new Command();
9.   commande.Connection=connexion;
10.  // exécution ordre select
11.  commande.CommandText="select ...";
12.  IDbDataReader reader=commande.ExecuteReader();
13.  // exploitation résultats
14.  try{
15.     while(reader.Read()){
16.     // exploiter ligne courante
17.     ...
18.  }finally{
19.     // fermeture reader
20.     reader.Close();
21.  }
22. }catch(Exception ex){
23.  // gestion exception
24.  ...
25. }finally{
26.  // fermeture connexion
27.  connexion.Close();
28. }
29. ...
```

Quoiqu'il arrive, les objets *IDataReader* et *IDbConnection* doivent être fermés. C'est pourquoi cette fermeture est faite dans les clauses *finally*.

La fermeture de la connexion et celle du de l'objet *IDataReader* peuvent être automatisées avec une clause **using** :

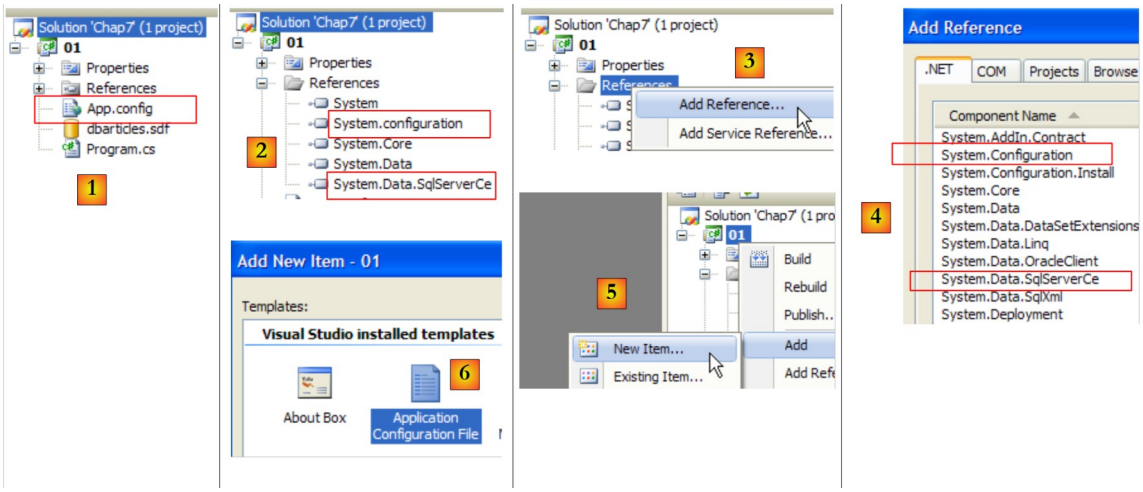
```
1.  // exploitation connexion
2.  try{
3.   using(IDbConnection connexion=...){
4.      // ouverture
5.      connexion.Open();
6.      // préparation commande
7.      IDbCommand commande=new Command();
8.      commande.Connection=connexion;
9.      // exécution ordre select
10.     commande.CommandText="select ...";
11.     using(IDbDataReader reader=commande.ExecuteReader()){
```

```
12.         // exploitation résultats
13.         while(reader.Read()){
14.         // exploiter ligne courante
15.         ...
16.     }// using IData
17.  }//using IDbConnection
18. }catch(Exception ex){
19.  // gestion exception
20.  ...
21. }
22. ..
```

- Ligne 3, la clause **using** nous assure que la connexion ouverte dans le bloc **using(...){...}** sera fermée en-dehors de celui-ci, ceci quelque soit la façon dont on sort du bloc : normalement ou par l'arrivée d'une exception. On économise un *finally*, mais l'intérêt n'est pas dans cette économie mineure. L'utilisation d'un *using* évite au développeur de fermer lui-même la connexion. Or oublier de fermer une connexion peut passer inaperçu et "planter" l'application d'une façon qui apparaîtra aléatoire, à chaque fois que le SGBD atteindra le nombre maximum de connexions ouvertes qu'il peut supporter.
- Ligne 11 : on procède de façon analogue pour fermer l'objet *IDataReader*.

9.3.5 Configuration du projet exemple

Le projet final sera le suivant :



- [1] : le projet aura un fichier de configuration [App.config]
- [2] : il utilise des classes de deux DLL non référencées par défaut et qu'il faut don ajouter aux références du projet :
  - [System.Configuration] pour exploiter le fichier de configuration [App.config]
  - [System.Data.SqlServerCe] pour exploiter la base de données Sql Server Compact
- [3, 4] : rappellent comment ajouter des références à un projet.
- [5, 6] : rappellent comment ajouter le fichier [App.config] à un projet.

Le fichier de configuration [App.config] sera le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.   <connectionStrings>
4.    <add name="dbSqlServerCe" connectionString="Data Source=|DataDirectory|
    \dbarticles.sdf;Password=dbarticles;" />
5.   </connectionStrings>
6.  </configuration>
```

- lignes 3-5 : la balise <**connectionStrings**> au pluriel définit des chaînes de connexion à des bases de données. Une chaîne de connexion à la forme "paramètre1=valeur1;paramètre2=valeur2;...". Elle définit tous les paramètres nécessaires à l'établissement d'une connexion avec une base de données particulière. Ces chaînes de connexion changent avec chaque SGBD. Le site [**http://www.connectionstrings.com/**] donne la forme de celles-ci pour les principaux SGBD.
- ligne 4 : définit une chaîne de connexion particulière, ici celle de la base SQL Server Compact *dbarticles.sdf* que nous avons créée précédemment :
  - **name** = nom de la chaîne de connexion. C'est via ce nom qu'une chaîne de connexion est récupérée par le programme C#
  - **connectionString** : la chaîne de connexion pour une base SQL Server Compact

- **DataSource** : désigne le chemin de la base. La syntaxe |DataDirectory| désigne le dossier d'exécution du projet.
- **Password** : le mot de passe de la base. Ce paramètre est absent s'il n'y a pas de mot de passe.

Le code C# pour récupérer la chaîne de connexion précédente est le suivant :

```
string connectionString = ConfigurationManager.ConnectionStrings["dbSqlServerCe"].ConnectionString;
```

- **ConfigurationManager** est la classe de la DLL [System.Configuration] qui permet d'exploiter le fichier [App.config].
- **ConnectionsStrings["nom"].ConnectionString** : désigne l'attribut *connectionString* de la balise < *add name="nom" connectionString="..."*> de la section <*connectionStrings*> de [App.config]

Le projet est désormais configuré. Nous étudions maintenant la classe [Program.cs] dont nous avons vu précédemment un exemple d'exécution.

9.3.6 Le programme exemple

Le programme [program.cs] est le suivant :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Data.SqlServerCe;
4.  using System.Text;
5.  using System.Text.RegularExpressions;
6.  using System.Configuration;
7.
8.  namespace Chap7 {
9.   class SqlCommands {
10.     static void Main(string[] args) {
11.
12.         // application console - exécute des requêtes SQL tapées au clavier
13.         // sur une base de données dont la chaîne de connexion est obtenue dans un fichier de
    configuration
14.
15.         // exploitation du fichier de configuration [App.config]
16.         string connectionString = null;
17.         try {
18.            connectionString =
    ConfigurationManager.ConnectionStrings["dbSqlServerCe"].ConnectionString;
19.         } catch (Exception e) {
20.            Console.WriteLine("Erreur de configuration : {0}", e.Message);
21.            return;
22.         }
23.
24.         // affichage chaîne de connexion
25.         Console.WriteLine("Chaîne de connexion à la base : [{0}]\n", connectionString);
26.
27.         // on construit un dictionnaire des commandes sql acceptées
28.         string[] commandesSQL = new string[] { "select", "insert", "update", "delete" };
29.         Dictionary<string, bool> dicoCommandes = new Dictionary<string, bool>();
30.         for (int i = 0; i < commandesSQL.Length; i++) {
31.            dicoCommandes.Add(commandesSQL[i], true);
32.         }
33.
34.         // lecture-exécution des commandes SQL tapées au clavier
35.         string requête = null;  // texte de la requête SQL
36.         string[] champs;        // les champs de la requête
37.         Regex modèle = new Regex(@"\s+");   // suite d'espaces
38.
39.         // boucle de saisie-exécution des commandes SQL tapées au clavier
40.         while (true) {
41.            // demande de la requête
42.            Console.Write("\nRequête SQL (rien pour arrêter) : ");
43.            requête = Console.ReadLine().Trim().ToLower();
44.            // fini ?
45.            if (requête == "")
46.               break;
47.            // on décompose la requête en champs
48.            champs = modèle.Split(requête);
49.            // requête valide ?
50.            if (champs.Length == 0 || ! dicoCommandes.ContainsKey(champs[0])) {
51.               // msg d'erreur
52.               Console.WriteLine("Requête invalide. Utilisez select, insert, update, delete ou rien
    pour arrêter");
53.               // requête suivante
54.               continue;
```

```
55.           }
56.           // exécution de la requête
57.           if (champs[0] == "select") {
58.              ExecuteSelect(connectionString, requête);
59.           } else
60.              ExecuteUpdate(connectionString, requête);
61.        }
62.     }
63.
64.     // exécution d'une requête de mise à jour
65.     static void ExecuteUpdate(string connectionString, string requête) {
66.  ...
67.     }
68.
69.     // exécution d'une requête Select
70.     static void ExecuteSelect(string connectionString, string requête) {
71. ....
72.     }
73.  }
74. }
```

- lignes 1-6 : les espaces de nom utilisés dans l'application. La gestion d'une base de données SQL Server Compact nécessite l'espace de noms [System.Data.SqlServerCe] de la ligne 3. On a là une dépendance sur un espace de noms propriétaire à un SGBD. On peut en déduire que le programme devra être modifié si on change de SGBD.
- ligne 18 : la chaîne de connexion à la base est lue dans le fichier [App.config] et affichée ligne 25. Elle servira pour l'établissement d'une connexion avec la base de données.
- lignes 28-32 : un dictionnaire mémorisant les noms des quatre ordres SQL autorisés : select, insert, update, delete.
- lignes 40-62 : la boucle de saisie des ordres SQL tapés au clavier et leur exécution sur la base de données
- ligne 48 : la ligne tapée au clavier est décomposée en champs afin d'en connaître le premier terme qui doit être : select, insert, update, delete
- lignes 50-55 : si la requête est invalide, un message d'erreur est affiché et on passe à la requête suivante.
- lignes 57-61 : on exécute l'ordre SQL saisi. Cette exécution prend une forme différente selon que l'on a affaire à un ordre *select* ou à un ordre *insert*, *update*, *delete*. Dans le premier cas, l'ordre ramène des données de la base sans modifier celle-ci, dans le second il la met à jour sans ramener des données. Dans les deux cas, on délègue l'exécution à une méthode qui a besoin de deux paramètres :
  - la chaîne de connexion qui va lui permettre de se connecter à la base
  - l'ordre SQL à exécuter sur cette connexion

9.3.7 Exécution d'une requête SELECT

L'exécution d'ordres SQL nécessite les étapes suivantes :

1. Connexion à la base de données
2. Émission des ordres SQL vers la base
3. Traitement des résultats de l'ordre SQL
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. Les connexions ouvertes sont des ressources limitées d'un SGBD. Il faut les économiser. Aussi cherchera-t-on toujours à limiter la durée de vie d'une connexion ouverte. Dans l'exemple étudié, la connexion est fermée après chaque ordre SQL. Une nouvelle connexion est ouverte pour l'ordre SQL suivant. L'ouverture / fermeture d'une connexion est coûteuse. Pour diminuer ce coût, certains SGBD offrent la notion de **pools de connexions** ouvertes : lors de l'initialisation de l'application, N connexions sont ouvertes et sont affectées au pool. Elles resteront ouvertes jusqu'à la fin de l'application. Lorsque l'application ouvre une connexion, elle reçoit l'une des N connexions déjà ouvertes du pool. Lorsqu'elle ferme la connexion, celle-ci est simplement remise dans le pool. L'intérêt de ce système est qu'il est transparent pour le développeur : le programme n'a pas à être modifié pour utiliser le pool de connexions. La configuration du pool de connexions est dépendant du SGBD.

Nous nous intéressons tout d'abord à l'exécution des ordres SQL *Select*. La méthode *ExecuteSelect* de notre programme exemple est la suivante :

```
1.  // exécution d'une requête Select
2.        static void ExecuteSelect(string connectionString, string requête) {
3.           // on gère les éventuelles exceptions
4.           try {
5.              using (SqlCeConnection connexion = new SqlCeConnection(connectionString)) {
6.                 // ouverture connexion
7.                 connexion.Open();
8.                 // exécute sqlCommand avec requête select
9.                 SqlCeCommand sqlCommand = new SqlCeCommand(requête, connexion);
10.                SqlCeDataReader reader= sqlCommand.ExecuteReader();
```

```
11.                // affichage résultats
12.                AfficheReader(reader);
13.             }
14.          } catch (Exception ex) {
15.             // msg d'erreur
16.             Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
17.          }
18.       }
19.
20.       // affichage reader
21.       static void AfficheReader(IDataReader reader) {
22. ...
23.       }
```

- ligne 2 : la méthode reçoit deux paramètres :
  - la chaîne de connexion [connectionString] qui va lui permettre de se connecter à la base
  - l'ordre SQL *Select* [requête] à exécuter sur cette connexion
- ligne 4 : toute opération avec une base de données peut générer une exception qu'on peut vouloir gérer. C'est d'autant plus important ici que les ordres SQL donnés par l'utilisateur peuvent être syntaxiquement erronés. Il faut qu'on puisse le lui dire. Tout le code est donc à l'intérieur d'un *try / catch*.
- ligne 5 : il y a plusieurs choses ici :
  - la connexion avec la base est initialisée avec la chaîne de connexion [connectionString]. Elle n'est pas encore ouverte. Elle le sera ligne 7.
  - la clause **using (Ressource) {...}** est une facilité syntaxique garantissant la libération de la ressource *Ressource*, ici une connexion, à la sortie du bloc contrôlé par le *using*.
  - la connexion est d'un type propriétaire : *SqlCeConnection*, propre au SGBD SQL Server Compact.
- ligne 7 : la connexion est ouverte. C'est à ce moment que les paramètres de la chaîne de connexion sont utilisés.
- ligne 9 : un ordre SQL est émis via un objet propriétaire *SqlCeCommand*. La ligne 9 initialise cet objet avec deux informations : la connexion à utiliser et l'ordre SQL à émettre dessus. L'objet *SqlCeCommand* sert aussi bien à exécuter un ordre *Select* qu'un ordre *Update*, *Insert*, *Delete*. Ses propriétés et méthodes ont été présentées page 225.
- ligne 10 : un ordre SQL *Select* est exécuté via la méthode *ExecuteReader* de l'objet *SqlCeCommand* qui rend un objet *IDataReader* dont a présenté les méthodes et propriétés page 226.
- ligne 12 : l'affichage des résultats est confiée à la méthode *AfficheReader* suivante :

```
1.        // affichage reader
2.        static void AfficheReader(IDataReader reader) {
3.           using (reader) {
4.              // exploitation des résultats
5.              // -- colonnes
6.              StringBuilder ligne = new StringBuilder();
7.              int i;
8.              for (i = 0; i < reader.FieldCount - 1; i++) {
9.                 ligne.Append(reader.GetName(i)).Append(",");
10.             }
11.             ligne.Append(reader.GetName(i));
12.             Console.WriteLine("\n{0}\n{1}\n{2}\n", "".PadLeft(ligne.Length, '-'), ligne,
    "".PadLeft(ligne.Length, '-'));
13.             // -- données
14.             while (reader.Read()) {
15.                // exploitation ligne courante
16.                ligne = new StringBuilder();
17.                for (i = 0; i < reader.FieldCount; i++) {
18.                   ligne.Append(reader[i].ToString()).Append(" ");
19.                }
20.                Console.WriteLine(ligne);
21.             }
22.          }
23. }
```

- ligne 2 : la méthode reçoit un objet *IDataReader*. On notera qu'ici nous avons utilisé une interface et non une classe spécifique.
- ligne 3 : la clause *using* est utilisée pour gérer de façon automatique la fermeture de l'objet *IDataReader*.
- lignes 8-10 : on affiche les noms des colonnes de la table résultat du *Select*. Ce sont les colonnes *coli* de la requête *select col1, col2, ... from table ...*
- lignes 14-21 : on parcourt la table des résultats et on affiche les valeurs de chaque ligne de la table.
- ligne 18 : on ne connaît pas le type de la colonne n° i du résultat parce qu'on ne connaît pas la table interrogée. On ne peut donc utiliser la syntaxe *reader.GetXXX(i)* où *XXX* est le type de la colonne n° i, car on ne connaît pas ce type. On utilise alors la syntaxe *reader.Item[i].ToString()* pour avoir la représentation de la colonne n° i sous forme de chaîne de caractères. La syntaxe *reader.Item[i].ToString()* peut être abrégée en *reader[i].ToString()*.

9.3.8 Exécution d'un ordre de mise à jour : INSERT, UPDATE, DELETE

Le code de la méthode *ExecuteUpdate* est le suivant :

```
1.        // exécution d'une requête de mise à jour
2.        static void ExecuteUpdate(string connectionString, string requête) {
3.           // on gère les éventuelles exceptions
4.           try {
5.              using (SqlCeConnection connexion = new SqlCeConnection(connectionString)) {
6.                 // ouverture connexion
7.                 connexion.Open();
8.                 // exécute sqlCommand avec requête de mise à jour
9.                 SqlCeCommand sqlCommand = new SqlCeCommand(requête, connexion);
10.                int nbLignes = sqlCommand.ExecuteNonQuery();
11.                // affichage résultat
12.                Console.WriteLine("Il y a eu {0} ligne(s) modifiée(s)", nbLignes);
13.             }
14.          } catch (Exception ex) {
15.             // msg d'erreur
16.             Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
17.          }
18. }
```

Nous avons dit que l'exécution d'un ordre d'interrogation *Select* ne différait de celle d'un ordre de mise à jour *Update*, *Insert*, *Delete* que par la méthode de l'objet *SqlCeCommand* utilisée : **ExecuteReader** pour *Select*, **ExecuteNonQuery** pour *Update*, *Insert*, *Delete*. Nous ne commentons que cette dernière méthode dans le code ci-dessus :

- ligne 10 : l'ordre *Update*, *Insert*, *Delete* est exécuté par la méthode *ExecuteNonQuery* de l'objet *SqlCeCommand*. Si elle réussit, cette méthode rend le nombre de lignes mises à jour (update) ou insérées (insert) ou détruites (delete).
- ligne 12 : ce nombre de lignes est affiché à l'écran

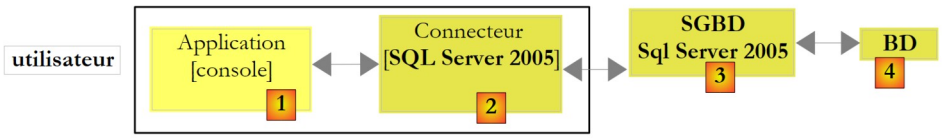
Le lecteur est invité à revoir un exemple d'exécution de ce code, page 224.

9.4 Autres connecteurs ADO.NET

Le code que nous avons étudié est propriétaire : il dépend de l'espace de noms [System.Data.SqlServerCe] destiné au SGBD SQL Server Compact. Nous allons maintenant construire le même programme avec différents connecteurs .NET et voir ce qui change.

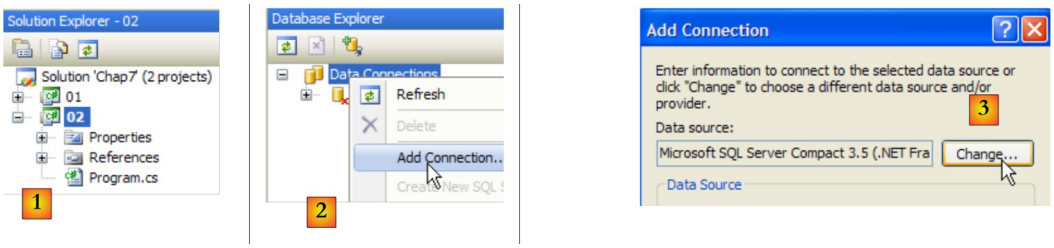
9.4.1 Connecteur SQL Server 2005

L'architecture utilisée sera la suivante :



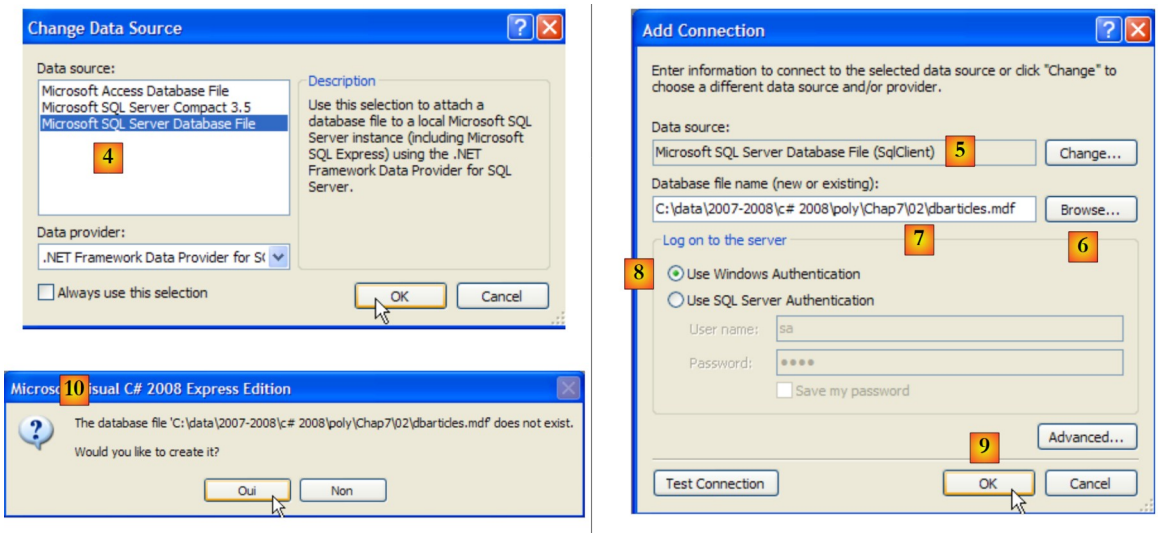
L'installation de SQL Server 2005 est décrite en annexes au paragraphe 1.1, page 427.

Nous créons un second projet dans la même solution que précédemment puis nous créons la base de données SQL server 2005. Le SGBD SQL Server 2005 doit être lancé avant les opérations qui suivent :

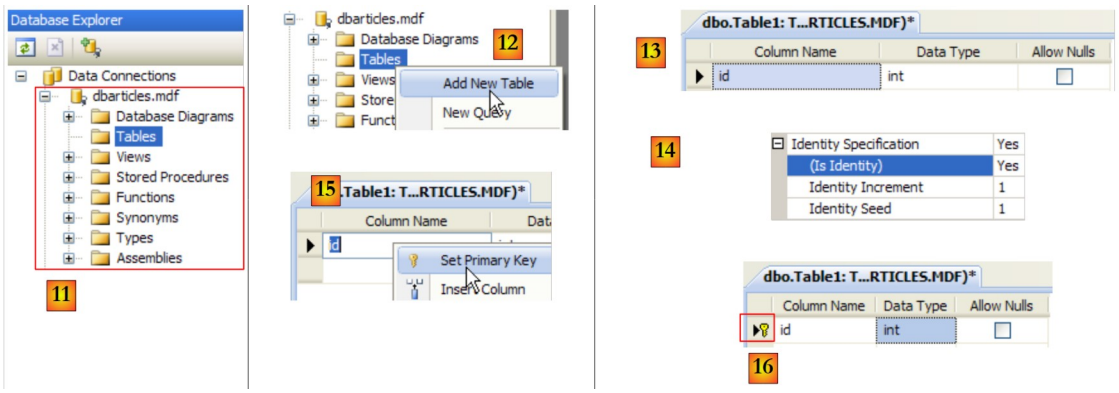


- [1] : créer un nouveau projet dans la solution actuelle et en faire le projet courant.
- [2] : créer une nouvelle connexion

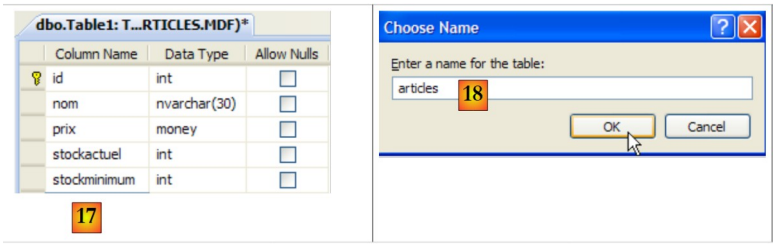
- [3] : choisir le type de connexion



- [4] : choisir le SGBD SQL Server
- [5] : résultat du choix précédent
- [6] : utiliser le bouton [Browse] pour indiquer où créer la base SQL Server 2005. La base de données est encapsulée dans un fichier .mdf.
- [7] : choisir la racine du nouveau projet et appeler la base [dbarticles.mdf].
- [8] : utiliser une authentification windows.
- [9] : valider la page de renseignements

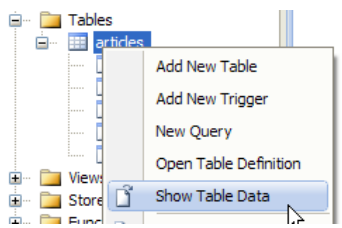


- [11] : la base SQL Server
- [12] : créer une table. Celle-ci sera identique à la base SQL Server Compact construite précédemment.
- [13] : le champ [id]
- [14] : le champ [id] est de type *Identity*.
- [15,16] : le champ [id] est clé primaire



- [17] : les autres champs de la table
- [18] : donner le nom [articles] à la table au moment de sa sauvegarde (Ctrl+S).

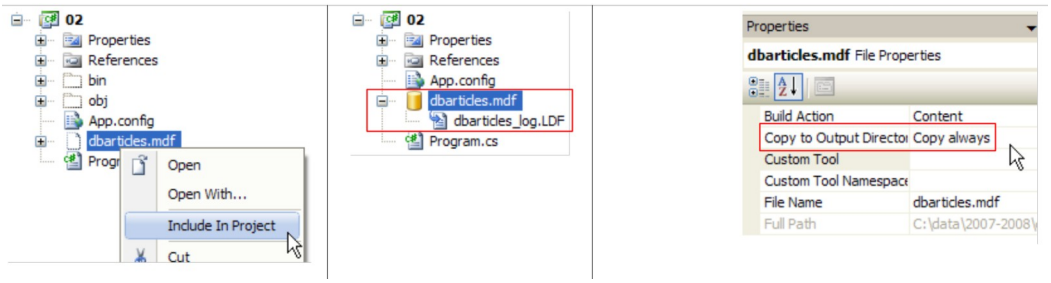
Il nous reste à mettre des données dans la table :



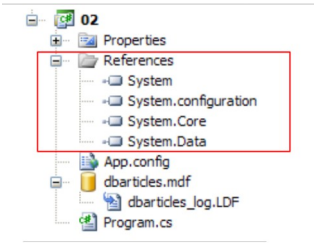
articles: Que...BARTICLES.MDF)

id	nom	prix	stockactuel	stockminimum
1	vélo	500,0000	10	5
2	pompe	10,0000	10	2
3	arc	610,0000	4	1
4	flèches - lot de 6	100,0000	12	20
5	combinaison de plongée	300,0000	8	2
6	Bouteilles d'oxygène	120,0000	10	5

Nous incluons la base de données dans le projet :



Les références du projet sont les suivantes :



Le fichier de configuration [App.config] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.   <connectionStrings>
4.    <add name="connectString1" connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|
    DataDirectory|\dbarticles.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True;" />
5.    <add name="connectString2" connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|
    DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;Connect Timeout=30;" />
6.   </connectionStrings>
7.  </configuration>
```

- ligne 4 : la chaîne de connexion à la base [dbarticles.mdf] avec une authentification windows
- ligne 5 : la chaîne de connexion à la base [dbarticles.mdf] avec une authentification SQL Server. [sa,msde] est le couple (login,password) de l'administrateur du serveur SQL Server tel que défini au paragraphe 1.1, page 427.

Le programme [Program.cs] évolue de la façon suivante :

```
1.  using System.Data.SqlClient;
2.  ...
3.
4.  namespace Chap7 {
5.   class SqlCommands {
6.      static void Main(string[] args) {
7.  ...
8.         // exploitation du fichier de configuration [App.config]
9.         string connectionString = null;
10.        try {
```

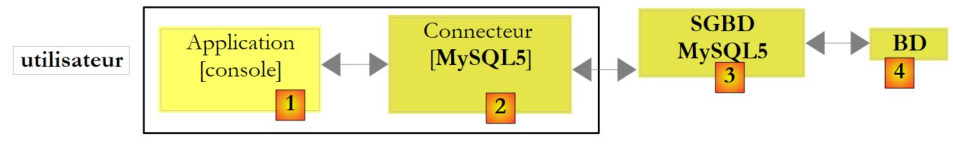
```
11.           connectionString =
    ConfigurationManager.ConnectionStrings["connectString2"].ConnectionString;
12.       } catch (Exception e) {
13. ...
14.       }
15. ...
16.       // lecture-exécution des commandes SQL tapées au clavier
17. ...
18.    }
19.
20.    // exécution d'une requête de mise à jour
21.    static void ExecuteUpdate(string connectionString, string requête) {
22.       // on gère les éventuelles exceptions
23.       try {
24.          using (SqlConnection connexion = new SqlConnection(connectionString)) {
25.             // ouverture connexion
26.             connexion.Open();
27.             // exécute sqlCommand avec requête de mise à jour
28.             SqlCommand sqlCommand = new SqlCommand(requête, connexion);
29.             int nbLignes = sqlCommand.ExecuteNonQuery();
30.             // affichage résultat
31.             Console.WriteLine("Il y a eu {0} ligne(s) modifiée(s)", nbLignes);
32.          }
33.       } catch (Exception ex) {
34. ....
35.       }
36.    }
37.
38.    // exécution d'une requête Select
39.    static void ExecuteSelect(string connectionString, string requête) {
40.       // on gère les éventuelles exceptions
41.       try {
42.          using (SqlConnection connexion = new SqlConnection(connectionString)) {
43.             // ouverture connexion
44.             connexion.Open();
45.             // exécute sqlCommand avec requête select
46.             SqlCommand sqlCommand = new SqlCommand(requête, connexion);
47.             SqlDataReader reader = sqlCommand.ExecuteReader();
48.             // exploitation des résultats
49. ...
50.          }
51.       } catch (Exception ex) {
52. ...
53.       }
54.    }
55. }
56. }
```

- ligne 1 : l'espace de noms [**System.Data.SqlClient**] contient les classes permettant de gérer une base SQL Server 2005
- ligne 24 : la connexion est de type **SQLConnection**
- ligne 28 : l'objet encapsulant les ordres SQL est de type **SQLCommand**
- ligne 47 : l'objet encapsulant le résultat d'un ordre SQL Select est de type **SQLDataReader**

Le code est identique à celui utilisé avec le SGBD SQL Server Compact au nom des classes près. Pour l'exécuter, on peut utiliser (ligne 11) l'une ou l'autre des deux chaînes de connexion définies dans [App.config].

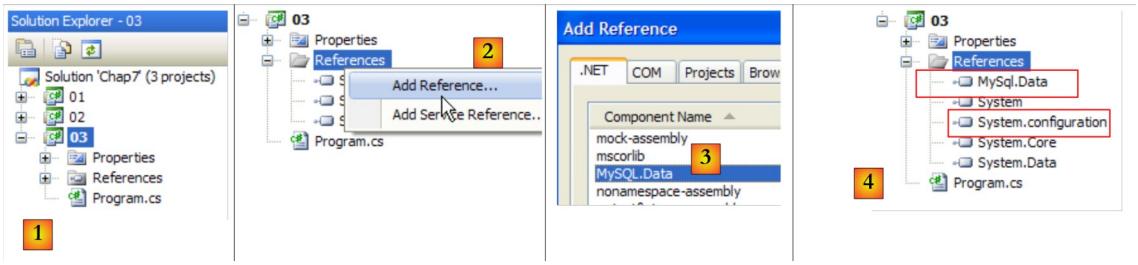
9.4.2 Connecteur MySQL5

L'architecture utilisée sera la suivante :



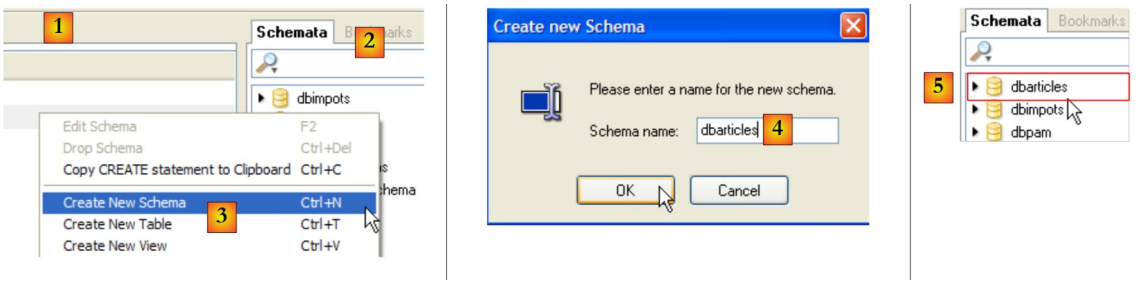
L'installation de MySQL5 est décrite en annexes au paragraphe 1.2, page 433 et celle du connecteur Ado.Net au paragraphe 1.2.5, page 442.

Nous créons un troisième projet dans la même solution que précédemment et nous lui ajoutons les références dont il a besoin :



- [1] : le nouveau projet
- [2] : auquel on ajoute des références
- [3] : la DLL [MySQL.Data] du connecteur Ado.Net de MySql5 ainsi que celle de [System.Configuration] [4].

Nous créons maintenant la base de données [dbarticles] et sa table [articles]. Le SGBD MySQL5 doit être lancé. Par ailleurs, on lance le client [Query Browser] (cf paragraphe 1.2.3, page 437).

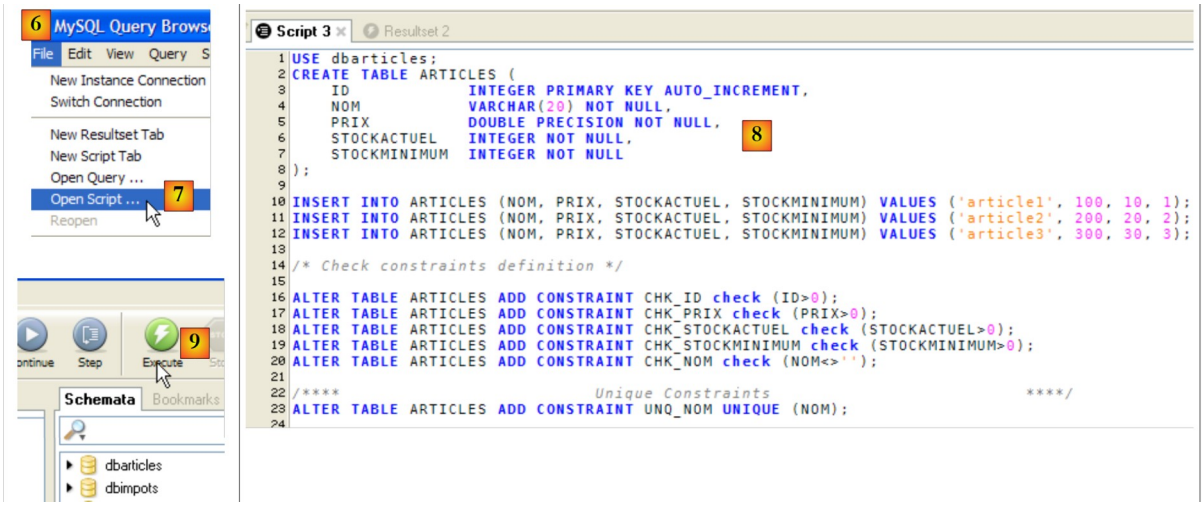


- [1] : dans [Query Browser], cliquer droit dans la zone [Schemata] [2] pour créer [3] un nouveau schéma, terme qui désigne une base de données.
- [4] : la base de données s'appellera [dbarticles]. En [5], on la voit. Elle est pour l'instant sans tables. Nous allons exécuter le script SQL suivant :

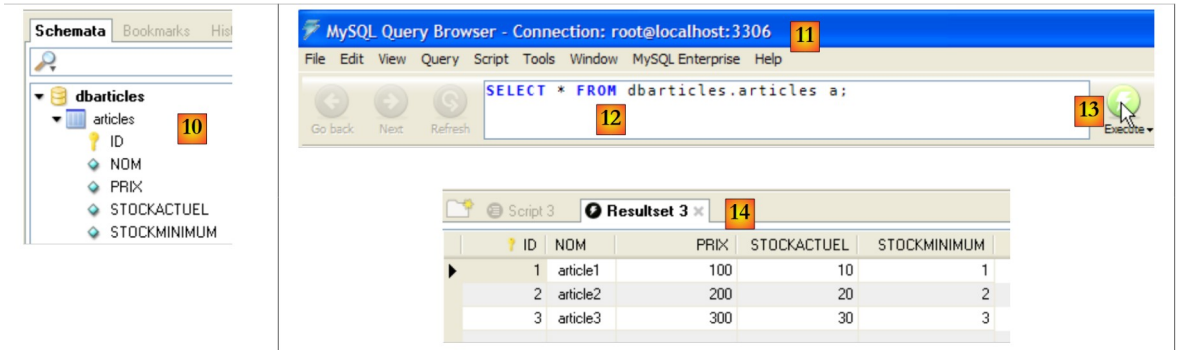
```
1.  /* choix de la base de données courante */
2.  USE dbarticles;
3.  /* création de la table des articles */
4.  CREATE TABLE ARTICLES (
5.      ID            INTEGER PRIMARY KEY AUTO_INCREMENT,
6.      NOM           VARCHAR(20) NOT NULL,
7.      PRIX          DOUBLE PRECISION NOT NULL,
8.      STOCKACTUEL   INTEGER NOT NULL,
9.      STOCKMINIMUM  INTEGER NOT NULL
10. );
11. /* insertion de données dans la table */
12. INSERT INTO ARTICLES (NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES ('article1', 100, 10, 1);
13. INSERT INTO ARTICLES (NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES ('article2', 200, 20, 2);
14. INSERT INTO ARTICLES (NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES ('article3', 300, 30, 3);
15. /* ajout de contraintes */
16. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID check (ID>0);
17. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX check (PRIX>0);
18. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL check (STOCKACTUEL>0);
19. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM check (STOCKMINIMUM>0);
20. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM check (NOM<>'');
21. ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
```

- ligne 1 : la base [dbarticles] devient la base courante. Les ordres SQL qui suivent s'exécuteront sur elle.
- lignes 4-10 : définition de la table [ARTICLES]. On notera le SQL propriétaire de MySQL. Les types des colonnes, la génération automatique de la clé primaire (attribut AUTO_INCREMENT) diffèrent de ce qui a été rencontré avec les SGBD SQL Server Compact et Express.
- lignes 12-14 : insertion de trois lignes
- lignes 16-21 : ajout de contraintes d'intégrité sur les colonnes.

Ce script est exécuté dans [MySQL Query Browser] :

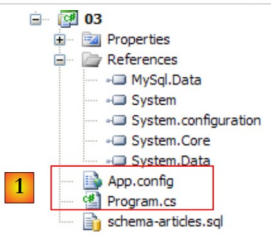


- dans [MySQL Query Browser] [6], on charge le script [7]. On le voit en [8]. En [9], il est exécuté.



- en [10], la table [articles] a été créée. On double-clique dessus. Cela fait apparaître la fenêtre [11] avec la requête [12] dedans prête à être exécutée par [13]. En [14], le résultat de l'exécution. On a bien les trois lignes attendues. On notera que les valeurs du champ [ID] ont été générées automatiquement (attribut AUTO_INCREMENT du champ).

Maintenant que la base de données est prête, nous pouvons revenir au développement de l'application dans Visual studio.



En [1], le programme [Program.cs] et le fichier de configuration [App.config]. Celui-ci est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.   <connectionStrings>
4.    <add name="dbArticlesMySql5"
    connectionString="Server=localhost;Database=dbarticles;Uid=root;Pwd=root;" />
5.   </connectionStrings>
6.  </configuration>
```

Ligne 4, les éléments de la chaîne de connexion sont les suivants :

- **Server** : nom de la machine sur laquelle se trouve le SGBD MySQL, ici *localhost*, c.a.d. la machine sur laquelle va être exécutée le programme.
- **Database** : le nom de la base de données gérée, ici *dbarticles*
- **Uid** : le login de l'utilisateur, ici *root*

- **Pwd** : son mot de passe, ici *root*. Ces deux informations désignent l'administrateur créé au paragraphe 1.2, page 433.

Le programme [Program.cs] est identique à celui des versions précédentes aux détails près suivants :

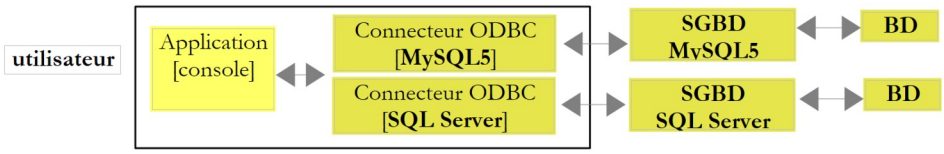
espace de noms	MySql.Data.MySqlClient
classe Connection	MySqlConnection
classe Command	MySqlCommand
classe DataReader	MySqlDataReader

Le programme utilise la chaîne de connexion nommée *dbArticlesMySql5* dans le fichier [App.config]. L'exécution donne les résultats suivants :

```
1.  Chaîne de connexion à la base : [Server=localhost;Database=dbarticles;Uid=root;Pwd=root;]
2.
3.  Requête SQL (rien pour arrêter) : select * from articles
4.
5.  -----------------------------------
6.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
7.  -----------------------------------
8.
9.  1 article1 100 10 1
10. 2 article2 200 20 2
11. 3 article3 300 30 3
```

9.4.3 Connecteur ODBC

L'architecture utilisée sera la suivante :

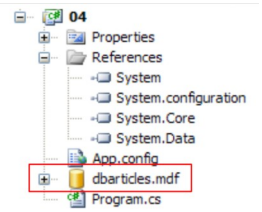


L'intérêt des connecteurs ODBC est qu'ils présentent une interface standard aux applications qui les utilisent. Ainsi la nouvelle application va-t-elle, avec un unique code, pouvoir dialoguer avec tout SGBD ayant un connecteur ODBC, c.a.d. la plupart des SGBD. Les performances des connecteurs ODBC sont moins bonnes que celles des connecteurs "propriétaires" qui savent exploiter toutes les caractéristiques d'un SGBD particulier. En contre-partie, on obtient une grande souplesse de l'application : on peut changer de SGBD sans changer le code.

Nous étudions un exemple où l'application exploite une base MySQL5 ou une base SQL server Express selon la chaîne de connexion qu'on lui donne. Dans ce qui suit, nous supposons que :

- les SGBD SQL Server Express et MySQL5 ont été lancés
- que le pilote ODBC de MySQL5 est présent sur la machine (cf paragraphe 1.2.6, page 443). Celui de SQL Server 2005 est présent par défaut.
- les bases de données utilisées sont celles du paragraphe 9.4.2, page 235 pour la base MySQL5, celle du paragraphe 9.4.1, page 232 pour la base SQL Server Express.

Le nouveau projet Visual studio est le suivant :



Ci-dessus, la base SQL Server [dbarticles.mdf] créée au paragraphe 9.4.1, page 232 a été recopiée dans le dossier du projet.

Le fichier de configuration [App.config] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
```

```
2.  <configuration>
3.   <connectionStrings>
4.    <add name="dbArticlesOdbcMySql5" connectionString="Driver={MySQL ODBC 3.51
    Driver};Server=localhost;Database=dbarticles; User=root;Password=root;" />
5.    <add name="dbArticlesOdbcSqlServer2005" connectionString="Driver={SQL Native Client};Server=.
    \SQLExpress;AttachDbFilename=|DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;" />
6.   </connectionStrings>
7.  </configuration>
```

- ligne 4 : la chaîne de connexion de la source ODBC MySQL5. C'est une chaîne déjà étudiée dans laquelle on trouve un nouveau paramètre **Driver** qui définit le pilote ODBC à utiliser.
- ligne 5 : la chaîne de connexion de la source ODBC SQL Server Express. C'est la chaîne déjà utilisée dans un exemple précédent à laquelle le paramètre **Driver** a été ajouté.

Le programme [Program.cs] est identique à celui des versions précédentes aux détails près suivants :

`espace de noms`	System.Data.Odbc
`classe Connection`	OdbcConnection
`classe Command`	OdbcCommand
`classe DataReader`	OdbcDataReader

Le programme utilise l'une des deux chaînes de connexion définies dans le fichier [App.config]. L'exécution donne les résultats suivants :

Avec la chaîne de connexion [dbArticlesOdbcSqlServer2005] :

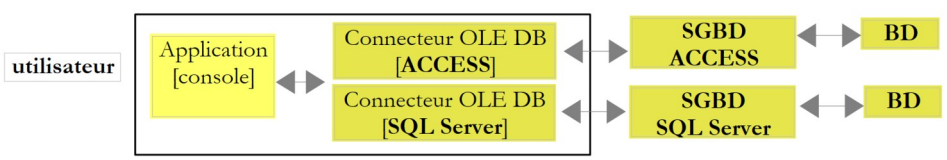
```
1.  Chaîne de connexion à la base : [Driver={SQL Native Client};Server=.\SQLExpress;AttachDbFilename=|
    DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;]
2.
3.  Requête SQL (rien pour arrêter) : select * from articles
4.
5.  -----------------------------------
6.  id,nom,prix,stockactuel,stockminimum
7.  -----------------------------------
8.
9.  1 vélo 500,0000 10 5
10. 2 pompe 10,0000 10 2
11. 3 arc 610,0000 4 1
12. 4 flèches - lot de 6 100,0000 12 20
13. 5 combinaison de plongée 300,0000 8 2
14. 6 Bouteilles d'oxygène 120,0000 10 5
```

Avec la chaîne de connexion [dbArticlesOdbcMySql5] :

```
1.  Chaîne de connexion à la base : [Driver={MySQL ODBC 3.51
    Driver};Server=localhost;Database=dbarticles; User=root;Password=root;]
2.
3.  Requête SQL (rien pour arrêter) : select * from articles
4.
5.  -----------------------------------
6.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
7.  -----------------------------------
8.
9.  1 article1 100 10 1
10. 2 article2 200 20 2
11. 3 article3 300 30 3
```

9.4.4 Connecteur OLE DB

L'architecture utilisée sera la suivante :



Comme les connecteurs ODBC, les connecteurs OLE DB (Object Linking and Embedding DataBase) présentent une interface standard aux applications qui les utilisent. Les pilotes ODBC permettent l'accès à des bases de données. Les sources de données

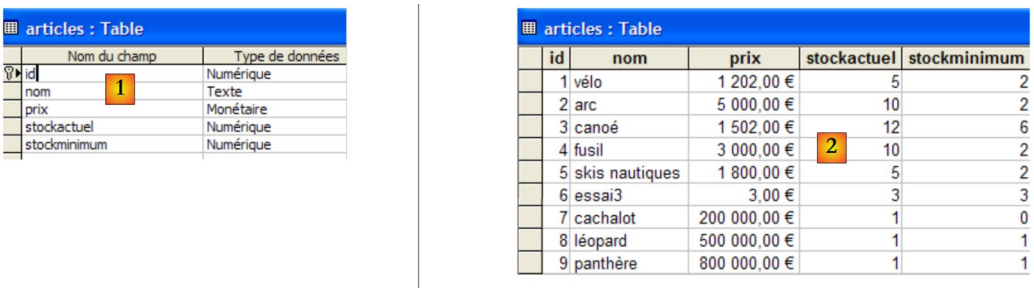
pour les pilotes OLE DB sont plus variées : bases de données, messageries, annuaires, ... Toute source de données peut faire l'objet d'un pilote Ole DB si un éditeur le décide. On a ainsi un accès standard à une grande variété de données.

Nous étudions un exemple où l'application exploite une base ACCESS ou une base SQL server Express selon la chaîne de connexion qu'on lui donne. Dans ce qui suit, nous supposons que le SGBD SQL Server Express a été lancé et que la base de données utilisée est celle de l'exemple précédent.

Le nouveau projet Visual studio est le suivant :

- en [1] : l'espace de noms nécessaire aux connecteurs OLE DB est [System.Data.OleDb] présent dans la référence [System.Data] ci-dessus. La base SQL Server [dbarticles.mdf] a été recopiée à partir du projet précédent. La base [dbarticles.mdb] a été créée avec Access.
- en [2] : comme la base SQL Server, la base ACCESS a la propriété [Copy to Output Directory=Copy Always] afin qu'elle soit automatiquement recopiée dans le dossier d'exécution du projet.

La base de données ACCESS [dbarticles.mdb] est la suivante :



En [1], la structure de la table [articles] et en [2] son contenu.

Le fichier de configuration [App.config] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.    <connectionStrings>
4.     <add name="dbArticlesOleDbAccess"  connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data
    Source=|DataDirectory|\dbarticles.mdb;"/>
5.     <add name="dbArticlesOleDbSqlServer2005" connectionString="Provider=SQLNCLI;Server=.
    \SQLEXPRESS;AttachDbFilename=|DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;" />
6.    </connectionStrings>
7.  </configuration>
```

- ligne 4 : la chaîne de connexion de la source OLE DB ACCESS. On y trouve le paramètre **Provider** qui définit le pilote OLE DB à utiliser ainsi que le chemin de la base de données
- ligne 5 : la chaîne de connexion de la source OLE DB Server Express.

Le programme [Program.cs] est identique à celui des versions précédentes aux détails près suivants :

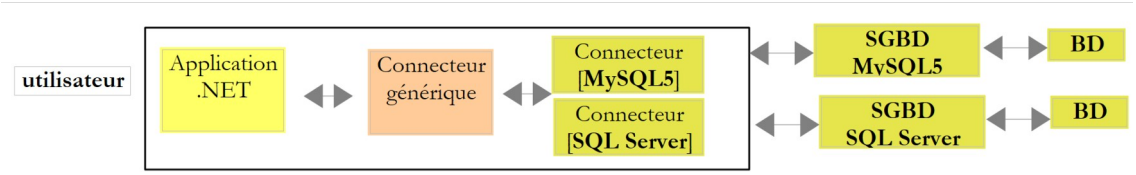
`espace de noms`	System.Data.OleDb
`classe Connection`	OleDbConnection
`classe Command`	OleDbCommand
`classe DataReader`	OleDbDataReader

Le programme utilise l'une des deux chaînes de connexion définies dans le fichier [App.config]. L'exécution donne les résultats suivants avec la chaîne de connexion [dbArticlesOleDbAccess] :

```
1.  Chaîne de connexion à la base : [Provider=Microsoft.Jet.OLEDB.4.0;Data Source=|DataDirectory|
    \dbarticles.mdb;]
2.
3.  Requête SQL (rien pour arrêter) : select * from articles
4.
5.  -----------------------------------
6.  id,nom,prix,stockactuel,stockminimum
7.  -----------------------------------
8.
9.  1 vélo 1202 5 2
10. 2 arc 5000 10 2
11. 3 canoé 1502 12 6
12. 4 fusil 3000 10 2
13. 5 skis nautiques 1800 5 2
14. 6 essai3 3 3 3
15. 7 cachalot 200000 1 0
16. 8 léopard 500000 1 1
17. 9 panthère 800000 1 1
```

9.4.5 Connecteur générique

L'architecture utilisée sera la suivante :



Comme les connecteurs ODBC et OLE DB, le connecteur générique présente une interface standard aux applications qui l'utilise mais améliore les performances sans sacrifier la souplesse. En effet, le connecteur générique s'appuie sur les connecteurs propriétaires des SGBD. L'application utilise des classes du connecteur générique. Ces classes servent d'intermédiaires entre l'application et le connecteur propriétaire.

Ci-dessus, lorsque l'application demande par exemple une connexion au connecteur générique, celui-ci lui rend une instance **IDbConnection**, l'interface des connexions décrite page 225, implémentée par une classe **MySQLConnection** ou **SQLConnection** selon la nature de la demande qui lui a été faite. On dit que le connecteur générique a des classes de type **factory** : on utilise une classe **factory** pour lui demander de créer des objets et en donner des références (pointeurs). D'où son nom (**factory**=usine, usine de production d'objets).

Il n'existe pas de connecteur générique pour tous les SGBD (avril 2008). Pour connaître ceux installés sur une machine, on pourra utiliser le programme suivant :

```
1.  using System;
2.  using System.Data;
3.  using System.Data.Common;
4.
5.  namespace Chap7 {
6.   class Providers {
7.      public static void Main() {
8.         DataTable dt = DbProviderFactories.GetFactoryClasses();
9.         foreach (DataColumn col in dt.Columns) {
10.           Console.Write("{0}|", col.ColumnName);
11.        }
12.        Console.WriteLine("\n".PadRight(40, '-'));
13.        foreach (DataRow row in dt.Rows) {
14.           foreach (object item in row.ItemArray) {
15.              Console.Write("{0}|", item);
16.           }
17.           Console.WriteLine("\n".PadRight(40, '-'));
18.        }
19.     }
20.  }
21. }
```

- ligne 8 : la méthode statique [DbProviderFactories.GetFactoryClasses()] rend la liste des connecteurs génériques installés, sous la forme d'une table de base de données placée en mémoire (DataTable).
- lignes 9-11 : affichent les noms des colonnes de la table *dt* :
  - *dt.Columns* est la liste des colonnes de la table. Une colonne C est de type *DataColumn*
  - *[DataColumn].ColumnName* est le nom de la colonne
- lignes 13-18 : affichent les lignes de la table *dt* :
  - *dt.Rows* est la liste des lignes de la table. Une ligne L est de type *DataRow*
  - *[DataRow].ItemArray* est un tableau d'objets ou chaque objet représente une colonne de la ligne

Le résultat de l'exécution sur ma machine est le suivant :

```
1.  Name|Description|InvariantName|AssemblyQualifiedName|
2.  -------------------------------------
3.  Odbc Data Provider|.Net Framework Data Provider for Odbc|System.Data.Odbc|
    System.Data.Odbc.OdbcFactory, System.Data, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089|
4.  -------------------------------------
5.  OleDb Data Provider|.Net Framework Data Provider for OleDb|System.Data.OleDb|
    System.Data.OleDb.OleDbFactory, System.Data, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089|
6.  -------------------------------------
7.  OracleClient Data Provider|.Net Framework Data Provider for Oracle|System.Data.OracleClient|
    System.Data.OracleClient.OracleClientFactory, System.Data.OracleClient, Version=2.0.0.0,
    Culture=neutral, PublicKeyToken=b77a5c561934e089|
8.  -------------------------------------
9.  SqlClient Data Provider|.Net Framework Data Provider for SqlServer|System.Data.SqlClient|
    System.Data.SqlClient.SqlClientFactory, System.Data, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089|
10. -------------------------------------
11. Microsoft SQL Server Compact Data Provider|.NET Framework Data Provider for Microsoft SQL Server
    Compact|System.Data.SqlServerCe.3.5|System.Data.SqlServerCe.SqlCeProviderFactory,
    System.Data.SqlServerCe, Version=3.5.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91|
12. -------------------------------------
13. MySQL Data Provider|.Net Framework Data Provider for MySQL|MySql.Data.MySqlClient|
    MySql.Data.MySqlClient.MySqlClientFactory, MySql.Data, Version=5.2.1.0, Culture=neutral,
    PublicKeyToken=c5687fc88969c44d|
```

- ligne 1 : la table a quatre colonnes. Les trois premières sont les plus utiles pour nous ici.

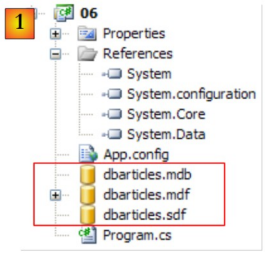
L'affichage suivant montre que l'on dispose des connecteurs génériques suivants :

Nom	Identifiant
Odbc Data Provider	System.Data.Odbc
OleDb Data Provider	System.Data.OleDb
OracleClient Data Provider	System.Data.OracleClient
SqlClient Data Provider	System.Data.SqlClient
Microsoft SQL Server Compact Data Provider	System.Data.SqlServerCe.3.5
MySQL Data Provider	MySql.Data.MySqlClient

Un connecteur générique est accessible dans un programme C# via son identifiant.

Nous étudions un exemple où l'application exploite les diverses bases de données que nous avons construites jusqu'à maintenant. L'application recevra deux paramètres :
- un premier paramètre précise le type de SGBD utilisé afin que la bonne bibliothèque de classes soit utilisée
- le second paramètre précise la base de données gérée, via une chaîne de connexion.

Le nouveau projet Visual studio est le suivant :



- en [1] : l'espace de noms nécessaire aux connecteurs génériques est [System.Data.common] présent dans la référence [System.Data].

Le fichier de configuration [App.config] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.   <connectionStrings>
4.      <add name="dbArticlesSqlServerCe" connectionString="Data Source=|DataDirectory|
    \dbarticles.sdf;Password=dbarticles;" />
5.      <add name="dbArticlesSqlServer" connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|
    DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;" />
6.      <add name="dbArticlesMySql5"
    connectionString="Server=localhost;Database=dbarticles;Uid=root;Pwd=root;" />
7.      <add name="dbArticlesOdbcMySql5" connectionString="Driver={MySQL ODBC 3.51
    Driver};Server=localhost;Database=dbarticles; User=root;Password=root;Option=3;" />
8.      <add name="dbArticlesOleDbSqlServer2005" connectionString="Provider=SQLNCLI;Server=.
    \SQLExpress;AttachDbFilename=|DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;" />
9.      <add name="dbArticlesOdbcSqlServer2005" connectionString="Driver={SQL Native Client};Server=.
    \SQLExpress;AttachDbFilename=|DataDirectory|\dbarticles.mdf;Uid=sa;Pwd=msde;" />
10.     <add name="dbArticlesOleDbAccess"  connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data
    Source=|DataDirectory|\dbarticles.mdb;Persist Security Info=True"/>
11.  </connectionStrings>
12.  <appSettings>
13.     <add key="factorySqlServerCe" value="System.Data.SqlServerCe.3.5"/>
14.     <add key="factoryMySql" value="MySql.Data.MySqlClient"/>
15.     <add key="factorySqlServer" value="System.Data.SqlClient"/>
16.     <add key="factoryOdbc" value="System.Data.Odbc"/>
17.     <add key="factoryOleDb" value="System.Data.OleDb"/>
18.  </appSettings>
19. </configuration>
```

- lignes 3-11 : les chaînes de connexion des diverses bases de données exploitées.
- lignes 13-17 : les noms des connecteurs génériques à utiliser

Le programme [Program.cs] est le suivant :

```
1.  ...
2.  using System.Data.Common;
3.
4.  namespace Chap7 {
5.   class SqlCommands {
6.      static void Main(string[] args) {
7.
8.         // application console - exécute des requêtes SQL tapées au clavier
9.         // sur une base de données dont la chaîne de connexion est obtenue dans un fichier de
    configuration ainsi que le nom du connecteur du SGBD associé
10.
11.        // vérification paramètres
12.        if (args.Length != 2) {
13.           Console.WriteLine("Syntaxe : pg factory connectionString");
14.           return;
15.        }
16.
17.        // exploitation du fichier de configuration
18.        string factory = null;
19.        string connectionString = null;
20.        DbProviderFactory connecteur = null;
21.        try {
22.           // factory
23.           factory = ConfigurationManager.AppSettings[args[0]];
24.           // chaîne de connexion
25.           connectionString = ConfigurationManager.ConnectionStrings[args[1]].ConnectionString;
26.           // on récupère un connecteur générique pour le SGBD
27.           connecteur = DbProviderFactories.GetFactory(factory);
```

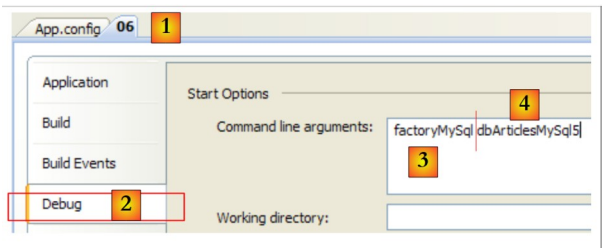
```
28.         } catch (Exception e) {
29.            Console.WriteLine("Erreur de configuration : {0}", e.Message);
30.            return;
31.         }
32.
33.         // affichages
34.         Console.WriteLine("Provider factory : [{0}]\n", factory);
35.         Console.WriteLine("Chaîne de connexion à la base : [{0}]\n", connectionString);
36.
37. ...
38.            // exécution de la requête
39.            if (champs[0] == "select") {
40.               ExecuteSelect(connecteur,connectionString, requête);
41.            } else
42.               ExecuteUpdate(connecteur, connectionString, requête);
43.         }
44.     }
45.
46.     // exécution d'une requête de mise à jour
47.     static void ExecuteUpdate(DbProviderFactory connecteur, string connectionString, string
    requête) {
48.         // on gère les éventuelles exceptions
49.         try {
50.            using (DbConnection connexion = connecteur.CreateConnection()) {
51.               // configuration connexion
52.               connexion.ConnectionString = connectionString;
53.               // ouverture connexion
54.               connexion.Open();
55.               // configuration Command
56.               DbCommand sqlCommand = connecteur.CreateCommand();
57.               sqlCommand.CommandText = requête;
58.               sqlCommand.Connection = connexion;
59.               // exécution requête
60.               int nbLignes = sqlCommand.ExecuteNonQuery();
61.               // affichage résultat
62.               Console.WriteLine("Il y a eu {0} ligne(s) modifiée(s)", nbLignes);
63.            }
64.         } catch (Exception ex) {
65.            // msg d'erreur
66.            Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
67.         }
68.     }
69.
70.     // exécution d'une requête Select
71.     static void ExecuteSelect(DbProviderFactory connecteur, string connectionString, string
    requête) {
72.         // on gère les éventuelles exceptions
73.         try {
74.            using (DbConnection connexion = connecteur.CreateConnection()) {
75.               // configuration connexion
76.               connexion.ConnectionString = connectionString;
77.               // ouverture connexion
78.               connexion.Open();
79.               // configuration Command
80.               DbCommand sqlCommand = connecteur.CreateCommand();
81.               sqlCommand.CommandText = requête;
82.               sqlCommand.Connection = connexion;
83.               // exécution requête
84.               DbDataReader reader = sqlCommand.ExecuteReader();
85.               // affichage des résultats
86. ...
87.            }
88.         } catch (Exception ex) {
89.            // msg d'erreur
90.            Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
91.         }
92.     }
93.  }
94. }
```

- lignes 12-14 : l'application reçoit deux paramètres : le nom du connecteur générique ainsi que la chaîne de connexion à la base de données sous la forme de clés du fichier [App.config].
- lignes 23, 25 : on récupère dans [App.config], le nom du connecteur générique ainsi que la chaîne de connexion
- ligne 27 : le connecteur générique est instancié. A partir de ce moment, il est associé à un SGBD particulier.
- lignes 39-43 : l'exécution de l'ordre SQL saisi au clavier est déléguée à deux méthodes auxquelles on passe :
    - la requête à exécuter
    - la chaîne de connexion qui identifie la base sur laquelle la requête sera exécutée
    - le connecteur générique qui identifie les classes à utiliser pour dialoguer avec le SGBD gérant la base.

- lignes 50-54 : une connexion est obtenue avec la méthode *CreateConnection* (ligne 50) du connecteur générique puis configurée avec la chaîne de connexion de la base à gérer (ligne 52). Elle est ensuite ouverte (ligne 54).
- lignes 56-58 : l'objet *Command* nécessaire à l'exécution de l'ordre SQL est créé avec la méthode *CreateCommand* du connecteur générique. Il est ensuite configuré avec le texte de l'ordre SQL à exécuter (ligne 57) et la connexion sur laquelle exécuter celui-ci (ligne 58).
- ligne 60 : l'ordre SQL de mise à jour est exécuté
- lignes 74-87 : on trouve un code analogue. La nouveauté se trouve ligne 84. L'objet *Reader* obtenu par l'exécution de l'ordre *Select* est de type *DbDataReader* qui s'utilise comme les objets *OleDbDataReader*, *OdbcDataReader*, ... que nous avons déjà rencontrés.

Voici quelques exemples d'exécution.

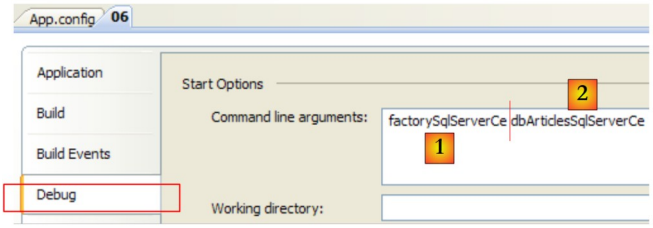
Avec la base MySQL5 :



On ouvre la page de propriétés du projet [1] et on sélectionne l'onglet [Debug] [2]. En [3], la clé du connecteur de la ligne 14 de [App.config]. En [4], la clé de la chaîne de connexion de la ligne 6 de [App.config]. Les résultats de l'exécution sont les suivants :

```
1.  Provider factory : [MySql.Data.MySqlClient]
2.  Chaîne de connexion à la base : [Server=localhost;Database=dbarticles;Uid=root;Pwd=root;]
3.
4.  Requête SQL (rien pour arrêter) : select * from articles
5.
6.  -----------------------------------
7.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
8.  -----------------------------------
9.
10. 1 article1 100 10 1
11. 2 article2 200 20 2
12. 3 article3 300 30 3
```

Avec la base SQL Server Compact :



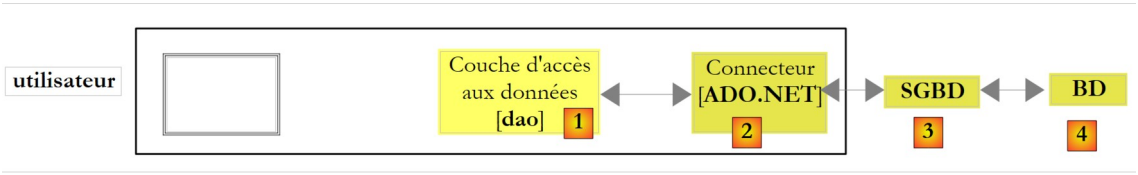
En [1], la clé du connecteur de la ligne 13 de [App.config]. En [2], la clé de la chaîne de connexion de la ligne 4 de [App.config]. Les résultats de l'exécution sont les suivants :

```
1.  Provider factory : [System.Data.SqlServerCe.3.5]
2.  Chaîne de connexion à la base : [Data Source=|DataDirectory|\dbarticles.sdf;Password=dbarticles;]
3.
4.  Requête SQL (rien pour arrêter) : select * from articles
5.
6.  -----------------------------------
7.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
8.  -----------------------------------
9.
10. 1 vélo 500 10 5
11. 2 pompe 10 10 2
12. 3 arc 600 4 1
13. 4 flèches - lot de 6 100 12 20
14. 5 combinaison de plongée 300 8 2
15. 6 bouteilles d'oxygène 120 10 5
```

Le lecteur est invité à tester les autres bases de données.

9.4.6 Quel connecteur choisir ?

Revenons à l'architecture d'une application avec bases de données :



Nous avons vu divers types de connecteurs ADO.NET :

- les connecteurs propriétaires sont les plus performants mais rendent la couche [dao] dépendante de classes propriétaires. Changer le SGBD implique de changer la couche [dao].
- les connecteurs ODBC ou OLE DB permettent de travailler avec de multiples bases de données sans changer la couche [dao]. Ils sont moins performants que les connecteurs propriétaires.
- le connecteur générique s'appuie sur les connecteurs propriétaires tout en présentant une interface standard à la couche [dao].

Il semble donc que le connecteur générique soit le connecteur idéal. Dans la pratique, le connecteur générique n'arrive cependant pas à cacher toutes les particularités d'un SGBD derrière une interface standard. Nous allons voir dans le paragraphe suivant, la notion de requête paramétrée. Avec SQL Server, une requête paramétrée à la forme suivante :

```
insert into articles(nom,prix,stockactuel,stockminimum) values(@nom,@prix,@sa,@sm)
```

Avec MySQL5, la même requête s'écrirait :

```
insert into articles(nom,prix,stockactuel,stockminimum) values(?,?,?,?)
```

Il y a donc une différence de syntaxe. La propriété de l'interface **IDbCommand** décrite page 225, liée aux paramètres est la suivante :

`Parameters`	la liste des paramètres d'un ordre SQL paramétré. L'ordre *update articles set prix=prix*1.1 where id=@id* a le paramètre @id.

La propriété *Parameters* est de type *IDataParameterCollection*, une interface. Elle représente l'ensemble des paramètres de l'ordre SQL *CommandText*. La propriété *Parameters* a une méthode *Add* pour ajouter des paramètres de type *IDataParameter*, de nouveau une interface. Celle-ci a les propriétés suivantes :

- *ParameterName* : nom du paramètre
- *DbType* : le type SQL du paramètre
- *Value* : la valeur affectée au paramètre
- ...

Le type *IDataParameter* convient bien aux paramètres de l'ordre SQL

```
insert into articles(nom,prix,stockactuel,stockminimum) values(@nom,@prix,@sa,@sm)
```

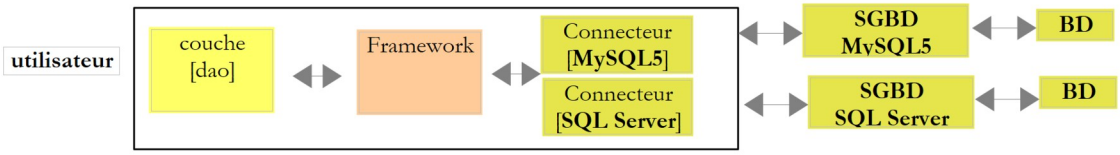
car on y trouve des paramètres nommés. La propriété *ParameterName* peut être utilisée.

Le type *IDataParameter* ne convient pas à l'ordre SQL

```
insert into articles(nom,prix,stockactuel,stockminimum) values(?,?,?,?)
```

car les paramètres ne sont pas nommés. C'et l'ordre d'ajout des paramètres dans la collection [IDbCommand.Parameters] qui est alors pris en compte. Dans cet exemple, il faudra insérer les 4 paramètres dans l'ordre *nom*, *prix*, *stockactuel*, *stockminimum*. Dans la requête avec paramètres nommés, l'ordre d'ajout des paramètres n'a pas d'importance. Au final, le développeur ne peut faire totalement abstraction du SGBD qu'il utilise lorsqu'il initialise les paramètres d'une requête paramétrée. On a là une des limites actuelles du connecteur générique.

Il existe des *frameworks* qui s'affranchissent de ces limites et qui apportent par ailleurs de nouvelles fonctionnalités à la couche [dao] :



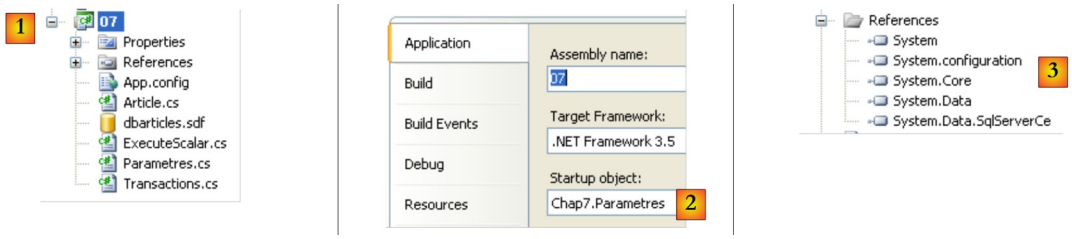
Un framework est un ensemble de bibliothèques de classes visant à faciliter une certaine façon d'architecturer l'application. Il en existe plusieurs qui permettent l'écriture de couches [dao] à la fois performantes et insensibles au changement de SGBD :

- **Spring.Net** [http://www.springframework.net/] déjà présenté dans ce document offre l'équivalent du connecteur générique étudié, sans ses limitations, ainsi que des facilités diverses qui simplifient l'accès aux données. Il existe une version Java.
- **iBatis.Net** [http://ibatis.apache.org] est plus ancien et plus riche que **Spring.Net**. Il existe une version Java.
- **NHibernate** [http://www.hibernate.org/] est un portage de la version Java **Hibernate** très connue dans le monde Java. *NHibernate* permet à la couche [dao] d'échanger avec le SGBD sans émettre d'ordres SQL. La couche [dao] travaille avec des objets Hibernate. Un langage de requêtes HBL (Hibernate Query language) permet de requêter les objets gérés par Hibernate. Ces sont ces derniers qui émettent les ordres SQL. Hibernate sait s'adapter aux SQL propriétaires des SGBD.
- **LINQ** (Language INtegrated Query), intégrée à la version 3.5 .NET et disponible dans C# 2008. *LINQ* marche sur les pas de *NHibernate*, mais pour l'instant (mai 2008) seul le SGBD SQL Server est supporté. Ceci devrait évoluer avec le temps. LINQ va plus loin que NHibernate : son langage de requêtes permet d'interroger de façon standard trois types différents de sources de données :
  - des collections d'objets (**LINQ to Objects**)
  - un fichier Xml (**LINQ to Xml**)
  - une base de données (**LINQ to SQL**)

Ces frameworks ne seront pas abordés dans ce document. Il est cependant vivement conseillé de les utiliser dans les applications professionnelles.

9.5 Requêtes paramétrées

Nous avons évoqué dans le paragraphe précédent les requêtes paramétrées. Nous les présentons ici avec un exemple pour le SGBD SQL Server Compact. Le projet est le suivant



- en [1], le projet. Seuls [App.config], [Article.cs] et [Parametres.cs] sont utilisés. On notera également la base SQL Server Ce [dbarticles.sdf].
- en [2], le projet est configuré pour exécuter [Parametres.cs]
- en [3], les références du projet

Le fichier de configuration [App.config] définit la chaîne de connexion à la base de données :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.   <connectionStrings>
4.      <add name="dbArticlesSqlServerCe" connectionString="Data Source=|DataDirectory|
    \dbarticles.sdf;Password=dbarticles;" />
5.   </connectionStrings>
6.  </configuration>
```

Le fichier [Article.cs] définit une classe [Article]. Un objet *Article* sera utilisé pour encapsuler les informations d'une ligne de la table *ARTICLES* de la base de données [dbarticles.sdf] :

```
1.  namespace Chap7 {
```

```
2.   class Article {
3.      // propriétés
4.      public int Id { get; set; }
5.      public string Nom { get; set; }
6.      public decimal Prix { get; set; }
7.      public int StockActuel { get; set; }
8.      public int StockMinimum { get; set; }
9.
10.     // constructeurs
11.     public Article() {
12.     }
13.
14.     public Article(int id, string nom, decimal prix, int stockActuel, int stockMinimum) {
15.        Id = id;
16.        Nom = nom;
17.        Prix = prix;
18.        StockActuel = stockActuel;
19.        StockMinimum = stockMinimum;
20.     }
21.
22.  }
23. }
```

L'application [Parametres.cs] met en oeuvre les requêtes paramétrées :

```
1.  using System;
2.  using System.Data.SqlServerCe;
3.  using System.Text;
4.  using System.Data;
5.  using System.Configuration;
6.
7.  namespace Chap7 {
8.   class Parametres {
9.      static void Main(string[] args) {
10.
11.        // exploitation du fichier de configuration
12.        string connectionString = null;
13.        try {
14.           // chaîne de connexion
15.           connectionString =
    ConfigurationManager.ConnectionStrings["dbArticlesSqlServerCe"].ConnectionString;
16.        } catch (Exception e) {
17.           Console.WriteLine("Erreur de configuration : {0}", e.Message);
18.           return;
19.        }
20.
21.        // affichages
22.        Console.WriteLine("Chaîne de connexion à la base : [{0}]\n", connectionString);
23.
24.        // création d'un tableau d'articles
25.        Article[] articles = new Article[5];
26.        for (int i = 1; i <= articles.Length; i++) {
27.           articles[i-1] = new Article(0, "article" + i, i * 100, i * 10, i);
28.        }
29.
30.        // on gère les éventuelles exceptions
31.        try {
32.
33.           // on supprime les articles existants de la base
34.           ExecuteUpdate(connectionString, "delete from articles");
35.
36.           // on affiche les articles de la table
37.           ExecuteSelect(connectionString, "select id,nom,prix,stockactuel,stockminimum from
    articles");
38.
39.           // on insère le tableau des articles dans la base
40.           InsertArticles(connectionString, articles);
41.
42.           // on affiche les articles de la table
43.           ExecuteSelect(connectionString, "select id,nom,prix,stockactuel,stockminimum from
    articles");
44.        } catch (Exception ex) {
45.           // msg d'erreur
46.           Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
47.        }
48.     }
49.
50.     // insertion d'un tableau d'articles
51.     static void InsertArticles(string connectionString, Article[] articles) {
52.        using (SqlCeConnection connexion = new SqlCeConnection(connectionString)) {
53.           // ouverture connexion
```

```
54.           connexion.Open();
55.           // configuration commande
56.           string requête = "insert into articles(nom,prix,stockactuel,stockminimum)
    values(@nom,@prix,@sa,@sm)";
57.           SqlCeCommand sqlCommand = new SqlCeCommand(requête, connexion);
58.           sqlCommand.Parameters.Add("@nom",SqlDbType.NVarChar,30);
59.           sqlCommand.Parameters.Add("@prix", SqlDbType.Money);
60.           sqlCommand.Parameters.Add("@sa", SqlDbType.Int);
61.           sqlCommand.Parameters.Add("@sm", SqlDbType.Int);
62.           // compilation de la commande
63.           sqlCommand.Prepare();
64.           // insertion des lignes
65.           for (int i = 0; i < articles.Length; i++) {
66.              // initialisation paramètres
67.              sqlCommand.Parameters["@nom"].Value = articles[i].Nom;
68.              sqlCommand.Parameters["@prix"].Value = articles[i].Prix;
69.              sqlCommand.Parameters["@sa"].Value = articles[i].StockActuel;
70.              sqlCommand.Parameters["@sm"].Value = articles[i].StockMinimum;
71.              // exécution requête
72.              sqlCommand.ExecuteNonQuery();
73.           }
74.        }
75.     }
76.
77.     // exécution d'une requête de mise à jour
78.     static void ExecuteUpdate(string connectionString, string requête) {
79. ...
80.     }
81.
82.     // exécution d'une requête Select
83.     static void ExecuteSelect(string connectionString, string requête) {
84. ...
85.     }
86.
87.     // affichage reader
88.     static void AfficheReader(IDataReader reader) {
89. ...
90.  }
91. }
```

La nouveauté par rapport à ce qui a été vu précédemment est la procédure [InsertArticles] des lignes 51-75 :

- ligne 51 : la procédure reçoit deux paramètres :
  - la chaîne de connexion *connectionString* qui va permettre à la procédure de se connecter à la base
  - un tableau d'objets *Article* qu'il faut ajouter à la table *Articles* de la base de données
- ligne 56 : la requête d'insertion d'un objet [Article]. Elle a quatre paramètres :
  - *@nom* : le nom de l'article
  - *@prix* : son prix
  - *@sa* : son stock actuel
  - *@sm* : son stock minimum

  La syntaxe de cette requête paramétrée est propriétaire à SQL Server Compact. Nous avons vu dans le paragraphe précédent qu'avec MySQL5, la syntaxe serait la suivante :

  ```
  insert into articles(nom,prix,stockactuel,stockminimum) values(?,?,?,?)
  ```

  Avec SQL Server Compact, chaque paramètre doit être précédé du caractère @. Le nom des paramètres est libre.
- lignes 58-61 : on définit les caractéristiques de chacun des 4 paramètres et on les ajoute, un par un, à la liste des paramètres de l'objet *SqlCeCommand* qui encapsule l'ordre SQL qui va être exécuté.

  On utilise ici la méthode **[SqlCeCommand].Parameters.Add** qui possède six signatures. Nous utilisons les deux suivantes :

  Add(string parameterName, SQLDbType type)
  ajoute et configure le paramètre nommé *parameterName*. Ce nom doit être l'un de ceux de la requête paramétrée configurée : (@nom, ...). *type* désigne le type SQL de la colonne concernée par le paramètre. On dispose de nombreux types dont les suivants :

type SQL	type C#	commentaire
BigInt	Int64	
DateTime	DateTime	
Decimal	Decimal	

type SQL	type C#	commentaire
Float	Double	
Int	Int32	
Money	Decimal	
NChar	String	chaîne de longueur fixe
NVarChar	String	chaîne de longueur variable
Real	Single	

Add(string parameterName, SQLDbType type, int size)
le troisième paramètre **size** fixe la taille de la colonne. Cette information n'est utile que pour certains types SQL, le type *NVarChar* par exemple.

- ligne 63 : on compile la requête paramétrée. On dit aussi qu'on la **prépare**, d'où le nom de la méthode. Cette opération n'est pas indispensable. Elle est là pour améliorer les performances. Lorsqu'un SGBD exécute un ordre SQL, il fait un certain travail d'optimisation avant de l'exécuter. Une requête paramétrée est destinée à être exécutée plusieurs fois avec des paramètres différents. Le texte de la requête lui ne change pas. Le travail d'optimisation peut alors n'être fait qu'une fois. Certains SGBD ont la possibilité de "préparer" ou "compiler" des requêtes paramétrées. Un plan d'exécution est alors défini pour cette requête. C'est la phase d'optimisation dont on a parlé. Une fois compilée, la requête est exécutée de façon répétée avec à chaque fois de nouveaux paramètres effectifs mais le même plan d'exécution.

  La compilation n'est pas l'unique avantage des requêtes paramétrées. Reprenons la requête étudiée :

```
insert into articles(nom,prix,stockactuel,stockminimum) values(@nom,@prix,@sa,@sm)
```

  On pourrait vouloir construire le texte de la requête par programme :

```
string requête="insert into articles(nom,prix,stockactuel,stockminimum)
values('"+nom+"',"+prix+","+sa+","+sm+")";
```

  Ci-dessus si (nom,prix,sa,sm) vaut ("article1",100,10,1), la requête précédente devient :

```
string requête="insert into articles(nom,prix,stockactuel,stockminimum)
values('article1',100,10,1)";
```

  Maintenant si (nom,prix,sa,sm) vaut ("l'article1",100,10,1), la requête précédente devient :

```
string requête="insert into articles(nom,prix,stockactuel,stockminimum)
values('l'article1',100,10,1)";
```

  et devient syntaxiquement incorrecte à cause de l'apostrophe du nom *l'article1*. Si *nom* provient d'une saisie de l'utilisateur, cela veut dire que nous sommes amenés à vérifier si la saisie n'a pas d'apostrophes et si elle en a, à les neutraliser. Cette neutralisation est dépendante du SGBD. L'intérêt de la requête préparée est qu'elle fait elle-même ce travail. Cette facilité justifie à elle seule l'utilisation d'une requête préparée.

- lignes 65-73 : les articles du tableau sont insérés un à un
- lignes 67-70 : chacun des quatre paramètres de la requête reçoit sa valeur via sa propriété *Value*.
- ligne 72 : la requête d'insertion maintenant complète est exécutée de la façon habituelle.

Voici un exemple d'exécution :

```
1.  Chaîne de connexion à la base : [Data Source=|DataDirectory|\dbarticles.sdf;Password=dbarticles;]
2.
3.  Il y a eu 5 ligne(s) modifiée(s)
4.
5.  -----------------------------------
6.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
7.  -----------------------------------
8.
9.
10. -----------------------------------
11. ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
12. -----------------------------------
13.
14. 117 article1 100 10 1
15. 118 article2 200 20 2
16. 119 article3 300 30 3
17. 120 article4 400 40 4
```

```
18. 121 article5 500 50 5
```

- ligne 3 : message après la suppression de toutes les lignes de la table
- lignes 5-7 : montrent que la table est vide
- lignes 10-18 : montrent la table après l'insertion des 5 articles

9.6 Transactions

9.6.1 Généralités

Une transaction est une suite d'ordres SQL exécutée de façon "atomique" :
- soit toutes les opérations réussissent
- soit l'une d'elles échoue et alors toutes celles qui ont précédé sont annulées

Au final, les opérations d'une transaction ont soit toutes été appliquées avec succès, soit aucune n'a été appliquée. Lorsque l'utilisateur a lui-même la maîtrise de la transaction, il valide une transaction par un ordre COMMIT ou l'annule par un ordre ROLLBACK.

Dans nos exemples précédents, nous n'avons pas utilisé de transaction. Et pourtant il y en avait, car dans un SGBD un ordre SQL s'exécute toujours au sein d'une transaction. Si le client .NET ne démarre pas lui-même une transaction **explicite**, le SGBD utilise une transaction **implicite**. Il y a alors deux cas courants :

1. chaque ordre SQL individuel fait l'objet d'une transaction, initiée par le SGBD avant l'ordre et fermée ensuite. On dit qu'on est en mode **autocommit**. Tout se passe donc comme si le client .NET faisait des transactions pour chaque ordre SQL.
2. le SGBD n'est pas en mode **autocommit** et commence une transaction implicite au 1er ordre SQL que le client .NET émet en dehors d'une transaction et il laisse le client la fermer. Tous les ordres SQL émis par le client .NET font alors partie de la transaction **implicite**. Celle-ci peut se terminer sur différents événements : le client ferme la connexion, commence une nouvelle transaction, ... mais on est alors dans une situation dépendante du SGBD. Cest un mode à éviter.

Le mode par défaut est généralement fixé par configuration du SGBD. Certains SGBD sont par défaut en mode **autocommit**, d'autres pas. SQLServer Compact est par défaut en mode **autocommit**.

Les ordres SQL des différents utilisateurs s'exécutent en même temps dans des transactions qui travaillent en parallèle. Les opérations faites par une transaction peuvent affecter celles faites par une autre transaction. On distingue quatre niveaux d'étanchéité entre les transactions des différents utilisateurs :

- **Uncommitted Read**
- **Committed Read**
- **Repeatable Read**
- **Serializable**

Uncommitted Read

Ce mode d'isolation est également appelé "Dirty Read". Voici un exemple de ce qui se peut se passer dans ce mode :
1. un utilisateur U1 commence une transaction sur une table T
2. un utilisateur U2 commence une transaction sur cette même table T
3. l'utilisateur U1 modifie des lignes de la table T mais ne les valide pas encore
4. l'utilisateur U2 "voit" ces modifications et prend des décisions à partir de ce qu'il voit
5. l'utilisateur annule sa transaction par un ROLLBACK

On voit qu'en 4, l'utilisateur U2 a pris une décision à partir de données qui s'avèreront fausses ultérieurement.

Committed Read

Ce mode d'isolation évite l'écueil précédent. Dans ce mode, l'utilisateur U2 à l'étape 4 ne "verra" pas les modifications apportées par l'utilisateur U1 à la table T. Il ne les verra qu'après que U1 ait fait un COMMIT de sa transaction.

Dans ce mode, également appelé "Unrepeatable Read", on peut néanmoins rencontrer les situations suivantes :

1. un utilisateur U1 commence une transaction sur une table T
2. un utilisateur U2 commence une transaction sur cette même table T
3. l'utilisateur U2 fait un SELECT pour obtenir la moyenne d'une colonne C des lignes de T vérifiant une certaine condition

4. l'utilisateur U1 modifie (UPDATE) certaines valeurs de la colonne C de T et les valide (COMMIT)
5. l'utilisateur U2 refait le même SELECT qu'en 3. Il découvrira que la moyenne de la colonne C a changé à cause des modifications faites par U1.

Maintenant l'utilisateur U2 ne voit que les modifications "validées" par U1. Mais alors qu'il reste dans la même transaction, deux opérations identiques 3 et 5 donnent des résultats différents. Le terme "Unrepeatable Read" désigne cette situation. C'est une situation ennuyeuse pour quelqu'un qui désire avoir une image stable de la table T.

Repeatable Read

Dans ce mode d'isolation, un utilisateur est assuré d'avoir les mêmes résultats pour ses lectures de la base tant qu'il reste dans la même transaction. Il travaille sur une photo sur laquelle ne sont jamais répercutées les modifications apportées par les autres transactions, mêmes validées. Il ne verra celles-ci que lorsque lui-même terminera sa transaction par un COMMIT ou ROLLBACK.

Ce mode d'isolation n'est cependant pas encore parfait. Après l'opération 3 ci-dessus, les lignes consultées par l'utilisateur U2 sont verrouillées. Lors de l'opération 4, l'utilisateur U1 ne pourra pas modifier (UPDATE) les valeurs de la colonne C de ces lignes. Il peut cependant rajouter des lignes (INSERT). Si certaines des lignes ajoutées vérifient la condition testée en 3, l'opération 5 donnera une moyenne différente de celle trouvée en 3 à cause des lignes rajoutées. On appelle parfois ces lignes des **lignes fantômes**.

Pour résoudre ce nouveau problème, il faut passer en isolation "Serializable".

Serializable

Dans ce mode d'isolation, les transactions sont complètement étanches les unes des autres. Il assure que le résultat de deux transactions menées simultanément donneront le même résultat que si elles étaient faites l'une après l'autre. Pour arriver à ce résultat, lors de l'opération 4 où l'utilisateur U1 veut ajouter des lignes qui changeraient le résultat du SELECT de l'utilisateur U1, il en sera empêché. Un message d'erreur lui indiquera que l'insertion n'est pas possible. Elle le deviendra lorsque l'utilisateur U2 aura validé sa transaction.

Les quatres niveaux SQL d'isolation des transactions ne sont pas disponibles dans tous les SGBD. Le niveau d'étanchéité par défaut est en général le niveau **Committed Read.** Le niveau d'étanchéité désiré pour une transaction peut être indiqué explicitement lors de la création d'une transaction explicite par un client .NET.

9.6.2 L'API de gestion des transactions

Une connexion implémente l'interface *IDbConnection* présentée page 225. Cette interface à la méthode suivante :

`BeginTransaction`	M	démarre une transaction.

Cette méthode a deux signatures :

1. *IDbTransaction BeginTransaction()* : démarre une transaction et rend l'objet *IDbTransaction* permettant de la contrôler
2. *IDbTransaction BeginTransaction(IsolationLevel level)* : précise de plus le niveau d'étanchéité désiré pour la transaction. *level* prend ses valeurs dans l'énumération suivante :

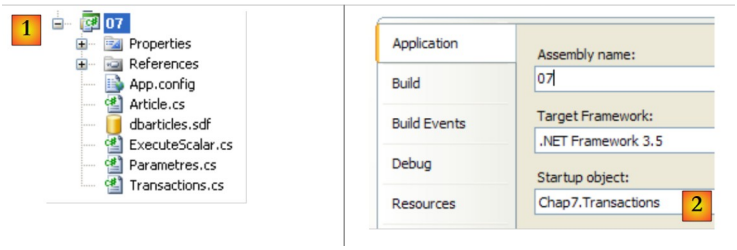
`ReadUncommitted`	la transaction peut lire des données écrites par une autre transaction que celle-ci n'a pas encore validées - à éviter
`ReadCommitted`	la transaction ne peut pas lire des données écrites par une autre transaction que celle-ci n'a pas encore validées. Les données lues deux fois de suite dans la transaction peuvent cependant changer (not repeatable reads) car une autre transaction a pu les modifier entre-temps (les lignes lues ne sont pas verrouillées - seules les lignes mises à jour le sont). Par ailleurs, une autre transaction a pu ajouter des lignes (lignes fantômes) qui seront intégrées dans la seconde lecture.
`RepeatableRead`	les lignes lues par la transaction sont verrouillées à l'instar des lignes mises à jour. Cela empêche une autre transaction de les modifier. Cela n'évite pas l'ajout de lignes.
`Serializable`	les tables exploitées par la transaction sont verrouillées empêchant l'ajout de nouvelles lignes par une autre transaction. Tout se passe comme si la transaction était seule. Diminue les performances car les transactions ne travaillent plus en parallèle.
`Snapshot`	la transaction travaille sur une copie des données faite au temps T. Utilisée lorsque la transaction est en lecture seule. Donne le même résultat que *serializable* en évitant son coût.

Une fois la transaction démarrée, elle est contrôlée par l'objet de type *IDbTransaction*, une interface dont nous utiliserons les propriétés P et méthodes M suivantes :

Nom	Type	Rôle
`Connection`	P	la connexion *IDbConnection* qui supporte la transaction
`Commit`	M	valide la transaction - les résultats des ordres SQL émis dans la transaction sont copiés dans la base.
`Rollback`	M	invalide la transaction - les résultats des ordres SQL émis dans la transaction ne sont pas copiés dans la base.

9.6.3 Le programme exemple

Nous reprenons le projet précédent pour nous intéresser maintenant au programme [Transactions.cs] :



- en [1], le projet.
- en [2], le projet est configuré pour exécuter [Transactions.cs]

Le code de [Transactions.cs] est le suivant :

```
1.  using System;
2.  using System.Configuration;
3.  using System.Data;
4.  using System.Data.SqlServerCe;
5.  using System.Text;
6.
7.  namespace Chap7 {
8.   class Transactions {
9.      static void Main(string[] args) {
10.
11.        // exploitation du fichier de configuration
12.        string connectionString = null;
13.        try {
14.           // chaîne de connexion
15.           connectionString =
    ConfigurationManager.ConnectionStrings["dbArticlesSqlServerCe"].ConnectionString;
16.        } catch (Exception e) {
17.           Console.WriteLine("Erreur de configuration : {0}", e.Message);
18.           return;
19.        }
20.
21.        // affichages
22.        Console.WriteLine("Chaîne de connexion à la base : [{0}]\n", connectionString);
23.
24.        // création d'un tableau de 2 articles de même nom
25.        Article[] articles = new Article[2];
26.        for (int i = 1; i <= articles.Length; i++) {
27.           articles[i - 1] = new Article(0, "article", i * 100, i * 10, i);
28.        }
29.        // on gère les éventuelles exceptions
30.        try {
31.           Console.WriteLine("Insertion sans transaction...");
32.           // on insère le tableau des articles dans la base d'abord sans transaction
33.           ExecuteUpdate(connectionString, "delete from articles");
34.           try {
35.              InsertArticlesOutOfTransaction(connectionString, articles);
36.           } catch (Exception ex) {
37.              // msg d'erreur
38.              Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
39.           }
40.           ExecuteSelect(connectionString, "select id,nom,prix,stockactuel,stockminimum from
    articles");
41.
```

```
42.           // on refait la même chose mais dans une transaction cette fois
43.           Console.WriteLine("\n\nInsertion dans une transaction...");
44.           ExecuteUpdate(connectionString, "delete from articles");
45.           InsertArticlesInTransaction(connectionString, articles);
46.           ExecuteSelect(connectionString, "select id,nom,prix,stockactuel,stockminimum from
    articles");
47.        } catch (Exception ex) {
48.           // msg d'erreur
49.           Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
50.        }
51.     }
52.
53.     // insertion d'un tableau d'articles sans transaction
54.     static void InsertArticlesOutOfTransaction(string connectionString, Article[] articles) {
55. ....
56.     }
57.
58.     // insertion d'un tableau d'articles dans une transaction
59.     static void InsertArticlesInTransaction(string connectionString, Article[] articles) {
60. ....
61.     }
62.
63.     // exécution d'une requête de mise à jour
64.     static void ExecuteUpdate(string connectionString, string requête) {
65. ....
66.     }
67.
68.     // exécution d'une requête Select
69.     static void ExecuteSelect(string connectionString, string requête) {
70. ...
71.     }
72.
73.     // affichage reader
74.     static void AfficheReader(IDataReader reader) {
75. ...
76.        }
77.     }
78.  }
79. }
```

- lignes 12-19 : la chaîne de connexion à la base SQLServer Ce est lue dans [App.config]
- lignes 25-28 : un tableau de deux objets *Article* est créé. Ces deux articles ont le même nom "article". Or, la base [dbarticles.sdf] a une contrainte d'unicité sur sa colonne [nom] (cf page 222). Donc ces deux articles ne peuvent être présents en même temps dans la base. Les deux articles de nom "article" sont ajoutés dans la table *articles*. Il va donc y avoir un problème, c.a.d. une exception lancée par le SGBD et relayée par son connecteur ADO.NET. Pour montrer l'effet de la transaction, les deux articles vont être insérés dans deux environnements différents :
  - d'abord en-dehors de toute transaction. Il faut se rappeler ici que, dans ce cas, SQLServer Compact travaille en mode **autocommit**, c.a.d. insère chaque ordre SQL dans une transaction **implicite**. Le 1er article va être inséré. Le second ne le sera pas.
  - ensuite dans une transaction **explicite** encapsulant les deux insertions. Parce que la deuxième insertion va échouer, la première sera alors défaite. Au final aucune insertion ne sera faite.
- ligne 33 : la table *articles* est vidée
- ligne 35 : l'insertion des deux articles sans transaction explicite. Parce qu'on sait que la deuxième insertion va provoquer une exception, celle-ci est gérée par un try / catch
- ligne 46 : affichage de la table *articles*
- lignes 44-46 : on refait la même séquence mais cette fois ci une transaction explicite est utilisée pour faire les insertions. L'exception qui est rencontrée est ici gérée par la méthode *InsertArticlesInTransaction*.
- lignes 54-56 : la méthode *InsertArticlesOutOfTransaction* est la méthode *InsertArticles* du programme [Parametres.cs] étudié précédemment.
- lignes 64-66 : la méthode *ExecuteUpdate* est la même que précédemment. L'ordre SQL exécuté l'est dans une transaction implicite. C'est possible ici car on sait que dans ce cas, SQLServer Compact travaille en mode **autocommit**.
- lignes 69-71 : idem pour la méthode *ExecuteSelect*.

La méthode *InsertArticlesInTransaction* est la suivante :

```
1.  // insertion d'un tableau d'articles dans une transaction
2.        static void InsertArticlesInTransaction(string connectionString, Article[] articles) {
3.           using (SqlCeConnection connexion = new SqlCeConnection(connectionString)) {
4.              // ouverture connexion
5.              connexion.Open();
6.              // configuration commande
7.              string requête = "insert into articles(nom,prix,stockactuel,stockminimum)
    values(@nom,@prix,@sa,@sm)";
8.              SqlCeCommand sqlCommand = new SqlCeCommand(requête, connexion);
9.              sqlCommand.Parameters.Add("@nom", SqlDbType.NVarChar, 30);
```

```
10.             sqlCommand.Parameters.Add("@prix", SqlDbType.Money);
11.             sqlCommand.Parameters.Add("@sa", SqlDbType.Int);
12.             sqlCommand.Parameters.Add("@sm", SqlDbType.Int);
13.             // compilation de la commande
14.             sqlCommand.Prepare();
15.             // transaction
16.             SqlCeTransaction transaction = null;
17.             try {
18.                // début transaction
19.                transaction = connexion.BeginTransaction(IsolationLevel.ReadCommitted);
20.                // la commande SQL doit être exécutée dans cette transaction
21.                sqlCommand.Transaction = transaction;
22.                // insertion des lignes
23.                for (int i = 0; i < articles.Length; i++) {
24.                   // initialisation paramètres
25.                   sqlCommand.Parameters["@nom"].Value = articles[i].Nom;
26.                   sqlCommand.Parameters["@prix"].Value = articles[i].Prix;
27.                   sqlCommand.Parameters["@sa"].Value = articles[i].StockActuel;
28.                   sqlCommand.Parameters["@sm"].Value = articles[i].StockMinimum;
29.                   // exécution requête
30.                   sqlCommand.ExecuteNonQuery();
31.                }
32.                // on valide la transaction
33.                transaction.Commit();
34.                Console.WriteLine("transaction validée...");
35.             } catch {
36.                // on défait la transaction
37.                if (transaction != null)transaction.Rollback();
38.                Console.WriteLine("transaction invalidée...");
39.             }
40.          }
41.       }
```

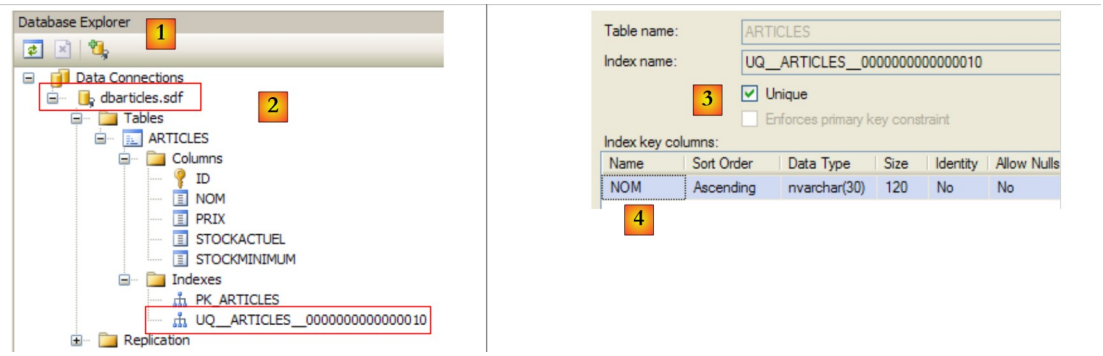
Nous ne détaillons que ce qui la différencie de la méthode *InsertArticles* du programme [Parametres.cs] étudié précédemment :

- ligne 16 : une transaction *SqlCeTransaction* est déclarée.
- lignes 17, 35 : le try / catch pour gérer l'exception qui va surgir à l'issue de la 2ième insertion
- ligne 19 : la transaction est créée. Elle appartient à la connexion courante.
- ligne 21 : la commande SQL paramétrée est mise dans la transaction
- lignes 23-31 : les insertions sont faites
- ligne 33 : tout s'est bien passé - la transaction est validée - les insertions vont être définitivement intégrées à la base de données.
- ligne 37 : on a eu un problème. La transaction est défaite si elle existait.

L'exécution donne les résultats suivants :

```
1.  Chaîne de connexion à la base : [Data Source=|DataDirectory|\dbarticles.sdf;Password=dbarticles;]
2.
3.  Insertion sans transaction...
4.  Il y a eu 0 ligne(s) modifiée(s)
5.  Erreur d'accès à la base de données (A duplicate value cannot be inserted into a unique index.
    [ Table name = ARTICLES,Constraint name = UQ__ARTICLES__0000000000000010 ])
6.
7.  -----------------------------------
8.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
9.  -----------------------------------
10.
11. 126 article 100 10 1
12.
13.
14. Insertion dans une transaction...
15. Il y a eu 1 ligne(s) modifiée(s)
16. transaction invalidée...
17.
18. -----------------------------------
19. ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
20. -----------------------------------
```

- ligne 4 : affichée par le *ExecuteUpdate*("delete from articles") - il n'y avait pas de lignes dans la table
- ligne 5 : l'exception provoquée par la deuxième insertion. Le message indique que la contrainte UQ__ARTICLES__0000000000000010 n'a pas été vérifiée. On peut en savoir plus en regardant les propriétés de la base :



- en [1] dans la vue [Database Explorer] de Visual Studio, on a créé une connexion [2] sur la base [dbarticles.sdf]. Celle-ci a un index *UQ__ARTICLES__0000000000000010*. En cliquant droit sur cet index, on a accès à ses propriétés (Index properties)
- en [3,4], on voit que l' index *UQ__ARTICLES__0000000000000010* corespond à une contrainte d'unicité sur la colonne [NOM]

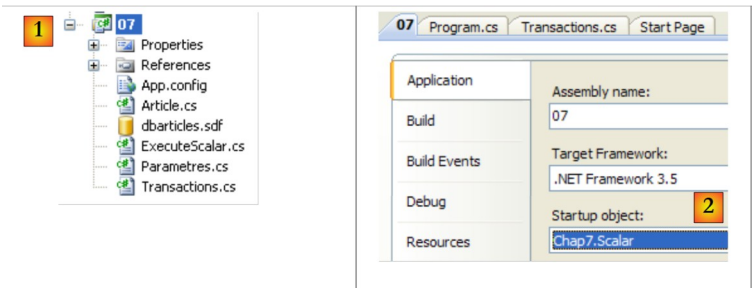
- lignes 7-11 : affichage de la table *articles* après les deux insertions. Elle n'est pas vide : le 1er article a été inséré.
- ligne 15 : affichée par le *ExecuteUpdate*("delete from articles") - il y avait une ligne dans la table
- ligne 16 : message affiché par *InsertArticlesInTransaction* lorsque la transaction échoue.
- lignes 18-20 : montrent qu'aucune insertion n'a été faite. Le *Rollback* de la transaction a défait la 1ère insertion.

9.7 La méthode ExecuteScalar

Parmi les méthodes de l'interface *IDbCommand* décrite page 225, il y avait la méthode suivante :

`ExecuteScalar`	M	pour exécuter un ordre SQL *Select* ne rendant qu'un unique résultat comme dans : *select count(*) from articles*.

Nous montrons ici un exemple d'utilisation de cette méthode. Revenons au projet :



- en [1], le projet.
- en [2], le projet est configuré pour exécuter [ExecuteScalar.cs]

Le programme [ExecuteScalar.cs] est le suivant :

```
1.  ...
2.  namespace Chap7 {
3.   class Scalar {
4.      static void Main(string[] args) {
5.
6.         // exploitation du fichier de configuration
7.         string connectionString = null;
8.  ...
9.
10.        // affichages
11.        Console.WriteLine("Chaîne de connexion à la base : [{0}]\n", connectionString);
12.
13.        // création d'un tableau de 5 articles
14.        Article[] articles = new Article[5];
15.        for (int i = 1; i <= articles.Length; i++) {
16.           articles[i - 1] = new Article(0, "article" + i, i * 100, i * 10, i);
```

```
17.        }
18.
19.        // on gère les éventuelles exceptions
20.        try {
21.           // on insère le tableau des articles dans une transaction
22.           ExecuteUpdate(connectionString, "delete from articles");
23.           InsertArticlesInTransaction(connectionString, articles);
24.           ExecuteSelect(connectionString, "select id,nom,prix,stockactuel,stockminimum from
    articles");
25.           // on calcule la moyenne des prix des articles
26.           decimal prixMoyen = (decimal)ExecuteScalar(connectionString, "select avg(prix) from
    articles");
27.           Console.WriteLine("Prix moyen des articles={0}", prixMoyen);
28.           // ou le nombre des articles
29.           int nbArticles = (int)ExecuteScalar(connectionString, "select count(id) from articles");
30.           Console.WriteLine("Nombre d'articles={0}", nbArticles);
31.        } catch (Exception ex) {
32.           // msg d'erreur
33.           Console.WriteLine("Erreur d'accès à la base de données (" + ex.Message + ")");
34.        }
35.     }
36.
37.     // insertion d'un tableau d'articles dans une transaction
38.     static void InsertArticlesInTransaction(string connectionString, Article[] articles) {
39. ...
40.     }
41.
42.
43.     // exécution d'une requête de mise à jour
44.     static object ExecuteScalar(string connectionString, string requête) {
45.        using (SqlCeConnection connexion = new SqlCeConnection(connectionString)) {
46.           // ouverture connexion
47.           connexion.Open();
48.           // exécution requête
49.           return new SqlCeCommand(requête, connexion).ExecuteScalar();
50.        }
51.     }
52.
53.     // exécution d'une requête de mise à jour
54.     static void ExecuteUpdate(string connectionString, string requête) {
55. ...
56.     }
57.
58.     // exécution d'une requête Select
59.     static void ExecuteSelect(string connectionString, string requête) {
60. ...
61.     }
62.
63.     // affichage reader
64.     static void AfficheReader(IDataReader reader) {
65. ...
66.     }
67.  }
68. }
```

- lignes 14-17 : création d'un tableau de 5 articles
- ligne 22 : la table *articles* est vidée
- ligne 23 : elle est remplie avec les 5 acticles
- ligne 24 : elle est affichée
- ligne 26 : demande le prix moyen des articles
- ligne 29 : demande le nombre d'articles
- ligne 49 : utilisation de la méthode *[IDbCommand].ExecuteScalar()* pour calculer chacune de ces valeurs.

Les résultats de l'exécution sont les suivants :

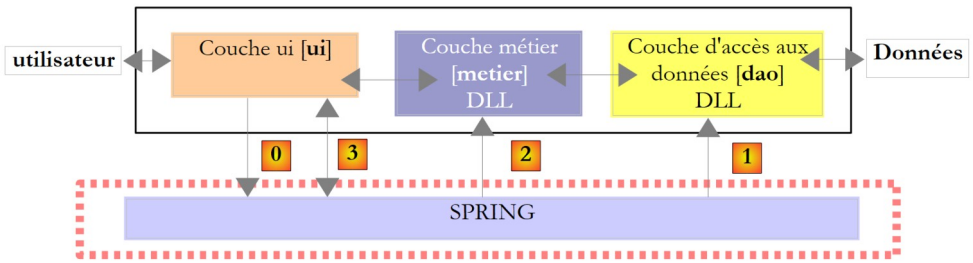
```
1.  Chaîne de connexion à la base : [Data Source=|DataDirectory|\dbarticles.sdf;Password=dbarticles;]
2.
3.  Il y a eu 5 ligne(s) modifiée(s)
4.  transaction validée...
5.
6.  -----------------------------------
7.  ID,NOM,PRIX,STOCKACTUEL,STOCKMINIMUM
8.  -----------------------------------
9.
10. 145 article1 100 10 1
11. 146 article2 200 20 2
12. 147 article3 300 30 3
13. 148 article4 400 40 4
14. 149 article5 500 50 5
```

```
15. Prix moyen des articles=300
16. Nombre d'articles=5
```

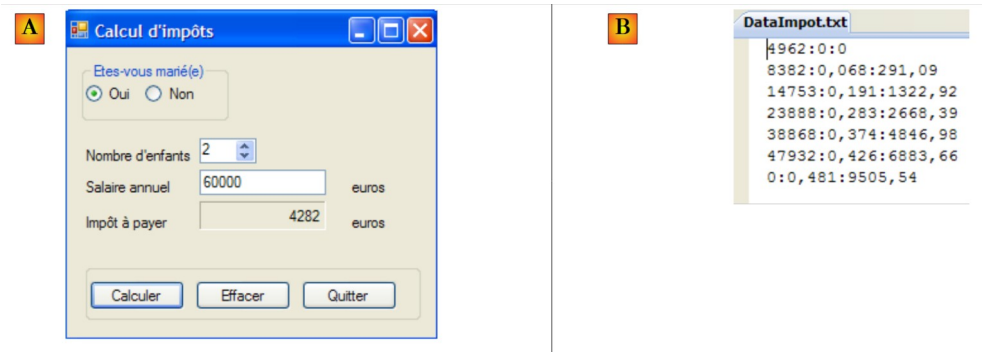
Les lignes 15 et 16 montrent les deux valeurs renvoyées par la méthode *ExecuteScalar*.

9.8 Application exemple - version 7

On reprend l'application exemple IMPOTS. La dernière version a été étudiée au paragraphe 7.6, page 203. C'était l'application à trois couches suivante :



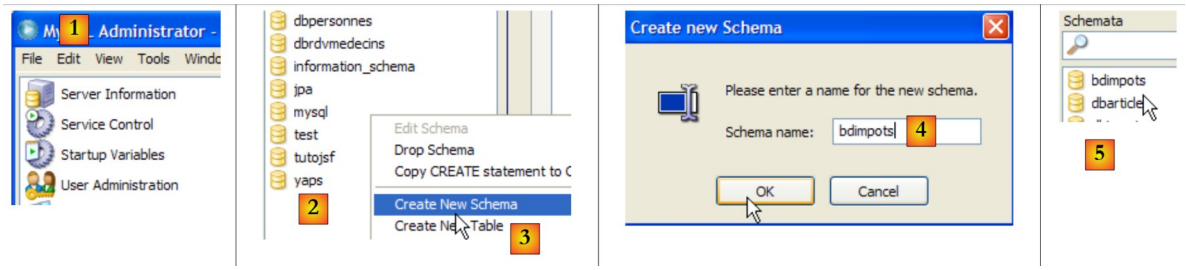
- la couche [ui] était une interface graphique [A] et la couche [dao] trouvait ses données dans un fichier texte [B].
- l'instanciation des couches et leur intégration dans l'application étaient assurées par Spring.



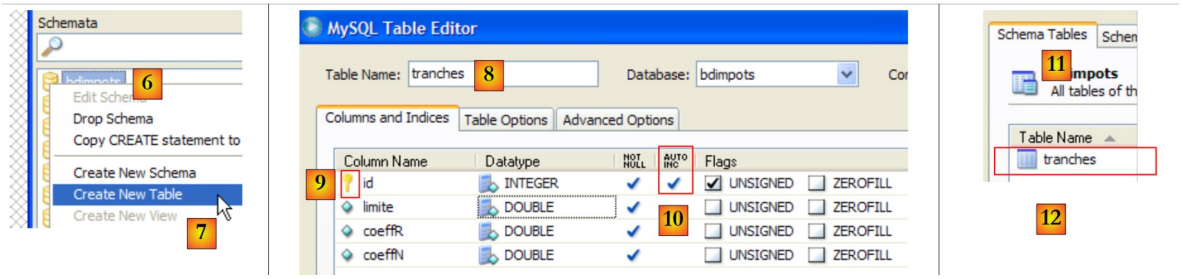
Nous modifions la couche [dao] afin qu'elle aille chercher ses données dans une base de données.

9.8.1 La base de données

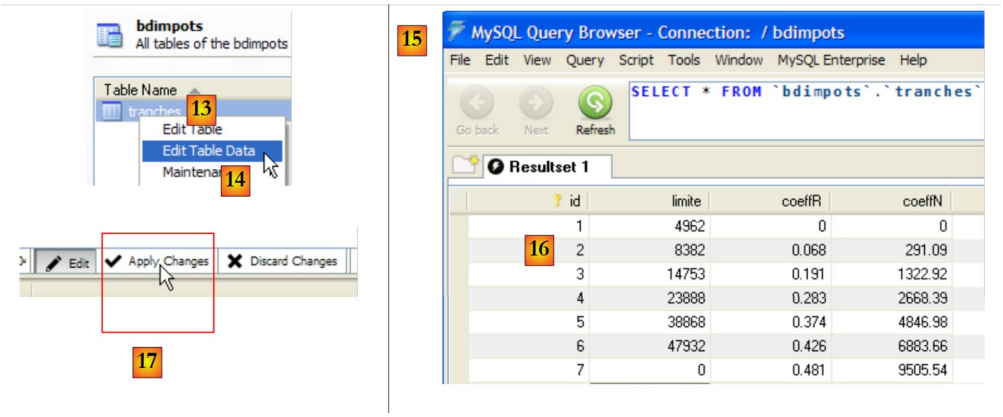
Le contenu du fichier texte [B] précédent est mis dans une base de données MySQL5. Nous montrons comment procéder :



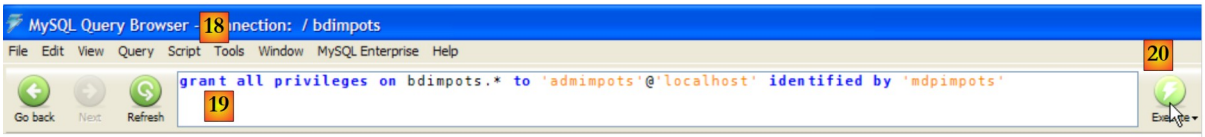
- [1] :MySQL Administrator a été lancé
- [2,3] : dans la zone [Schemata], cliquer droit et prendre l'option [Create Schema] pour créer une nouvelle base
- [4] : la base s'appellera [bdimpots]
- [5] : elle a été ajoutée aux bases de la zone [Schemata].



- [6,7] : cliquer droit sur la table et prendre l'option [Create New Table] pour créer une table
- [8] : la table s'appellera [tranches]. Elle aura les colonnes [id, limite, coeffR, coeffN].
- [9,10] : [id] est clé primaire de type INTEGER et a l'attribut AUTO_INCREMENT [10] : c'est le SGBD qui se chargera de remplir cette colonne lors d'ajout de lignes.
- les colonnes [limite, coeffR, coeffN] sont de type DOUBLE.
- [11,12] : la nouvelle table apparaît dans l'onglet [Schema Tables] de la base de données.

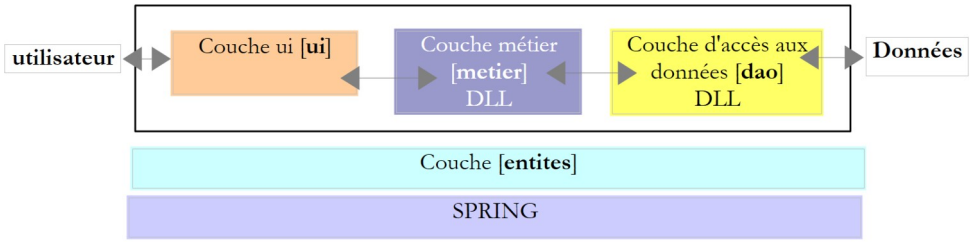


- [13,14] : pour mettre des données dans la table
- [15] : [Query Browser] a été lancé
- [16] : les données ont été entrées et validées pour les colonnes [limite, coeffR, coeffN]. La colonne [id] a été remplie par le SGBD. La validation a eu lieu avec [17].

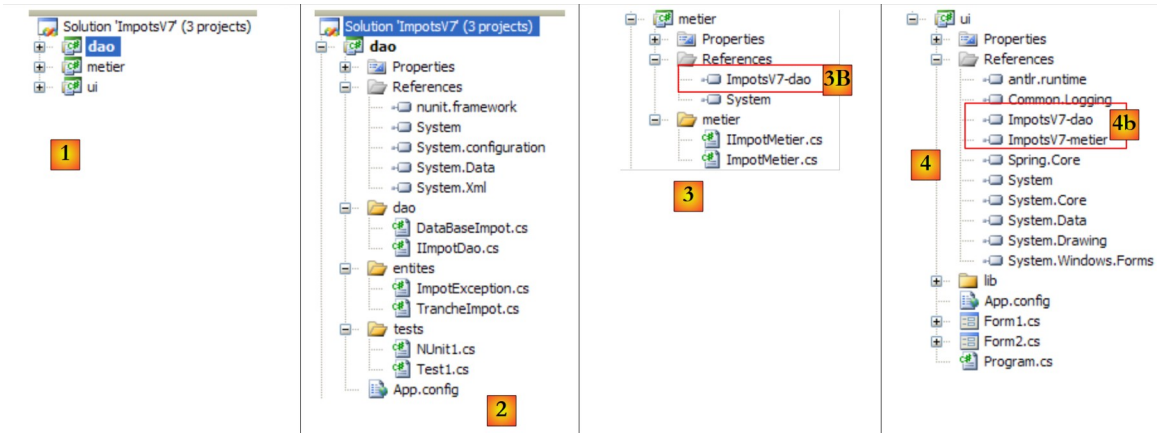


- toujours dans [Query Browser] [18], on exécute [20] la requête [19]. Celle-ci crée un utilisateur '*admimpots*' de mot de passe '*mdpimpots*' et lui donne tous les privilèges (*grant all privileges*) sur tous les objets de la base *bdimpots* (*on bdimpots.**). Cela va nous permettre de travailler sur la base [bdimpots] avec l'utilisateur [admimpots] plutôt qu'avec l'administrateur [root].

9.8.2 La solution Visual Studio



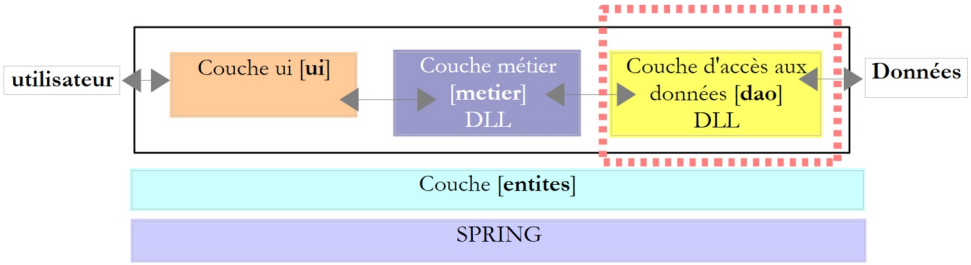
Nous suivrons la démarche étudiée pour la version 5 de l'application exemple (cf paragraphe 6.4, page 140). Nous allons construire progressivement la solution Visual Studio suivante :

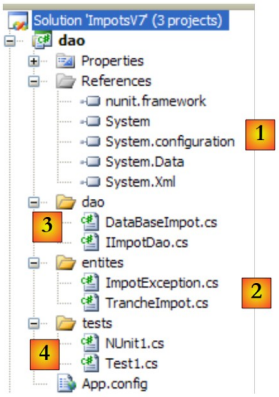


- en [1] : la solution *ImpotsV7* est formée de trois projets, un pour chacune des trois couches de l'application
- en [2] : le projet [dao] de la couche [dao] qui va désormais exploiter une base de données
- en [3] : le projet [metier] de la couche [metier]. Nous reprenons ici la couche [metier] de la version 5, décrite au paragraphe 6.4.4, page 154.
- en [4] : le projet [ui] de la couche [ui]. Nous reprenons ici la couche [ui] de la version 6, décrite au paragraphe 7.6, page 203.

Nous nous appuyons sur l'acquis pour récupérer deux couches déjà écrites, les couches [ui] et [metier]. Cela est rendu possible par l'architecture en couches choisie. Nous aurons néanmoins besoin des codes source des couches [ui] et [metier]. Il n'est en effet pas possible de se contenter des DLL des couches. Lorsque dans la version 5, la DLL de la couche [metier] a été créée, elle avait une dépendance sur la DLL de la couche [dao]. Cette dépendance a été inscrite en dur dans la DLL de la couche [metier] (nom de la DLL de la couche [dao], version, jeton d'identité, ...). Ainsi la DLL de la version 5 [ImpotsV5-metier.dll] n'accepte de travailler qu'avec la DLL [ImpotsV5-dao.dll] avec laquelle elle a été compilée. Si on change la DLL de la couche [dao] il faut recompiler la couche [metier] pour lui créer une nouvelle DLL. Il en est de même pour la couche [ui]. Les couches [ui] et [metier] ne seront donc pas modifiées mais elles seront recompilées pour travailler avec la DLL de la nouvelle couche [dao].

9.8.3 La couche [dao]





Les références du projet (cf [1] dans le projet)

- *nunit.framework* : pour le test NUnit
- *System.Configuration* : pour exploiter le fichier de configuration [App.config]
- *System.Data* : parce qu'on exploite une base de données.

Les entités (cf [2] dans le projet)

Les classes [TrancheImpot] et [ImpotException] sont celles des versions précédentes.

La couche [dao] (cf [3] dans le projet)

L'interface [IImpotDao] n'a pas changé :

```
1.  using Entites;
2.
3.  namespace Dao {
4.   public interface IImpotDao {
5.      // les tranches d'impôt
6.      TrancheImpot[] TranchesImpot{get;}
7.   }
8.  }
```

La classe d'implémentation [DataBaseImpot] de cette interface est la suivante :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Data.Common;
4.  using Entites;
5.
6.  namespace Dao {
7.   public class DataBaseImpot : IImpotDao {
8.      // tranches d'impôt
9.      private TrancheImpot[] tranchesImpot;
10.     public TrancheImpot[] TranchesImpot { get { return tranchesImpot; } }
11.
12.     // constructeur
13.     public DataBaseImpot(string factory, string connectionString, string requête) {
14.        // factory : la factory du SGBD cible
15.        // connectionString : la chaîne de connexion à la base des tranches d'impot
16.        // on gère les éventuelles exceptions
17.        try {
18.           // on récupère un connecteur générique pour le SGBD
19.           DbProviderFactory connecteur = DbProviderFactories.GetFactory(factory);
20.           using (DbConnection connexion = connecteur.CreateConnection()) {
21.              // configuration connexion
22.              connexion.ConnectionString = connectionString;
23.              // ouverture connexion
24.              connexion.Open();
25.              // configuration Command
26.              DbCommand sqlCommand = connecteur.CreateCommand();
27.              sqlCommand.CommandText = requête;
28.              sqlCommand.Connection = connexion;
29.              // exécution requête
30.              List<TrancheImpot> listTrancheImpot = new List<TrancheImpot>();
31.              using (DbDataReader reader = sqlCommand.ExecuteReader()) {
32.                 while (reader.Read()) {
```

```
33.                     // on crée une nouvelle trance d'impôt
34.                     listTrancheImpot.Add(new TrancheImpot() { Limite = reader.GetDecimal(0), CoeffR
    = reader.GetDecimal(1), CoeffN = reader.GetDecimal(2) });
35.                 }
36.             }
37.             // on met les tranches d'impôt dans son instance
38.             tranchesImpot = listTrancheImpot.ToArray();
39.         }
40.       } catch (Exception ex) {
41.         // on encapsule l'exception dans un type ImpotException
42.         throw new ImpotException("Erreur de lecture des tranches d'impôt", ex) { Code = 101 };
43.       }
44.
45.     }
46.  }
47. }
```

- ligne 7 : la classe [DataBaseImpot] implémente l'interface [IImpotDao].
- ligne 10 : l'implémentation de la méthode [TranchesImpot] de l'interface. Elle se contente de rendre une référence sur le tableau des tranches d'impôt de la ligne 9. Ce tableau va être construit par le constructeur de la classe.
- ligne 13 : le constructeur. Il utilise un connecteur générique (cf paragraphe 9.4.5, page 241) pour exploiter la base de données des tranches d'impôt. Le constructeur reçoit trois paramètres :
    1. le nom de la "factory" auprès de laquelle il va demander les classes pour se connecter à la base, émettre des ordres SQL, exploiter le résultat d'un Select.
    2. la chaîne de connexion qu'il doit utiliser pour se connecter à la base de données
    3. l'ordre SQL Select qu'il doit excécuter pour avoir les tranches d'impôt.
- ligne 19 : demande un connecteur à la "factory"
- ligne 20 : crée une connexion avec ce connecteur. Elle est créée mais pas encore opérationnelle
- ligne 22 : la chaîne de connexion de la connexion est initialisée. On peut désormais se connecter.
- ligne 24 : on se connecte
- ligne 26 : demande au connecteur, un objet [DbCommand] pour exécuter un ordre SQL
- ligne 27 : fixe l'ordre SQL à exécuter
- ligne 28 : fixe la connexion sur laquelle l'exécuter
- ligne 30 : une liste [listTrancheImpot] d'objets de type [TrancheImpot] est créée vide.
- ligne 31 : l'ordre SQL Select est exécuté
- lignes 32-35 : l'objet [DbDataReader] résultat du Select est exploité. Chaque ligne de la table résultat du Select sert à instancier un objet de type [TrancheImpot] qui est ajouté à la liste [listTrancheImpot].
- ligne 38 : la liste d'objets de type [TrancheImpot] est transférée dans le tableau de la ligne 9.
- lignes 40-43 : une éventuelle exception est encapsulée dans un type [ImpotException] et se voit attribuer le code d'erreur 101 (arbitraire).

Le test [Test1] (cf [4] dans le projet)

La classe [Test1] se contente d'afficher les tranches d'impôt à l'écran. C'est celle déjà utilisée dans la version 5 (page 149) sauf pour l'instruction qui instancie la couche [dao] (ligne 14).

```
1.  using System;
2.  using Dao;
3.  using Entites;
4.  using System.Configuration;
5.
6.  namespace Tests {
7.   class Test1 {
8.      static void Main() {
9.
10.       // on crée la couche [dao]
11.       IImpotDao dao = null;
12.       try {
13.         // création couche [dao]
14.         dao = new DataBaseImpot(ConfigurationManager.AppSettings["factoryMySql5"],
    ConfigurationManager.ConnectionStrings["dbImpotsMySql5"].ConnectionString,
    ConfigurationManager.AppSettings["requete"]);
15.       } catch (ImpotException e) {
16.         // affichage erreur
17.         string msg = e.InnerException == null ? null : String.Format(", Exception d'origine :
    {0}", e.InnerException.Message);
18.         Console.WriteLine("L'erreur suivante s'est produite : [Code={0},Message={1}{2}]",
    e.Code, e.Message, msg == null ? "" : msg);
19.         // arrêt programme
20.         Environment.Exit(1);
21.       }
22.       // on affiche les tranches d'impôt
23.       TrancheImpot[] tranchesImpot = dao.TranchesImpot;
24.       foreach (TrancheImpot t in tranchesImpot) {
```

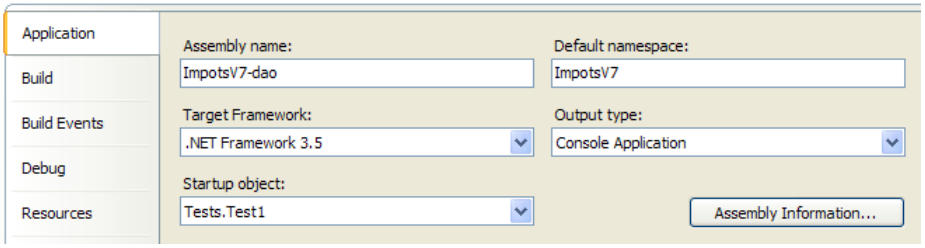
```
25.           Console.WriteLine("{0}:{1}:{2}", t.Limite, t.CoeffR, t.CoeffN);
26.       }
27.     }
28.  }
29. }
```

La ligne 14 exploite le fichier de configuration [App.config] suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.   <connectionStrings>
4.      <add name="dbImpotsMySql5"
    connectionString="Server=localhost;Database=bdimpots;Uid=admimpots;Pwd=mdpimpots;" />
5.   </connectionStrings>
6.   <appSettings>
7.      <add key="requete" value="select limite, coeffr, coeffn from tranches"/>
8.      <add key="factoryMySql5" value="MySql.Data.MySqlClient"/>
9.   </appSettings>
10. </configuration>
```

- ligne 4 : la chaîne de connexion à la base MySQL5. On notera que c'est l'utilisateur [admimpots] qui établira la connexion.
- ligne 8 : la "factory" pour travailler avec le SGBD MySQL5
- ligne 7 : la requête SQL Select pour obtenir les tranches d'impôt.

Le projet est configuré pour exécuter [Test1.cs] :



L'exécution du test donne les résultats suivants :

```
1.  4962:0:0
2.  8382:0,068:291,09
3.  14753:0,191:1322,92
4.  23888:0,283:2668,39
5.  38868:0,374:4846,98
6.  47932:0,426:6883,66
7.  0:0,481:9505,54
```

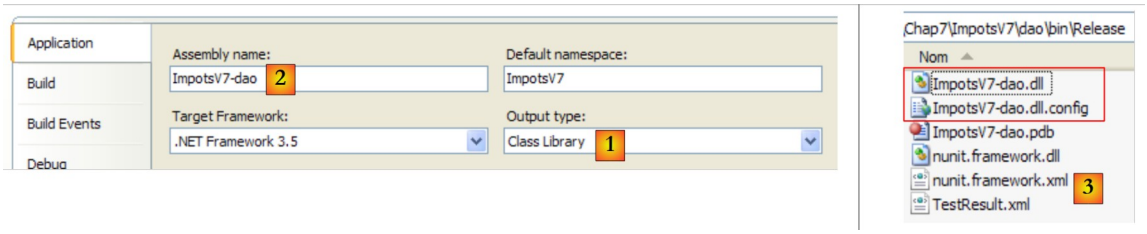
Le test NUnit [NUnit1] (cf [4] dans le projet)

Le test unitaire [NUnit1] est celui déjà utilisé dans la version 5 (page 149) sauf pour l'instruction qui instancie la couche [dao] (ligne 16).

```
1.  using System;
2.  using System.Configuration;
3.  using Dao;
4.  using Entites;
5.  using NUnit.Framework;
6.
7.  namespace Tests {
8.   [TestFixture]
9.   public class NUnit1 : AssertionHelper{
10.     // couche [dao] à tester
11.     private IImpotDao dao;
12.
13.     // constructeur
14.     public NUnit1() {
15.        // initialisation couche [dao]
16.        dao = new DataBaseImpot(ConfigurationManager.AppSettings["factoryMySql5"],
    ConfigurationManager.ConnectionStrings["dbImpotsMySql5"].ConnectionString,
    ConfigurationManager.AppSettings["requete"]);
17.     }
18.
19.     // test
20.     [Test]
21.     public void ShowTranchesImpot(){
```

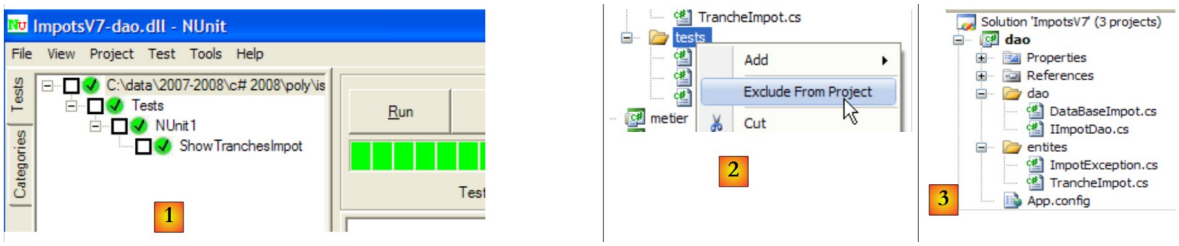
```
22.         // on affiche les tranches d'impôt
23.         TrancheImpot[] tranchesImpot = dao.TranchesImpot;
24.         foreach (TrancheImpot t in tranchesImpot) {
25.            Console.WriteLine("{0}:{1}:{2}", t.Limite, t.CoeffR, t.CoeffN);
26.         }
27.         // qqs tests
28.         Expect(tranchesImpot.Length,EqualTo(7));
29.         Expect(tranchesImpot[2].Limite,EqualTo(14753).Within(1e-6));
30.         Expect(tranchesImpot[2].CoeffR, EqualTo(0.191).Within(1e-6));
31.         Expect(tranchesImpot[2].CoeffN, EqualTo(1322.92).Within(1e-6));
32.     }
33.  }
34. }
```

Pour exécuter ce test unitaire, le projet doit être de type [Class Library] :



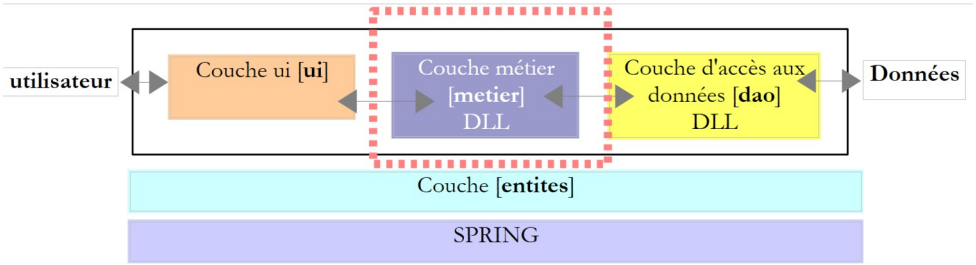
- en [1] : la nature du projet a été changée
- en [2] : la DLL générée s'appellera [ImpotsV7-dao.dll]
- en [3] : après génération (F6) du projet, le dossier [dao/bin/Release] contient la DLL [ImpotsV7-dao.dll]. Il contient aussi le fichier de configuration [App.config] renommé [nom DLL].config. C'est standard dans Visual studio.

La DLL [ImpotsV7-dao.dll] est ensuite chargée dans le framework *NUnit* et exécutée :



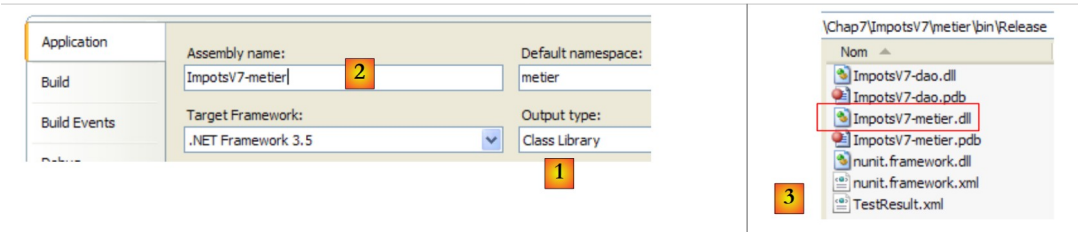
- en [1] : les tests ont été réussis. Nous considérons désormais la couche [dao] opérationnelle. Sa DLL contient toutes les classes du projet dont les classes de test. Celles-ci sont inutiles. Nous reconstruisons la DLL afin d'en exclure les classes de tests.
- en [2] : le dossier [tests] est exclu du projet
- en [3] : le nouveau projet. Celui-ci est régénéré par F6 afin de générer une nouvelle DLL. C'est cette DLL qui sera utilisée par les couches [metier] et [ui] de l'application.

9.8.3.1 La couche [metier]



- en [1], le projet [metier] est devenu le projet actif de la solution
- en [2] : les références du projet. On notera la référence sur la DLL de la couche [dao] créée précédemment. Cette procédure d'ajout de référence a été décrite dans la version 5, au paragraphe 6.4.4, page 154.
- en [3] : la couche [metier]. C'est celle de la version 5, décrite au paragraphe 6.4.4, page 154.

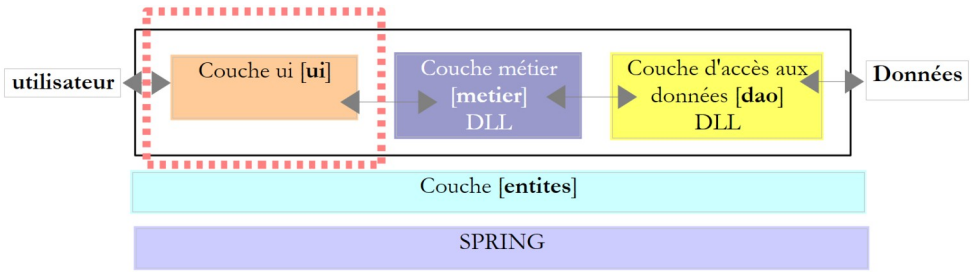
Le projet [metier] est configuré pour générer une DLL :

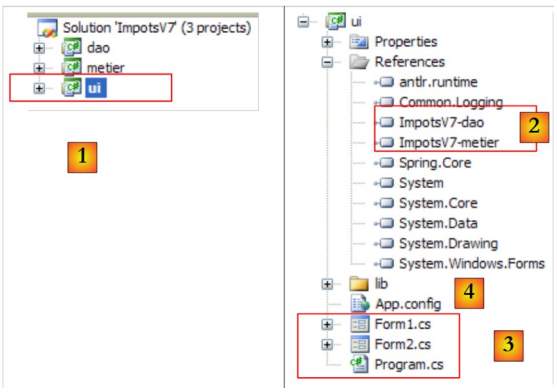


- [1] : le projet est de type "bibliothèque de classes"
- [2] : la génération du projet produira la DLL [ImpotsV7-metier.dll] [3].

Le projet est généré (F6).

9.8.4 La couche [ui]





- en [1], le projet [ui] est devenu le projet actif de la solution
- en [2] : les références du projet. On notera les références sur les DLL des couches [dao] et [metier].
- en [3] : la couche [ui]. C'est celle de la version 6 décrite au paragraphe 7.6, page 203.

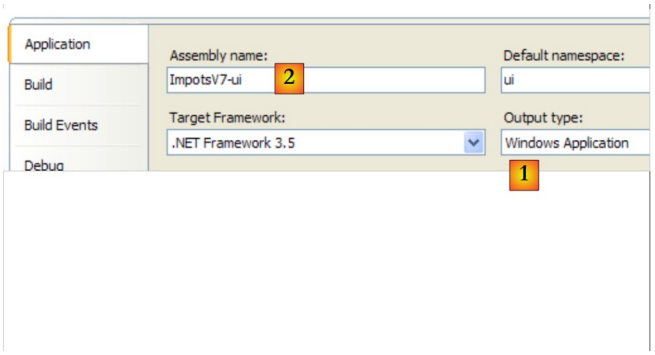
- en [4], le fichier de configuration [App.config] est analogue à celui de la version 6. Il n'en diffère que par la façon dont la couche [dao] est instanciée par Spring :

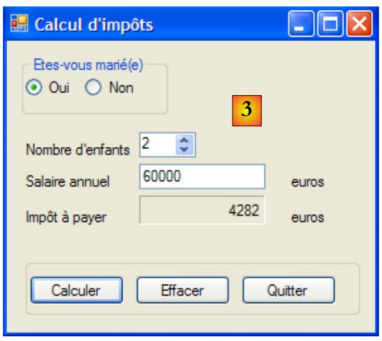
```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.   </configSections>
10.
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <object name="dao" type="Dao.DataBaseImpot, ImpotsV7-dao">
17.           <constructor-arg index="0" value="MySql.Data.MySqlClient"/>
18.           <constructor-arg index="1"
    value="Server=localhost;Database=bdimpots;Uid=admimpots;Pwd=mdpimpots;"/>
19.           <constructor-arg index="2" value="select limite, coeffr, coeffn from tranches"/>
20.        </object>
21.        <object name="metier" type="Metier.ImpotMetier, ImpotsV7-metier">
22.           <constructor-arg index="0" ref="dao"/>
23.        </object>
24.     </objects>
25.  </spring>
26. </configuration>
```

- lignes 11-25 : la configuration Spring
- lignes 15-24 : les objets instanciés par Spring
- lignes 16-20 : instanciation de la couche [dao]
- ligne 16 : la couche [dao] est instanciée par la classe [Dao.DataBaseImpot] qui se trouve dans la DLL [ImpotsV7-Dao]
- lignes 17-19 : les trois paramètres (factory du SGBD utilisé, chaîne de connexion, requête SQL) à fournir au constructeur de la classe [Dao.DataBaseImpot]
- lignes 21-23 : instanciation de la couche [metier]. C'est la même configuration que dans la version 6.

Tests

Le projet [ui] est configuré comme suit :

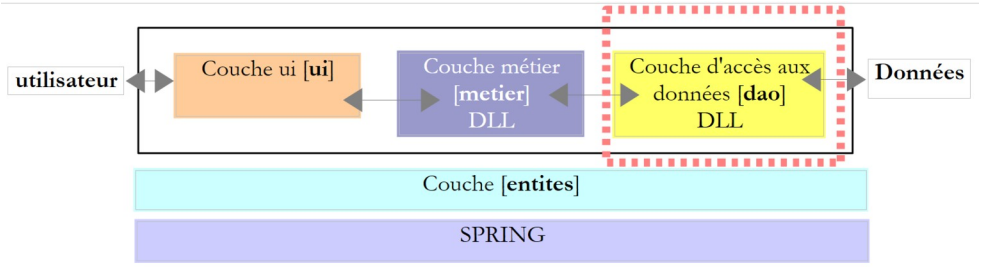




- [1] : le projet est de type "Windows Application"
- [2] : la génération du projet produira l'exécutable [ImpotsV7-ui.exe]

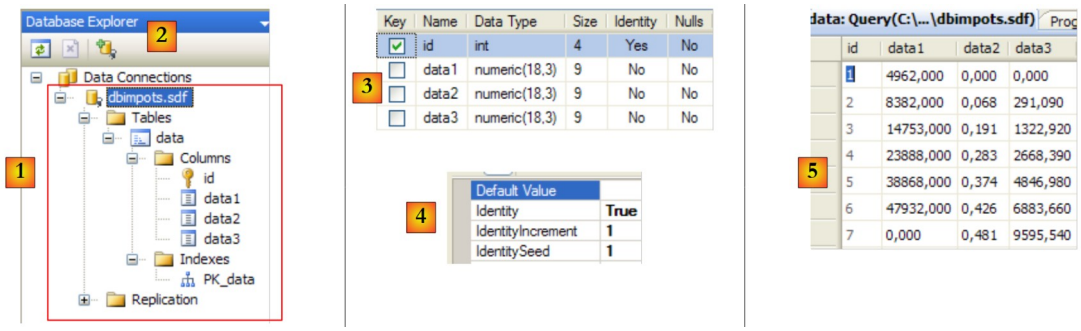
Un exemple d'exécution est donné en [3].

9.8.5 Changer la base de données

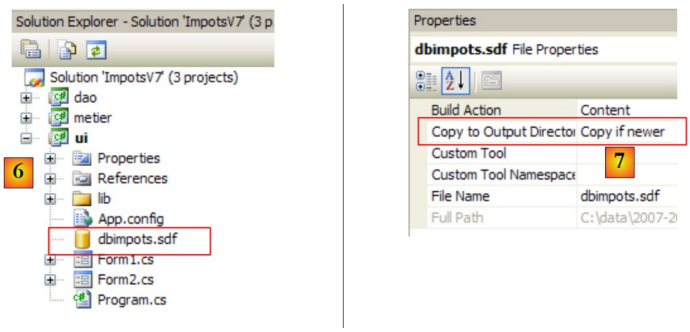


La couche [dao] ci-dessus a été écrite avec un connecteur générique et une base MySQL5. Nous nous proposons ici de passer à une base SQL Server Compact afin de montrer que seule la configuration va changer.

La base SQL Server Compact sera la suivante :



- [1] : la base [dbimpots.sdf] dans la vue [DataBase Explorer] de Visual studio [2]. Elle a été créée sans mot de passe.
- [3] : la table [data] qui contient les données. On a volontairement choisi des noms différents pour la table et les colonnes de ceux utilisés avec la base MySQL5 afin d'insister de nouveau sur l'intérêt de mettre ce genre de détails dans le fichier de configuration plutôt que dans le code.
- [4] : la colonne [id] est clé primaire et a l'attribut *Identity* : c'est le SGBD qui va lui attribuer ses valeurs.
- [5] : le contenu de la table [data].



- [6] : la base [dbimpots.sdf] a été placée dans le dossier du projet [ui] et intégrée à ce projet.
- [7] : la base [dbimpots.sdf] sera copiée dans le dossier d'exécution du projet.

Le fichier de configuration [App.config] pour la nouvelle base de données est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.   </configSections>
10.
```

```
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <!--
17.        <object name="dao" type="Dao.DataBaseImpot, ImpotsV7-dao">
18.           <constructor-arg index="0" value="MySql.Data.MySqlClient"/>
19.           <constructor-arg index="1"
    value="Server=localhost;Database=bdimpots;Uid=admimpots;Pwd=mdpimpots;"/>
20.           <constructor-arg index="2" value="select limite, coeffr, coeffn from tranches"/>
21.        </object>
22.        -->
23.        <object name="dao" type="Dao.DataBaseImpot, ImpotsV7-dao">
24.           <constructor-arg index="0" value="System.Data.SqlServerCe.3.5"/>
25.           <constructor-arg index="1" value="Data Source=|DataDirectory|\dbimpots.sdf;" />
26.           <constructor-arg index="2" value="select data1, data2, data3 from data"/>
27.        </object>
28.        <object name="metier" type="Metier.ImpotMetier, ImpotsV7-metier">
29.           <constructor-arg index="0" ref="dao"/>
30.        </object>
31.     </objects>
32.  </spring>
33. </configuration>
```

- lignes 23-27 : la configuration de la couche [dao] pour exploiter la base [dbimpots.sdf].

Les résultats de l'exécution sont identiques aux précédents. On notera l'intérêt d'utiliser un connecteur générique pour rendre la couche [dao] insensible au changement de SGBD. Nous avons vu cependant que ce connecteur ne convenait pas à toutes les situations, notamment celles où des requêtes paramétrées sont utilisées. Il y a alors d'autres solutions telle celle évoquée, des frameworks tiers d'accès aux données (Spring, iBatis, NHibernate, LINQ, ...).

9.9 Pour aller plus loin ...

- **LINQ** est présenté dans de nombreux ouvrages, notamment dans le livre : C# 3.0 in a Nutshell, Joseph et Ben Albahari, éditions O'Reilly déjà cité dans l'introduction de ce document.
- **iBatis** est présenté dans le livre : iBatis in Action, Clinton Begin, éditions Manning
- **Nhibernate** in Action aux éditions Manning est prévu pour juillet 2008

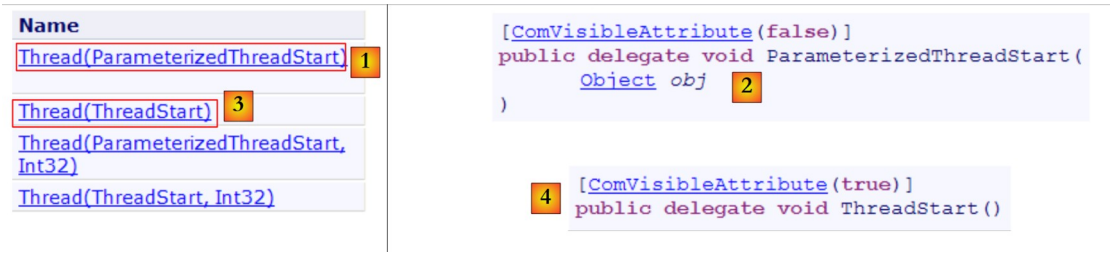
Spring, iBatis, NHibernate ont des manuels de référence disponibles sur le site de ces différents frameworks.

10 Les threads d'exécution

10.1 La classe Thread

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 :

Constructeurs



Nous n'utiliserons dans les exemples à suivre que les constructeurs [1,3]. Le constructeur [1] admet comme paramètre une méthode ayant la signature [2], c.a.d. ayant un paramètre de type *object* et ne rendant pas de résultat. Le constructeur [3] admet comme paramètre une méthode ayant la signature [4], c.a.d. n'ayant pas de paramètre et ne rendant pas de résultat.

Propriétés

Quelques propriétés utiles :

- *Thread CurrentThread* : propriété statique qui donne une référence sur le thread dans lequel se trouve le code ayant demandé cette propriété
- *string Name* : le nom du thread
- *bool IsAlive* : indique si le thread est en cours d'exécution ou non.

Méthodes

Les méthodes les plus utilisées sont les suivantes :

- *Start(), Start(object obj)* : lance l'exécution asynchrone du thread, éventuellement en lui passant de l'information dans un type *object*.
- *Abort(), Abort(object obj)* : pour terminer de force un thread
- *Join()* : le thread *T1* qui exécute *T2.Join* est bloqué jusqu'à ce que soit terminé le thread *T2*. Il existe des variantes pour terminer l'attente au bout d'un temps déterminé.
- *Sleep(int n)* : méthode statique - le thread exécutant la méthode est suspendu pendant *n* millisecondes. Il perd alors le processeur qui est donné à un autre thread.

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 :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program {
6.      static void Main(string[] args) {
7.         // init thread courant
8.         Thread main = Thread.CurrentThread;
9.         // affichage
10.        Console.WriteLine("Thread courant : {0}", main.Name);
11.        // on change le nom
12.        main.Name = "main";
13.        // vérification
14.        Console.WriteLine("Thread courant : {0}", main.Name);
15.
16.        // boucle infinie
17.        while (true) {
18.           // affichage
19.           Console.WriteLine("{0} : {1:hh:mm:ss}", main.Name, DateTime.Now);
20.           // arrêt temporaire
```

```
21.           Thread.Sleep(1000);
22.         }//while
23.     }
24.  }
25. }
```

- ligne 8 : on récupère une référence sur le thread dans lequel s'exécute la méthode [main]
- lignes 10-14 : on affiche et on modifie son nom
- lignes 17-22 : une boucle qui fait un affichage toutes les secondes
- ligne 21 : le thread dans lequel s'exécute la méthode [main] va être suspendu pendant 1 seconde

Les résultats écran sont les suivants :

```
1.  Thread courant :
2.  Thread courant : main
3.  main : 04:19:00
4.  main : 04:19:01
5.  main : 04:19:02
6.  main : 04:19:03
7.  main : 04:19:04
8.  ^CAppuyez sur une touche pour continuer...
```

- ligne 1 : le thread courant n'avait pas de nom
- ligne 2 : il en a un
- lignes 3-7 : l'affichage qui a lieu toutes les secondes
- ligne 8 : le programme est interrompu par Ctrl-C.

10.2 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 *thread*s 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 *thread*s 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*)
- en se mettant en sommeil pendant un temps déterminé (*Sleep*)

1. Un **thread** T est tout d'abord créé par l'un des constructeurs présentés plus haut, par exemple :

   ```
   Thread thread=new Thread(Start);
   ```

   où *Start* est une méthode ayant l'une des deux signatures suivantes :

   ```
   void Start();
   void Start(object obj);
   ```

   La création d'un thread ne lance pas celui-ci.
2. L'exécution du thread T est lancé par **T.Start()** : la méthode *Start* 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 T s'exécute de façon autonome. Il s'arrêtera lorsque la méthode S*tart* qu'il exécute aura fini son travail.
4. On peut forcer le thread T à se terminer :
   a. **T.Abort()** demande au thread T de se terminer.
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 :

```
1.  using System;
2.  using System.Threading;
3.
```

```
4.  namespace Chap8 {
5.   class Program {
6.      public static void Main() {
7.         // init Thread courant
8.         Thread main = Thread.CurrentThread;
9.         // on fixe un nom au Thread
10.        main.Name = "Main";
11.
12.        // création de threads d'exécution
13.        Thread[] tâches = new Thread[5];
14.        for (int i = 0; i < tâches.Length; i++) {
15.           // on crée le thread i
16.           tâches[i] = new Thread(Affiche);
17.           // on fixe le nom du thread
18.           tâches[i].Name =  i.ToString();
19.           // on lance l'exécution du thread i
20.           tâches[i].Start();
21.        }
22.
23.        // fin de main
24.        Console.WriteLine("Fin du thread {0} à {1:hh:mm:ss}",main.Name,DateTime.Now);
25.     }
26.
27.     public static void Affiche() {
28.        // affichage début d'exécution
29.        Console.WriteLine("Début d'exécution de la méthode Affiche dans le Thread {0} :
    {1:hh:mm:ss}",Thread.CurrentThread.Name,DateTime.Now);
30.        // mise en sommeil pendant 1 s
31.        Thread.Sleep(1000);
32.        // affichage fin d'exécution
33.        Console.WriteLine("Fin d'exécution de la méthode Affiche dans le Thread {0} :
    {1:hh:mm:ss}", Thread.CurrentThread.Name, DateTime.Now);
34.     }
35.  }
36. }
```

- lignes 8-10 : on donne un nom au thread qui exécute la méthode [Main]
- lignes 13-21 : on crée 5 threads et on les exécute. Les références des threads sont mémorisées dans un tableau afin de pouvoir les récupérer ultérieurement. Chaque thread exécute la méthode *Affiche* des lignes 27-35.
- ligne 20 : le thread n° i est lancé. Cette opération est non bloquante. Le thread n° i va s'exécuter en parallèle du thread de la méthode [Main] qui l'a lancé.
- ligne 24 : le thread qui exécute la méthode [Main] se termine.
- lignes 27-35 : la méthode [Affiche] fait des affichages. Elle affiche le nom du thread qui l'exécute ainsi que les heures de début et fin d'exécution.
- ligne 31 : tout thread exécutant la méthode [Affiche] va s'arrêter pendant 1 seconde. Le processeur va alors être donné à un autre thread en attente de processeur. A la fin de la seconde d'arrêt, le thread arrêté va âtre candidat au processeur. Il l'aura lorsque son tour sera venu. Cela dépend de divers facteurs dont la priorité des autres threads en attente de processeur.

Les résultats sont les suivants :

```
1.  Début d'exécution de la méthode Affiche dans le Thread 0 : 10:30:44
2.  Début d'exécution de la méthode Affiche dans le Thread 1 : 10:30:44
3.  Début d'exécution de la méthode Affiche dans le Thread 2 : 10:30:44
4.  Début d'exécution de la méthode Affiche dans le Thread 3 : 10:30:44
5.  Début d'exécution de la méthode Affiche dans le Thread 4 : 10:30:44
6.  Fin du thread Main à 10:30:44
7.  Fin d'exécution de la méthode Affiche dans le Thread 0 : 10:30:45
8.  Fin d'exécution de la méthode Affiche dans le Thread 1 : 10:30:45
9.  Fin d'exécution de la méthode Affiche dans le Thread 2 : 10:30:45
10. Fin d'exécution de la méthode Affiche dans le Thread 3 : 10:30:45
11. Fin d'exécution de la méthode Affiche dans le Thread 4 : 10:30:45
```

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 terminer la méthode *Main* par les instructions :

```
1.          // fin de main
2.          Console.WriteLine("Fin du thread " + main.Name);
3.          // on arrête tous les threads
4.  Environment.Exit(0);
```

L'exécution du nouveau programme donne les résultats suivants :

```
1.  Début d'exécution de la méthode Affiche dans le Thread 0 : 10:33:18
2.  Début d'exécution de la méthode Affiche dans le Thread 1 : 10:33:18
3.  Début d'exécution de la méthode Affiche dans le Thread 2 : 10:33:18
4.  Début d'exécution de la méthode Affiche dans le Thread 3 : 10:33:18
5.  Début d'exécution de la méthode Affiche dans le Thread 4 : 10:33:18
6.  Fin du thread Main à 10:33:18
```

- lignes 1-5 : les threads créés par la fonction *Main* commencent leur exécution et sont interrompus pendant 1 seconde
- ligne 6 : le thread [Main] récupère le processeur et exécute l'instruction :

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

Cette instruction arrête **tous les threads** de l'application et non simplement le thread *Main*.

Si la méthode *Main* veut attendre la fin d'exécution des threads qu'elle a créés, elle peut utiliser la méthode *Join* de la classe *Thread* :

```
1.        public static void Main() {
2.  ...
3.          // on attend tous les threads
4.          for (int i = 0; i < tâches.Length; i++) {
5.             // attente de la fin d'exécution du thread i
6.             tâches[i].Join();
7.          }
8.          // fin de main
9.          Console.WriteLine("Fin du thread {0} à {1:hh:mm:ss}", main.Name, DateTime.Now);
10. }
```

- ligne 6 : le thread [Main] attend chacun des threads. Il est d'abord bloqué en attente du thread n° 1, puis du thread n° 2, etc... Au final lorsqu'il sort de la boucle des lignes 2-5, c'est ce que les 5 threads qu'il a lancés sont finis.

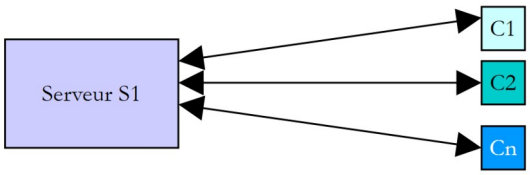
On obtient alors les résultats suivants :

```
1.  Début d'exécution de la méthode Affiche dans le Thread 0 : 10:35:18
2.  Début d'exécution de la méthode Affiche dans le Thread 1 : 10:35:18
3.  Début d'exécution de la méthode Affiche dans le Thread 2 : 10:35:18
4.  Début d'exécution de la méthode Affiche dans le Thread 3 : 10:35:18
5.  Début d'exécution de la méthode Affiche dans le Thread 4 : 10:35:18
6.  Fin d'exécution de la méthode Affiche dans le Thread 0 : 10:35:19
7.  Fin d'exécution de la méthode Affiche dans le Thread 1 : 10:35:19
8.  Fin d'exécution de la méthode Affiche dans le Thread 2 : 10:35:19
9.  Fin d'exécution de la méthode Affiche dans le Thread 3 : 10:35:19
10. Fin d'exécution de la méthode Affiche dans le Thread 4 : 10:35:19
11. Fin du thread Main à 10:35:19
```

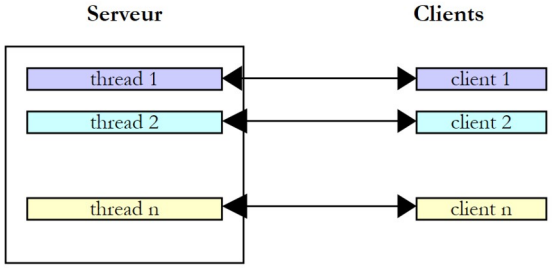
- ligne 11 : le thread [Main] s'est terminé après les threads qu'il avait lancés.

10.3 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 les raisons pour lesquelles 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. Nous allons les présenter dans le chapitre qui suit. Dans une application client-serveur de l'internet, 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 minutes. Il est bien sûr hors de question qu'un client monopolise tout seul le serveur pendant 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.



Dans la pratique, le serveur utilise un pool de threads avec un nombre limité de threads, 50 par exemple. Le 51 ième client est alors prié d'attendre.

10.4 Echange d'informations entre threads

Dans les exemples précédents, un thread était initialisé de la façon suivante :

```
Thread t=new Thread(Run);
```

où *Run* était une méthode ayant la signature suivante :

```
void Run();
```

Il est également possible d'utiliser la signature suivante :

```
void Run(object obj);
```

Cela permet de transmettre de l'information au thread lancé. Ainsi

```
t.Start(obj1);
```

va lancer le thread *t* qui va alors exécuter la méthode *Run* qui lui a été associée par construction, en lui passant le paramètre effectif *obj1*. Voici un exemple :

```
1.  using System;
2.  using System.Threading;
```

```
3.
4.  namespace Chap8 {
5.   class Program4 {
6.      public static void Main() {
7.         // init Thread courant
8.         Thread main = Thread.CurrentThread;
9.         // on fixe un nom au Thread
10.        main.Name = "Main";
11.
12.        // création de threads d'exécution
13.        Thread[] tâches = new Thread[5];
14.        Data[] data = new Data[5];
15.        for (int i = 0; i < tâches.Length; i++) {
16.           // on crée le thread i
17.           tâches[i] = new Thread(Sleep);
18.           // on fixe le nom du thread
19.           tâches[i].Name = i.ToString();
20.           // on lance l'exécution du thread i
21.           tâches[i].Start(data[i] = new Data { Début = DateTime.Now, Durée = i+1 });
22.        }
23.        // on attend tous les threads
24.        for (int i = 0; i < tâches.Length; i++) {
25.           // attente de la fin d'exécution du thread i
26.           tâches[i].Join();
27.           // affichage résultat
28.           Console.WriteLine("Thread {0} terminé : début {1:hh:mm:ss}, durée programmée {2} s, fin
    {3:hh:mm:ss}, durée effective {4}",
29.              tâches[i].Name,data[i].Début,data[i].Durée,data[i].Fin,(data[i].Fin-data[i].Début));
30.        }
31.        // fin de main
32.        Console.WriteLine("Fin du thread {0} à {1:hh:mm:ss}", main.Name, DateTime.Now);
33.     }
34.
35.     public static void Sleep(object infos) {
36.        // on récupère le paramètre
37.        Data data = (Data)infos;
38.        // mise en sommeil pendant Durée secondes
39.        Thread.Sleep(data.Durée*1000);
40.        // fin d'exécution
41.        data.Fin = DateTime.Now;
42.     }
43.  }
44.
45.  internal class Data {
46.     // informations diverses
47.     public DateTime Début { get; set; }
48.     public int Durée { get; set; }
49.     public DateTime Fin { get; set; }
50.  }
51. }
```

- lignes 45-50 : l'information de type [Data] passée aux threads :
  - *Début* : heure du début de l'exécution du thread - fixée par le thread lanceur
  - *Durée* : durée en secondes du Sleep exécuté par le thread lancé - fixée par le thread lanceur
  - *Fin* : heure du début de l'exécution du thread - fixée par le thread lancé

  Il y a là un échange d'informations entre le thread lanceur et le thread lancé.
- lignes 35-43 : la méthode *Sleep* exécutée par les threads a la signature *void Sleep(object obj)*. Le paramètre effectif *obj* sera du type [Data] défini ligne 45.
- lignes 15-22 : création de 5 threads
- ligne 17 : chaque thread est associé à la méthode Sleep de la ligne 35
- ligne 21 : un objet de type [Data] est passé à la méthode *Start* qui lance le thread. Dans cet objet on a noté l'heure de début de l'exécution du thread ainsi que la durée en secondes pendant laquelle il doit dormir. Cet objet est mémorisé dans le tableau de la ligne 14.
- lignes 24-30 : le thread [Main] attend la fin de tous les threads qu'il a lancés.
- lignes 28-29 : le thread [Main] récupère l'objet data[i] du thread n° i et en affiche le contenu.
- lignes 35-42 : la méthode *Sleep* exécutée par les threads
- ligne 37 : on récupère le paramètre de type [Data]
- ligne 39 : le champ *Durée* du paramètre est utilisé pour fixer la durée du *Sleep*
- ligne 41 : le champ *Fin* du paramètre est initialisé

Les résultats de l'exécution sont les suivants :

```
1.  Thread 0 terminé : début 11:18:50, durée programmée 1 s, fin 11:18:51, durée effective
    00:00:01.0156250
2.  Thread 1 terminé : début 11:18:50, durée programmée 2 s, fin 11:18:52, durée effective 00:00:02
3.  Thread 2 terminé : début 11:18:50, durée programmée 3 s, fin 11:18:53, durée effective 00:00:03
4.  Thread 3 terminé : début 11:18:50, durée programmée 4 s, fin 11:18:54, durée effective 00:00:04
5.  Thread 4 terminé : début 11:18:50, durée programmée 5 s, fin 11:18:55, durée effective 00:00:05
6.  Fin du thread Main à 11:18:55
```

Cet exemple montre que deux threads peuvent s'échanger de l'information :

- le thread lanceur peut contrôler l'exécution du thread lancé en lui donnant des informations
- le thread lancé peut rendre des résultats au thread lanceur.

Pour que le thread lancé sache à quel moment les résultats qu'il attend sont disponibles, il faut qu'il soit averti de la fin du thread lancé. Ici, il a attendu qu'il se termine en utilisant la méthode *Join*. Il y a d'autres façons de faire la même chose. Nous les verrons ultérieurement.

10.5 Accès concurrents à des ressources partagées

10.5.1 Accès concurrents non synchronisés

Dans le paragraphe sur l'échange d'informations entre threads, l'information échangée ne l'était que par deux threads et à des moments bien précis. On avait là un classique passage de paramètres. Il existe d'autres cas où une information est partagée par plusieurs threads qui peuvent vouloir la lire ou la mettre à jour au même moment. Se pose alors le problème de l'intégrité de cette information. Supposons que l'information partagée soit une structure S avec diverses informations I1, I2, ... In.

- un thread T1 commence à mettre à jour la structure S : il modifie le champ I1 et est interrompu avant d'avoir terminé la mise à jour complète de la structure S
- un thread T2 qui récupère le processeur lit alors la structure S pour prendre des décisions. Il lit une structure dans un état instable : certains champs sont à jour, d'autres pas.

On appelle cette situation, l'accès à une ressource partagée, ici la structure S, et elle est souvent assez délicate à gérer. Prenons l'exemple suivant pour illustrer les problèmes qui peuvent surgir :

- 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 affichée. On devrait donc trouver *n*.

Le programme est le suivant :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program {
6.
7.      // variables de classe
8.      static int cptrThreads = 0;   // compteur de threads
9.
10.     //main
11.     public static void Main(string[] args) {
12.        // mode d'emploi
13.        const string syntaxe = "pg nbThreads";
14.        const int nbMaxThreads = 100;
15.
16.        // vérification nbre d'arguments
17.        if (args.Length != 1) {
18.           // erreur
19.           Console.WriteLine(syntaxe);
20.           // arrêt
21.           Environment.Exit(1);
22.        }
23.        // vérification qualité de l'argument
24.        int nbThreads = 0;
25.        bool erreur = false;
26.        try {
27.           nbThreads = int.Parse(args[0]);
28.           if (nbThreads < 1 || nbThreads > nbMaxThreads)
29.              erreur = true;
30.        } catch {
31.           // erreur
```

```
32.           erreur = true;
33.        }
34.        // erreur ?
35.        if (erreur) {
36.           // erreur
37.           Console.Error.WriteLine("Nombre de threads incorrect (entre 1 et 100)");
38.           // fin
39.           Environment.Exit(2);
40.        }
41.        // création et génération des threads
42.        Thread[] threads = new Thread[nbThreads];
43.        for (int i = 0; i < nbThreads; i++) {
44.           // création
45.           threads[i] = new Thread(Incrémente);
46.           // nommage
47.           threads[i].Name = "" + i;
48.           // lancement
49.           threads[i].Start();
50.        }//for
51.        // attente de la fin des threads
52.        for (int i = 0; i < nbThreads; i++) {
53.           threads[i].Join();
54.        }
55.        // affichage compteur
56.        Console.WriteLine("Nombre de threads générés : " + cptrThreads);
57.     }
58.
59.     public static void Incrémente() {
60.        // augmente le compteur de threads
61.        // lecture compteur
62.        int valeur = cptrThreads;
63.        // suivi
64.        Console.WriteLine("A {0:hh:mm:ss}, le thread {1}  a lu la valeur du compteur : {2}",
    DateTime.Now, Thread.CurrentThread.Name, cptrThreads);
65.        // attente
66.        Thread.Sleep(1000);
67.        // incrémentation compteur
68.        cptrThreads = valeur + 1;
69.        // suivi
70.        Console.WriteLine("A {0:hh:mm:ss}, le thread {1}  a écrit la valeur du compteur : {2}",
    DateTime.Now, Thread.CurrentThread.Name, cptrThreads);
71.     }
72.  }
73. }
```

Nous ne nous attarderons pas sur la partie génération de threads déjà étudiée. Intéressons-nous plutôt à la méthode *Incrémente*, de la ligne 59 utilisée par chaque thread pour incrémenter le compteur statique *cptrThreads* de la ligne 8.

1. ligne 62 : le compteur est lu
2. ligne 66 : le thread s'arrête 1 s. Il perd donc le processeur
3. ligne 68 : 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 avec 5 threads sont les suivants :

```
1.  A 12:00:56, le thread 3  a lu la valeur du compteur : 0
2.  A 12:00:56, le thread 2  a lu la valeur du compteur : 0
3.  A 12:00:56, le thread 1  a lu la valeur du compteur : 0
4.  A 12:00:56, le thread 0  a lu la valeur du compteur : 0
5.  A 12:00:56, le thread 4  a lu la valeur du compteur : 0
6.  A 12:00:57, le thread 3  a écrit la valeur du compteur : 1
7.  A 12:00:57, le thread 2  a écrit la valeur du compteur : 1
8.  A 12:00:57, le thread 1  a écrit la valeur du compteur : 1
9.  A 12:00:57, le thread 0  a écrit la valeur du compteur : 1
10. A 12:00:57, le thread 4  a écrit la valeur du compteur : 1
11. Nombre de threads générés : 1
```

A la lecture de ces résultats, on voit bien ce qui se passe :

- ligne 1 : un premier thread lit le compteur. Il trouve 0. Il s'arrête 1 s donc perd le processeur
- ligne 2 : 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 incrémentée. Il s'arrête lui aussi 1 s et perd à son tour le processeur.
- lignes 1-5 : en 1 s, les 5 threads ont le temps de passer tous et de lire tous la valeur 0.
- lignes 6-10 : 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) en ligne 11.

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.

Dans notre exemple, la section critique est le code situé entre la lecture du compteur et l'écriture de sa nouvelle valeur :

```
1.          // lecture compteur
2.          int valeur = cptrThreads;
3.          // attente
4.          Thread.Sleep(1000);
5.          // incrémentation compteur
6.  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 divers outils pour assurer l'entrée unitaire dans les sections critiques de code. Nous en voyons quelques-uns maintenant.

10.5.2 La clause lock

La clause **lock** permet de délimiter une section critique de la façon suivante :

```
lock(obj){section critique}
```

obj doit être une référence d'objet visible par tous les threads exécutant la section critique. La clause **lock** assure qu'un seul thread à la fois exécutera la section critique. L'exemple précédent est réécrit comme suit :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program2 {
6.
7.      // variables de classe
8.      static int cptrThreads = 0;   // compteur de threads
9.      static object synchro = new object(); // objet de synchronisation
10.
11.     //main
12.     public static void Main(string[] args) {
13.  ...
14.         // attente de la fin des threads
15.         Thread.CurrentThread.Name = "Main";
16.         for (int i = nbThreads - 1; i >= 0; i--) {
17.            Console.WriteLine("A {0:hh:mm:ss}, le thread {1} attend la fin du thread {2}",
    DateTime.Now, Thread.CurrentThread.Name, threads[i].Name);
18.            threads[i].Join();
19.            Console.WriteLine("A {0:hh:mm:ss}, le thread {1} a été prévenu de la fin du thread {2}",
    DateTime.Now, Thread.CurrentThread.Name, threads[i].Name);
20.         }
21.         // affichage compteur
22.         Console.WriteLine("Nombre de threads générés : " + cptrThreads);
23.     }
24.
25.     public static void Incrémente() {
26.         // augmente le compteur de threads
27.         // un accès exclusif au compteur est demandé
28.         Console.WriteLine("A {0:hh:mm:ss}, le thread {1}  attend l'autorisation d'entrer dans la
    section critique", DateTime.Now, Thread.CurrentThread.Name);
29.         lock (synchro) {
```

```
30.           // lecture compteur
31.           int valeur = cptrThreads;
32.           // suivi
33.           Console.WriteLine("A {0:hh:mm:ss}, le thread {1}  a lu la valeur du compteur : {2}",
    DateTime.Now, Thread.CurrentThread.Name, cptrThreads);
34.           // attente
35.           Thread.Sleep(1000);
36.           // incrémentation compteur
37.           cptrThreads = valeur + 1;
38.           // suivi
39.           Console.WriteLine("A {0:hh:mm:ss}, le thread {1}  a écrit la valeur du compteur : {2}",
    DateTime.Now, Thread.CurrentThread.Name, cptrThreads);
40.       }
41.       Console.WriteLine("A {0:hh:mm:ss}, le thread {1} a quitté la section critique",
    DateTime.Now, Thread.CurrentThread.Name);
42.    }
43. }
44. }
```

- ligne 9 : *synchro* est l'objet qui va permettre la synchronisation de tous les threads.
- lignes 16-23 : la méthode [Main] attend les threads dans l'ordre inverse de leur création.
- lignes 29-40 : la section critique de la méthode *Incrémente* a été encadrée par la clause *lock*.

Les résultats obtenus avec 3 threads sont les suivants :

```
1.  A 09:37:09, le thread 0 attend l'autorisation d'entrer dans la section critique
2.  A 09:37:09, le thread 0 a lu la valeur du compteur : 0
3.  A 09:37:09, le thread 1 attend l'autorisation d'entrer dans la section critique
4.  A 09:37:09, le thread 2 attend l'autorisation d'entrer dans la section critique
5.  A 09:37:09, le thread Main attend la fin du thread 2
6.  A 09:37:10, le thread 0 a écrit la valeur du compteur : 1
7.  A 09:37:10, le thread 1 a lu la valeur du compteur : 1
8.  A 09:37:10, le thread 0 a quitté la section critique
9.  A 09:37:11, le thread 1 a écrit la valeur du compteur : 2
10. A 09:37:11, le thread 1 a quitté la section critique
11. A 09:37:11, le thread 2 a lu la valeur du compteur : 2
12. A 09:37:12, le thread 2 a écrit la valeur du compteur : 3
13. A 09:37:12, le thread 2 a quitté la section critique
14. A 09:37:12, le thread Main a été prévenu de la fin du thread 2
15. A 09:37:12, le thread Main attend la fin du thread 1
16. A 09:37:12, le thread Main a été prévenu de la fin du thread 1
17. A 09:37:12, le thread Main attend la fin du thread 0
18. A 09:37:12, le thread Main a été prévenu de la fin du thread 0
19. Nombre de threads générés : 3
```

- le thread 0 entre le 1er dans la section critique : lignes 1, 2, 6, 8
- les deux autres threads vont être bloqués tant que le thread 0 ne sera pas sorti de la section critique : lignes 3 et 4
- le thread 1 passe ensuite : lignes 7, 9, 10
- le thread 2 passe ensuite : lignes 11, 12, 13
- ligne 14 : le thread Main qui attendait la fin du thread 2 est prévenu
- ligne 15 : le thread Main attend maintenant la fin du thread 1. Celui-ci est déjà terminé. Le thread Main en est prévenu immédiatement, ligne 16.
- lignes 17-18 : le même processus se passe avec le thread 0
- ligne 19 : le nombre de threads est correct

10.5.3 La classe Mutex

La classe **System.Threading.Mutex** permet elle aussi de délimiter des sections critiques. Elle diffère de la clause **lock** en terme de visibilité :

- la clause **lock** permet de synchroniser des threads d'une même application
- la classe **Mutex** permet de synchroniser des threads de différentes applications.

Nous utiliserons le constructeur et les méthodes suivants :

`public Mutex()`	crée un *Mutex* 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

```
public void
ReleaseMutex()
```

le *Mutex* M qu'il détient. Plusieurs threads peuvent ainsi être bloqués en attente du *Mutex* M. 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 donner celui-ci 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 ne faut pas oublier de libérer un *Mutex* devenu inutile afin qu'un autre thread puisse entrer dans la section critique, sinon les threads en attente du *Mutex* jamais libéré n'auront jamais accès au processeur.

Si nous mettons en pratique sur l'exemple précédent ce que nous venons de voir, notre application devient la suivante :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program3 {
6.
7.      // variables de classe
8.      static int cptrThreads = 0;   // compteur de threads
9.      static Mutex synchro = new Mutex(); // objet de synchronisation
10.
11.     //main
12.     public static void Main(string[] args) {
13.  ...
14.     }
15.
16.     public static void Incrémente() {
17. ....
18.         synchro.WaitOne();
19.         try {
20. ...
21.         } finally {
22. ...
23.            synchro.ReleaseMutex();
24.         }
25.     }
26.  }
27. }
```

- ligne 9 : l'objet de synchronisation des threads est désormais un *Mutex*.
- ligne 18 : début de la section critique - un seul thread doit y entrer. On se bloque jusqu'à ce que le *Mutex synchro* soit libre.
- ligne 33 : parce qu'un *Mutex* doit toujours être libéré, exception ou pas, on gère la section critique avec un *try / finally* afin de libérer le *Mutex* dans le *finally*.
- ligne 23 : le *Mutex* est libéré une fois la section critique passée.

Les résultats obtenus sont les mêmes que précédemment.

10.5.4 La classe AutoResetEvent

Un objet *AutoResetEvent* est une barrière ne laissant passer qu'un thread à la fois, comme les deux outils précédents *lock* et *Mutex*. On construit un objet *AutoResetEvent* de la façon suivante :

```
AutoResetEvent barrière=new AutoresetEvent(bool état);
```

Le booléen *état* indique l'état fermé (false) ou ouvert (true) de la barrière. Un thread voulant passer la barrière l'indiquera de la façon suivante :

```
barrière.WaitOne();
```

- si la barrière est ouverte, le thread passe et la barrière est refermée derrière lui. Si plusieurs threads attendaient, on est assuré qu'un seul passera.
- si la barrière est fermée, le thread est bloqué. Un autre thread l'ouvrira lorsque le moment sera venu. Ce moment est entièrement dépendant du problème traité. La barrière sera ouverte par l'opération :

```
barrière.Set();
```

Il peut arriver qu'un thread veuille fermer une barrière. Il pourra le faire par :

```
barrière.Reset();
```

Si dans l'exemple précédent, on remplace l'objet *Mutex* par un objet de type *AutoResetEvent*, le code devient le suivant :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program4 {
6.
7.      // variables de classe
8.      static int cptrThreads = 0;   // compteur de threads
9.      static EventWaitHandle synchro = new AutoResetEvent(false); // objet de synchronisation
10.
11.     //main
12.     public static void Main(string[] args) {
13. ....
14.        // on ouvre la barrière de la section critique
15.        Console.WriteLine("A {0:hh:mm:ss}, le thread {1} ouvre la barrière de la section critique",
    DateTime.Now, Thread.CurrentThread.Name);
16.        synchro.Set();
17.        // attente de la fin des threads
18. ...
19.        // affichage compteur
20.        Console.WriteLine("Nombre de threads générés : " + cptrThreads);
21.     }
22.
23.     public static void Incrémente() {
24.        // augmente le compteur de threads
25.        // un accès exclusif au compteur est demandé
26. ...
27.        synchro.WaitOne();
28.        try {
29. ...
30.        } finally {
31.           // on relâche la ressource
32. ...
33.           synchro.Set();
34.        }
35.     }
36.  }
37. }
```

- ligne 9 : la barrière est créée fermée. Elle sera ouverte par le thread *Main* ligne 16.
- ligne 27 : le thread chargé d'incrémenter le compteur de threads demande l'autorisation d'entrer dans la section critique. Les différents threads vont s'accumuler devant la barrière fermée. Lorque le thread *Main* va l'ouvrir, l'un des threads en attente va passer.
- ligne 33 : lorsqu'il a terminé son travail, il rouvre la barrière permettant à un autre thread d'entrer.

On obtient des résultats analogues aux précédents.

10.5.5 La classe Interlocked

La classe **Interlocked** permet de rendre **atomique** un groupe d'opérations. Dans un groupe d'opérations *atomique*, soit toutes les opérations sont exécutées par le thread qui exécute le groupe soit aucune. On ne reste pas dans un état ou certaines ont été exécutées et d'autres pas. Les objets de synchronisation *lock*, *Mutex*, *AutoResetEvent* ont toutes pour but de rendre *atomique* un groupe d'opérations. Ce résultat est obtenu au prix du blocage de threads. La classe **Interlocked** permet, pour des opérations simples mais assez fréquentes, d'éviter le blocage de threads. La classe **Interlocked** offre les méthodes statiques suivantes :

Name	Description
Add	Overloaded. Adds two integers and replaces the first integer with the sum, as an atomic operation.
CompareExchange	Overloaded. Compares two values for equality and, if they are equal, replaces one of the values.
Decrement	Overloaded. Decrements a specified variable and stores the result, as an atomic operation.
Exchange	Overloaded. Sets a variable to a specified value as an atomic operation.
Increment	Overloaded. Increments a specified variable and stores the result, as an atomic operation.
Read	Returns a 64-bit value, loaded as an atomic operation.

La méthode *Increment* a la signature suivante :

```
public static int Increment(ref int location);
```

Elle permet d'incrémenter de 1 le paramètre *location*. L'opération est garantie *atomique*.

Notre programme de comptage de threads peut alors être le suivant :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program5 {
6.
7.      // variables de classe
8.      static int cptrThreads = 0;   // compteur de threads
9.
10.     //main
11.     public static void Main(string[] args) {
12. ...
13.     }
14.
15.     public static void Incrémente() {
16.        // incrémente le compteur de threads
17.        Interlocked.Increment(ref cptrThreads);
18.     }
19.  }
20. }
```

- ligne 17 : le compteur de threads est incrémenté de façon atomique.

10.6 Accès concurrents à des ressources partagées multiples

10.6.1 Un exemple

Dans nos exemples précédents, une unique ressource était partagée par les différents threads. La situation peut se compliquer s'il y en a plusieurs et qu'elles sont dépendantes les unes des autres. Une situation d'interblocage peut notamment survenir. Cette situation appelée également *deadlock* est celle 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.

Considérons l'exemple suivant :

1. On a un tableau dans lequel des threads viennent déposer des données (les écrivains) et d'autres viennent les lire (les lecteurs).
2. Les écrivains sont égaux entre-eux mais exclusifs : un seul écrivain à la fois peut déposer ses données dans le tableau.

3. Les lecteurs sont égaux entre-eux mais exclusifs : un seul lecteur à la fois peut lire les données déposées dans le tableau.
4. Un lecteur ne peut lire les données du tableau que lorsqu'un écrivain en a déposé dedans et un écrivain ne peut déposer de nouvelles données dans le tableau que lorsque celles qui y sont ont été lues par un lecteur.

On peut distinguer deux ressources partagées :
- le tableau en écriture : un seul écrivain à la fois doit y avoir accès.
- le tableau en lecture : un seul lecteur à la fois doit y avoir accès.

et un ordre d'utilisation de ces ressources :
- un lecteur doit toujours passer après un écrivain.
- un écrivain doit toujours passer après un lecteur, sauf la 1ère fois.

On peut contrôler l'accès à ces deux ressources avec deux barrières de type *AutoResetEvent* :

- la barrière *peutEcrire* contrôlera l'accès des écrivains au tableau.
- la barrière *peutLire* contrôlera l'accès des lecteurs au tableau.
- la barrière *peutEcrire* sera créée initialement ouverte laissant passer ainsi un 1er écrivain et bloquant tous les autres.
- la barrière *peutLire* sera créée initialement fermée bloquant tous les lecteurs.
- lorsqu'un écrivain aura terminé son travail, il ouvrira la barrière *peutLire* pour laisser entrer un lecteur.
- lorsqu'un lecteur aura terminé son travail, il ouvrira la barrière *peutEcrire* pour laisser entrer un écrivain.

Le programme illustrant cette synchronisation par événements est le suivant :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program {
6.      // utilisation de threads lecteurs et écrivains
7.      // illustre l'utilisation d'événements de synchronisation
8.
9.
10.     // variables de classe
11.     static int[] data = new int[3];   // ressource partagée entre threads lecteur et threads écrivain
12.     static Random objRandom = new Random(DateTime.Now.Second);    // un générateur de nombres aléatoires
13.     static AutoResetEvent peutLire;   // signale qu'on peut lire le contenu de data
14.     static AutoResetEvent peutEcrire; // signale qu'on peut écrire le contenu de data
15.
16.     //main
17.     public static void Main(string[] args) {
18.
19.        // le nbre de threads à générer
20.        const int nbThreads = 2;
21.
22.        // initialisation des drapeaux
23.        peutLire = new AutoResetEvent(false); // on ne peut pas encore lire
24.        peutEcrire = new AutoResetEvent(true);   // on peut déjà écrire
25.
26.        // création des threads lecteurs
27.        Thread[] lecteurs = new Thread[nbThreads];
28.        for (int i = 0; i < nbThreads; i++) {
29.           // création
30.           lecteurs[i] = new Thread(Lire);
31.           lecteurs[i].Name = "L" + i.ToString();
32.           // lancement
33.           lecteurs[i].Start();
34.        }
35.
36.        // création des threads écrivains
37.        Thread[] écrivains = new Thread[nbThreads];
38.        for (int i = 0; i < nbThreads; i++) {
39.           // création
40.           écrivains[i] = new Thread(Ecrire);
41.           écrivains[i].Name = "E" + i.ToString();
42.           // lancement
43.           écrivains[i].Start();
44.        }
45.
46.        //fin de main
47.        Console.WriteLine("Fin de Main...");
48.     }
49.
50.     // lire le contenu du tableau
51.     public static void Lire() {
52. ...
53.     }
54.
55.     // écrire dans le tableau
56.     public static void Ecrire() {
57. ....
58.     }
```

```
59.   }
60. }
```

- ligne 11 : le tableau *data* est la ressource partagée entre les threads lecteurs et écrivains. Elle est partagée en lecture par les threads lecteurs, en écriture par les threads écrivains.
- ligne 13 : l'objet *peutLire* sert à avertir les threads lecteurs qu'ils peuvent lire le tableau *data*. Il est mis à vrai par le thread écrivain ayant rempli le tableau *data*. Il est initialisé à *false*, ligne 23. Il faut qu'un thread écrivain remplisse d'abord le tableau avant de passer l'événement *peutLire* à *vrai*.
- ligne 14 : l'objet *peutEcrire* sert à avertir les threads écrivains qu'ils peuvent écrire dans le tableau *data*. Il est mis à vrai par le thread lecteur ayant exploité la totalité du tableau *data*. Il est initialisé à *true*, ligne 24. En effet, le tableau *data* est libre en écriture.
- lignes 27-34 : création et lancement des threads lecteurs
- lignes 37-44 : création et lancement des threads écrivains

La méthode *Lire* exécutée par les threads lecteurs est la suivante :

```
1.  public static void Lire() {
2.           // suivi
3.           Console.WriteLine("Méthode [Lire] démarrée par le thread n° {0}",
    Thread.CurrentThread.Name);
4.           // on doit attendre l'autorisation de lecture
5.           peutLire.WaitOne();
6.           // lecture tableau
7.           for (int i = 0; i < data.Length; i++) {
8.              //attente 1 s
9.              Thread.Sleep(1000);
10.             // affichage
11.             Console.WriteLine("{0:hh:mm:ss} : Le lecteur {1} a lu le nombre {2}", DateTime.Now,
    Thread.CurrentThread.Name, data[i]);
12.          }
13.          // on peut écrire
14.          peutEcrire.Set();
15.          // suivi
16.          Console.WriteLine("Méthode [Lire] terminée par le thread n° {0}",
    Thread.CurrentThread.Name);
17.       }
```

- ligne 5 : on attend qu'un thread écrivain signale que le tableau a été rempli. Lorsque ce signal sera reçu, un seul des threads lecteurs en attente de ce signal pourra passer.
- lignes 7-12 : exploitation du tableau *data* avec un *Sleep* au milieu pour forcer le thread à perdre le processeur.
- ligne 14 : indique aux threads écrivains que le tableau a été lu et qu'il peut être rempli de nouveau.

La méthode *Ecrire* exécutée par les threads écrivains est la suivante :

```
1.  public static void Ecrire() {
2.           // suivi
3.           Console.WriteLine("Méthode [Ecrire] démarrée par le thread n° {0}",
    Thread.CurrentThread.Name);
4.           // on doit attendre l'autorisation d'écriture
5.           peutEcrire.WaitOne();
6.           // écriture tableau
7.           for (int i = 0; i < data.Length; i++) {
8.              //attente 1 s
9.              Thread.Sleep(1000);
10.             // affichage
11.             data[i] = objRandom.Next(0, 1000);
12.             Console.WriteLine("{0:hh:mm:ss} : L'écrivain {1} a écrit le nombre {2}", DateTime.Now,
    Thread.CurrentThread.Name, data[i]);
13.          }
14.          // on peut lire
15.          peutLire.Set();
16.          // suivi
17.          Console.WriteLine("Méthode [Ecrire] terminée par le thread n° {0}",
    Thread.CurrentThread.Name);
18.       }
```

- ligne 5 : on attend qu'un thread lecteur signale que le tableau a été lu. Lorsque ce signal sera reçu, un seul des threads écrivains en attente de ce signal pourra passer.
- lignes 7-13 : exploitation du tableau *data* avec un *Sleep* au milieu pour forcer le thread à perdre le processeur.
- ligne 15 : indique aux threads lecteurs que le tableau a été rempli et qu'il peut être lu de nouveau.

L'exécution donne les résultats suivants :

```
1.  Méthode [Lire] démarrée par le thread n° L0
2.  Méthode [Lire] démarrée par le thread n° L1
3.  Méthode [Ecrire] démarrée par le thread n° E0
4.  Méthode [Ecrire] démarrée par le thread n° E1
5.  Fin de Main...
6.  02:29:18 : L'écrivain E0 a écrit le nombre 607
7.  02:29:19 : L'écrivain E0 a écrit le nombre 805
8.  02:29:20 : L'écrivain E0 a écrit le nombre 650
9.  Méthode [Ecrire] terminée par le thread n° E0
10. 02:29:21 : Le lecteur L0 a lu le nombre 607
11. 02:29:22 : Le lecteur L0 a lu le nombre 805
12. 02:29:23 : Le lecteur L0 a lu le nombre 650
13. Méthode [Lire] terminée par le thread n° L0
14. 02:29:24 : L'écrivain E1 a écrit le nombre 186
15. 02:29:25 : L'écrivain E1 a écrit le nombre 881
16. 02:29:26 : L'écrivain E1 a écrit le nombre 415
17. Méthode [Ecrire] terminée par le thread n° E1
18. 02:29:27 : Le lecteur L1 a lu le nombre 186
19. 02:29:28 : Le lecteur L1 a lu le nombre 881
20. 02:29:29 : Le lecteur L1 a lu le nombre 415
21. Méthode [Lire] terminée par le thread n° L1
```

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 *Ecrire*
- 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

10.6.2 La classe Monitor

Dans l'exemple précédent :
- il y a deux ressources partagées à gérer
- pour une ressource donnée, les threads sont égaux.

Lorsque les threads écrivains sont bloqués sur l'instruction *peutEcrire.WaitOne*, l'un d'entre-eux, n'importe lequel, est débloqué par l'opération *peutEcrire.Set*. Si l'opération précédente doit ouvrir la barrière à **un écrivain en particulier**, les choses deviennent plus compliquées.

On peut considérer l'analogie avec un établissement accueillant du public à des guichets où chaque guichet est spécialisé. Lorsque le client arrive, il prend un ticket au distributeur de tickets pour le guichet X puis va s'asseoir. Chaque ticket est numéroté et les clients sont appelés par leur numéro via un haut-parleur. Pendant son attente, le client fait ce qu'il veut. Il peut lire ou somnoler. Il est réveillé à chaque fois par le haut-parleur qui annonce que le n° Y est appelé au guichet X. S'il s'agit de lui, le client se lève et accède au guichet X, sinon il continue ce qu'il faisait.

On peut ici fonctionner de façon analogue. Prenons l'exemple des écrivains :

plusieurs écrivains attendent pour un même guichet	leurs threads sont bloqués
le guichet se libère et le n° de l'écrivain suivant est appelé	le thread qui utilisait le tableau en lecture indique aux écrivains que le tableau est disponible. Lui ou un autre thread a fixé le thread écrivain qui doit passer la barrière.
chaque écrivain regarde son n° et seul celui qui a le n° appelé va au guichet. Les autres se remettent en attente.	chaque thread vérifie s'il est l'élu. Si oui, il passe la barrière. Si non, il se remet en attente.

La classe **Monitor** permet de mettre en oeuvre ce scénario.

Name	Description
Enter	Acquires an exclusive lock on the specified object.
Exit	Releases an exclusive lock on the specified object.
Pulse	Notifies a thread in the waiting queue of a change in the locked object's state.
PulseAll	Notifies all waiting threads of a change in the object's state.
TryEnter	Overloaded. Attempts to acquire an exclusive lock on the specified object.
Wait	Overloaded. Releases the lock on an object and blocks the current thread until it reacquires the lock.

Nous décrivons maintenant une construction standard (*pattern*), proposée dans le chapitre *Threading* du livre *C# 3.0* référencé dans l'introduction de ce document, capable de résoudre les problèmes de barrière avec condition d'entrée.

- Tout d'abord, les threads qui se partagent une ressource (le guichet, ...) y accèdent via un objet que nous appellerons un **jeton**. Pour ouvrir la barrière qui mène au guichet, il faut avoir le jeton pour l'ouvrir et il n'y a qu'un seul jeton. Les threads doivent donc se passer le jeton entre-eux.

```
object jeton=new object();
```

- Pour aller au guichet, les threads demandent tout d'abord le jeton :

```
Monitor.Enter(jeton);
```

Si le jeton est libre, il est donné au thread ayant exécuté l'opération précédente, sinon le thread est mis en attente du jeton.

- Si l'accès au guichet se fait de façon non ordonnée, c.a.d. dans le cas où la personne qui entre n'importe pas, l'opération précédente est suffisante. Le thread ayant le jeton va au guichet. Si l'accès se fait de façon ordonnée, le thread qui a le jeton vérifie qu'il remplit la condition pour aller au guichet :

```
while (! jeNeSuisPasCeluiQuiEstAttendu) {Monitor.Wait(jeton);}
```

Si le thread n'est pas celui qui est attendu au guichet, il laisse son tour en redonnant le jeton. Il passe dans un état bloqué. Il sera réveillé dès que le jeton redeviendra disponible pour lui. Il vérifiera alors de nouveau s'il vérifie la condition pour aller au guichet. L'opération *Monitor.Wait(jeton)* qui relâche le jeton ne peut être faite que si le thread **est propriétaire** du jeton. Si ce n'est pas le cas, une exception est lancée.

- Le thread qui vérifie la condition pour aller au guichet y va :

1. // travail au guichet
2. ....

Avant de quitter le guichet, le thread doit rendre son jeton, sinon les threads bloqués en attente de celui-ci le resteront indéfiniment. Il y a deux situations différentes :

- la première situation est celle où le thread ayant le jeton est également celui qui signale aux threads en attente du jeton que celui-ci est libre. Il le fera de la façon suivante :

```
1.  // travail au guichet
2.  ....
3.  // modification condition d'accès au guichet
4.  ...
5.  // réveil des threads en attente du jeton
6.  Monitor.PulseAll(jeton);
7.  // libération du jeton
8.  Monitor.Exit(jeton);
```

Ligne 6, il réveille les threads en attente du jeton. Ce réveil signifie qu'ils deviennent éligibles pour recevoir le jeton. Cela ne veut pas dire qu'ils le reçoivent immédiatement. Ligne 8, le jeton est libéré. Tous les threads éligibles vont recevoir tour à tour le jeton, de façon indéterministe. Cela va leur donner l'occasion de vérifier de nouveau s'ils vérifient la condition d'accès. Le thread ayant libéré le jeton a modifié cette condition ligne 4 afin de permettre à un nouveau thread d'entrer. Le premier qui la vérifie garde le jeton et va au guichet à son tour.

- la seconde situation est celle où le thread ayant le jeton n'est pas celui qui doit signaler aux threads en attente du jeton que celui-ci est libre. Il doit néanmoins le libérer parce que le thread chargé d'envoyer ce signal doit être détenteur du jeton. Il le fera par l'opération :

```
Monitor.Exit(jeton);
```

Le jeton est désormais disponible, mais les threads qui l'attendent (ils ont fait une opération Wait(jeton)) n'en sont pas avertis. Cette tâche est confiée à un autre thread qui à un moment donné exécutera un code similaire au suivant :

```
1.  // acquisition jeton
2.  Monitor.Enter(jeton);
3.  // modification condition d'accès au guichet
4.  ....
5.  // réveil des threads en attente du jeton
6.  Monitor.PulseAll(jeton);
7.  // libération du jeton
8.  Monitor.Exit(jeton);
```

Au final, la construction standard proposée dans le chapitre *Threading* du livre *C# 3.0* est la suivante :

- définir le jeton d'accès au guichet :

```
object jeton=new object();
```

- demander l'accès au guichet :

```
lock(jeton){
   while (! jeNeSuisPasCeluiQuiEstAttendu)
      Monitor.Wait(jeton);
}
// passage au guichet
...
```

```
lock(jeton){...}
```

est équivalent à

```
Monitor.Enter(jeton);
try{...} finally{Monitor.Exit(jeton);}
```

On notera que dans ce schéma le jeton est relâché immédiatement, dès que la barrière est passée. Un autre thread peut alors tester la condition d'accès. La construction précédente laisse donc entrer tous les threads vérifiant la condition d'accès. Si ce n'est pas ce qui est désiré, on pourra écrire :

```
lock(jeton){
   while (! jeNeSuisPasCeluiQuiEstAttendu)
      Monitor.Wait(jeton);
   // passage au guichet
   ...
}
```

où le jeton n'est relâché qu'après le passage au guichet.

- modifier la condition d'accès au guichet et en avertir les autres threads

```
lock(jeton){
   // modifier la condition d'accès au guichet
   ...
   // en avertir les threads en attente du jeton
   Monitor.PulseAll(jeton);
}
```

Ci-dessus, la condition d'accès ne peut être modifiée que par le thread ayant le jeton. On pourra aussi écrire :

```
   // modifier la condition d'accès au guichet
   ...
   // en avertir les threads en attente du jeton
   Monitor.PulseAll(jeton);
   // libérer le jeton
   Monitor.Exit(jeton);
```

si le thread a déjà le jeton.

Muni de ces informations, nous pouvons réécrire l'application lecteurs / écrivains en fixant un ordre des lecteurs et des écrivains pour l'accès à leurs guichets respectifs. Le code est le suivant :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.   class Program2 {
6.      // utilisation de threads lecteurs et écrivains
7.      // illustre l'utilisation d'événements de synchronisation
8.
9.
10.     // variables de classe
11.     static int[] data = new int[3];        // ressource partagée entre threads lecteur et threads
    écrivain
12.     static Random objRandom = new Random(DateTime.Now.Second);  // un générateur de nombres
    aléatoires
13.     static object peutLire = new object();    // signale qu'on peut lire le contenu de data
14.     static object peutEcrire = new object();  // signale qu'on peut écrire le contenu de data
15.     static bool lectureAutorisée = false;  // pour autoriser la lecture du tableau
16.     static bool écritureAutorisée = false; // pour autoriser l'écriture dans le tableau
17.     static string[] ordreLecture; // fixe l'ordre des lecteurs
18.     static string[] ordreEcriture;   // fixe l'ordre des écrivains
19.     static int lecteurSuivant = 0;   // indique le n° du lecteur suivant
20.     static int écrivainSuivant = 0;  // indique le n° de l'écrivain suivant
21.
22.     //main
23.     public static void Main(string[] args) {
24.
25.        // le nbre de threads à générer
26.        const int nbThreads = 5;
27.
28.        // création des threads lecteurs
29.        Thread[] lecteurs = new Thread[nbThreads];
30.        for (int i = 0; i < nbThreads; i++) {
31.           // création
32.           lecteurs[i] = new Thread(Lire);
33.           lecteurs[i].Name = "L" + i.ToString();
34.           // lancement
35.           lecteurs[i].Start();
36.        }
37.
38.        // création de l'ordre de lecture
39.        ordreLecture = new string[nbThreads];
40.        for (int i = 0; i < nbThreads; i++) {
41.           ordreLecture[i] = lecteurs[nbThreads - i - 1].Name;
42.           Console.WriteLine("Le lecteur {0} est en position {1}", ordreLecture[i], i);
43.        }
44.
45.        // création des threads écrivains
46.        Thread[] écrivains = new Thread[nbThreads];
47.        for (int i = 0; i < nbThreads; i++) {
48.           // création
49.           écrivains[i] = new Thread(Ecrire);
50.           écrivains[i].Name = "E" + i.ToString();
51.           // lancement
52.           écrivains[i].Start();
53.        }
54.
55.        // création de l'ordre d'écriture
56.        ordreEcriture = new string[nbThreads];
57.        for (int i = 0; i < nbThreads; i++) {
58.           ordreEcriture[i] = écrivains[i].Name;
59.           Console.WriteLine("L'écrivain {0} est en position {1}", ordreEcriture[i], i);
60.        }
61.
62.        // autorisation d'écriture
63.        lock (peutEcrire) {
64.           écritureAutorisée = true;
65.           Monitor.Pulse(peutEcrire);
66.        }
67.
68.
69.        //fin de main
70.        Console.WriteLine("Fin de Main...");
71.     }
72.
73.     // lire le contenu du tableau
```

```
74.     public static void Lire() {
75. ...
76.     }
77.
78.     // écrire dans le tableau
79.     public static void Ecrire() {
80. ...
81.     }
82.  }
83. }
```

L'accès au guichet de lecture est conditionné par les éléments suivants :

- ligne 13 : le jeton *peutLire*
- ligne 15 : le booléen *lectureAutorisée*
- ligne 17 : le tableau ordonné des lecteurs. Les lecteurs vont au guichet de lecture dans l'ordre de ce tableau qui contient leurs noms.
- ligne 19 : *lecteurSuivant* indique le n° du prochain lecteur autorisé à aller au guichet.

L'accès au guichet d'écriture est conditionné par les éléments suivants :

- ligne 14 : le jeton *peutEcrire*
- ligne 16 : le booléen *écritureAutorisée*
- ligne 18 : le tableau ordonné des écrivains. Les écrivains vont au guichet d'écriture dans l'ordre de ce tableau qui contient leurs noms.
- ligne 20 : *écrivainSuivant* indique le n° du prochain écrivain autorisé à aller au guichet.

Les autres éléments du code sont les suivants :

- lignes 29-36 : création et lancement des threads lecteurs. Ils seront tous bloqués car la lecture n'est pas autorisée (ligne 15).
- lignes 39-43 : leur ordre de passage au guichet se fera dans l'ordre inverse de leur création.
- lignes 46-53 : création et lancement des threads érivains. Ils seront tous bloqués car l'écriture n'est pas autorisée (ligne 16).
- lignes 56-60 : leur ordre de passage au guichet se fera dans l'ordre de leur création.
- ligne 64 : on autorise l'écriture
- ligne 65 : on avertit les écrivains que quelque chose a changé.

La méthode *Lire* est la suivante :

```
1.      public static void Lire() {
2.         // suivi
3.         Console.WriteLine("Méthode [Lire] démarrée par le thread n° {0}",
    Thread.CurrentThread.Name);
4.         // on doit attendre l'autorisation de lecture
5.         lock (peutLire) {
6.            while (!lectureAutorisée || ordreLecture[lecteurSuivant] != Thread.CurrentThread.Name) {
7.               Monitor.Wait(peutLire);
8.            }
9.            // lecture tableau
10.           for (int i = 0; i < data.Length; i++) {
11.              //attente 1 s
12.              Thread.Sleep(1000);
13.              // affichage
14.              Console.WriteLine("{0:hh:mm:ss} : Le lecteur {1} a lu le nombre {2}", DateTime.Now,
    Thread.CurrentThread.Name, data[i]);
15.           }
16.           // lecteur suivant
17.           lectureAutorisée = false;
18.           lecteurSuivant++;
19.           // on prévient les écrivains qu'ils peuvent écrire
20.           lock (peutEcrire) {
21.              écritureAutorisée = true;
22.              Monitor.PulseAll(peutEcrire);
23.           }
24.
25.           // suivi
26.           Console.WriteLine("Méthode [Lire] terminée par le thread n° {0}",
    Thread.CurrentThread.Name);
27.        }
28. }
```

- l'ensemble de l'accès au guichet est contrôlé par le *lock* des lignes 5-27. Le lecteur qui récupère le jeton le garde pendant tout son passage au guichet
- lignes 6-8 : un lecteur ayant acquis le jeton ligne 5 le relâche si la lecture n'est pas autorisée ou si ce n'est pas à son tour de passer.
- lignes 10-15 : passage au guichet (exploitation du tableau)
- lignes 17-18 : le thread change les conditions d'accès au guichet de lecture. On notera qu'il a toujours le jeton de lecture et que ces modifications ne peuvent pas encore permettre à un lecteur de passer.
- lignes 20-23 : le thread change les conditions d'accès au guichet d'écriture et prévient tous les écrivains en attente que quelque chose a changé.
- ligne 27 : le *lock* se termine, le jeton *peutLire* est relâché. Un thread de lecture pourrait alors l'acquérir ligne 5 mais il ne passerait pas la condition d'accès puisque le booléen *lectureAutorisée* est à faux. Par ailleurs, tous les threads qui sont en attente du jeton *peutLire* le restent car l'opération *PulseAll*(*peutLire*) n'a pas encore eu lieu.

La méthode *Ecrire* est la suivante :

```
1.        public static void Ecrire() {
2.           // suivi
3.           Console.WriteLine("Méthode [Ecrire] démarrée par le thread n° {0}",
    Thread.CurrentThread.Name);
4.           // on doit attendre l'autorisation d'écriture
5.           lock (peutEcrire) {
6.              while (!écritureAutorisée || ordreEcriture[écrivainSuivant] !=
    Thread.CurrentThread.Name) {
7.                 Monitor.Wait(peutEcrire);
8.              }
9.              // écriture tableau
10.             for (int i = 0; i < data.Length; i++) {
11.                //attente 1 s
12.                Thread.Sleep(1000);
13.                // affichage
14.                data[i] = objRandom.Next(0, 1000);
15.                Console.WriteLine("{0:hh:mm:ss} : L'écrivain {1} a écrit le nombre {2}",
    DateTime.Now, Thread.CurrentThread.Name, data[i]);
16.             }
17.             // écrivain suivant
18.             écritureAutorisée = false;
19.             écrivainSuivant++;
20.             // on réveille les lecteurs en attente du jeton peutLire
21.             lock (peutLire) {
22.                lectureAutorisée = true;
23.                Monitor.PulseAll(peutLire);
24.             }
25.             // suivi
26.             Console.WriteLine("Méthode [Ecrire] terminée par le thread n° {0}",
    Thread.CurrentThread.Name);
27.          }
28. }
```

- l'ensemble de l'accès au guichet d'écriture est contrôlé par le *lock* des lignes 5-27. L'écrivain qui récupère le jeton le garde pendant tout son passage au guichet
- lignes 6-8 : un écrivain ayant acquis le jeton ligne 5 le relâche si l'écriture n'est pas autorisée ou si ce n'est pas à son tour de passer.
- lignes 10-16 : passage au guichet (exploitation du tableau)
- lignes 18-19 : le thread change les conditions d'accès au guichet d'écriture. On notera qu'il a toujours le jeton d'écriture et que ces modifications ne peuvent pas encore permettre à un écrivain de passer.
- lignes 21-24 : le thread change les conditions d'accès au guichet de lecture et prévient tous les lecteurs en attente que quelque chose a changé.
- ligne 27 : le *lock* se termine, le jeton *peutEcrire* est relâché. Un thread d'écriture pourrait alors l'acquérir ligne 5 mais il ne passerait pas la condition d'accès puisque le booléen *écritureAutorisée* est à faux. Par ailleurs, tous les threads qui sont en attente du jeton *peutEcrire* le restent dans l'attente d'une nouvelle opération *PulseAll*(*peutEcrire*).

Un exemple d'exécution est le suivant :

```
1.  Méthode [Lire] démarrée par le thread n° L0
2.  Méthode [Lire] démarrée par le thread n° L2
3.  Méthode [Lire] démarrée par le thread n° L1
4.  Le lecteur L2 est en position 0
5.  Le lecteur L1 est en position 1
6.  Le lecteur L0 est en position 2
7.  Méthode [Ecrire] démarrée par le thread n° E0
8.  Méthode [Ecrire] démarrée par le thread n° E1
```

```
9.  L'écrivain E0 est en position 0
10. L'écrivain E1 est en position 1
11. L'écrivain E2 est en position 2
12. Fin de Main...
13. Méthode [Ecrire] démarrée par le thread n° E2
14. 12:09:05 : L'écrivain E0 a écrit le nombre 815
15. 12:09:06 : L'écrivain E0 a écrit le nombre 990
16. 12:09:07 : L'écrivain E0 a écrit le nombre 563
17. Méthode [Ecrire] terminée par le thread n° E0
18. 12:09:08 : Le lecteur L2 a lu le nombre 815
19. 12:09:09 : Le lecteur L2 a lu le nombre 990
20. 12:09:10 : Le lecteur L2 a lu le nombre 563
21. Méthode [Lire] terminée par le thread n° L2
22. 12:09:11 : L'écrivain E1 a écrit le nombre 411
23. 12:09:12 : L'écrivain E1 a écrit le nombre 11
24. 12:09:13 : L'écrivain E1 a écrit le nombre 54
25. Méthode [Ecrire] terminée par le thread n° E1
26. 12:09:14 : Le lecteur L1 a lu le nombre 411
27. 12:09:15 : Le lecteur L1 a lu le nombre 11
28. 12:09:16 : Le lecteur L1 a lu le nombre 54
29. Méthode [Lire] terminée par le thread n° L1
30. 12:09:17 : L'écrivain E2 a écrit le nombre 698
31. 12:09:18 : L'écrivain E2 a écrit le nombre 448
32. 12:09:19 : L'écrivain E2 a écrit le nombre 472
33. Méthode [Ecrire] terminée par le thread n° E2
34. 12:09:20 : Le lecteur L0 a lu le nombre 698
35. 12:09:21 : Le lecteur L0 a lu le nombre 448
36. 12:09:22 : Le lecteur L0 a lu le nombre 472
37. Méthode [Lire] terminée par le thread n° L0
```

10.7 Les pools de threads

Jusqu'à maintenant, pour gérer des threads :

- nous les avons créés par *Thread T=new Thread(...)*
- puis exécutés par *T.Start()*

Nous avons vu au chapitre "Bases de données" qu'avec certains SGBD il était possible d'avoir des pools de connexions ouvertes :

- *n* connexions sont ouvertes au démarrage du pool
- lorsqu'un thread demande une connexion, on lui donne l'une des connexions ouvertes du pool
- lorsque le thread ferme la connexion, elle n'est pas fermée mais rendue au pool

L'usage d'un pool de connexions est transparent au niveau du code. L'intérêt réside dans l'amélioration des performances : l'ouverture d'une connexion coûte cher. Ici 10 connexions ouvertes peuvent servir des centaines de demandes.

Un système analogue existe pour les threads :

- *min* threads sont créés au démarrage du pool. La valeur de *min* est fixée avec la méthode *ThreadPool.SetMinThreads(min1,min2)*. Un pool de threads peut être utilisé pour exécuter des tâches bloquantes ou non bloquantes dites asynchrones. Le premier paramètre *min1* fixe le nombre de threads bloquants, le second *min2* le nombre de threads asynchrones. Les valeurs actuelles de ces deux valeurs peuvent être obtenues par *ThreadPool.GetMinThreads(out min1,out min2)*.
- si ce nombre n'est pas suffisant, le pool va créer d'autres threads pour répondre aux demande jusqu'à la limite de *max* threads. La valeur de *max* est fixée avec la méthode *ThreadPool.SetMaxThreads(max1,max2)*. Les deux paramètres ont la même signification que dans la méthode *SetMinThreads*. Les valeurs actuelles de ces deux valeurs peuvent être obtenues par *ThreadPool.GetMaxThreads(out max1,out max2)*. Lorsque les *max1* threads auront été atteints, les demandes de threads pour tâches bloquantes seront mises en attente d'un thread libre dans le pool.

Un pool de threads offre divers avantages :

- comme pour le pool de connexions, on économise sur le temps de création des threads : 10 threads peuvent servir des centaines de demandes.
- on sécurise l'application : en fixant un nombre maximum de threads, on évite l'asphyxie de l'application par des demandes trop nombreuses. Celles-ci seront mises en file d'attente.

Pour donner une tâche à un thread du pool, on utilise l'une des deux méthodes :

1. *ThreadPool.QueueWorkItem(WaitCallBack)*
2. *ThreadPool.QueueWorkItem(WaitCallBack,object)*

où *WaitCallBack* est toute méthode ayant la signature *void WaitCallBack(object)*. La méthode 1 demande à un thread d'exécuter la méthode *WaitCallBack* sans lui passer de paramètre. La méthode 2 fait la même chose mais en passant un paramètre de type *object* à la méthode *WaitCallBack*.

Voici un programme illustrant ces concepts :

```
1.  using System;
2.  using System.Threading;
3.
4.  namespace Chap8 {
5.     class Program {
6.        public static void Main() {
7.           // init Thread courant
8.           Thread main = Thread.CurrentThread;
9.           // on fixe un nom au Thread
10.          main.Name = "Main";
11.
12.          // on utilise un pool de threads
13.          int min1, min2;
14.          // on fixe le nombre minimal de threads bloquants
15.          ThreadPool.GetMinThreads(out min1, out min2);
16.          Console.WriteLine("Nombre minimum de tâches bloquantes dans le pool : {0}", min1);
17.          Console.WriteLine("Nombre minimum de tâches asynchrones dans le pool : {0}", min2);
18.          ThreadPool.SetMinThreads(3, min2);
19.          ThreadPool.GetMinThreads(out min1, out min2);
20.          Console.WriteLine("Nombre minimum de tâches bloquantes dans le pool après changement :
    {0}", min1);
21.          // on fixe le nombre maximal de threads bloquants
22.          int max1, max2;
23.          ThreadPool.GetMaxThreads(out max1, out max2);
24.          Console.WriteLine("Nombre maximum de tâches bloquantes dans le pool : {0}", max1);
25.          Console.WriteLine("Nombre maximum de tâches asynchrones dans le pool : {0}", max2);
26.          ThreadPool.SetMaxThreads(5, max2);
27.          ThreadPool.GetMaxThreads(out max1, out max2);
28.          Console.WriteLine("Nombre maximum de tâches bloquantes dans le pool après changement :
    {0}", max1);
29.          // on exécute 7 threads
30.          for (int i = 0; i < 7; i++) {
31.             // on lance l'exécution du thread i dans un pool
32.             ThreadPool.QueueUserWorkItem(Sleep, new Data2 { Numéro = i.ToString(), Début =
    DateTime.Now, Durée = i + 10 });
33.          }
34.          // fin de main
35.          Console.Write("Tapez [entrée] pour terminer le thread {0} à {1:hh:mm:ss:FF}", main.Name,
    DateTime.Now);
36.          // attente
37.          Console.ReadLine();
38.       }
39.
40.       public static void Sleep(object infos) {
41.          // on récupère le paramètre
42.          Data2 data = infos as Data2;
43.          Console.WriteLine("A {2:hh:mm:ss:FF}, le thread n° {0} va dormir pendant {1} seconde(s)",
    data.Numéro, data.Durée,DateTime.Now);
44.          // état du pool
45.          int cpt1, cpt2;
46.          ThreadPool.GetAvailableThreads(out cpt1, out cpt2);
47.          Console.WriteLine("Nombre de threads pour tâches bloquantes disponibles dans le pool :
    {0}", cpt1);
48.          // mise en sommeil pendant Durée secondes
49.          Thread.Sleep(data.Durée * 1000);
50.          // fin d'exécution
51.          data.Fin = DateTime.Now;
52.          Console.WriteLine("A {3:hh:mm:ss:FF}, le thread n° {0} se termine. Il était programmé pour
    durer {1} seconde(s). Il a duré {2} seconde(s)", data.Numéro, data.Durée, data.Fin -
    data.Début,DateTime.Now);
53.       }
54.    }
55.
56.    internal class Data2 {
57.       // informations diverses
58.       public string Numéro { get; set; }
59.       public DateTime Début { get; set; }
60.       public int Durée { get; set; }
61.       public DateTime Fin { get; set; }
62.    }
```

```
63. }
```

- ligne 15-17 : on demande et affiche le nombre minimal actuel des deux types de threads du pool de threads
- ligne 18 : on change le nombre minimal de threads pour tâches bloquantes : 2
- lignes 19-21 : on affiches les nouveaux minima
- lignes 22-28 : on fait de même pour fixer le nombre maximal de threads pour tâches bloquantes : 5
- lignes 30-33 : on fait exécuter 7 tâches dans un pool de 5 threads. 5 tâches devraient obtenir 1 thread, les 2 premières rapidement puisque 2 threads sont toujours présents, les 3 autres avec un délai d'attente de 0.5 seconde. 2 tâches devraient attendre qu'un thread se libère.
- ligne 32 : les tâches exécutent la méthode *Sleep* des lignes 40-54 en lui passant un paramètre de type *Data2* défini lignes 56-62.
- ligne 40 : la méthode *Sleep* exécutée par les tâches
- ligne 42 : on récupère le paramètre passé à la méthode *Sleep*.
- ligne 43 : la tâche s'identifie sur la console
- lignes 45-47 : on affiche le nombre de threads actuellement disponibles. On veut voir comment il évolue.
- ligne 49 : la tâche s'arrête quelques secondes (tâche bloquante).
- ligne 52 : lorsqu'elle se réveille, on fait afficher quelques informations sur son compte.

Les résultats obtenus sont les suivants.

Pour les nombres *min* et *max* de threads dans le pool :

```
1.  Nombre minimum de tâches bloquantes dans le pool : 2
2.  Nombre minimum de tâches asynchrones dans le pool : 2
3.  Nombre minimum de tâches bloquantes dans le pool après changement : 3
4.  Nombre maximum de tâches bloquantes dans le pool : 500
5.  Nombre maximum de tâches asynchrones dans le pool : 1000
6.  Nombre maximum de tâches bloquantes dans le pool après changement : 5
```

Pour l'exécution des 7 threads :

```
1.  A 03:07:37:04, le thread n° 0 va dormir pendant 10 seconde(s)
2.  Nombre de threads pour tâches bloquantes disponibles dans le pool : 3
3.  A 03:07:37:04, le thread n° 2 va dormir pendant 12 seconde(s)
4.  Nombre de threads pour tâches bloquantes disponibles dans le pool : 2
5.  A 03:07:37:04, le thread n° 1 va dormir pendant 11 seconde(s)
6.  Nombre de threads pour tâches bloquantes disponibles dans le pool : 2
7.  A 03:07:38:04, le thread n° 3 va dormir pendant 13 seconde(s)
8.  Nombre de threads pour tâches bloquantes disponibles dans le pool : 1
9.  A 03:07:38:54, le thread n° 4 va dormir pendant 14 seconde(s)
10. Nombre de threads pour tâches bloquantes disponibles dans le pool : 0
11. A 03:07:47:04, le thread n° 0 se termine. Il était programmé pour durer 10 seconde(s). Il a duré
    00:00:10 seconde(s)
12. A 03:07:47:04, le thread n° 5 va dormir pendant 15 seconde(s)
13. Nombre de threads pour tâches bloquantes disponibles dans le pool : 0
14. A 03:07:48:04, le thread n° 1 se termine. Il était programmé pour durer 11 seconde(s). Il a duré
    00:00:11 seconde(s)
15. A 03:07:48:04, le thread n° 6 va dormir pendant 16 seconde(s)
16. Nombre de threads pour tâches bloquantes disponibles dans le pool : 0
17. A 03:07:49:04, le thread n° 2 se termine. Il était programmé pour durer 12 seconde(s). Il a duré
    00:00:12 seconde(s)
18. A 03:07:51:04, le thread n° 3 se termine. Il était programmé pour durer 13 seconde(s). Il a duré
    00:00:14 seconde(s)
19. A 03:07:52:54, le thread n° 4 se termine. Il était programmé pour durer 14 seconde(s). Il a duré
    00:00:15.5000000 seconde(s)
20. A 03:08:02:04, le thread n° 5 se termine. Il était programmé pour durer 15 seconde(s). Il a duré
    00:00:25 seconde(s)
21. A 03:08:04:04, le thread n° 6 se termine. Il était programmé pour durer 16 seconde(s). Il a duré
    00:00:27 seconde(s)
```

- lignes 1-6 : les 3 premières tâches sont exécutées tour à tour. Elles trouvent imédiatement 1 thread disponible (*MinThreads*=3) puis se mette en sommeil.
- lignes 7-9 : pour les tâches 3 et 4, c'est un peu plus long. Pour chacun d'eux il n'y avait pas de thread libre. Il a fallu en créer un. Ce mécanisme est possible jusqu'à 5 (MaxThreads=5).
- ligne 10 : il n'y a plus de threads disponibles : les tâches 5 et 6 vont devoir attendre.
- lignes 11-12 : la tâche 0 se termine. La tâche 5 prend son thread.
- lignes 13-14 : la tâche 1 se termine. La tâche 6 prend son thread.
- lignes 17-21 : les tâches se terminent les unes après les autres.

10.8 La classe BackgroundWorker

10.8.1 Exemple 1

La classe *BackgroundWorker* appartient à l'espace de noms [System.ComponentModel]. Elle s'utilise comme un thread mais présente des particularités qui peuvent la rendre, dans certains cas, plus intéressante que la classe [Thread] :

- elle émet les événements suivants :
  - *DoWork* : un thread a demandé l'exécution du *BackgroundWorker*
  - *ProgressChanged* : l'objet *BackgroundWorker* a exécuté la méthode *ReportProgress*. Celle-ci sert à donner un pourcentage d'exécution.
  - *RunWorkerCompleted* : l'objet *BackgroundWorker* a terminé son travail. Il a pu le terminer normalement ou sur annulation ou exception.

  Ces événements rendent le *BackgroundWorker* utile dans les interfaces graphiques : une tâche longue sera confiée à un *BackgroundWorker* qui pourra rendre compte de son avancement avec l'événement *ProgressChanged* et sa fin avec l'événement *RunWorkerCompleted*. Le travail à effectuer par le *BackgroundWorker* sera gfait par une méthode qui aura été associée à l'événement *DoWork*.
- il est possible de demander son annulation. Dans une interface graphique, une tâche longue pourra ainsi être annulée par l'utilisateur.
- les objets *BackgroundWorker* appartiennent à un pool et sont recyclés selon les besoins. Une application qui a besoin d'un objet *BackgroundWorker* l'obtiendra auprès du pool qui lui donnera un thread déjà existant mais inutilisé. Le fait de recycler ainsi les threads plutôt que de créer à chaque fois un thread neuf, améliore les performances.

Nous utilisons cet outil sur l'application précédente dans le cas où l'accès au guichet est non contrôlé :

```
1.  using System;
2.  using System.Threading;
3.  using System.ComponentModel;
4.
5.  namespace Chap8 {
6.   class Program2 {
7.      // utilisation de threads lecteurs et écrivains
8.      // illustre l'utilisation simultanée de ressources partagées et de synchronisation
9.
10.     // variables de classe
11.     const int nbThreads = 2;                // nbre de threads au total
12.     static int nbLecteursTerminés = 0;      // nbre de threads terminés
13.     static int[] data = new int[5];         // tableau partagé entre threads lecteur et threads écrivain
14.     static object appli;                    // synchronise l'accès au nbre de threads terminés
15.     static Random objRandom = new Random(DateTime.Now.Second);   // un générateur de nombres aléatoires
16.     static AutoResetEvent peutLire;      // signale qu'on peut lire le contenu du tableau
17.     static AutoResetEvent peutEcrire;    // signale qu'on peut écrire dans le tableau
18.     static AutoResetEvent finLecteurs;   // signale la fin des lecteurs
19.
20.     //main
21.     public static void Main(string[] args) {
22.
23.        // on donne un nom au thread
24.        Thread.CurrentThread.Name = "Main";
25.
26.        // initialisation des drapeaux
27.        peutLire = new AutoResetEvent(false);    // on ne peut pas encore lire
28.        peutEcrire = new AutoResetEvent(true);   // on peut déjà écrire
29.        finLecteurs = new AutoResetEvent(false); // appli non terminée
30.
31.        // synchronise l'accès au compteur de threads terminés
32.        appli = new object();
33.
34.        // création des threads lecteurs
35.        MyBackgroundWorker[] lecteurs = new MyBackgroundWorker[nbThreads];
36.        for (int i = 0; i < nbThreads; i++) {
37.           // création
38.           lecteurs[i] = new MyBackgroundWorker();
39.           lecteurs[i].Numéro = "L" + i;
40.           lecteurs[i].DoWork += Lire;
41.           lecteurs[i].RunWorkerCompleted += EndLecteur;
42.           // lancement
43.           lecteurs[i].RunWorkerAsync();
44.        }
45.
46.        // création des threads écrivains
47.        MyBackgroundWorker[] écrivains = new MyBackgroundWorker[nbThreads];
48.        for (int i = 0; i < nbThreads; i++) {
49.           // création
50.           écrivains[i] = new MyBackgroundWorker();
51.           écrivains[i].Numéro = "E" + i;
```

```
52.             écrivains[i].DoWork += Ecrire;
53.             // lancement
54.             écrivains[i].RunWorkerAsync();
55.         }
56.
57.         // attente de la fin de tous les threads
58.         finLecteurs.WaitOne();
59.         //fin de main
60.         Console.WriteLine("Fin de Main...");
61.     }
62.
63.     public static void EndLecteur(object sender, RunWorkerCompletedEventArgs infos) {
64. ...
65.     }
66.
67.     // lire le contenu du tableau
68.     public static void Lire(object sender, DoWorkEventArgs infos) {
69. ...
70.     }
71.
72.     // écrire dans le tableau
73.     public static void Ecrire(object sender, DoWorkEventArgs infos) {
74. ...
75.     }
76.  }
77.
78.  // thread
79.  internal class MyBackgroundWorker : BackgroundWorker {
80.     // informations diverses
81.     public string Numéro { get; set; }
82.  }
83.
84. }
```

Nous ne détaillons que les changements :

- la classe *Thread* est remplacée par la classe *MyBackgroundWorker* des lignes 79-82. La classe *BackgroundWorker* a été dérivée afin de donner un numéro au thread. On aurait pu procéder différemment en passant un objet à la méthode *RunWorkerAsync* des lignes 43 et 54, objet contenant le n° du thread.
- ligne 58 : la méthode *Main* se termine après que tous les threads lecteurs ont fait leur travail. Pour cela, ligne 12, le compteur *nbLecteursTerminés* compte le nombre de threads lecteurs ayant terminé leur travail. Ce compteur est incrémenté par la méthode *EndLecteur* des lignes 63-65 qui est exécutée à chaque fois qu'un thread lecteur se termine. C'est cette procédure qui contrôle l'événement *AutoResetEvent finLecteurs* de la ligne 18 sur lequel se synchronise, ligne 59, la méthode *Main*.
- ligne 16 : parce que plusieurs threads lecteurs peuvent vouloir incrémenter en même temps le compteur *nbLecteursTerminés*, un accès exclusif à celui-ci est assuré par l'objet de synchronisation *appli*. Ce cas est improbable mais théoriquement possible.
- lignes 35-44 : création des threads lecteurs
  - ligne 38 : création du thread de type *MyBackgroundWorker*
  - ligne 39 : on lui donne un N°
  - ligne 40 : on lui assigne la méthode *Lire* à exécuter
  - ligne 41 : la méthode *EndLecteur* sera exécutée après la fin du thread
  - ligne 43 : le thread est lancé
- lignes 47-55 : création des threads écrivains
  - ligne 50 : création du thread de type *MyBackgroundWorker*
  - ligne 51 : on lui donne un N°
  - ligne 52 : on lui assigne la méthode *Ecrire* à exécuter
  - ligne 54 : le thread est lancé

Les méthodes *Lire* et *Ecrire* restent inchangées. La méthode *EndLecteur* est exécutée à la fin de chaque thread lecteur. Son code est le suivant :

```
1.         public static void EndLecteur(object sender, RunWorkerCompletedEventArgs infos) {
2.             // incrémentation nbre de lecteurs terminés
3.             lock (appli) {
4.                 nbLecteursTerminés++;
5.                 if (nbLecteursTerminés == nbThreads)
6.                     finLecteurs.Set();
7.             }
8.  }
```

Le rôle de la méthode *EndLecteur* est d'avertir la méthode *Main* que tous les lecteurs ont fait leur travail.

- ligne 4 : le compteur *nbLecteursTerminés* est incrémenté.

- lignes 5-6 : si tous les lecteurs ont fait leur travail, alors l'événement *finLecteurs* est positionné à vrai afin de prévenir la méthode *Main* qui attend cet événement.
- parce que la procédure *EndLecteur* est exécutée par plusieurs threads, la section critique précédente est protégée par la clause *lock* de la ligne 3.

L'exécution donne des résultats analogues à ceux de la version utilisant des threads.

10.8.2 Exemple 2

Le code suivant illustre d'autres points de la classe *BackgroundWorker* :

- la possibilité d'annuler la tâche
- la remontée d'une exception lancée dans la tâche
- le passage d'un paramètre d'E/S à la tâche

```
1.  using System;
2.  using System.Threading;
3.  using System.ComponentModel;
4.
5.  namespace Chap8 {
6.   class Program3 {
7.
8.      // threads
9.      static BackgroundWorker[] tâches = new BackgroundWorker[5];
10.
11.     public static void Main() {
12.        // init Thread courant
13.        Thread main = Thread.CurrentThread;
14.        // on fixe un nom au Thread
15.        main.Name = "Main";
16.
17.        // création de threads
18.        for (int i = 0; i < tâches.Length; i++) {
19.           // on crée le thread n° i
20.           tâches[i] = new BackgroundWorker();
21.           // on l'initialise
22.           tâches[i].DoWork += Sleep;
23.           tâches[i].RunWorkerCompleted += End;
24.           tâches[i].WorkerSupportsCancellation = true;
25.           // on le lance
26.           tâches[i].RunWorkerAsync(new Data { Numéro = i, Début = DateTime.Now, Durée = i + 1 });
27.        }
28.        // on annule le dernier thread
29.        tâches[4].CancelAsync();
30.
31.        // fin de main
32.        Console.WriteLine("Fin du thread {0}, tapez [entrée] pour terminer...", main.Name);
33.        Console.ReadLine();
34.        return;
35.     }
36.
37.     public static void Sleep(object sender, DoWorkEventArgs infos) {
38. ...
39.     }
40.
41.     public static void End(object sender, RunWorkerCompletedEventArgs infos) {
42. ...
43.     }
44.
45.     internal class Data {
46.        // informations diverses
47.        public int Numéro { get; set; }
48.        public DateTime Début { get; set; }
49.        public int Durée { get; set; }
50.        public DateTime Fin { get; set; }
51.     }
52.  }
53. }
```

- ligne 9 : le tableau de *BackgroundWorker*
- lignes 18-27 : création des threads
- ligne 20 : création du thread
- ligne 22 : le thread exécutera la méthode *Sleep* des lignes 39-41

- ligne 23 : la méthode *End* des lignes 43-45 sera exécutée à la fin du thread
- ligne 24 : le thread pourra être annulé
- ligne 26 : le thread est lancé avec un paramètre de type [Data], défini lignes 49-52. Cet objet a les champs suivants :
  - *Numéro* (entrée) : n° du thread
  - *Début* (entrée) : heure de début d'exécution du thread
  - *Durée* (entrée) : durée d'exécution du *Sleep*
  - *Fin* (sortie) : fin d'exécution du thread
- ligne 29 : le thread n° 4 est annulé

Tous les threads exécutent la méthode *Sleep* suivante :

```
1.        public static void Sleep(object sender, DoWorkEventArgs infos) {
2.           // on exploite le paramètre infos
3.           Data data = (Data)infos.Argument;
4.           // exception pour la tâche n° 3
5.           if (data.Numéro == 3) {
6.              throw new Exception("test....");
7.           }
8.           // mise en sommeil pendant Durée secondes avec un arrêt ttes les secondes
9.           for (int i = 1; i <= data.Durée && !tâches[data.Numéro].CancellationPending; i++) {
10.             // attente d'1 seconde
11.             Thread.Sleep(1000);
12.          }
13.          // fin d'exécution
14.          data.Fin = DateTime.Now;
15.          // on initialise le résultat
16.          infos.Result = data;
17.          infos.Cancel = tâches[data.Numéro].CancellationPending;
18. }
```

- ligne 1 : la méthode *Sleep* a la signature standard des gestionnaires d'événements. Elle reçoit deux paramètres :
  - *sender* : l'émetteur de l'événement, ici le *BackgroundWorker* qui exécute la méthode
  - *infos* : de type *DoWorkEventArgs* qui donne des informations sur l'événement *DoWork*. Ce paramètre sert aussi bien à transmettre des informations au thread qu'à récupérer ses résultats.
- ligne 3 : le paramètre passé à la méthode *RunWorkerAsync* de la tâche est retrouvé dans la propriété *infos.Argument*.
- lignes 5-7 : on lance une exception pour la tâche n° 3
- lignes 9-12 : le thread "dort" *Durée* secondes par tranches d'une seconde afin de permettre le test d'annulation de la ligne 9. Cela simule un travail de longue durée au cours duquel le thread vérifierait régulièrement s'il existe une demande d'annulation. Pour indiquer qu'il a été annulé, le thread doit mettre la propriété *infos.Cancel* à vrai (ligne 17).
- ligne 16 : le thread peut rendre un résultat au thread qui l'a lancé. Il place ce résultat dans *infos.Result*.

Une fois terminés, les threads exécutent la méthode *End* suivante :

```
1.  public static void End(object sender, RunWorkerCompletedEventArgs infos) {
2.           // on exploite le paramètre infos pour afficher le résultat de l'exécution
3.           // exception ?
4.           if (infos.Error != null) {
5.              Console.WriteLine("Le thread {1} a rencontré l'erreur suivante : {0}",
    infos.Error.Message, sender);
6.           } else
7.              if (!infos.Cancelled) {
8.                 Data data = (Data)infos.Result;
9.                 Console.WriteLine("Thread {0} terminé : début {1:hh:mm:ss}, durée programmée {2} s,
    fin {3:hh:mm:ss}, durée effective {4}",
10.                data.Numéro, data.Début, data.Durée, data.Fin, (data.Fin - data.Début));
11.             } else {
12.                Console.WriteLine("Thread {0} annulé", sender);
13.             }
14.       }
```

- ligne 1 : la méthode *End* a la signature standard des gestionnaires d'événements. Elle reçoit deux paramètres :
  - *sender* : l'émetteur de l'événement, ici le *BackgroundWorker* qui exécute la méthode
  - *infos* : de type *RunWorkerCompletedEventArgs* qui donne des informations sur l'événement *RunWorkerCompleted*.
- ligne 4 : le champ *infos.Error* de type *Exception* est renseigné seulement si une exception s'est produite.
- ligne 7 : le champ *infos.Cancelled* de type booléen à la valeur *true* si le thread a été annulé.
- ligne 8 : s'il y a pas eu exception ou annulation, alors *infos.Result* est le résultat du thread exécuté. Utiliser ce résultat s'il y a eu annulation du thread ou si le thread a lancé une exception, provoque une exception. Ainsi lignes 5 et 13, on n'est pas capables d'afficher le n° du thread annulé ou qui a lancé une exception car ce n° est dans *infos.Result*. Ce problème peut être contourné en dérivant la classe *BackgroundWorker* pour y mettre les informations à échanger entre le thread appelant et

le thread appelé comme il a été fait dans l'exemple précédent. On utilise alors l'argument *sender* qui représente le *BackgroundWorker* au lieu de l'argument *infos*.

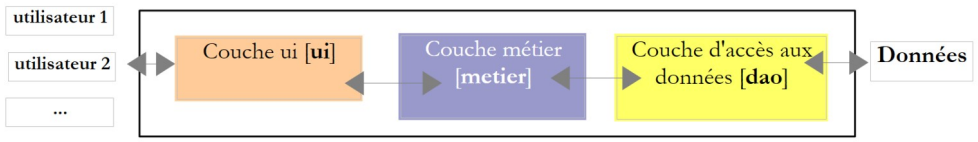
Les résultats d'exécution sont les suivants :

```
1.  Fin du thread Main. Laissez les autres threads se terminer puis tapez [entrée] pour terminer...
2.  Thread 0 terminé : début 05:19:46, durée programmée 1 s, fin 05:19:47, durée effective 00:00:01
3.  Le thread System.ComponentModel.BackgroundWorker a rencontré l'erreur suivante : test....
4.  Thread System.ComponentModel.BackgroundWorker annulé
5.  Thread 1 terminé : début 05:19:46, durée programmée 2 s, fin 05:19:49, durée effective 00:00:03
6.  Thread 2 terminé : début 05:19:46, durée programmée 3 s, fin 05:19:50, durée effective 00:00:04
```

10.9 Données locales à un thread

10.9.1 Le principe

Considérons une application à trois couches :



Supposons que l'application soit multi-utilisateurs, une application web par exemple. Chaque utilisateur est servi par un thread qui lui est dédié. La vie du thread est la suivante :

1. le thread est créé ou demandé à un pool de threads pour satisfaire une demande d'un utilisateur
2. si cette demande nécessite des données, le thread va exécuter une méthode de la couche [ui] qui va appeler une méthode de la couche [metier] qui va à son tour appeler une méthode de la couche [dao].
3. le thread rend la réponse à l'utilisateur. Il disparaît ensuite ou il est recyclé dans un pool de threads.

Dans l'opération 2, il peut être intéressant que le thread ait des données qui lui soient propres, c.a.d. non partagées avec les autres threads. Ces données pourraient par exemple appartenir à l'utilisateur particulier que le thread sert. Ces données pourraient alors être utilisées dans les différentes couches [ui, metier, dao].

La classe *Thread* permet ce scénario grâce à une sorte de dictionnaire privé où les clés seraient de type *LocalDataStoreSlot* :

`public static LocalDataStoreSlot GetNamedDataSlot(`
`  string name`
`)`
	crée une entrée dans le dictionnaire privé du thread pour la clé *name*.

`public static void SetData(`
`  LocalDataStoreSlot slot,`
`  Object data`
`)`
	associe la valeur *data* à la clé *name* du dictionnaire privé du thread

`public static Object GetData(`
`  LocalDataStoreSlot slot`
`)`
	récupère la valeur associée à la clé *name* du dictionnaire privé du thread

Un modèle d'utilisation pourrait être le suivant :

- pour créer un couple (*clé*,*valeur*) associé au thread courant :

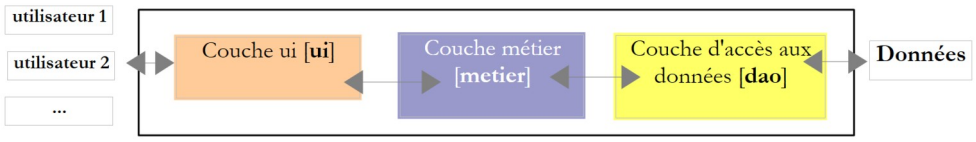
```
Thread.SetData(Thread.GetNamedDataSlot("clé"),valeur);
```

- pour récupérer la valeur associée à *clé* :

```
Thread.GetData(Thread.GetNamedDataSlot("clé"));
```

10.9.2 Application du principe

Considérons l'application à trois couches suivantes :



Supposons que la couche [dao] gère une base d'articles et que son interface soit initialement la suivante :

```
1.  using System.Collections.Generic;
2.
3.  namespace Chap8 {
4.   public interface IDao {
5.      int InsertArticle(Article article);
6.      List<Article> GetAllArticles();
7.      void DeleteAllArticles();
8.   }
9.  }
```

- ligne 5 : pour insérer un article dans la base
- ligne 6 : pour récupérer tous les articles de la base
- ligne 7 : pour supprimer tous les articles de la base

Ultérieurement, apparaît le besoin d'une méthode pour insérer un tableau d'articles à l'aide d'une transaction parce qu'on souhaite fonctionner en tout ou rien : soit tous les articles sont insérés soit aucun. On peut alors modifier l'interface pour intégrer ce nouveau besoin :

```
1.  using System.Collections.Generic;
2.
3.  namespace Chap8 {
4.   public interface IDao {
5.      int InsertArticle(Article article);
6.      void insertArticles(Article[] articles);
7.      List<Article> GetAllArticles();
8.      void DeleteAllArticles();
9.   }
10. }
```

- ligne 6 : pour ajouter un tableau d'articles dans la base

Ultérieurement, pour une autre application, apparaît le besoin de supprimer une liste d'articles enregistrée dans une liste, toujours dans une transaction. On voit que pour répondre à des besoins métier différents, la couche [dao] va être amenée à grossir. On peut prendre une autre voie :

- ne mettre dans la couche [dao] que les opérations basiques *InsertArticle*, *DeleteArticle*, *UpdateArticle*, *SelectArticle*, *SelectArticles*
- déporter dans la couche [métier] les opérations de mise à jour simultanée de plusieurs articles. Celles-ci utiliseraient les opérations élémentaires de la couche [dao].

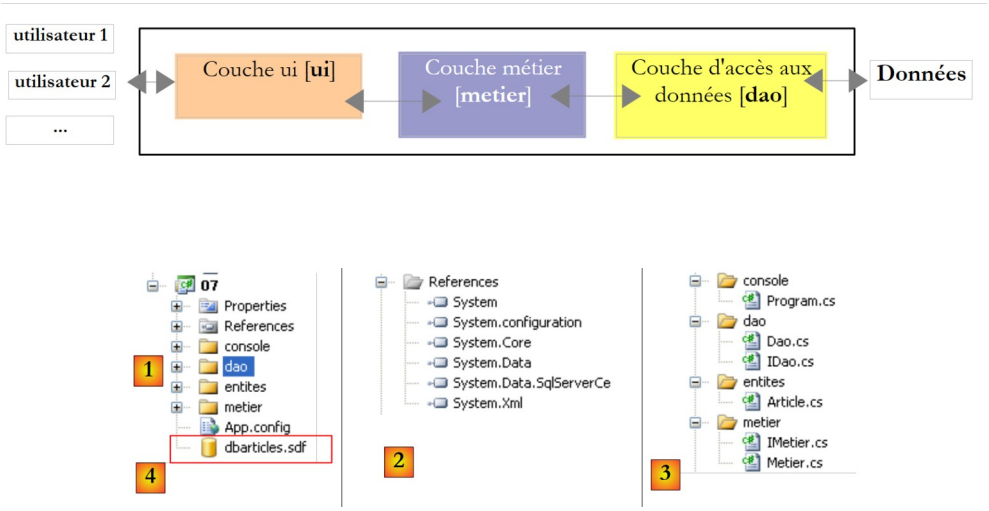
L'avantage de cette solution est que la même couche [dao] pourrait être utilisée sans changement avec différentes couches [metier]. Elle amène une difficulté dans la gestion de la transaction qui regroupe des mises à jour à faire de façon atomique sur la base :

- la transaction doit être initiée par la couche [metier] avant qu'elle n'appelle les méthodes de la couche [dao]
- les méthodes de la couche [dao] doivent connaître l'existence de la transaction afin d'y prendre part si elle existe
- la transaction doit êter terminée par la couche [métier].

Pour que les méthodes de la couche [dao] connaissent l'existence d'une éventuelle transaction en cours, on pourrait ajouter la transaction comme paramètre de chaque méthode de la couche [dao]. Ce paramètre va alors apparaître dans la signature des méthodes de l'interface, ce qui va lier celle-ci à une source de données particulière : la base de données. Les données locales du

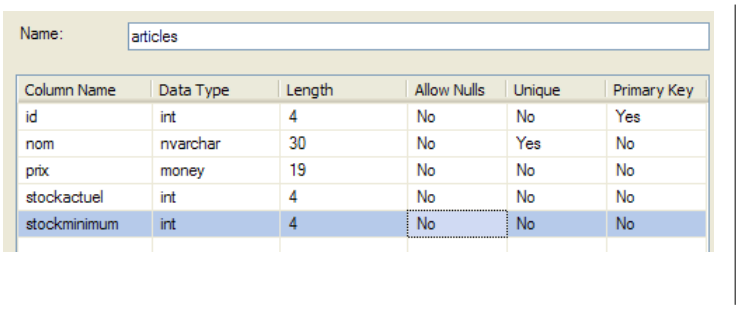
thread nous apportent une solution plus élégante : la couche [métier] mettra la transaction dans les données locales du thread et c'est là que la couche [dao] ira la chercher. La signature des méthodes de la couche [dao] n'a alors pas besoin d'être changée.

Nous mettons en oeuvre cette solution avec le projet Visual studio suivant :



- en [1] : la solution dans son ensemble
- en [2] : les références utilisées. La base [4] étant une base SQL Server Compact, il est nécessaire d'avoir la référence [System.Data.SqlServerCe].
- en [3] : les différentes couches de l'application.

La base [4] est la base SQL Server Compact déjà utilisée dans le chapitre précédent notamment au paragraphe 9.3.1, page 220.



La classe Article

Une ligne de la table [articles] précédente est encapsulée dans un objet de type *Article* :

```
1.  namespace Chap8 {
2.   public class Article {
3.      // propriétés
4.      public int Id { get; set; }
5.      public string Nom { get; set; }
6.      public decimal Prix { get; set; }
7.      public int StockActuel { get; set; }
8.      public int StockMinimum { get; set; }
9.
10.     // constructeurs
11.     public Article() {
12.     }
13.
14.     public Article(int id, string nom, decimal prix, int stockActuel, int stockMinimum) {
15.        Id = id;
16.        Nom = nom;
17.        Prix = prix;
18.        StockActuel = stockActuel;
19.        StockMinimum = stockMinimum;
```

```
20.     }
21.
22.     // identité
23.     public override string ToString() {
24.         return string.Format("[{0},{1},{2},{3},{4}]", Id, Nom, Prix, StockActuel, StockMinimum);
25.     }
26.  }
27. }
```

Interface de la couche [dao]

L'interface **IDao** de la couche [dao] sera la suivante :

```
1.  using System.Collections.Generic;
2.
3.  namespace Chap8 {
4.   public interface IDao {
5.      int InsertArticle(Article article);
6.      List<Article> GetAllArticles();
7.      void DeleteAllArticles();
8.   }
9.  }
```

- ligne 5 : pour insérer un article dans la table [articles]
- ligne 6 : pour mettre toutes les lignes de la table [articles] dans une liste d'objets *Article*
- ligne 7 : pour supprimer toutes les lignes de la table [articles]

Interface de la couche [metier]

L'interface **IMetier** de la couche [metier] sera la suivante :

```
1.  using System.Collections.Generic;
2.
3.  namespace Chap8 {
4.   interface IMetier {
5.      void InsertArticlesInTransaction(Article[] articles);
6.      void InsertArticlesOutOfTransaction(Article[] articles);
7.      List<Article> GetAllArticles();
8.      void DeleteAllArticles();
9.   }
10. }
```

- ligne 5 : pour insérer, à l'intérieur d'une transaction, un ensemble d'articles
- ligne 6 : idem mais sans transaction
- ligne 7 : pour obtenir la liste de tous les articles
- ligne 8 : pour supprimer tous les articles

Implémentation de la couche [metier]

L'implémentation **Metier** de l'interface **IMetier** sera la suivante :

```
1.  using System.Collections.Generic;
2.  using System.Data;
3.  using System.Data.SqlServerCe;
4.  using System.Threading;
5.
6.  namespace Chap8 {
7.   public class Metier : IMetier {
8.      // couche [dao]
9.      public IDao Dao { get; set; }
10.     // chaîne de connexion
11.     public string ConnectionString { get; set; }
12.
13.     // insertion d'un tableau d'articles à l'intérieur d'une transaction
14.     public void InsertArticlesInTransaction(Article[] articles) {
15.         // on crée la connexion à la base
16.         using (SqlCeConnection connexion = new SqlCeConnection(ConnectionString)) {
17.             // ouverture connexion
18.             connexion.Open();
19.             // transaction
20.             SqlCeTransaction transaction = null;
21.             try {
```

```
22.             // début transaction
23.             transaction = connexion.BeginTransaction(IsolationLevel.ReadCommitted);
24.             // on enregistre la transaction dans le thread
25.             Thread.SetData(Thread.GetNamedDataSlot("transaction"), transaction);
26.             // insertion des articles
27.             foreach (Article article in articles) {
28.                Dao.InsertArticle(article);
29.             }
30.             // on valide la transaction
31.             transaction.Commit();
32.          } catch {
33.             // on défait la transaction
34.             if (transaction != null)
35.                transaction.Rollback();
36.          }
37.       }
38.    }
39.
40.    // insertion d'un tableau d'articles sans transaction
41.    public void InsertArticlesOutOfTransaction(Article[] articles) {
42.       // insertion des articles
43.       foreach (Article article in articles) {
44.          Dao.InsertArticle(article);
45.       }
46.    }
47.
48.    // liste des articles
49.    public List<Article> GetAllArticles() {
50.       return Dao.GetAllArticles();
51.    }
52.    // supprimer tous les articles
53.    public void DeleteAllArticles() {
54.       Dao.DeleteAllArticles();
55.    }
56. }
57. }
```

La classe a les propriétés suivantes :

- ligne 9 : une référence sur la couche [dao]
- ligne 11 : la chaîne de connexion qui permet de se connecter à la base de données des articles

Nous ne commentons que la méthode *InsertArticlesInTransaction* qui seule présente des difficultés :

- ligne 16 : une connexion avec la base est créée
- ligne 18 : elle est ouverte
- ligne 23 : une transaction est créée
- ligne 25 : elle est enregistrée dans les données locales du thread, associée à la clé "transaction"
- lignes 27-29 : la méthode d'insertion unitaire de la couche [dao] est appelée pour chaque article à insérer
- lignes 21 et 32 : l'ensemble de l'insertion du tableau est contrôlée par un try / catch
- ligne 31 : si on arrive là, c'est qu'il n'y a pas eu d'exception. On valide alors la transaction.
- lignes 34-35 : il y a eu exception, on défait la transaction
- ligne 37 : on sort de la clause *using*. La connexion ouverte en ligne 18 est automatiquement fermée.

Implémentation de la couche [dao]

L'implémentation **Dao** de l'interface **IDao** sera la suivante :

```
1.  using System.Collections.Generic;
2.  using System.Data;
3.  using System.Data.SqlServerCe;
4.  using System.Threading;
5.
6.  namespace Chap8 {
7.   public class Dao : IDao {
8.      // chaîne de connexion
9.      public string ConnectionString { get; set; }
10.     // requêtes
11.     public string InsertText { get; set; }
12.     public string DeleteAllText { get; set; }
13.     public string GetAllText { get; set; }
14.
15.     // implémentation interface
```

```
16.
17.     // insertion article
18.     public int InsertArticle(Article article) {
19.         // y-a-t-il une transaction en cours ?
20.         SqlCeTransaction transaction = Thread.GetData(Thread.GetNamedDataSlot("transaction")) as
    SqlCeTransaction;
21.         // récupérer la connexion ou la créer
22.         SqlCeConnection connexion = null;
23.         if (transaction != null) {
24.             // récupérer la connexion
25.             connexion = transaction.Connection as SqlCeConnection;
26.         } else {
27.             // la créer
28.             connexion = new SqlCeConnection(ConnectionString);
29.             connexion.Open();
30.         }
31.         try {
32.             // préparation commande d'insertion
33.             SqlCeCommand sqlCommand = new SqlCeCommand();
34.             sqlCommand.Transaction = transaction;
35.             sqlCommand.Connection = connexion;
36.             sqlCommand.CommandText = InsertText;
37.             sqlCommand.Parameters.Add("@nom", SqlDbType.NVarChar, 30);
38.             sqlCommand.Parameters.Add("@prix", SqlDbType.Money);
39.             sqlCommand.Parameters.Add("@sa", SqlDbType.Int);
40.             sqlCommand.Parameters.Add("@sm", SqlDbType.Int);
41.             sqlCommand.Parameters["@nom"].Value = article.Nom;
42.             sqlCommand.Parameters["@prix"].Value = article.Prix;
43.             sqlCommand.Parameters["@sa"].Value = article.StockActuel;
44.             sqlCommand.Parameters["@sm"].Value = article.StockMinimum;
45.             // exécution
46.             return sqlCommand.ExecuteNonQuery();
47.         } finally {
48.             // si on n'était pas dans une transaction, on ferme la connexion
49.             if (transaction == null) {
50.                 connexion.Close();
51.             }
52.         }
53.     }
54.
55.     // liste des articles
56.     public List<Article> GetAllArticles() {
57. ...
58.     }
59.
60.     // suppression des articles
61.     public void DeleteAllArticles() {
62. ...
63.     }
64.  }
65. }
```

La classe a les propriétés suivantes :

- ligne 9 : la chaîne de connexion qui permet de se connecter à la base de données des articles
- ligne 11 : l'ordre SQL pour insérer un article
- ligne 12 : l'ordre SQL pour suprrimer tous les articles
- ligne 13 : l'ordre SQL pour obtenir tous les articles

Ces propriétés seront initialisées à partir du fichier de configuration [App.config] suivant :

```
(a) <?xml version="1.0" encoding="utf-8" ?>
(b) <configuration>
(c)  <connectionStrings>
(d)     <add name="dbArticlesSqlServerCe" connectionString="Data Source=|DataDirectory|
    \dbarticles.sdf;Password=dbarticles;" />
(e)  </connectionStrings>
(f)  <appSettings>
(g)     <add key="insertText" value="insert into articles(nom,prix,stockactuel,stockminimum)
    values(@nom,@prix,@sa,@sm)"/>
(h)     <add key="getAllText" value="select id,nom,prix,stockactuel,stockminimum from articles"/>
(i)     <add key="deleteAllText" value="delete from articles"/>
(j)  </appSettings>
(k) </configuration>
```

Nous commentons la méthode *InsertArticle* :

- ligne 20 : on récupère l'éventuelle transaction qu'a pu placer la couche [metier] dans le thread
- lignes 23-25 : si la transaction est présente, on récupère la connexion à laquelle elle a été liée.
- lignes 26-30 : sinon, une connexion nouvelle est créée et ouverte.
- lignes 33-44 : on prépare la commande d'insertion. Celle-ci est paramétrée (cf ligne g de *App.config*).
- ligne 33 : l'objet *Command* est créé.
- ligne 34 : il est associé à la transaction courante. Si celle-ci n'existe pas (transaction=null), cela revient à exécuter l'ordre SQL sans transaction explicite. On rappelle qu'alors il y a quand même une transaction implicite. Avec SQL Server CE, cette transaction implicite est par défaut en mode *autocommit* : l'ordre SQL est *committé* après son exécution.
- ligne 35 : l'objet *Command* est associé à la connexion courante
- ligne 36 : le texte SQl à exécuter est fixé. C'est la requête paramétrée de la ligne g de *App.config*.
- lignes 37-44 : les 4 paramètres de la requête sont initialisés
- ligne 46 : la requête est exécutée.
- lignes 49-51 : il faut se souvenir que s'il n'y avait pas de transaction, une nouvelle connexion a été ouverte avec la base, lignes 26-30. Dans ce cas, elle doit être fermée. S'il y avait une transaction, la connexion ne doit pas être fermée car c'est la couche [metier] qui la gère.

Les deux autres méthodes reprennent ce qui a été vu dans le chapitre "Bases de données" :

```
1.        // liste des articles
2.        public List<Article> GetAllArticles() {
3.           // liste des articles - vide au départ
4.           List<Article> articles = new List<Article>();
5.           // exploitation connexion
6.           using (SqlCeConnection connexion = new SqlCeConnection(ConnectionString)) {
7.              // ouverture connexion
8.              connexion.Open();
9.              // exécute sqlCommand avec requête select
10.             SqlCeCommand sqlCommand = new SqlCeCommand(GetAllText, connexion);
11.             using (SqlCeDataReader reader = sqlCommand.ExecuteReader()) {
12.                // exploitation résultat
13.                while (reader.Read()) {
14.                   // exploitation ligne courante
15.                   articles.Add(new Article(reader.GetInt32(0), reader.GetString(1),
    reader.GetDecimal(2), reader.GetInt32(3), reader.GetInt32(4)));
16.                }
17.             }
18.          }
19.          // on rend le résultat
20.          return articles;
21.       }
22.
23.       // suppression des articles
24.       public void DeleteAllArticles() {
25.          using (SqlCeConnection connexion = new SqlCeConnection(ConnectionString)) {
26.             // ouverture connexion
27.             connexion.Open();
28.             // exécute sqlCommand avec requête de mise à jour
29.             new SqlCeCommand(DeleteAllText, connexion).ExecuteNonQuery();
30.          }
31. }
```

L'application [console] de test

L'application [console] de test est la suivante :

```
1.  using System;
2.  using System.Configuration;
3.
4.  namespace Chap8 {
5.   class Program {
6.      static void Main(string[] args) {
7.         // exploitation du fichier de configuration
8.         string connectionString = null;
9.         string insertText;
10.        string getAllText;
11.        string deleteAllText;
12.        try {
13.           // chaîne de connexion
14.           connectionString =
    ConfigurationManager.ConnectionStrings["dbArticlesSqlServerCe"].ConnectionString;
```

```
15.           // autres paramètres
16.           insertText = ConfigurationManager.AppSettings["insertText"];
17.           getAllText = ConfigurationManager.AppSettings["getAllText"];
18.           deleteAllText = ConfigurationManager.AppSettings["deleteAllText"];
19.        } catch (Exception e) {
20.           Console.WriteLine("Erreur de configuration : {0}", e.Message);
21.           return;
22.        }
23.        // création couche [dao]
24.        Dao dao = new Dao();
25.        dao.ConnectionString = connectionString;
26.        dao.DeleteAllText = deleteAllText;
27.        dao.GetAllText = getAllText;
28.        dao.InsertText = insertText;
29.        // création couche [métier]
30.        Metier metier = new Metier();
31.        metier.Dao = dao;
32.        metier.ConnectionString = connectionString;
33.        // on crée un tableau d'articles
34.        Article[] articles = new Article[2];
35.        for (int i = 0; i < articles.Length; i++) {
36.           articles[i] = new Article(0, "article", 100, 10, 1);
37.        }
38.        // on supprime tous les articles
39.        Console.WriteLine("Suppression de tous les articles...");
40.        metier.DeleteAllArticles();
41.        // on insère le tableau hors transaction
42.        Console.WriteLine("Insertion des articles hors transaction...");
43.        try {
44.           metier.InsertArticlesOutOfTransaction(articles);
45.        } catch (Exception e){
46.           Console.WriteLine("Exception : {0}", e.Message);
47.        }
48.        // on affiche les articles
49.        Console.WriteLine("Liste des articles");
50.        AfficheArticles(metier);
51.        // on supprime tous les articles
52.        Console.WriteLine("Suppression de tous les articles...");
53.        metier.DeleteAllArticles();
54.        // on insère le tableau dans une transaction
55.        Console.WriteLine("Insertion des articles dans une transaction...");
56.        metier.InsertArticlesInTransaction(articles);
57.        // on affiche les articles
58.        Console.WriteLine("Liste des articles");
59.        AfficheArticles(metier);
60.     }
61.
62.     private static void AfficheArticles(IMetier metier) {
63.        // on affiche les articles
64.        foreach(Article article in metier.GetAllArticles()){
65.           Console.WriteLine(article);
66.        }
67.     }
68.
69.  }
70. }
```

- lignes 12-22 : le fichier [App.config] est exploité.
- lignes 24-28 : la couche [dao] est instanciée et initialisée
- lignes 30-32 : il est fait de même pour la couche [metier]
- lignes 34-37 : on crée un tableau de 2 articles avec le même nom. La table [articles] de la base SQL server Ce [dbarticles.sdf] a une contrainte d'unicité sur le nom. L'insertion du 2ième article sera donc refusée. Si l'insertion du tableau se fait hors transaction, le 1er article sera d'abord inséré puis le restera. Si l'insertion du tableau se fait dans une transaction, le 1er article sera d'abord inséré puis sera retiré, lors du *Rollback* de la transaction.
- lignes 39-50 : insertion hors transaction du tableau de 2 articles et vérification.
- lignes 52-59 : idem mais dans une transaction

Les résultats à l'exécution sont les suivants :

```
1.  Suppression de tous les articles...
2.  Insertion des articles hors transaction...
3.  Exception : A duplicate value cannot be inserted into a unique index. [ Table na
4.  me = ARTICLES,Constraint name = UQ__ARTICLES__0000000000000010 ]
5.  Liste des articles
6.  [7,article,100,10,1]
```

```
7.  Suppression de tous les articles...
8.  Insertion des articles dans une transaction...
9.  Liste des articles
```

- lignes 5-6 : l'insertion hors transaction a laissé le 1er article dans la base
- ligne 9 : l'insertion faite dans une transaction n'a laissé aucun article dans la base

10.9.3 Conclusion

L'exemple précédent a montré l'intérêt des données locales à un thread pour la gestion des transactions. Il n'est pas à reproduire tel quel. Des frameworks tels que Spring, Nhibernate, ... utilisent cette technique mais la rendent encore plus transparente : il est possible pour la couche [metier] d'utiliser des transactions sans que la couche [dao] n'ait besoin de le savoir. Il n'y a alors aucun objet *Transaction* dans le code de la couche [dao]. Cela est obtenu au moyen d'une technique de proxy appelée AOP (Aspects Oriented Programming). De nouveau on ne peut qu'inciter le lecteur à utiliser ces frameworks.

10.10 Pour approfondir...

Pour approfondir le domaine difficile de la synchronisation de threads, on pourra lire le chapitre *Threading* du livre *C# 3.0* référencé dans l'introduction de ce document. On y présente de nombreuses techniques de synchronisation pour différents types de situation.

11 Programmation Internet

11.1 Généralités

11.1.1 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 peut être utile 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

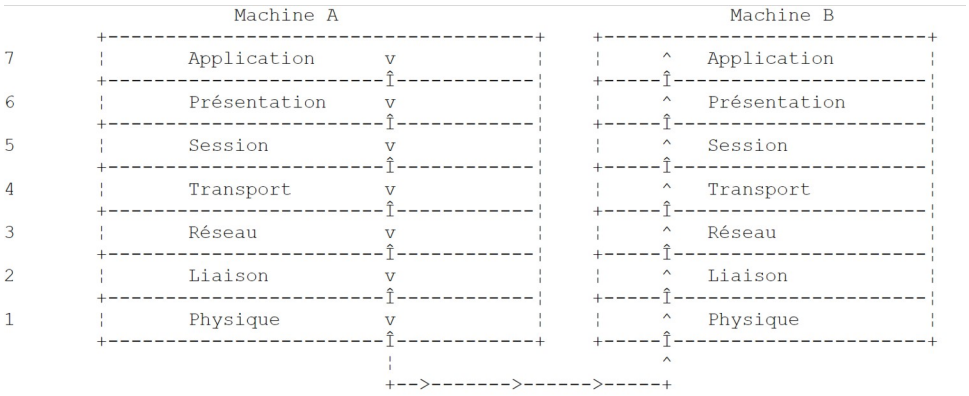
11.1.2 Le modèle OSI

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 :



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.

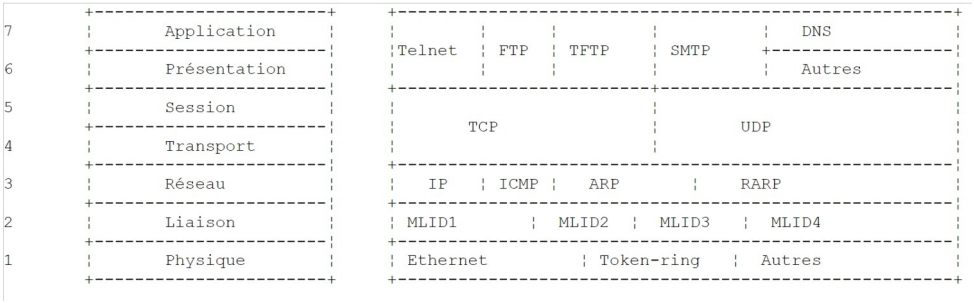
`Session` On va trouver dans cette couche des services permettant à une application d'ouvrir et de maintenir une session de travail sur une machine distante.

`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.

11.1.3 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 :



Couche Physique

En réseau local, on trouve généralement une technologie **Ethernet** ou **Token-Ring**. Nous ne présentons ici que la technologie Ethernet.

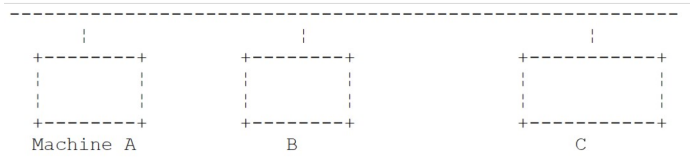
Ethernet

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



- 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

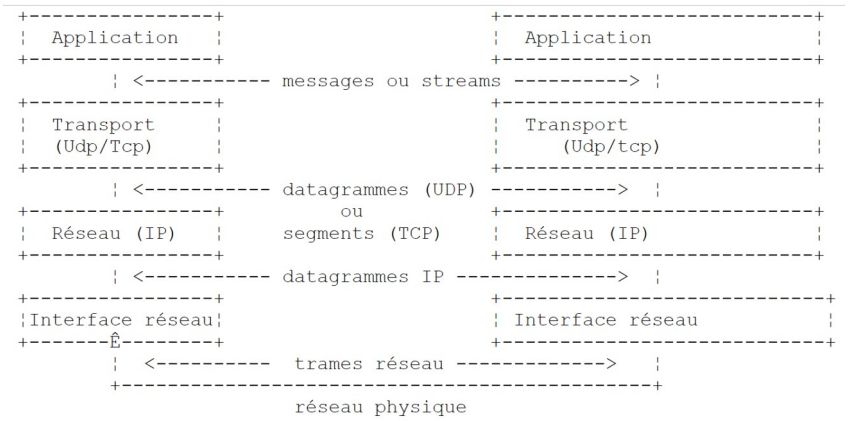
On trouve ici divers protocoles :

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

11.1.4 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 :



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.

11.1.5 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.

11.1.5.1 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

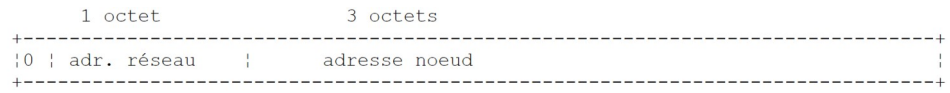
Selon la taille de ces deux champs, les adresses IP sont divisées en 3 classes : classes A, B et C.

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 :



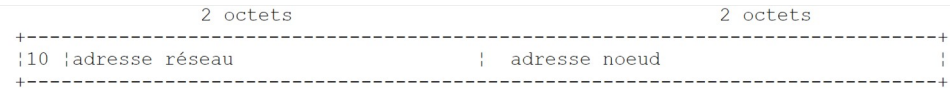
L'adresse réseau est sur 7 bits et l'adresse du noeud sur 24 bits. On peut donc avoir 127 réseaux de classe A, chacun comportant jusqu'à 2^{24} noeuds.

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 :



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 :

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.

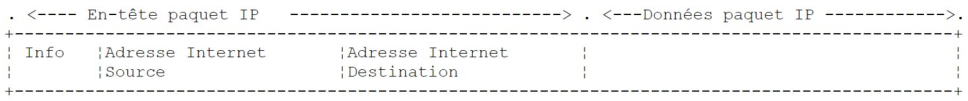
L'adresse de la machine *Lagaffe* de la faculté des sciences d'Angers étant 193.49.144.1, on voit que l'octet de poids fort vaut 193, c'est à dire en binaire 11000001. On en déduit que le réseau est de classe C.

Adresses réservées

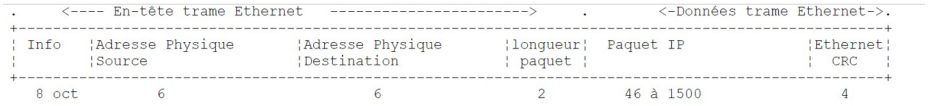
- Certaines adresses IP sont des adresses de réseaux plutôt que des adresses de noeuds dans le réseau. Ce sont celles, où l'adresse du noeud est mise à 0. Ainsi, l'adresse 193.49.144.0 est l'adresse IP du réseau de la Faculté des Sciences d'Angers. En conséquence, aucun noeud d'un réseau ne peut avoir 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.

11.1.5.2 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 :



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 :



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*).

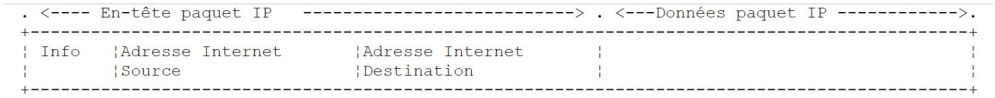
- Elle envoie un paquet d'un type spécial appelé paquet ARP contenant l'adresse IP de la machine dont on cherche l'adresse physique. Elle a pris soin également d'y placer sa propre adresse IP ainsi que son adresse physique.
- Ce paquet est envoyé à tous les noeuds du réseau.
- Ceux-ci reconnaissent la 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.

11.1.6 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 :



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 :

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.

11.1.6.1 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.

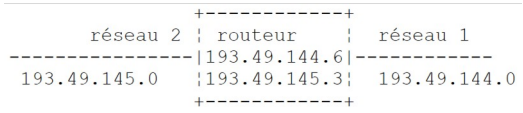
Routage direct

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.



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.

11.1.6.2 Messages d'erreur et de contrôle

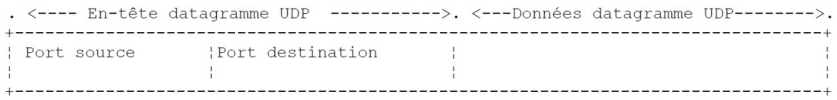
Toujours dans la couche réseau, au même niveau donc que le protocole IP, existe le protocole **ICMP** (*Internet Control Message Protocol*). Il sert à envoyer des messages sur le fonctionnement interne du réseau : noeuds en panne, embouteillage à un routeur, etc ... Les messages ICMP sont encapsulés 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.

11.1.7 La couche transport : les protocoles UDP et TCP

11.1.7.1 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 :



Ces datagrammes seront encapsulés dans des paquets IP, puis dans des trames physiques.

11.1.7.2 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.

11.1.8 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.

11.1.9 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

11.2 Les classes .NET de la gestion des adresses IP

Une machine sur le réseau Internet est définie de façon unique par une adresse IP (Internet Protocol) qui peut prendre deux formes :

- IPv4 : codée sur 32 bits et représentée par une chaîne de la forme "I1.I2.I3.I4" où I_n est un nombre entre 1 et 254. Ce sont les adresses IP les plus courantes actuellement.
- IPv6 : codée sur 128 bits et représentée par une chaîne de la forme "[I1.I2.I3.I4.I5.I6.I7.I8]" où I_n est une chaîne de 4 chiffres hexadécimaux. Dans ce document, nous n'utiliserons pas les adresses IPv6.

Une machine peut être aussi 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.

Une machine peut avoir plusieurs adresses IP si elle est physiquement connectée à plusieurs réseaux en même temps. Elle a alors une adresse IP sur chaque réseau.

Une adresse IP peut être représentée de deux façons dans .NET :
- sous la forme d'une chaîne de caractères "I1.I2.I3.I4" ou "[I1.I2.I3.I4.I5.I6.I7.I8]"
- sous la forme d'un objet de type **IPAddress**

La classe IPAddress

Parmi les méthodes M, propriétés P et constantes C de la classe **IPAddress**, on trouve les suivantes :

`AddressFamily AddressFamily`	P	famille de l'adresse IP. Le type *AddressFamily* est une énumération. Les deux valeurs courantes sont : *AddressFamily.InterNetwork* : pour une adresse IPv4 *AddressFamily.InterNetworkV6* : pour une adresse IPv6
`IPAddress Any`	C	l'adresse IP "0.0.0.0". Lorsqu'un service est associé à cette adresse, cela signifie qu'il accepte des clients sur toutes les adresses IP de la machine sur laquelle il opère.
`IPAddress LoopBack`	C	l'adresse IP "127.0.0.1". Appelée "adresse de boucle". Lorsqu'un service est associé à cette adresse, cela signifie qu'il n'accepte que les clients qui sont sur la même machine que lui.
`IPAdress None`	C	l'adresse IP "255.255.255.255". Lorsqu'un service est associé à cette adresse, cela signifie qu'il n'accepte aucun client.
`bool TryParse(string ipString, out IPAddress address)`	M	essaie de passer l'adresse IP *ipString* de forme "I1.I2.I3.I4" sous la forme d'un objet *IPAddress address*. Rend *true* si l'opération a réussi.

`bool IsLoopBack`	M	rend true si l'adresse IP est "127.0.0.1"
`string ToString()`	M	rend l'adrresse IP sous la forme "I1.I2.I3.I4" ou "[I1.I2.I3.I4.I5.I6.I7.I8]"

L'association *adresse IP* <--> *nomMachine* est assurée par un service distribué de l'internet appelé **DNS** (Domain Name System). Les méthodes statiques de la classe **Dns** permettent de faire l'association *adresse IP* <--> *nomMachine* :

`GetHostEntry (string hostNameOrdAddress)`	rend une adresse *IPHostEntry* à partir d'une adresse IP sous la forme d'une chaîne ou à partir d'un nom de machine. Lance une exception si la machine ne peut être trouvée.
`GetHostEntry (IPAddress ip)`	rend une adresse *IPHostEntry* à partir d'une adresse IP de type **IPAddress**. Lance une exception si la machine ne peut être trouvée.
`string GetHostName()`	rend le nom de la machine sur laquelle s'exécute le programme qui joue cette instruction
`IPAddress[] GetHostAddresses(string hostNameOrdAddress)`	rend les adresses IP de la machine identifiée par son nom ou l'une de ses adresses IP.

Une instance *IPHostEntry* encapsule les adresses IP, les alias et le nom d'une machine. Le type *IPHostEntry* est le suivant :

`IPAddress[] AddressList`	P	tableau des adresses IP de la machine
`String[] Aliases`	P	les alias DNS de la machine. Ceux-ci sont les noms correspondant aux différentes adresses IP de la machine.
`string HostName`	P	le nom d'hôte principal de la machine

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* :

```
1.  using System;
2.  using System.Net;
3.
4.  namespace Chap9 {
5.   class Program {
6.      static void Main(string[] args) {
7.         // affiche le nom de la machine locale
8.         // puis donne interactivement des infos sur les machines réseau
9.         // identifiées par un nom ou une adresse IP
10.
11.        // machine locale
12.        Console.WriteLine("Machine Locale= {0}" ,Dns.GetHostName());
13.
14.        // question-réponses interactives
15.        string machine;
16.        IPHostEntry ipHostEntry;
17.        while (true) {
18.           // saisie du nom ou de l'adresse IP de la machine recherchée
19.           Console.Write("Machine recherchée (rien pour arrêter) : ");
20.           machine = Console.ReadLine().Trim().ToLower();
21.           // fini ?
22.           if (machine == "") return;
23.           // gestion exception
24.           try {
25.              // recherche machine
26.              ipHostEntry = Dns.GetHostEntry(machine);
27.              // le nom de la machine
28.              Console.WriteLine("Machine : " + ipHostEntry.HostName);
29.              // les adresses IP de la machine
30.              Console.Write("Adresses IP : {0}" , ipHostEntry.AddressList[0]);
31.              for (int i = 1; i < ipHostEntry.AddressList.Length; i++) {
32.                 Console.Write(", {0}" , ipHostEntry.AddressList[i]);
33.              }
34.              Console.WriteLine();
35.              // les alias de la machine
36.              if (ipHostEntry.Aliases.Length != 0) {
37.                 Console.Write("Alias : {0}" , ipHostEntry.Aliases[0]);
38.                 for (int i = 1; i < ipHostEntry.Aliases.Length; i++) {
39.                    Console.Write(", {0}" , ipHostEntry.Aliases[i]);
40.                 }
41.                 Console.WriteLine();
42.              }
43.           } catch {
44.              // la machine n'existe pas
45.              Console.WriteLine("Impossible de trouver la machine [{0}]",machine);
46.           }
47.        }
```

```
48.     }
49.  }
50. }
```

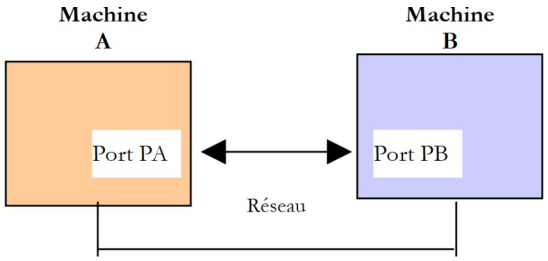
L'exécution donne les résultats suivants :

```
1.  Machine Locale= LISA-AUTO2005A
2.  Machine recherchée (rien pour arrêter) : localhost
3.  Machine : LISA-AUTO2005A
4.  Adresses IP : 127.0.0.1
5.  Machine recherchée (rien pour arrêter) : 127.0.0.1
6.  Machine : LISA-AUTO2005A
7.  Adresses IP : 127.0.0.1
8.  Machine recherchée (rien pour arrêter) : istia.univ-angers.fr
9.  Machine : istia.univ-angers.fr
10. Adresses IP : 193.49.146.171
11. Machine recherchée (rien pour arrêter) : 193.49.146.171
12. Machine : istia.istia.univ-angers.fr
13. Adresses IP : 193.49.146.171
14. Machine recherchée (rien pour arrêter) : xx
15. Impossible de trouver la machine [xx]
```

11.3 Les bases de la programmation internet

11.3.1 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.

11.3.2 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.

11.3.3 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.

11.3.4 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
fermer la connexion
```

11.3.5 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
```

11.4 Découvrir les protocoles de communication de l'internet

11.4.1 Introduction

Lorsqu'un client s'est connecté à un serveur, s'établit ensuite un dialogue entre-eux. La nature de celui-ci forme ce qu'on appelle le protocole de communication du serveur. Parmi les protocoles les plus courants de l'internet on trouve les suivants :

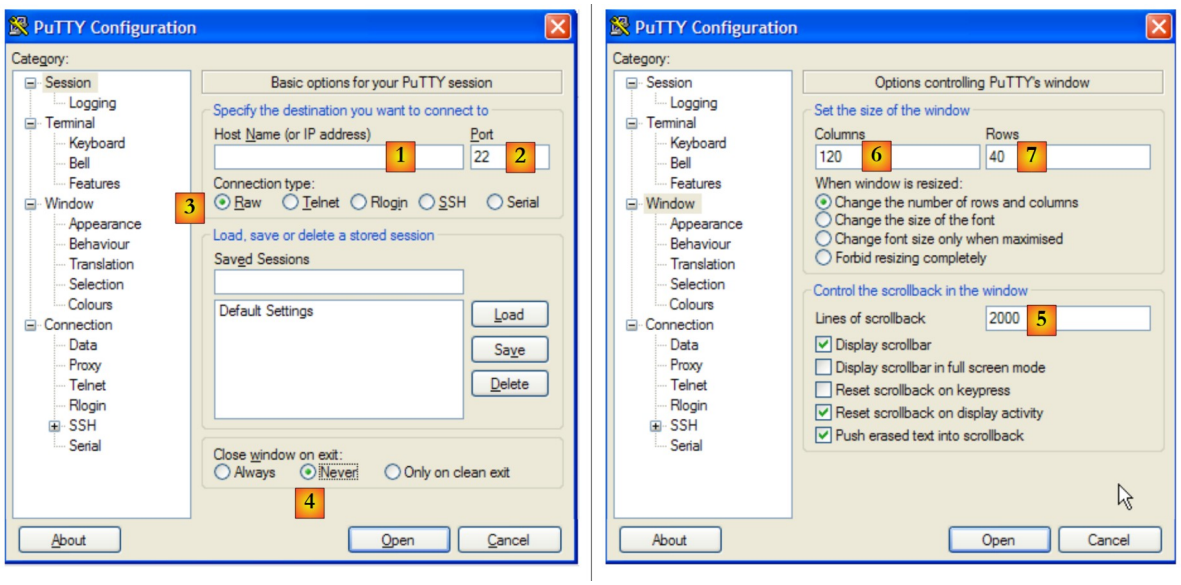
- HTTP : HyperText Transfer Protocol - le protocole de dialogue avec un serveur web (serveur HTTP)
- SMTP : Simple Mail Transfer Protocol - le protocole de dialogue avec un serveur d'envoi de courriers électroniques (serveur SMTP)
- POP : Post Office Protocol - le protocole de dialogue avec un serveur de stockage du courrier électronique (serveur POP). Il s'agit là de récupérer les courriers électroniques reçus et non d'en envoyer.
- FTP : File Transfer Protocol - le protocole de dialogue avec un serveur de stockage de fichiers (serveur FTP).

Tous ces protocoles ont la particularité d'être des protocoles à lignes de texte : le client et le serveur s'échangent des lignes de texte. Si on a un client capable de :

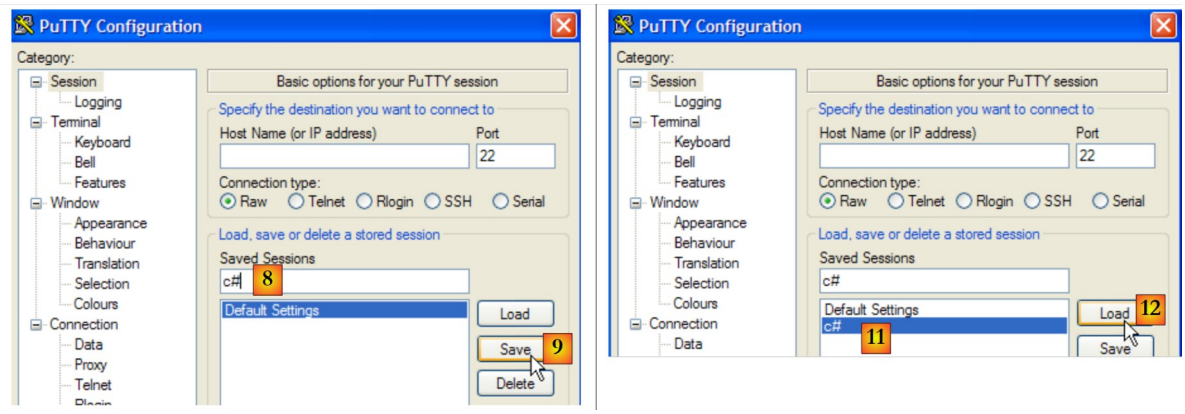
- créer une connexion avec un serveur Tcp
- afficher à la console les lignes de texte que le serveur lui envoie
- envoyer au serveur les lignes de texte qu'un utilisateur saisirait

alors on est capable de dialoguer avec un serveur Tcp ayant un protocole à lignes de texte pour peu qu'on connaisse les règles de ce protocole.

Le programme **telnet** qu'on trouve sur les machines Unix ou Windows est un tel client. Sur les machines Windows, on trouve également un outil appelé **putty** et c'est lui que nous allons utiliser ici. **putty** est téléchargeable à l'adresse [http://www.putty.org/]. C'est un exécutable (.exe) directement utilisable. Nous le configurerons de la façon suivante :



- [1] : l'adresse IP du serveur Tcp auquel on veut se connceter ou son nom
- [2] : le port d'écoute du serveur Tcp
- [3] : prendre le mode *Raw* qui désigne une connexion Tcp brute.
- [4] : prendre le mode *Never* pour empêcher la fenêtre du client *putty* de se fermer si le serveur ferme la connexion.
- [6,7] : nombre de colonnes / lignes de la console
- [5] : le nombre maximal de lignes conservées en mémoire. Un serveur HTTP peut envoyer beaucoup de lignes. Il faut pouvoir "scroller" dessus.

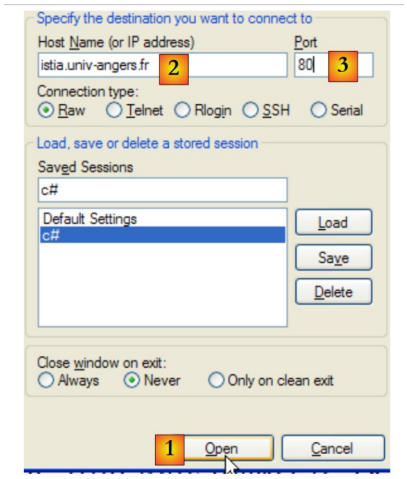


- [8,9] : pour conserver les paramètres précédents, donner un nom à la configuration [8] et la sauvegarder [9].
- [11,12] : pour récupérer une configuration sauvegardée, la sélectionner [11] et la charger [12].

Avec cet outil ainsi configuré, découvrons quelques protocoles TCP.

11.4.2 Le protocole HTTP (HyperText Transfer Protocol)

Connectons [1] notre client TCP sur le serveur web de la machine *istia.univ-angers.fr* [2], port 80 [3] :



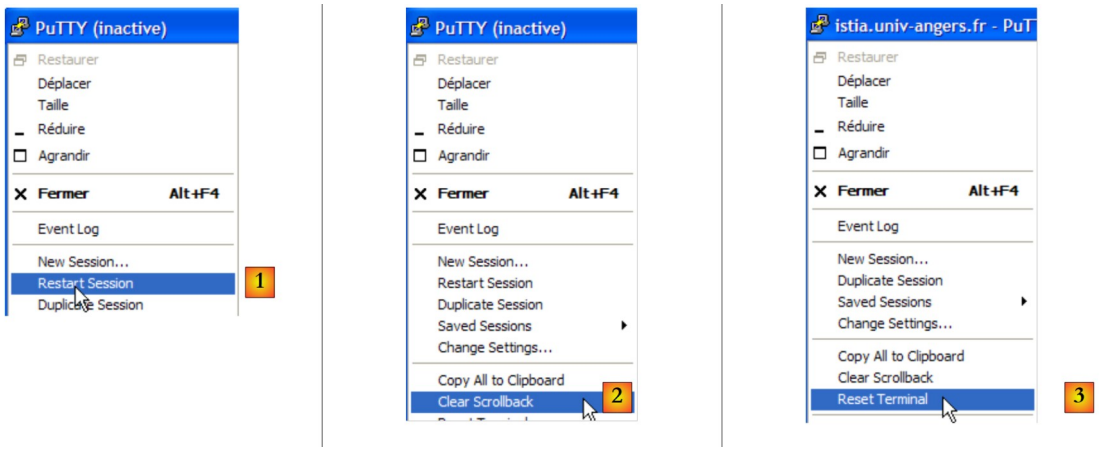
Dans la console de *putty*, nous construisons le dialogue HTTP suivant :

```
1.  GET / HTTP/1.1
2.  Host: istia.univ-angers.fr:80
3.  Connection: close
4.
5.  HTTP/1.1 200 OK
6.  Date: Sat, 03 May 2008 07:53:47 GMT
7.  Server: Apache/1.3.34 (Debian) PHP/4.4.4-8+etch4 mod_jk/1.2.18 mod_perl/1.29
8.  X-Powered-By: PHP/4.4.4-8+etch4
9.  Set-Cookie: fe_typo_user=0d2e64b317; path=/
10. Connection: close
11. Transfer-Encoding: chunked
12. Content-Type: text/html;charset=iso-8859-1
13.
14. 693f
15. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
16.         <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr_FR" lang="fr_FR">
17. ....
18.         </html>
19. 0
```

- les lignes 1-4 sont la demande du client, tapée au clavier

- les lignes 5-19 sont la réponse du serveur
- ligne 1 : syntaxe *GET UrlDocument HTTP/1.1* - nous demandons l'Url /, c.a.d. la racine du site web [istia.univ-angers.fr].
- ligne 2 : syntaxe *Host: machine:port*
- ligne 3 : syntaxe *Connection: [mode de la connexion]*. Le mode [close] indique au serveur de fermer la connexion une fois qu'il aura envoyée sa réponse. Le mode [Keep-Alive] demande de la laisser ouverte.
- ligne 4 : ligne vide. Les lignes 1-3 sont appelées **entêtes HTTP**. Il peut y en avoir d'autres que ceux présentés ici. La fin des entêtes HTTP est signalée avec une ligne vide.
- lignes 5-13 : les entêtes HTTP de la réponse du serveur - se terminent là également par une ligne vide.
- lignes 14-19 : le document envoyé par le serveur, ici un document HTML
- ligne 5 : syntaxe *HTTP/1.1 code msg* - le code 200 indique que le document demandé a été trouvé.
- ligne 6 : les date et heure du serveur
- ligne 7 : identification logiciel assurant le service web - ici un serveur Apache sur un Linux / Debian
- ligne 8 : le document a été généré dynamiquement par PHP
- ligne 9 : cookie d'identification du client - si celui-ci veut se faire reconnaître à sa prochaine connexion, il devra renvoyer ce cookie dans ses entêtes HTTP.
- ligne 10 : indique qu'après avoir servi le document demandé, le serveur fermera la connexion
- ligne 11 : le document va être transmis par morceaux (chunked) et non d'un seul bloc.
- ligne 12 : nature du document : ici un document HTML
- ligne 13 : la ligne vide qui signale la fin des entêtes HTTP du serveur
- ligne 14 : nombre hexadécimal indiquant le nombre de caractères du 1er bloc du document. Lorsque ce nombre vaudra 0 (ligne 19), le client saura qu'il a reçu tout le document.
- lignes 15-18 : partie du document reçu.

La connexion a été fermée et le client *putty* est inactif. Reconnectons-nous [1] et nettoyons l'écran des affichages précédents [2,3] :



Le dialogue cette fois-ci est le suivant :

```
1.  GET /inconnu HTTP/1.1
2.  Host: istia.univ-angers.fr:80
3.  Connection: Close
4.
5.  HTTP/1.1 404 Not Found
6.  Date: Sat, 03 May 2008 08:16:02 GMT
7.  Server: Apache/1.3.34 (Debian) PHP/4.4.4-8+etch4 mod_jk/1.2.18 mod_perl/1.29
8.  Connection: close
9.  Transfer-Encoding: chunked
10. Content-Type: text/html; charset=iso-8859-1
11.
12. 11a
13. <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
14.                                                 <HTML><HEAD>
15.                                                             <TITLE>404 Not Found</TITLE>
16.
    </HEAD><BODY>
17.
    <H1>Not Found</H1>
18.  The requested URL /inconnu was not found on this server.<P>
19.                                                                 <HR>
20.                                                                     <ADDRESS>Apache/1.3.34 Server at
    www.istia.univ-angers.fr Port 80</ADDRESS>
21.                     </BODY></HTML>
22.
```

```
23. 0
```

- ligne 1 : on a demandé un document inexistant
- ligne 5 : le serveur HTTP a répondu avec le code 404 signifiant que le document demandé n'a pas été trouvé.

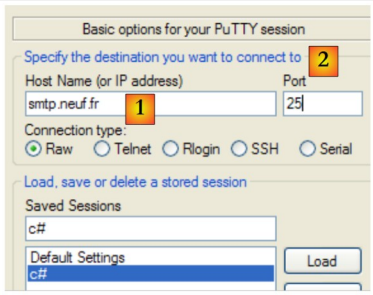
Si on demande ce document avec un navigateur Firefox :

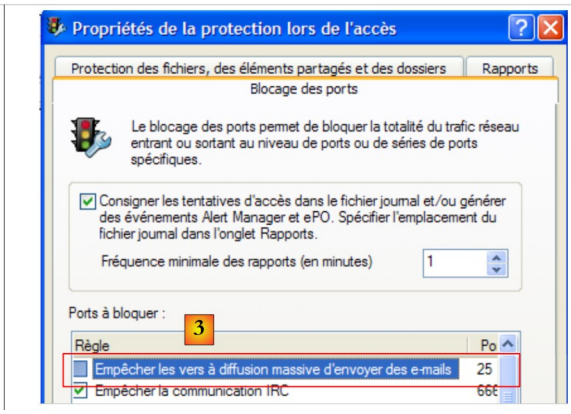
Si nous demandons à voir le code source [Affichage/Code source] :

```
1.  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
2.  <HTML><HEAD>
3.  <TITLE>404 Not Found</TITLE>
4.  </HEAD><BODY>
5.  <H1>Not Found</H1>
6.  The requested URL /inconnu was not found on this server.<P>
7.  <HR>
8.  <ADDRESS>Apache/1.3.34 Server at www.istia.univ-angers.fr Port 80</ADDRESS>
9.  </BODY></HTML>
```

Nous obtenons les lignes 13-22 reçues par notre client *putty*. L'intérêt de celui-ci est de nous montrer en plus, les entêtes HTTP de la réponse. Il est également possible d'avoir ceux-ci avec Firefox.

11.4.3 Le protocole SMTP (Simple Mail Transfer Protocol)





Les serveurs SMTP opèrent en général sur le port 25 [2]. On se connecte sur le serveur [1]. Ici, il faut en général prendre un serveur appartenant au même domaine IP que la machine car le plus souvent les serveurs SMTP sont configurés pour n'accepter que les demandes des machines appartenant au même domaine qu'eux. Par ailleurs, assez souvent également, les pare-feu ou antivirus des machines personnelles sont configurés pour ne pas accepter de connexion vers le port 25 d'une machine extérieure. Il peut être alors nécessaire de reconfigurer [3] ce pare-feu ou antivirus.

Le dialogue SMTP dan la fenêtre du client *putty* est le suivant :

```
1.  220 neuf-infra-smtp-out-sp604001av.neufgp.fr neuf telecom Service relais mail ready
2.  HELO istia.univ-angers.fr
3.  250 neuf-infra-smtp-out-sp604002av.neufgp.fr hello [84.100.189.193], Banniere OK , pret pour
    envoyer un mail
4.  mail from: @expéditeur
```

```
5.  250 2.1.0 <@expéditeur> sender ok
6.  rcpt to: @destinataire
7.  250 2.1.5 <@destinataire> destinataire ok
8.  data
9.  354 enter mail, end with "." on a line by itself
10. ligne1
11. ligne2
12. .
13. 250 2.0.0 LwiU1Z00V4AoCxw0200000 message ok
14. quit
15. 221 2.0.0 neuf-infra-smtp-out-sp604002av.neufgp.fr neuf telecom closing connection
```

Ci-dessous (D) est une demande du client, (R) une réponse du serveur.

- ligne 1 : (R) message d'accueil du serveur SMTP
- ligne 2 : (D) commande HELO pour dire bonjour
- ligne 3 : (R) réponse du serveur
- ligne 4 : (D) adresse expéditeur, par exemple *mail from: someone@gmail.com*
- ligne 5 : (R) réponse du serveur
- ligne 6 : (D) adresse destinataire, par exemple *rcpt to: someoneelse@gmail.com*
- ligne 7 : (R) réponse du serveur
- ligne 8 : (D) signale le début du message
- ligne 9 : (R) réponse du serveur
- lignes 10-12 : (D) le message à envoyer terminé par une ligne contenant uniquement un point.
- ligne 13 : (R) réponse du serveur
- ligne 14 : (D) le client signale qu'il a terminé
- ligne 15 : (R) réponse du serveur qui ensuite ferme la connexion

11.4.4 Le protocole POP (Post Office Protocol)

Les serveurs POP opèrent en général sur le port 110 [2]. On se connecte sur le serveur [1]. Le dialogue POP dans la fenêtre du client *putty* est le suivant :

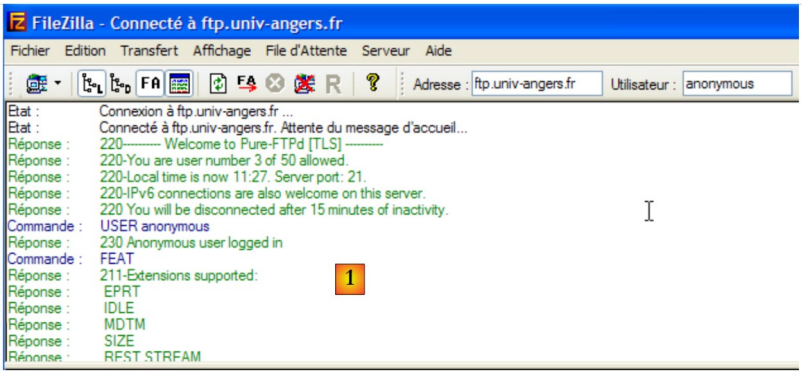
```
1.  +OK Hello there.
2.  user xx
3.  +OK Password required.
4.  pass yy
5.  +OK logged in.
6.  list
7.  +OK POP3 clients that break here, they violate STD53.
8.  1 10105
9.  2 55875
10. ...
11. 64 1717
12. .
13. retr 64
14. +OK 1717 octets follow.
15. Return-Path: <xx@neuf.fr>
16. X-Original-To: xx@univ-angers.fr
17. Delivered-To: xx@univ-angers.fr
18. ....
19. Date: Sat,  3 May 2008 10:59:25 +0200 (CEST)
20. From: xx@neuf.fr
21. To: undisclosed-recipients:;
22.
23. ligne1
24. ligne2
25. .
26. quit
27. +OK Bye-bye.
```

- ligne 1 : (R) message de bienvenue du serveur POP
- ligne 2 : (D) le client donne son identifiant POP, c.a.d. le login avec lequel il lit son courrier

- ligne 3 : (R) la réponse du serveur
- ligne 4 : (D) le mot de passe du client
- ligne 5 : (R) la réponse du serveur
- ligne 6 : (D) le client demande la liste de ses courriers
- lignes 7-12 : (R) la liste des messages dans la boîte à lettre du client, sous la forme [N° du message taille en octets du message]
- ligne 13 : (D) on demande le message n° 64
- lignes 14-25 : (R) le message n° 64 avec lignes 15-22, les entêtes du message, et lignes 23-24 le corps du message.
- ligne 26 : (D) le client indique qu'il a fini
- ligne 27 : (R) réponse du serveur qui va ensuite fermer la connexion.

11.4.5 Le protocole FTP (File Transfer Protocol)

Le protocole FTP est plus complexe que ceux présentés précédemment. Pour découvrir les lignes de texte échangées entre le client et le serveur, on pourra utiliser un outil tel que FileZilla [http://www.filezilla.fr/].

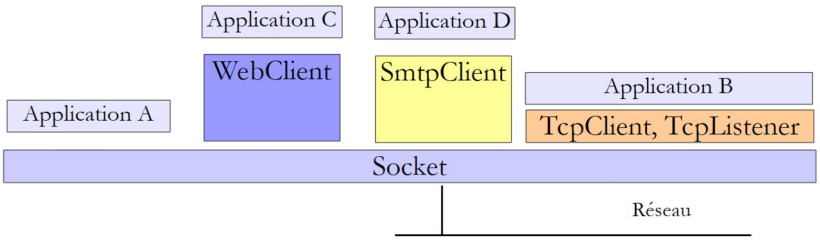


Filezilla est un client FTP offrant une interface windows pour faire des transferts de fichiers. Les actions de l'utilisateur sur l'interface windows sont traduites en commandes FTP qui sont loguées en [1]. C'est une bonne façon de découvrir les commandes du protocole FTP.

11.5 Les classes .NET de la programmation internet

11.5.1 Choisir la classe adaptée

Le framework .NET offre différentes classes pour travailler avec le réseau :



- la classe **Socket** est celle qui opère le plus près du réseau. Elle permet de gérer finement la connexion réseau. Le terme *socket* désigne une prise de courant. Le terme a été étendu pour désigner une prise de réseau logicielle. Dans une communication TCP-IP entre deux machines A et B, ce sont deux *sockets* qui communiquent entre-eux. Une application peut travailler directement avec les *sockets*. C'est le cas de l'application A ci-dessus. Un socket peut être un socket *client* ou *serveur*.
- si on souhaite travailler à un niveau moins fin que celui de la classe *Socket*, on pourra utiliser les classes
  - **TcpClient** pour créer un client Tcp
  - **TcpListener** pour créer un serveur Tcp

Ces deux classes offrent à l'application qui les utilisent, une vue plus simple de la communication réseau en gérant pour elle les détails techniques de gestion des sockets.

- .NET offre des classes spécifiques à certains protocoles :
  - la classe **SmtpClient** pour gérer le protocole SMTP de communication avec un serveur SMTP d'envoi de courriers électroniques
  - la classe **WebClient** pour gérer les protocoles HTTP ou FTP de communication avec un serveur web.

On retiendra que la classe *Socket* est suffisante en elle-même pour gérer toute communication tcp-ip mais on cherchera avant tout à utiliser les classes de plus haut niveau afin de faciliter l'écriture de l'application tcp-ip.

11.5.2 La classe TcpClient

La classe T**cpClient** est la classe qui convient dans la plupart des cas pour créer le client d'un service TCP. Elle a parmi ses constructeurs C, méthodes M et propriétés P, les suivants :

`TcpClient(string hostname, int port)`	C	crée une liaison tcp avec le service 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*
`Socket Client`	P	le socket utilisé par le client pour communiquer avec le serveur.
`NetworkStream GetStream()`	M	obtient un flux de lecture et d'écriture vers le serveur. C'est ce flux qui permet les échanges client-serveur.
`void Close()`	M	ferme la connexion. Le socket et le flux *NetworkStream* sont également fermés
`bool Connected()`	P	vrai si la connexion a été établie

La classe **NetworkStream** représente le flux réseau entre le client et le serveur. Elle 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 est-il intéressant d'utiliser des objets *StreamReader* et *StreamWriter* pour lire et écrire ces lignes dans le flux réseau. Ainsi si une machine M1 a établi une liaison avec une machine M2 à l'aide d'un objet TcpClient *client1* et 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 à la machine M2.

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();
```

Nous avons maintenant les éléments pour écrire l'architecture de base d'un client internet ayant le protocole de communication basique suivant avec le serveur :

- le client envoie une demande contenue dans une unique ligne
- le serveur envoie une réponse contenue dans une unique ligne

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.
5.  namespace ... {
6.   class ... {
7.      static void Main(string[] args) {
8.         ...
9.         try {
10.           // on se connecte au service
```

```
11.          using (TcpClient tcpClient = new TcpClient(serveur, port)) {
12.             using (NetworkStream networkStream = tcpClient.GetStream()) {
13.                using (StreamReader reader = new StreamReader(networkStream)) {
14.                   using (StreamWriter writer = new StreamWriter(networkStream)) {
15.                      // flux de sortie non bufferisé
16.                      writer.AutoFlush = true;
17.                      // boucle demande - réponse
18.                      while (true) {
19.                         // la demande vient du clavier
20.                         Console.Write("Demande (bye pour arrêter) : ");
21.                         demande = Console.ReadLine();
22.                         // fini ?
23.                         if (demande.Trim().ToLower() == "bye")
24.                            break;
25.                         // on envoie la demande au serveur
26.                         writer.WriteLine(demande);
27.                         // on lit la réponse du serveur
28.                         réponse = reader.ReadLine();
29.                         // on traite la réponse
30.                         ...
31.                      }
32.                   }
33.                }
34.             }
35.          }
36.       } catch (Exception e) {
37.          // erreur
38.          ...
39.       }
40.    }
41. }
42. }
```

- ligne 11 : création connexion du client - la clause *using* assure que les ressources liées à celle-ci seront libérées à la sortie du *using*.
- ligne 12 : ouverture du flux réseau dans une clause *using*
- ligne 13 : création et exploitation du flux de lecture dans une clause *using*
- ligne 14 : création et exploitation du flux d'écriture dans une clause *using*
- ligne 16 : ne pas bufferiser le flux de sortie
- lignes 18-31 : le cycle demande client / réponse serveur
- ligne 26 : le client envoie sa demande au serveur
- ligne 28 : le client attend la réponse du serveur. C'est une opération bloquante comme celle de la lecture au clavier. L'attente se termine par l'arrivée d'une chaîne terminée par "\n" ou bien par une fin de flux. Celle-ci se produira si le serveur ferme la connexion qu'il a ouverte avec le client.

11.5.3 La classe TcpListener

La classe **TcpListener** est la classe qui convient dans la plupart des cas pour créer un service TCP. Elle a parmi ses constructeurs C, méthodes M et propriétés P, les suivants :

`TcpListener(int port)`	C	crée un service TCP qui va attendre (listen) les demandes des clients sur un port passé en paramètre (port) appelé **port d'écoute**. Si la machine est connectée à plusieurs réseaux IP, le service écoute sur chacun des réseaux.
`TcpListener(IPAddress ip, int port)`	C	idem mais l'écoute n'a lieu que sur l'adresse ip précisée.
`void Start()`	M	lance l'écoute des demandes clients
`TcpClient AcceptTcpClient()`	M	accepte la demande d'un client. Ouvre alors une nouvelle connexion avec celui-ci, appelée connexion de service. Le port utilisé côté serveur est aléatoire et choisi par le système. On l'appelle le port de service. *AcceptTcpClient* rend comme résultat l'objet *TcpClient* associé côté serveur à la connexion de service.
`void Stop()`	M	arrête d'écouter les demandes clients
`Socket Server`	P	le socket d'écoute du serveur

La structure de base d'un serveur TCP qui échangerait avec ses clients selon le protocole suivant :

- le client envoie une demande contenue dans une unique ligne
- le serveur envoie une réponse contenue dans une unique ligne

pourrait ressembler à ceci :

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.  using System.Threading;
5.  using System.Net;
6.
7.  namespace ... {
8.   public class ... {
9.         ...
10.        // on crée le service d'écoute
11.        TcpListener ecoute = null;
12.        try {
13.           // on crée le service - il écoutera sur toutes les interfaces réseau de la machine
14.           ecoute = new TcpListener(IPAddress.Any, port);
15.           // on le lance
16.           ecoute.Start();
17.           // boucle de service
18.           TcpClient tcpClient = null;
19.           // boucle infinie - sera arrêtée par Ctrl-C
20.           while (true) {
21.              // attente d'un client
22.              tcpClient = ecoute.AcceptTcpClient();
23.              // le service est assuré par une autre tâche
24.              ThreadPool.QueueUserWorkItem(Service, tcpClient);
25.              // client suivant
26.           }
27.        } catch (Exception ex) {
28.           // on signale l'erreur
29.           ...
30.        } finally {
31.           // fin du service
32.           ecoute.Stop();
33.        }
34.     }
35.
36.     // ------------------------------------------------------
37.     // assure le service à un client
38.     public static void Service(Object infos) {
39.        // on récupère le client qu'il faut servir
40.        Client client = infos as Client;
41.        // exploitation liaison TcpClient
42.        try {
43.           using (TcpClient tcpClient = client.CanalTcp) {
44.              using (NetworkStream networkStream = tcpClient.GetStream()) {
45.                 using (StreamReader reader = new StreamReader(networkStream)) {
46.                    using (StreamWriter writer = new StreamWriter(networkStream)) {
47.                       // flux de sortie non bufferisé
48.                       writer.AutoFlush = true;
49.                       // boucle lecture demande/écriture réponse
50.                       bool fini=false;
51.                       while (! fini) != null) {
52.                          // attente demande client - opération bloquante
53.                          demande=reader.ReadLine();
54.                          // préparation réponse
55.                          réponse=...;
56.                          // envoi réponse au client
57.                          writer.WriteLine(réponse);
58.                          // demande suivante
59.                       }
60.                    }
61.                 }
62.              }
63.           }
64.        } catch (Exception e) {
65.           // erreur
66.           ...
67.        } finally {
68.           // fin client
69.           ...
70.        }
71.     }
72.  }
73. }
```

- ligne 14 : le service d'écoute est créé pour un port donné et une adresse IP donnée. Il faut se rappeler ici qu'une machine a au moins deux adresses IP : l'adresse "127.0.0.1" qui est son adresse de bouclage sur elle-même et l'adresse "I1.I2.I3.I4"

qu'elle a sur le réseau auquel elle est connectée. Elle peut avoir d'autres adresses IP si elle connectée à plusieurs réseaux IP. *IPAddress.Any* désigne toutes les adresses IP d'une machine.

- ligne 16 : le service d'écoute démarre. Auparavant il avait été créé mais il n'écoutait pas encore. Ecouter signifie attendre les demandes des clients.
- lignes 20-26 : la boucle attente demande client / service client répétée pour chaque nouveau client
- ligne 22 : la demande d'un client est acceptée. La méthode *AcceptTcpClient* rend une instance *TcpClient* dite de service :
    - le client a fait sa demande avec sa propre instance *TcpClient* côté client que nous appellerons *TcpClientDemande*
    - le serveur accepte cette demande avec *AcceptTcpClient*. Cette méthode crée une instance *TcpClient* côté serveur, que nous appellerons *TcpClientService*. On a alors une connexion Tcp **ouverte** avec aux deux bouts les instances *TcpClientDemande* <--> *TcpClientService*.
    - la communication client / serveur qui prend place ensuite se fait sur cette connexion. Le service d'écoute n'intervient plus.
- ligne 24 : afin que le serveur puisse traiter plusieurs clients à la fois, le service est assuré par des threads, 1 thread par client.
- ligne 32 : le service d'écoute est fermé
- ligne 38 : la méthode exécutée par le thread de service à un client. Elle reçoit en paramètre l'instance *TcpClient* déjà connectée au client qui doit être servi.
- lignes 38-71 : on retrouve un code similaire à celui du client Tcp basique étudié précédemment.

11.6 Exemples de clients / serveurs TCP

11.6.1 Un 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 :

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.  using System.Threading;
5.  using System.Net;
6.
7.  // appel : serveurEcho port
8.  // serveur d'écho
9.  // renvoie au client la ligne que celui-ci lui a envoyée
10.
11. namespace Chap9 {
12.  public class ServeurEcho {
13.     public const string syntaxe = "Syntaxe : [serveurEcho] port";
14.
15.     // programme principal
16.     public static void Main(string[] args) {
17.
18.        // y-a-t-il un argument ?
19.        if (args.Length != 1) {
20.           Console.WriteLine(syntaxe);
21.           return;
22.        }
23.        // cet argument doit être entier >0
24.        int port = 0;
25.        if (!int.TryParse(args[0], out port) || port<=0) {
26.           Console.WriteLine("{0} : {1}Port incorrect", syntaxe, Environment.NewLine);
27.           return;
28.        }
29.        // on crée le service d'écoute
30.        TcpListener ecoute = null;
31.        int numClient = 0;   // n° client suivant
32.        try {
33.           // on crée le service - il écoutera sur toutes les interfaces réseau de la machine
34.           ecoute = new TcpListener(IPAddress.Any, port);
35.           // on le lance
36.           ecoute.Start();
37.           // suivi
38.           Console.WriteLine("Serveur d'écho lancé sur le port {0}", ecoute.LocalEndpoint);
39.           // threads de service
40.           ThreadPool.SetMinThreads(10, 10);
41.           ThreadPool.SetMaxThreads(10, 10);
```

```
42.          // boucle de service
43.          TcpClient tcpClient = null;
44.          // boucle infinie - sera arrêtée par Ctrl-C
45.          while (true) {
46.             // attente d'un client
47.             tcpClient = ecoute.AcceptTcpClient();
48.             // le service est assuré par une autre tâche
49.             ThreadPool.QueueUserWorkItem(Service, new Client() { CanalTcp = tcpClient, NumClient
    = numClient });
50.             // client suivant
51.             numClient++;
52.          }
53.       } catch (Exception ex) {
54.          // on signale l'erreur
55.          Console.WriteLine("L'erreur suivante s'est produite sur le serveur : {0}", ex.Message);
56.       } finally {
57.          // fin du service
58.          ecoute.Stop();
59.       }
60.    }
61.
62.    // ------------------------------------------------------
63.    // assure le service à un client du serveur d'écho
64.    public static void Service(Object infos) {
65.       // on récupère le client qu'il faut servir
66.       Client client = infos as Client;
67.       // rend le service au client
68.       Console.WriteLine("Début de service au client {0}", client.NumClient);
69.       // exploitation liaison TcpClient
70.       try {
71.          using (TcpClient tcpClient = client.CanalTcp) {
72.             using (NetworkStream networkStream = tcpClient.GetStream()) {
73.                using (StreamReader reader = new StreamReader(networkStream)) {
74.                   using (StreamWriter writer = new StreamWriter(networkStream)) {
75.                      // flux de sortie non bufferisé
76.                      writer.AutoFlush = true;
77.                      // boucle lecture demande/écriture réponse
78.                      string demande = null;
79.                      while ((demande = reader.ReadLine()) != null) {
80.                         // suivi console
81.                         Console.WriteLine("<--- Client {0} : {1}", client.NumClient, demande);
82.                         // écho de la demande vers le client
83.                         writer.WriteLine("[{0}]", demande);
84.                         // suivi console
85.                         Console.WriteLine("---> Client {0} : {1}", client.NumClient, demande);
86.                         // le service s'arrête lorsque le client envoie "bye"
87.                         if (demande.Trim().ToLower() == "bye")
88.                            break;
89.                      }
90.                   }
91.                }
92.             }
93.          }
94.       } catch (Exception e) {
95.          // erreur
96.          Console.WriteLine("L'erreur suivante s'est produite lors du service au client {0} :
    {1}", client.NumClient, e.Message);
97.       } finally {
98.          // fin client
99.          Console.WriteLine("Fin du service au client {0}", client.NumClient);
100.      }
101.   }
102. }
103.
104. // infos client
105. internal class Client {
106.    public TcpClient CanalTcp { get; set; }      // liaison avec le client
107.    public int NumClient { get; set; }        // n° de client
108. }
109.}
```

La structure du serveur d'écho est conforme à l'architecture basique des serveurs Tcp exposée précédemment. Nous ne commenterons que la partie "service au client" :

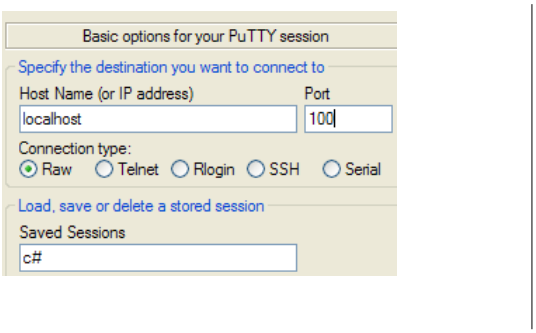
- ligne 79 : la demande du client est lue
- ligne 83 : elle est renvoyée au client entourée de crochets
- ligne 79 : le service s'arrête lorsque le client ferme la connexion

Dans une fenêtre Dos, nous utilisons l'exécutable du projet C# :

```
...\Chap9\02\bin\Release>dir
 03/05/2008  11:46             7 168 ServeurEcho.exe
```

```
...>ServeurEcho 100
Serveur d'écho lancé sur le port 0.0.0.0:100
```

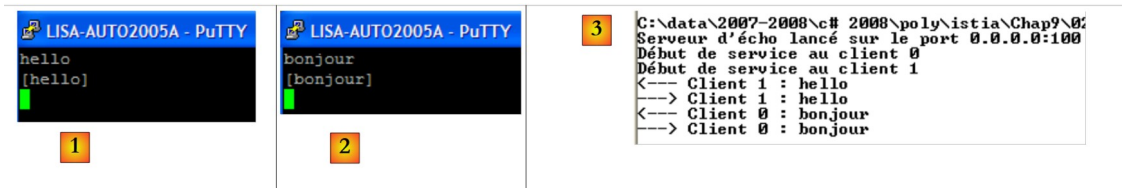
Nous lançons ensuite deux clients *putty* que nous connectons au port 100 de la machine *localhost* :



L'affichage console du serveur d'écho devient :

```
1.  Serveur d'écho lancé sur le port 0.0.0.0:100
2.  Début de service au client 0
3.  Début de service au client 1
```

Le client 1 puis le client 0 envoient les textes suivants :



- [1] : le client n° 1
- [2] : le client n° 0
- [3] : la console du serveur d'écho

- en [4] : le client 1 se déconnecte avec la commande *bye*.
- en [5] : le serveur le détecte

Le serveur peut être arrêté par Ctrl-C. Le client n° 0 le détecte alors [6].

11.6.2 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.

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.
5.  namespace Chap9 {
6.   // se connecte à un serveur d'écho
7.   // toute ligne tapée au clavier est reçue en écho
8.   class ClientEcho {
9.      static void Main(string[] args) {
10.        // syntaxe
11.        const string syntaxe = "pg machine port";
12.
13.        // nombre d'arguments
14.        if (args.Length != 2) {
15.           Console.WriteLine(syntaxe);
16.           return;
17.        }
18.
19.        // on note le nom du serveur
20.        string serveur = args[0];
21.
22.        // le port doit être entier >0
23.        int port = 0;
24.        if (!int.TryParse(args[1], out port) || port <= 0) {
25.           Console.WriteLine("{0}{1}port incorrect", syntaxe, Environment.NewLine);
26.           return;
27.        }
28.
29.        // on peut travailler
30.        string demande = null;     // demande du client
31.        string réponse = null;     // réponse du serveur
32.        try {
33.           // on se connecte au service
34.           using (TcpClient tcpClient = new TcpClient(serveur, port)) {
35.              using (NetworkStream networkStream = tcpClient.GetStream()) {
36.                 using (StreamReader reader = new StreamReader(networkStream)) {
37.                    using (StreamWriter writer = new StreamWriter(networkStream)) {
38.                       // flux de sortie non bufferisé
39.                       writer.AutoFlush = true;
40.                       // boucle demande - réponse
41.                       while (true) {
42.                          // la demande vient du clavier
43.                          Console.Write("Demande (bye pour arrêter) : ");
44.                          demande = Console.ReadLine();
45.                          // fini ?
46.                          if (demande.Trim().ToLower() == "bye")
47.                             break;
48.                          // on envoie la demande au serveur
49.                          writer.WriteLine(demande);
50.                          // on lit la réponse du serveur
51.                          réponse = reader.ReadLine();
52.                          // on traite la réponse
53.                          Console.WriteLine("Réponse : {0}", réponse);
54.                       }
55.                    }
56.                 }
57.              }
58.           }
59.        } catch (Exception e) {
60.           // erreur
61.           Console.WriteLine("L'erreur suivante s'est produite : {0}", e.Message);
62.        }
63.     }
64.  }
65. }
```

La structure de ce client est conforme à l'architecture générale basique proposée pour les clients T*cp*. 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 A (n° 0) on a les affichages suivants :

```
1.  ...\Chap9\03\bin\Release>ClientEcho localhost 100
2.  Demande (bye pour arrêter) : ligne1A
3.  Réponse : [ligne1A]
```

```
4.  Demande (bye pour arrêter) : ligne2A
5.  Réponse : [ligne2A]
6.  Demande (bye pour arrêter) :
```

Dans celle du client B (n° 1) :

```
1.  ...\Chap9\03\bin\Release>ClientEcho localhost 100
2.  Demande (bye pour arrêter) : ligne1B
3.  Réponse : [ligne1B]
4.  Demande (bye pour arrêter) : ligne2B
5.  Réponse : [ligne2B]
6.  Demande (bye pour arrêter) :
```

Dans celle du serveur :

```
1.  ...\Chap9\02\bin\Release>ServeurEcho 100
2.  Serveur d'écho lancé sur le port 0.0.0.0:100
3.  Début de service au client 0
4.  <--- Client 0 : ligne1A
5.  ---> Client 0 : ligne1A
6.  <--- Client 0 : ligne2A
7.  ---> Client 0 : ligne2A
8.  Début de service au client 1
9.  <--- Client 1 : ligne1B
10. ---> Client 1 : ligne1B
11. <--- Client 1 : ligne2B
12. ---> Client 1 : ligne2B
```

Le client A n° 0 se déconnecte :

```
1.  Demande (bye pour arrêter) : ligne1A
2.  Réponse : [ligne1A]
3.  ...
4.  Demande (bye pour arrêter) : bye
```

La console du serveur :

```
1.  Serveur d'écho lancé sur le port 0.0.0.0:100
2.  ...
3.  Fin du service au client 0
```

11.6.3 Un client TCP générique

Nous allons écrire un client Tcp générique qui sera lancé de la façon suivante : **ClientTcpGenerique serveur port**. Il aura un fonctionnement analogue au client putty mais aura une interface console et ne présentera pas d'option de configuration.

Dans l'application précédente, 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 le cycle envoi d'une unique ligne au serveur / réception d'une unique ligne envoyée par le serveur, ne convient-il pas toujours. Pour gérer les protocoles plus complexes que celui d'écho, le client Tcp générique aura deux threads :

- le thread principal lira les lignes de texte tapées au clavier et les enverra au serveur.
- un thread secondaire travaillera en parallèle et sera consacré à la lecture des lignes de texte envoyées par le serveur. Dès qu'il en reçoit une, il l'affiche sur la console. Le thread ne s'arrête que lorsque le serveur clôt la connexion. Il travaille donc en continu.

Le code est le suivant :

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.  using System.Threading;
5.
6.  namespace Chap9 {
7.   // reçoit en paramètre les caractéristiques d'un service sous la forme : serveur port
8.   // se connecte au service
9.   // envoie au serveur chaque ligne tapée au clavier
10.  // crée un thread pour lire en continu les lignes de texte envoyées par le serveur
11.  class ClientTcpGenerique {
12.     static void Main(string[] args) {
```

```
13.         // syntaxe
14.         const string syntaxe = "pg serveur port";
15.
16.         // nombre d'arguments
17.         if (args.Length != 2) {
18.             Console.WriteLine(syntaxe);
19.             return;
20.         }
21.
22.         // on note le nom du serveur
23.         string serveur = args[0];
24.
25.         // le port doit être entier >0
26.         int port = 0;
27.         if (!int.TryParse(args[1], out port) || port <= 0) {
28.             Console.WriteLine("{0}{1}port incorrect", syntaxe, Environment.NewLine);
29.             return;
30.         }
31.         // on se connecte au service
32.         TcpClient tcpClient = null;
33.         try {
34.             tcpClient = new TcpClient(serveur, port);
35.         } catch (Exception ex) {
36.             // erreur
37.             Console.WriteLine("Impossible de se connecter au service ({0},{1}) : erreur {2}", serveur, port,
    ex.Message);
38.             // fin
39.             return;
40.         }
41.
42.         // on lance un thread à part pour lire les lignes de texte envoyées par le serveur
43.         ThreadPool.QueueUserWorkItem(Receive, tcpClient);
44.
45.         // la lecture des commandes clavier se fait dans le thread principal
46.         Console.WriteLine("Tapez vos commandes (bye pour arrêter) : ");
47.         string demande = null;      // demande du client
48.         try {
49.             // on exploite la connexion client
50.             using (tcpClient) {
51.                 // on crée un flux d'écriture vers le serveur
52.                 using (NetworkStream networkStream = tcpClient.GetStream()) {
53.                     using (StreamWriter writer = new StreamWriter(networkStream)) {
54.                         // flux de sortie non bufferisé
55.                         writer.AutoFlush = true;
56.                         // boucle demande - réponse
57.                         while (true) {
58.                             demande = Console.ReadLine();
59.                             // fini ?
60.                             if (demande.Trim().ToLower() == "bye")
61.                                 break;
62.                             // on envoie la demande au serveur
63.                             writer.WriteLine(demande);
64.                         }
65.                     }
66.                 }
67.             }
68.         } catch (Exception e) {
69.             // erreur
70.             Console.WriteLine("L'erreur suivante s'est produite dans le thread principal : {0}", e.Message);
71.         }
72.     }
73.
74.     // thread de lecture client <-- serveur
75.     public static void Receive(object infos) {
76.         // données locales
77.         string réponse = null;  // réponse du serveur
78.         // création flux d'entrée
79.         try {
80.             using (TcpClient tcpClient = infos as TcpClient) {
81.                 using (NetworkStream networkStream = tcpClient.GetStream()) {
82.                     using (StreamReader reader = new StreamReader(networkStream)) {
83.                         // boucle lecture en continu des lignes de texte du flux d'entrée
84.                         while ((réponse = reader.ReadLine()) != null) {
85.                             // affichage console
86.                             Console.WriteLine("<-- {0}", réponse);
87.                         }
88.                     }
89.                 }
90.             }
91.         } catch (Exception ex) {
92.             // erreur
93.             Console.WriteLine("Flux de lecture : l'erreur suivante s'est produite : {0}", ex.Message);
94.         } finally {
95.             // on signale la fin du thread de lecture
96.             Console.WriteLine("Fin du thread de lecture des réponses du serveur. Si besoin est, arrêtez le thread
    de lecture console avec la commande bye.");
97.         }
98.     }
99.  }
100.}
```

- ligne 34 : le client se connecte au serveur
- ligne 43 : un thread de lecture des lignes de texte du serveur est lancé. Il doit exécuter la méthode *Receive* de la ligne 73. On passe à cette méthode l'instance *TcpClient* qui a été connectée au serveur.
- lignes 57-64 : la boucle saisie commande clavier / envoi commande au serveur. La saisie des commandes clavier est assurée par le thread principal.
- lignes 75-98 : la méthode *Receive* exécutée par le thread de lecture des lignes de texte. Cette méthode reçoit en paramètre l'instance *TcpClient* qui a été connectée au serveur.
- lignes 84-87 : la boucle en continu de lecture des lignes de texte envoyées par le serveur. Elle ne s'arrête que lorsque le serveur clôt la connexion ouverte avec le client.

Voici quelques exemples reprenant ceux utilisés avec le client *putty* au paragraphe 11.4, page 320. Le client est exécuté dans une console Dos.

Protocole HTTP

```
1.  ...\Chap9\04\bin\Release>ClientTcpGenerique istia.univ-angers.fr 80
2.  Tapez vos commandes (bye pour arrêter) :
3.  GET /inconnu HTTP/1.1
4.  Host: istia.univ-angers.fr:80
5.  Connection: Close
6.
7.  <-- HTTP/1.1 404 Not Found
8.  <-- Date: Sat, 03 May 2008 12:35:11 GMT
9.  <-- Server: Apache/1.3.34 (Debian) PHP/4.4.4-8+etch4 mod_jk/1.2.18 mod_perl/1.29
10.
11. <-- Connection: close
12. <-- Transfer-Encoding: chunked
13. <-- Content-Type: text/html; charset=iso-8859-1
14. <--
15. <-- 11a
16. <-- <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
17. <-- <HTML><HEAD>
18. <-- <TITLE>404 Not Found</TITLE>
19. <-- </HEAD><BODY>
20. <-- <H1>Not Found</H1>
21. <-- The requested URL /inconnu was not found on this server.<P>
22. <-- <HR>
23. <-- <ADDRESS>Apache/1.3.34 Server at www.istia.univ-angers.fr Port 80</ADDRESS>
24. <-- </BODY></HTML>
25. <--
26. <-- 0
27. <--
28. [Fin du thread de lecture des réponses du serveur]
29. bye
30.
31. ...\Chap9\04\bin\Release>
```

Le lecteur est invité à relire les explications données au paragraphe 11.4.2, page 321. Nous ne commentons que ce qui est propre à l'application :

- ligne 28 : après l'envoi de la ligne 27, le serveur HTTP a fermé la connexion, ce qui a provoqué la fin du thread de lecture. Le thread principal qui lit les commandes tapées au clavier est lui toujours actif. La commande de la ligne 29, tapée au clavier, l'arrête.

Protocole SMTP

```
1.  ...\Chap9\04\bin\Release>ClientTcpGenerique smtp.neuf.fr 25
2.  Tapez vos commandes (bye pour arrêter) :
3.  <-- 220 neuf-infra-smtp-out-sp604002av.neufgp.fr neuf telecom Service relais mail ready
4.  HELO istia.univ-angers.fr
5.  <-- 250 neuf-infra-smtp-out-sp604002av.neufgp.fr hello [84.100.189.193], Banniere OK , pret pour
    envoyer un mail
6.  mail from: xx@neuf.fr
7.  <-- 250 2.1.0 <xx@neuf.fr> sender ok
8.  rcpt to: yy@univ-angers.fr
9.  <-- 250 2.1.5 <yy@univ-angers.fr> destinataire ok
10. data
11. <-- 354 enter mail, end with "." on a line by itself
12. ligne1
13. ligne2
14. .
15. <-- 250 2.0.0 M0jL1Z0044AoCxw0200000 message ok
16. quit
17. <-- 221 2.0.0 neuf-infra-smtp-out-sp604002av.neufgp.fr neuf telecom closing connection
```

```
18. [Fin du thread de lecture des réponses du serveur]
19. bye
20.
21. ...\Chap9\04\bin\Release>
```

Le lecteur est invité à relire les explications données au paragraphe 11.4.3, page 323 et à tester les autres exemples utilisés avec le client *putty*.

11.6.4 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 dans une fenêtre Dos par : **ServeurTcpGenerique portEcoute**, où *portEcoute* est le port sur lequel les clients doivent se connecter. Le service au client sera assuré par deux threads :

- le thread principal qui :
  - traitera les clients les uns après les autres et non en parallèle.
  - qui lira les lignes tapées au clavier par l'utilisateur et les enverra au client. L'utilisateur signalera par la commande **bye** qu'il clôt la connexion avec le client. C'est parce que la console ne peut être utilisée pour deux clients simultanément que notre serveur ne traite qu'un client à la fois.
- un thread secondaire se consacrant exclusivement à la lecture des lignes de texte envoyées par le client

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 du paragraphe 11.6.3, page 333, pour lui parler. La fenêtre du client est la suivante :

```
1.  ...\Chap9\04\bin\Release>ClientTcpGenerique localhost 100
2.  Tapez vos commandes (bye pour arrêter) :
3.  commande 1 du client 1
4.  <-- réponse 1 au client 1
5.  commande 2 du client 1
6.  <-- réponse 2 au client 1
7.  bye
```

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 :

```
1.  ...\Chap9\05\bin\Release>ServeurTcpGenerique 100
2.  Serveur générique lancé sur le port 0.0.0.0:100
3.  Client 127.0.0.1:4165
4.  Tapez vos commandes (bye pour arrêter) :
5.  <-- commande 1 du client 1
6.  réponse 1 au client 1
7.  <-- commande 2 du client 1
8.  réponse 2 au client 1
9.  [Fin du thread de lecture des demandes du client]
10. bye
```

Les lignes commençant par <-- sont celles envoyées du client au serveur, les autres celles envoyées par le serveur au client. La ligne 9 indique que le thread de lecture des demandes du client s'est arrêté. Le thread principal du serveur est toujours en attente de commandes tapées au clavier pour les envoyer au client. Il faut alors taper au clavier la commande *bye* de la ligne 10 pour passer au client suivant. Le serveur est encore actif alors que le client 1 est terminé. On lance un second client pour le même serveur :

```
1.  ...\Chap9\04\bin\Release>ClientTcpGenerique localhost 100
2.  Tapez vos commandes (bye pour arrêter) :
3.  commande 3 du client 2
4.  <-- réponse 3 au client 2
5.  bye
```

La fenêtre du serveur est alors celle-ci :

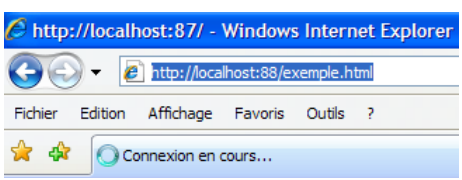
```
1.  Tapez vos commandes (bye pour arrêter) :
2.  Client 127.0.0.1:4166
3.  <-- commande 3 du client 2
4.  réponse 3 au client 2
5.  [Fin du thread de lecture des demandes du client]
6.  bye
```

Après la ligne 6 ci-dessus, le serveur est passé en attente d'un nouveau client. On peut l'arrêter par Ctrl-C.

Simulons maintenant un serveur web en lançant notre serveur générique sur le port 88 :

```
1.  ...\Chap9\05\bin\Release>ServeurTcpGenerique 88
2.
3.  Serveur générique lancé sur le port 0.0.0.0:88
```

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 :

```
1.  Serveur générique lancé sur le port 0.0.0.0:88
2.  Client 127.0.0.1:4167
3.  Tapez vos commandes (bye pour arrêter) :
4.  <-- GET /exemple.html HTTP/1.1
5.  <-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash,
    application/vnd.ms-excel, application/msword, application/xaml+xml, application/vnd.ms-
    xpsdocument, application/x-ms-xbap, application/x-ms-appl
6.  ication, application/x-silverlight, */*
7.  <-- Accept-Language: fr,en-US;q=0.7,fr-FR;q=0.3
8.  <-- UA-CPU: x86
9.  <-- Accept-Encoding: gzip, deflate
10. <-- User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.
11. 4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.590; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
12. <-- Host: localhost:88
13. <-- Connection: Keep-Alive
14. <--
```

On découvre les entêtes HTTP envoyés par le navigateur. Cela nous permet de découvrir d'autres entêtes HTTP que ceux déjà rencontrés. 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 :

```
1.  HTTP/1.1 200 OK
2.  Date: Sat, 03 May 2008 07:53:47 GMT
3.  Server: Apache/1.3.34 (Debian) PHP/4.4.4-8+etch4 mod_jk/1.2.18 mod_perl/1.29
4.  X-Powered-By: PHP/4.4.4-8+etch4
5.  Set-Cookie: fe_typo_user=0d2e64b317; path=/
6.  Connection: close
7.  Transfer-Encoding: chunked
8.  Content-Type: text/html;charset=iso-8859-1
9.
10. 693f
11. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
12.          <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr_FR" lang="fr_FR">
13. ....
14.          </html>
15. 0
```

Essayons de donner une réponse analogue en s'en tenant au strict mimimum :

```
1.  HTTP/1.1 200 OK
2.  Server: serveur tcp generique
3.  Connection: close
4.  Content-Type: text/html
5.
6.  <html>
7.  <head><title>Serveur generique</title></head>
8.  <body><h2>Reponse du serveur generique</h2></body>
9.  </html>
10. bye
```

```
11. Flux de lecture des lignes de texte du client : l'erreur suivante s'est produite : Unable to read
    data from the transport connection: Une opération de blocage a été interrompue par un appel à
    WSACancelBlockingCall.
12. [Fin du thread de lecture des demandes du client]
```

Nous nous sommes limités dans notre réponse aux entêtes HTTP des lignes 1-4. Nous ne donnons pas la taille du document 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 celle des lignes 6-9. L'utilisateur au clavier ferme ensuite la connexion au client en tapant la commande *bye*, ligne 10. Sur cette commande clavier le thread principal ferme la connexion avec le client. Ceci provoque l'exception de la ligne 11. Le thread de lecture des lignes de texte du client a été interrompu brutalement par la fermeture de la liaison avec le client et a lancé une exception. Après la ligne 12, le serveur se met en attente d'un nouveau client.

Le navigateur client affiche désormais la chose suivante :

Si ci-dessus, on fait *Affichage/Source* pour voir ce qu'a reçu le navigateur, on obtient [2], c'est à dire exactement ce qu'on a envoyé depuis le serveur générique.

Le code du serveur TCP générique est le suivant :

```
1.  using System;
2.  using System.IO;
3.  using System.Net;
4.  using System.Net.Sockets;
5.  using System.Threading;
6.
7.  namespace Chap9 {
8.   public class ServeurTcpGenerique {
9.      public const string syntaxe = "Syntaxe : ServeurGénérique Port";
10.
11.     // programme principal
12.     public static void Main(string[] args) {
13.
14.        // y-a-t-il un argument ?
15.        if (args.Length != 1) {
16.           Console.WriteLine(syntaxe);
17.           Environment.Exit(1);
18.        }
19.        // cet argument doit être entier >0
20.        int port = 0;
21.        if (!int.TryParse(args[0], out port) || port <= 0) {
22.           Console.WriteLine("{0} : {1}Port incorrect", syntaxe, Environment.NewLine);
23.           Environment.Exit(2);
24.        }
25.        // on crée le service d'écoute
26.        TcpListener ecoute = null;
27.        try {
28.           // on crée le service
29.           ecoute = new TcpListener(IPAddress.Any, port);
30.           // on le lance
31.           ecoute.Start();
32.           // suivi
33.           Console.WriteLine("Serveur générique lancé sur le port {0}", ecoute.LocalEndpoint);
34.           while (true) {
35.              // attente d'un client
36.              Console.WriteLine("Attente du client suivant...");
37.              TcpClient tcpClient = ecoute.AcceptTcpClient();
38.              Console.WriteLine("Client {0}", tcpClient.Client.RemoteEndPoint);
39.              // on lance un thread à part pour lire les lignes de texte envoyées par le client
40.              ThreadPool.QueueUserWorkItem(Receive, tcpClient);
41.              // la lecture des commandes clavier se fait dans le thread principal
42.              Console.WriteLine("Tapez vos commandes (bye pour arrêter) : ");
```

```
43.             string réponse = null;     // réponse serveur
44.             // on exploite la connexion client
45.             using (tcpClient) {
46.                // on crée un flux d'écriture vers le client
47.                using (NetworkStream networkStream = tcpClient.GetStream()) {
48.                   using (StreamWriter writer = new StreamWriter(networkStream)) {
49.                      // flux de sortie non bufferisé
50.                      writer.AutoFlush = true;
51.                      // boucle de saisie des réponses au clavier
52.                      while (true) {
53.                         réponse = Console.ReadLine();
54.                         // fini ?
55.                         if (réponse.Trim().ToLower() == "bye")
56.                            break;
57.                         // on envoie la demande au client
58.                         writer.WriteLine(réponse);
59.                      }
60.                   }
61.                }
62.             }
63.          }
64.       } catch (Exception ex) {
65.          // on signale l'erreur
66.          Console.WriteLine("Main : l'erreur suivante s'est produite : {0}", ex.Message);
67.       } finally {
68.          // fin de l'écoute
69.          ecoute.Stop();
70.       }
71.    }
72.
73.    // thread de lecture serveur <-- client
74.    public static void Receive(object infos) {
75.       // données locales
76.       string demande = null;  // demande du client
77.       string idClient=null;   // identité du client
78.
79.       // exploitation connexion client
80.       try {
81.          using (TcpClient tcpClient = infos as TcpClient) {
82.             // identité client
83.             idClient = tcpClient.Client.RemoteEndPoint.ToString();
84.             using (NetworkStream networkStream = tcpClient.GetStream()) {
85.                using (StreamReader reader = new StreamReader(networkStream)) {
86.                   // boucle lecture en continu des lignes de texte du flux d'entrée
87.                   while ((demande = reader.ReadLine()) != null) {
88.                      // affichage console
89.                      Console.WriteLine("<-- {0}", demande);
90.                   }
91.                }
92.             }
93.          }
94.       } catch (Exception ex) {
95.          // erreur
96.          Console.WriteLine("Flux de lecture des lignes de texte du client {1} : l'erreur suivante
   s'est produite : {0}", ex.Message,idClient);
97.       } finally {
98.          // on signale la fin du thread de lecture
99.          Console.WriteLine("Fin du thread de lecture des lignes de texte du client {0}. Si besoin
   est, arrêtez le thread de lecture console du serveur pour ce client, avec la commande bye.",
   idClient);
100.      }
101.   }
102. }
103.}
```

- ligne 29 : le service d'écoute est créé mais pas démarré. Il écoute toutes les interfaces réseau de la machine.
- ligne 31 : le service d'écoute est démarré
- ligne 34 : boucle infini d'attente des clients. L'utilisateur arrêtera le serveur par Ctrl-C.
- ligne 37 : attente d'un client - opération bloquante. Lorsque le client arrive, l'instance *TcpClient* rendue par la méthode *AcceptTcpClient* représente le côté serveur d'une connexion ouverte avec le client.
- ligne 40 : le flux de lecture des demandes du client est confié à un thread à part.
- ligne 45 : utilisation de la connexion au client dans une clause *using* afin d'être sûr qu'elle sera fermée quoiqu'il arrive.
- ligne 47 : utilisation du flux réseau dans une clause *using*
- ligne 48 : création dans une clause *using* d'un flux d'écriture sur le flux réseau
- ligne 50 : le flux d'écriture sera non bufferisé
- lignes 52-59 : boucle de saisie au clavier des commandes à envoyer au client
- ligne 69 : fin du service d'écoute. Cette instruction ne sera jamais exécutée ici puisque le serveur est arrêté par Ctrl-C.

- ligne 78 : la méthode *Receive* qui affiche en continu sur la console les lignes de texte envoyées par le client. On retrouve là ce qui a été vu pour le client TCP générique.

11.6.5 Un client Web

Nous avons vu dans l'exemple précédent, certains des entêtes HTTP qu'envoyait un navigateur :

```
1.  <-- GET /exemple.html HTTP/1.1
2.  <-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash,
    application/vnd.ms-excel, application/msword, application/xaml+xml, application/vnd.ms-
    xpsdocument, application/x-ms-xbap, application/x-ms-appl
3.  ication, application/x-silverlight, */*
4.  <-- Accept-Language: fr,en-US;q=0.7,fr-FR;q=0.3
5.  <-- UA-CPU: x86
6.  <-- Accept-Encoding: gzip, deflate
7.  <-- User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.
8.  4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.590; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
9.  <-- Host: localhost:88
10. <-- Connection: Keep-Alive
11. <--
```

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 :

```
1.  <-- GET /exemple.html HTTP/1.1
2.  <-- Host: localhost:88
3.  <-- Connection: close
4.  <--
```

- le premier entête indique le document désiré
- le second le serveur interrogé
- le troisième que nous souhaitons que le serveur ferme la connexion après nous avoir répondu.

Si ci-dessus ligne 1, nous remplaçons GET par HEAD, le serveur ne nous enverra que les entêtes HTTP et pas le document précisé ligne 1.

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 :

```
1.  ...\Chap9\06\bin\Release>ClientWeb http://istia.univ-angers.fr:80 HEAD
2.  HTTP/1.1 200 OK
3.  Date: Sat, 03 May 2008 14:05:24 GMT
4.  Server: Apache/1.3.34 (Debian) PHP/4.4.4-8+etch4 mod_jk/1.2.18 mod_perl/1.29
5.  X-Powered-By: PHP/4.4.4-8+etch4
6.  Set-Cookie: fe_typo_user=e668408ac1; path=/
7.  Connection: close
8.  Content-Type: text/html;charset=iso-8859-1
9.
10. ...\Chap9\06\bin\Release>
```

- ligne 1, nous ne demandons que les entêtes HTTP (HEAD)
- lignes 2-9 : la réponse du serveur

Si nous utilisons GET au lieu de HEAD dans l'appel au client Web, nous obtenons le même résultat qu'avec HEAD avec de plus le corps du document demandé.

Le code du client web est le suivant :

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.
5.  namespace Chap9 {
6.   class ClientWeb {
7.      static void Main(string[] args) {
8.         // syntaxe
9.         const string syntaxe = "pg URI GET/HEAD";
10.
11.        // nombre d'arguments
12.        if (args.Length != 2) {
```

```
13.           Console.WriteLine(syntaxe);
14.           return;
15.        }
16.
17.        // on note l'URI demandée
18.        string stringURI = args[0];
19.        string commande = args[1].ToUpper();
20.
21.        // vérification validité de l'URI
22.        if(! stringURI.StartsWith("http://")){
23.           Console.WriteLine("Indiquez une Url de la forme http://machine[:port]/document");
24.           return;
25.        }
26.        Uri uri = null;
27.        try {
28.           uri = new Uri(stringURI);
29.        } catch (Exception ex) {
30.           // URI incorrecte
31.           Console.WriteLine("L'erreur suivante s'est produite : {0}", ex.Message);
32.           return;
33.        }
34.        // vérification de la commande
35.        if (commande != "GET" && commande != "HEAD") {
36.           // commande incorrecte
37.           Console.WriteLine("Le second paramètre doit être GET ou HEAD");
38.           return;
39.        }
40.
41.        try {
42.           // on se connecte au service
43.           using (TcpClient tcpClient = new TcpClient(uri.Host, uri.Port)) {
44.              using (NetworkStream networkStream = tcpClient.GetStream()) {
45.                 using (StreamReader reader = new StreamReader(networkStream)) {
46.                    using (StreamWriter writer = new StreamWriter(networkStream)) {
47.                       // flux de sortie non bufferisé
48.                       writer.AutoFlush = true;
49.                       // on demande l'URL - envoi des entêtes HTTP
50.                       writer.WriteLine(commande + " " + uri.PathAndQuery + " HTTP/1.1");
51.                       writer.WriteLine("Host: " + uri.Host + ":" + uri.Port);
52.                       writer.WriteLine("Connection: close");
53.                       writer.WriteLine();
54.                       // on lit la réponse
55.                       string réponse = null;
56.                       while ((réponse = reader.ReadLine()) != null) {
57.                          // on affiche la réponse sur la console
58.                          Console.WriteLine(réponse);
59.                       }
60.                    }
61.                 }
62.              }
63.           }
64.        } catch (Exception e) {
65.           // on affiche l'exception
66.           Console.WriteLine("L'erreur suivante s'est produite : {0}", e.Message);
67.        }
68.     }
69.  }
70. }
```

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.

- lignes 26-33 : un objet *Uri* est construit à partir de la chaîne *stringURI* reçue en paramètre. 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/document?param1=val1&param2=val2;...* on aura :
  - **uri.Host**=*serveur*,
  - **uri.Port**=*port*,
  - **uri.Path**=*document*,
  - **uri.Query**=*param1=val1&param2=val2;...*,
  - **uri.pathAndQuery**= *cheminPageHTML?param1=val1&param2=val2;...*,
  - **uri.Scheme**=*http*.

11.6.6 Un client Web gérant les redirections

Le client Web précédent ne gère pas une éventuelle redirection de l'URL qu'il a demandée. Voici un exemple :

```
1.  ...\Chap9\06\bin\Release>ClientWeb http://www.ibm.com GET
2.  HTTP/1.1 302 Found
3.  Date: Sat, 03 May 2008 14:50:52 GMT
4.  Server: IBM_HTTP_Server
5.  Location: http://www.ibm.com/us/
6.  Content-Length: 206
7.  Kp-eeAlive: timeout=10, max=73
8.  Connection: Keep-Alive
9.  Content-Type: text/html
10.
11. <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
12. <html><head>
13. <title>302 Found</title>
14. </head><body>
15. <h1>Found</h1>
16. <p>The document has moved <a href="http://www.ibm.com/us/">here</a>.</p>
17. </body></html>
```

- ligne 2 : le code *302 Found* indique une redirection. L'adresse vers laquelle le navigateur doit se rediriger est dans le corps du document, ligne 16.

Un deuxième exemple :

```
1.  ...\Chap9\06\bin\Release>ClientWeb http://www.bull.com GET
2.  HTTP/1.1 301 Moved Permanently
3.  Date: Sat, 03 May 2008 14:52:31 GMT
4.  Server: Apache/1.3.33 (Unix) WS_filter/2.1.15 PHP/4.3.4
5.  X-Powered-By: PHP/4.3.4
6.  Location: http://www.bull.com/index.php
7.  Connection: close
8.  Transfer-Encoding: chunked
9.  Content-Type: text/html
10.
11. 0
```

- ligne 2 : le code *301 Moved Permanently* indique une redirection. L'adresse vers laquelle le navigateur doit se rediriger est indiquée ligne 6, dans l'entête HTTP **Location**.

Un troisième exemple :

```
1.  ...\Chap9\06\bin\Release>ClientWeb http://www.gouv.fr GET
2.  HTTP/1.1 302 Moved Temporarily
3.  Server: AkamaiGHost
4.  Content-Length: 0
5.  Location: http://www.premier-ministre.gouv.fr/fr/
6.  Date: Sat, 03 May 2008 14:56:53 GMT
7.  Connection: close
```

- ligne 2 : le code *302 Moved Temporarily* indique une redirection. L'adresse vers laquelle le navigateur doit se rediriger est indiquée ligne 5, dans l'entête HTTP **Location**.

Un quatrième exemple avec un serveur IIS local à la machine :

```
1.  ...\istia\Chap9\06\bin\Release>ClientWeb.exe http://localhost HEAD
2.  HTTP/1.1 302 Object moved
3.  Server: Microsoft-IIS/5.1
4.  Date: Sun, 04 May 2008 10:16:56 GMT
5.  Connection: close
6.  Location: localstart.asp
7.  Content-Length: 121
8.  Content-Type: text/html
9.  Set-Cookie: ASPSESSIONIDQQASDQAB=FDJLADLCOLDHGKGNIPMLHIIA; path=/
10. Cache-control: private
```

- ligne 2 : le code *302 Object moved* indique une redirection. L'adresse vers laquelle le navigateur doit se rediriger est indiquée ligne 5, dans l'entête HTTP **Location**. On notera que contrairement aux exemples précédents, l'adresse de redirection est relative. L'adresse complète est en fait *http://localhost/localstart.asp*.

Nous nous proposons de gérer les redirections lorsque la première ligne des entêtes HTTP contient le mot clé *moved* (insensible à la casse) et que l'adresse de redirection est dans l'entête HTTP **Location**.

Si nous reprenons les trois derniers exemples, nous avons les résultats suivants :

Url : http://www.bull.com

```
1.  ...\Chap9\06B\bin\Release>ClientWebAvecRedirection http://www.bull.com HEAD
2.  HTTP/1.1 301 Moved Permanently
3.  Date: Sun, 04 May 2008 10:22:48 GMT
4.  Server: Apache/1.3.33 (Unix) WS_filter/2.1.15 PHP/4.3.4
5.  X-Powered-By: PHP/4.3.4
6.  Location: http://www.bull.com/index.php
7.  Connection: close
8.  Content-Type: text/html
9.
10.
11. <--Redirection vers l'URL http://www.bull.com/index.php-->
12.
13. HTTP/1.1 200 OK
14. Date: Sun, 04 May 2008 10:22:49 GMT
15. Server: Apache/1.3.33 (Unix) WS_filter/2.1.15 PHP/4.3.4
16. X-Powered-By: PHP/4.3.4
17. Connection: close
18. Content-Type: text/html
```

- ligne 11 : la redirection a lieu vers l'adresse de la ligne 6

Url : http://www.gouv.fr

```
1.  ...\Chap9\06B\bin\Release>ClientWebAvecRedirect
2.  ion http://www.gouv.fr HEAD
3.  HTTP/1.1 302 Moved Temporarily
4.  Server: AkamaiGHost
5.  Content-Length: 0
6.  Location: http://www.premier-ministre.gouv.fr/fr/
7.  Date: Sun, 04 May 2008 10:30:38 GMT
8.  Connection: close
9.
10.
11. <--Redirection vers l'URL http://www.premier-ministre.gouv.fr/fr/-->
12.
13. HTTP/1.1 200 OK
14. Server: Apache
15. X-Powered-By: PHP/4.4.1
16. Last-Modified: Sun, 04 May 2008 10:29:48 GMT
17. Content-Type: text/html
18. Expires: Sun, 04 May 2008 10:40:38 GMT
19. Date: Sun, 04 May 2008 10:30:38 GMT
20. Connection: close
```

- ligne 11 : la redirection a lieu vers l'adresse de la ligne 6

Url : http://localhost

```
1.  ...\Chap9\06B\bin\Release>ClientWebAvecRedirection.exe http://localhost HEAD
2.  HTTP/1.1 302 Object moved
3.  Server: Microsoft-IIS/5.1
4.  Date: Sun, 04 May 2008 10:37:11 GMT
5.  Connection: close
6.  Location: localstart.asp
7.  Content-Length: 121
8.  Content-Type: text/html
9.  Set-Cookie: ASPSESSIONIDQQASDQAB=GDJLADLCJCMPCHFFEJEFPKMK; path=/
10. Cache-control: private
11.
12.
13. <--Redirection vers l'URL http://localhost/localstart.asp-->
14.
15. HTTP/1.1 401 Access Denied
16. Server: Microsoft-IIS/5.1
17. Date: Sun, 04 May 2008 10:37:11 GMT
18. WWW-Authenticate: Negotiate
19. WWW-Authenticate: NTLM
20. WWW-Authenticate: Basic realm="localhost"
21. Connection: close
```

```
22. Content-Length: 4766
23. Content-Type: text/html
```

- ligne 13 : la redirection a lieu vers l'adresse de la ligne 6
- ligne 15 : l'accès à la page *http://localhost/localstart.asp* nous a été refusé.

Le programme gérant la redirection est le suivant :

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.  using System.Text.RegularExpressions;
5.
6.  namespace Chap9 {
7.   class ClientWebAvecRedirection {
8.      static void Main(string[] args) {
9.         // syntaxe
10.        const string syntaxe = "pg URI GET/HEAD";
11.
12.        // nombre d'arguments
13.        if (args.Length != 2) {
14.           Console.WriteLine(syntaxe);
15.           return;
16.        }
17.
18.        // on note l'URI demandée
19.        string stringURI = args[0];
20.        string commande = args[1].ToUpper();
21.
22.        // vérification validité de l'URI
23.        if (!stringURI.StartsWith("http://")) {
24.           Console.WriteLine("Indiquez une Url de la forme http://machine[:port]/document");
25.           return;
26.        }
27.        Uri uri = null;
28.        try {
29.           uri = new Uri(stringURI);
30.        } catch (Exception ex) {
31.           // URI incorrecte
32.           Console.WriteLine("L'erreur suivante s'est produite : {0}", ex.Message);
33.           return;
34.        }
35.        // vérification de la commande
36.        if (commande != "GET" && commande != "HEAD") {
37.           // commande incorrecte
38.           Console.WriteLine("Le second paramètre doit être GET ou HEAD");
39.           return;
40.        }
41.
42.        const int nbRedirsMax = 1;    // pas plus d'une redirection acceptée
43.        int nbRedirs = 0;                  // nombre de redirections en cours
44.
45.        // expression régulière pour trouver une URL de redirection
46.        Regex location = new Regex(@"^Location: (.+?)$");
47.        try {
48.           // on peut avoir plusieurs URL à demander s'il y a des redirections
49.           while (nbRedirs <= nbRedirsMax) {
50.              // gestion redirection
51.              bool redir = false;
52.              bool locationFound = false;
53.              string locationString = null;
54.              // on se connecte au service
55.              using (TcpClient tcpClient = new TcpClient(uri.Host, uri.Port)) {
56.                 using (StreamReader reader = new StreamReader(tcpClient.GetStream())) {
57.                    using (StreamWriter writer = new StreamWriter(tcpClient.GetStream())) {
58.                       // flux de sortie non bufferisé
59.                       writer.AutoFlush = true;
60.                       // on demande l'URL - envoi des entêtes HTTP
61.                       writer.WriteLine(commande + " " + uri.PathAndQuery + " HTTP/1.1");
62.                       writer.WriteLine("Host: " + uri.Host + ":" + uri.Port);
63.                       writer.WriteLine("Connection: close");
64.                       writer.WriteLine();
65.                       // on lit la première ligne de la réponse
66.                       string premièreLigne = reader.ReadLine();
67.                       // écho écran
68.                       Console.WriteLine(premièreLigne);
69.
70.                       // redirection ?
71.                       if (Regex.IsMatch(premièreLigne.ToLower(), @"\s+moved\s*")) {
```

```
72.                         // il y a une redirection
73.                         redir = true;
74.                         nbRedirs++;
75.                      }
76.
77.                      // entêtes HTTP suivants jusqu'à trouver la ligne vide signalant la fin des
   entêtes
78.                      string réponse = null;
79.                      while ((réponse = reader.ReadLine()) != "") {
80.                         // on affiche la réponse
81.                         Console.WriteLine(réponse);
82.                         // s'il y a redirection, on recherche l'entête Location
83.                         if (redir && !locationFound) {
84.                            // on compare la ligne courante à l'expression relationnelle location
85.                            Match résultat = location.Match(réponse);
86.                            if (résultat.Success) {
87.                               // si on a trouvé, on note l'URL de redirection
88.                               locationString = résultat.Groups[1].Value;
89.                               // on note qu'on a trouvé
90.                               locationFound = true;
91.                            }
92.                         }
93.                      }
94.
95.                      // les entêtes HTTP ont été épuisés - on écrit la ligne vide
96.                      Console.WriteLine(réponse);
97.                      // puis on passe au corps du document
98.                      while ((réponse = reader.ReadLine()) != null) {
99.                         Console.WriteLine(réponse);
100.                     }
101.                  }
102.               }
103.            }
104.            // a-t-on fini ?
105.            if (!locationFound || nbRedirs > nbRedirsMax)
106.               break;
107.            // il y a une redirection à opérer - on construit la nouvelle Uri
108.            try {
109.               if (locationString.StartsWith("http")) {
110.                  // adresse http complète
111.                  uri = new Uri(locationString);
112.               } else {
113.                  // adresse http relative à l'uri courante
114.                  uri = new Uri(uri, locationString);
115.               }
116.               // log console
117.               Console.WriteLine("\n<--Redirection vers l'URL {0}-->\n", uri);
118.            } catch (Exception ex) {
119.               // pb avec l'Uri
120.               Console.WriteLine("\n<--L'adresse de redirection {0} n'a pas été comprise : {1} --
   >\n", locationString, ex.Message);
121.            }
122.         }
123.      } catch (Exception e) {
124.         // on affiche l'exception
125.         Console.WriteLine("L'erreur suivante s'est produite : {0}", e.Message);
126.      }
127.   }
128. }
129.}
```

Par rapport à la version précédente, les changements sont les suivants :

- ligne 46 : l'expression régulière pour récupérer l'adresse de redirection dans l'entête HTTP *Location: adresse*.
- ligne 49 : le code qui était exécuté précédemment pour une unique Uri peut l'être maintenant successivement pour plusieurs Uri.
- ligne 66 : on lit la 1ère ligne des entêtes HTTP envoyés par le serveur. C'est elle qui contient le mot clé *moved* si le document demandé a été déplacé.
- lignes 71-75 : on vérifie si la 1ère ligne contient le mot clé *moved*. Si oui, on le note.
- lignes 79-93 : lecture des autres entêtes HTTP jusqu'à rencontrer la ligne vide qui signale leur fin. Si la 1ère ligne annonçait une redirection, on s'attarde alors sur l'entête HTTP *Location: adresse* pour mémoriser l'adresse de redirection dans *locationString*.
- lignes 98-100 : le reste de la réponse du serveur HTTP est affiché à la console.
- lignes 105-106 : l'Uri demandée a été entièrement exploitée et affichée. S'il n'y a pas de redirection à faire ou si le nombre de redirections autorisées est dépassé, on quitte le programme.

- lignes 108-122 : s'il y a redirection, on calcule la nouvelle Uri à demander. Il y a une petite gymnastique à faire selon que l'adresse de redirection trouvée était absolue (ligne 111) ou relative (ligne 114).

11.7 Les classes .NET spécialisées dans un protocole particulier de l'internet

Dans les exemples précédents du client web, le protocole HTTP était géré avec un client TCP. Il nous fallait donc gérer nous-mêmes le protocole de communication particulier utilisé. Nous aurions pu construire de façon analogue, un client SMTP ou POP. Le framework .NET offre des classes spécialisées pour les protocoles HTTP et SMTP. Ces classes connaissent le protocole de communication entre le client et le serveur et évitent au développeur d'avoir à les gérer. Nous les présentons maintenant.

11.7.1 La classeWebClient

Il existe une classe *WebClient* sachant dialoguer avec un serveur web. Considérons l'exemple du client web du paragraphe 11.6.5, page 340, traité ici avec la classe *WebClient*.

```
1.  using System;
2.  using System.IO;
3.  using System.Net;
4.  namespace Chap9 {
5.   public class Program {
6.      public static void Main(string[] args) {
7.         // syntaxe : [prog] Uri
8.         const string syntaxe = "pg URI";
9.
10.        // nombre d'arguments
11.        if (args.Length != 1) {
12.           Console.WriteLine(syntaxe);
13.           return;
14.        }
15.
16.        // on note l'URI demandée
17.        string stringURI = args[0];
18.
19.        // vérification validité de l'URI
20.        if (!stringURI.StartsWith("http://")) {
21.           Console.WriteLine("Indiquez une Url de la forme http://machine[:port]/document");
22.           return;
23.        }
24.        Uri uri = null;
25.        try {
26.           uri = new Uri(stringURI);
27.        } catch (Exception ex) {
28.           // URI incorrecte
29.           Console.WriteLine("L'erreur suivante s'est produite : {0}", ex.Message);
30.           return;
31.        }
32.
33.        try {
34.           // création client web
35.           using (WebClient client = new WebClient()) {
36.              // ajout d'un entête HTTP
37.              client.Headers.Add("user-agent", "st");
38.              using (Stream stream = client.OpenRead(uri)) {
39.                 using (StreamReader reader = new StreamReader(stream)) {
40.                    // affichage réponse du serveur web
41.                    Console.WriteLine(reader.ReadToEnd());
42.                    // affichage entêtes réponse du serveur
43.                    Console.WriteLine("----------------------");
44.                    foreach (string clé in client.ResponseHeaders.Keys) {
45.                       Console.WriteLine("{0}: {1}", clé, client.ResponseHeaders[clé]);
46.                    }
47.                    Console.WriteLine("----------------------");
48.                 }
49.              }
50.           }
51.        } catch (WebException e1) {
52.           Console.WriteLine("L'exception suivante s'est produite : {0}", e1);
53.        } catch (Exception e2) {
54.           Console.WriteLine("L'exception suivante s'est produite : {0}", e2);
55.        }
56.     }
57.  }
58. }
```

- ligne 35 : le client web est créé mais pas encore configuré

- ligne 37 : on ajoute un entête HTTP à la demande HTTP qui va être faite. Nous allons découvrir que d'autres entêtes seront envoyés par défaut.
- ligne 38 : le client web demande l'Uri donnée par l'utilisateur et lit le document envoyé. *[WebClient].OpenRead(Uri)* ouvre la connexion avec *Uri* et lit la réponse. C'est là l'intérêt de la classe. Elle s'occupe du dialogue avec le serveur web. Le résultat de la méthode *OpenRead* est de type *Stream* et représente le document demandé. Les entêtes HTTP envoyés par le serveur et qui précèdent le document dans la réponse n'en font pas partie.
- ligne 39 : on utilise un *StreamReader* et ligne 41, sa méthode *ReadToEnd* pour lire la totalité de la réponse.
- lignes 44-46 : on affiche les entêtes HTTP de la réponse du serveur. *[WebClient].ResponseHeaders* représente une collection valuée dont les clés sont les noms des entêtes HTTP et les valeurs, les chaînes de caractères associées à ces entêtes.
- ligne 51 : les exceptions qui sont levées lors d'un échange client / serveur sont de type *WebException*.

Voyons quelques exemples.

On lance le serveur TCP générique construit au paragraphe 6.4.6, page 158 :

```
1.  ...\Chap9\05\bin\Release>ServeurTcpGenerique.exe 88
2.  Serveur générique lancé sur le port 0.0.0.0:88
```

On lance le client web précédent de la façon suivante :

```
...\Chap9\09\bin\Release>09 http://localhost:88
```

L'Uri demandée est celle du serveur générique. Celui-ci affiche alors les entêtes HTTP que lui a envoyés le client web :

```
1.  Client 127.0.0.1:1415
2.  Tapez vos commandes (bye pour arrêter) :
3.  <-- GET / HTTP/1.1
4.  <-- User-Agent: st
5.  <-- Host: localhost:88
6.  <-- Connection: Keep-Alive
7.  <--
```

On voit ainsi :
- que le client web envoie 3 entêtes HTTP par défaut (lignes 3, 5, 6)
- ligne 4 : l'entête que nous avons généré nous-mêmes (ligne 37 du code)
- que le client web utilise par défaut la méthode GET (ligne 3). Il existe d'autres méthodes parmi lesquelles POST et HEAD.

Maintenant demandons une ressource inexistante :

```
1.  ...\Chap9\09\bin\Release>09 http://istia.univ-angers.fr/inconnu
2.  L'exception suivante s'est produite : System.Net.WebException: The remote server returned an
    error: (404) Not Found.
3.     at System.Net.WebClient.OpenRead(Uri address)
4.     at System.Net.WebClient.OpenRead(String address)
5.     at Chap9.WebClient1.Main(String[] args) in C:\data\2007-2008\c#
    2008\poly\istia\Chap9\09\Program.cs:line 16
```

- ligne 2 : on a eu une exception de type *WebException* parce que le serveur a répondu par le code *404 Not Found* pour indiquer que la ressource demandée n'existait pas.

Enfin terminons en demandant une ressource existante :

```
...\istia\Chap9\09\bin\Release>09 http://istia.univ-angers.fr >istia.univ-angers.txt
```

Le fichier *istia.univ-angers.txt* produit par la commande est le suivant :

```
1.  <!DOCTYPE html
         PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr_FR" lang="fr_FR">
    <head>
     <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    ...
    </html>
2.  ---------------------
3.  Keep-Alive: timeout=15, max=100
4.  Connection: Keep-Alive
5.  Transfer-Encoding: chunked
6.  Content-Type: text/html;charset=iso-8859-1
7.  Date: Sun, 04 May 2008 14:30:53 GMT
```

```
8.  Set-Cookie: fe_typo_user=22eaaf283a; path=/
9.  Server: Apache/1.3.34 (Debian) PHP/4.4.4-8+etch4 mod_jk/1.2.18 mod_perl/1.29
10. X-Powered-By: PHP/4.4.4-8+etch4
11. ----------------------
```

- ligne 1 : le document HTML demandé.
- lignes 3-10 : les entêtes de la réponse HTTP dans un ordre qui n'est pas forcément celui dans lequel ils ont été envoyés.

La classe *WebClient* dispose de méthodes permettant de recevoir un document (méthodes *DownLoad*) ou d'en envoyer (méthodes *UpLoad*) :

`DownLoadData`	pour télécharger une ressource en tant que tableau d'octets (image par exemple)
`DownLoadFile`	pour télécharger une ressource et la sauvegarder dans un fichier local
`DownLoadString`	pour télécharger une ressource et la récupérer en tant que chaîne de caractères (fichier html par exemple)
`OpenWrite`	le pendant d'*OpenRead* mais pour envoyer des données au serveur
`UpLoadData`	le pendant de *DownLoadData* mais vers le serveur
`UpLoadFile`	le pendant de *DownLoadFile* mais vers le serveur
`UpLoadString`	le pendant de *DownLoadString* mais vers le serveur

`UpLoadValues`	pour envoyer au serveur les données d'une commande POST et en récupérer les résultats sous la forme d'un tableau d'octets. La commande POST demande un document tout en transmettant au serveur des informations qui lui sont nécessaires pour déterminer le document réel à envoyer. Ces informations sont envoyées comme document au serveur, d'où le nom *UpLoad* de la méthode. Elles sont envoyées derrière la ligne vide des entêtes HTTP sous la forme *param1=valeur1&param2=valeur2&...* :
	`POST /document HTTP/1.1`
	`...`
	`[ligne vide]`
	`param1=valeur1&param2=valeur2&...`
	Le même document pourrait être demandé avec la méthode GET :
	`GET /document?param1=valeur1&param2=valeur2&...`
	`...`
	`[ligne vide]`
	La différence entre les deux méthodes est que le navigateur affichant l'Uri demandée, affichera */document* dans le cas du POST et */document?param1=valeur1&param2=valeur2&...* dans le cas du GET.

11.7.2 Les classes WebRequest / WebResponse

Parfois la classe *WebClient* n'est pas suffisamment souple pour faire ce que l'on souhaite. Reprenons l'exemple du client web avec redirection étudié au paragraphe 11.6.6, page 342. Il nous faut émettre l'entête HTTP :

```
HEAD /document HTTP/1.1
```

Nous avons vu que les entêtes HTTP émis par défaut par le client web étaient les suivants :

```
1.  <-- GET / HTTP/1.1
2.  <-- Host: machine:port
3.  <-- Connection: Keep-Alive
```

Nous avons vu également qu'il était possible d'ajouter des entêtes HTTP aux précédents avec la collection *[WebClient].Headers*. Seulement la ligne 1 n'est pas un entête appartenant à la collection *Headers* car elle n'a pas la forme *clé: valeur*. Je n'ai pas trouvé comment changer le GET en HEAD dans la ligne 1 en partant de la classe *WebClient* (j'ai peut-être mal cherché ?). Lorsque la classe *WebClient* a atteint ses limites, on peut passer aux classes **WebRequest** / **WebResponse** :

- **WebRequest** : représente la totalité de la demande du client Web.
- **WebResponse** : représente la totalité de la réponse du serveur Web

Nous avons dit que la classe *WebClient* gérait les schémas **http:**, **https:**, **ftp:**, **file:**. Les requêtes et réponses de ces différents protocoles n'ont pas la même forme. Aussi est-il nécessaire de manipuler le type exact de ces éléments plutôt que leur type générique *WebRequest* et *WebResponse*. Aussi utilisera-t-on les classes :

- **HttpWebRequest**, **HttpWebResponse** pour un client HTTP
- **FtpWebRequest**, **FtpWebResponse** pour un client FTP

Nous traitons maintenant avec les classes *HttpWebRequest* et *HttpWebresponse* l'exemple du client web avec redirection étudié au paragraphe 11.6.6, page 342. Le code est le suivant :

```
1.  using System;
2.  using System.IO;
3.  using System.Net.Sockets;
4.  using System.Net;
5.
6.  namespace Chap9 {
7.   class WebRequestResponse {
8.      static void Main(string[] args) {
9.         // syntaxe
10.        const string syntaxe = "pg URI GET/HEAD";
11.
12.        // nombre d'arguments
13.        if (args.Length != 2) {
14.           Console.WriteLine(syntaxe);
15.           return;
16.        }
17.
18.        // on note l'URI demandée
19.        string stringURI = args[0];
20.        string commande = args[1].ToUpper();
21.
22.        // vérification validité de l'URI
23.        Uri uri = null;
24.        try {
25.           uri = new Uri(stringURI);
26.        } catch (Exception ex) {
27.           // URI incorrecte
28.           Console.WriteLine("L'erreur suivante s'est produite : {0}", ex.Message);
29.           return;
30.        }
31.        // vérification de la commande
32.        if (commande != "GET" && commande != "HEAD") {
33.           // commande incorrecte
34.           Console.WriteLine("Le second paramètre doit être GET ou HEAD");
35.           return;
36.        }
37.
38.        try {
39.           // on configure la requête
40.           HttpWebRequest httpWebRequest = WebRequest.Create(uri) as HttpWebRequest;
41.           httpWebRequest.Method = commande;
42.           httpWebRequest.Proxy = null;
43.           // on l'exécute
44.           HttpWebResponse httpWebResponse = httpWebRequest.GetResponse() as HttpWebResponse;
45.           // résultat
46.           Console.WriteLine("----------------------");
47.           Console.WriteLine("Le serveur {0} a répondu : {1} {2}", httpWebResponse.ResponseUri,
    (int)httpWebResponse.StatusCode, httpWebResponse.StatusDescription);
48.           // entêtes HTTP
49.           Console.WriteLine("----------------------");
50.           foreach (string clé in httpWebResponse.Headers.Keys) {
51.              Console.WriteLine("{0}: {1}", clé, httpWebResponse.Headers[clé]);
52.           }
53.           Console.WriteLine("----------------------");
54.           // document
55.           using (Stream stream = httpWebResponse.GetResponseStream()) {
56.              using (StreamReader reader = new StreamReader(stream)) {
57.                 // on affiche la réponse sur la console
58.                 Console.WriteLine(reader.ReadToEnd());
59.              }
60.           }
61.        } catch (WebException e1) {
62.           // on récupère la réponse
63.           HttpWebResponse httpWebResponse = e1.Response as HttpWebResponse;
64.           Console.WriteLine("Le serveur {0} a répondu : {1} {2}", httpWebResponse.ResponseUri,
    (int)httpWebResponse.StatusCode, httpWebResponse.StatusDescription);
65.        } catch (Exception e2) {
66.           // on affiche l'exception
67.           Console.WriteLine("L'erreur suivante s'est produite : {0}", e2.Message);
68.        }
69.     }
70.  }
71. }
```

- ligne 40 : un objet de type *WebRequest* est créé avec la méthode statique *WebRequest.Create(Uri uri)* ou uri est l'uri du document à télécharger. Parce que l'on sait que le protocole de l'Uri est HTTP, le type du résultat est changé en *HttpWebRequest* afin d'avoir accès aux éléments spécifiques du protocole Http.
- ligne 41 : nous fixons la méthode GET / POST / HEAD de la 1ère ligne des entêtes HTTP. Ici ce sera GET ou HEAD.
- ligne 42 : dans un réseau privé d'entreprise, il est fréquent que les machines de l'entreprise soit isolées de l'internet pour des raisons de sécurité. Pour cela, le réseau privé utilise des adresses internet que les routeurs de l'internet ne routent pas. Le réseau privé est relié à l'internet par des machines particulières appelées **proxy** qui sont reliées à la fois au réseau privé de l'entreprise et à l'internet. C'est un exemple de machines à plusieurs adresses IP. Une machine du réseau privé ne peut établir elle-même une connexion avec un serveur de l'internet, un serveur web par exemple. Elle doit demander à une machine proxy de le faire pour elle. Une machine *proxy* peut abriter des serveurs *proxy* pour différents protocoles. On parle de **proxy HTTP** pour désigner le service qui s'occupe de faire les requêtes HTTP pour le compte des machines du réseau privé. Si un tel serveur proxy HTTP existe, il faut l'indiquer dans le champ **[WebRequest].proxy**. On écrira par exemple :

```
[WebRequest].proxy=new WebProxy("pproxy.istia.uang:3128");
```

si le proxy HTTP opère sur le port 3128 de la machine *pproxy.istia.uang*. On met **null** dans le champ *[WebRequest].proxy* si la machine a un accès direct à l'internet et n'a pas à passer par un proxy.

- ligne 44 : la méthode **GetResponse()** demande le document identifié par son Uri et rend un objet *WebRequestResponse* qu'on transforme ici en objet *HttpWebResponse*. Cet objet représente la réponse du serveur à la demande du document.
- ligne 47 :
  - *[HttpWebResponse].ResponseUri* : est l'Uri du serveur ayant envoyé le document. En cas de redirection, celle-ci peut être différente de l'Uri du serveur interrogé initialement. On notera que le code ne gère pas la redirection. Elle est gérée automatiquement par la méthode *GetResponse*. De nouveau, c'est l'avantage des classes de haut niveau vis à vis des classes basiques du protocole Tcp.
  - *[HttpWebResponse].StatusCode*, *[HttpWebResponse].StatusDescription* représentent la 1ère ligne de la réponse, par exemple : *HTTP/1.1 200 OK*. *StatusCode* est 200 et *StatusDescription* est OK.
- ligne 50 : *[HttpWebResponse].Headers* est la collection des entêtes HTTP de la réponse.
- ligne 55 : *[HttpWebResponse].GetResponseStream* : est le flux qui permet d'obtenir le document contenu dans la réponse.
- ligne 61 : il peut se produire une exception de type *WebExcception*
- ligne 63 : *[WebExcception].Response* est la réponse qui a provoqué la levée de l'exception.

Voici un exemple d'exécution :

```
1.  ...\Chap9\09B\bin\Release>09B http://www.gouv.fr HEAD
2.  ---------------------
3.  Le serveur http://www.premier-ministre.gouv.fr/fr/ a répondu : 200 OK
4.  ---------------------
5.  Connection: keep-alive
6.  Content-Type: text/html; charset=iso-8859-1
7.  Date: Mon, 05 May 2008 13:02:29 GMT
8.  Expires: Mon, 05 May 2008 13:07:20 GMT
9.  Last-Modified: Mon, 05 May 2008 12:56:59 GMT
10. Server: Apache
11. X-Powered-By: PHP/4.4.1
12. ---------------------
```

- lignes 1 et 3 : le serveur qui a répondu n'est pas le même que celui qui avait été interrogé. Il y a donc eu redirection.
- lignes 5-11 : les entêtes HTTP envoyés par le serveur

11.7.3 Application : un client proxy d'un serveur web de traduction

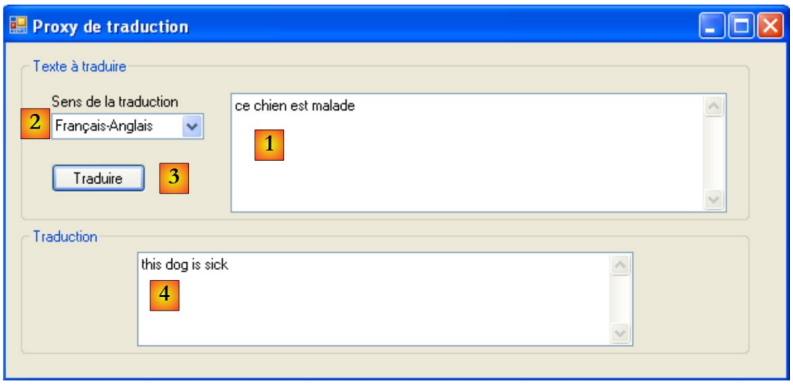
Nous montrons maintenant comment les classes précédentes nous permettent d'exploiter les ressources du web.

11.7.3.1 L'application

Il existe sur le web des sites de traduction. Celui qui sera utilisé ici est le site **http://trans.voila.fr/traduction_voila.php** :

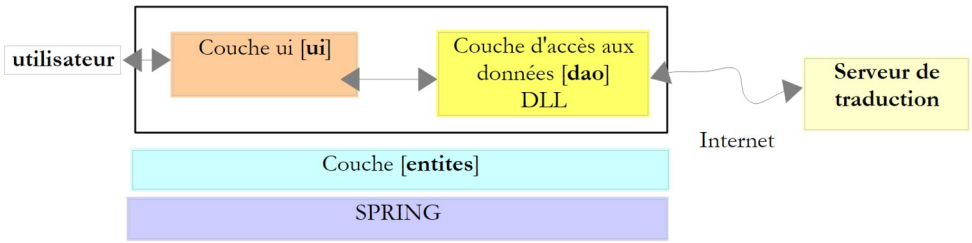
Le texte à traduire est inséré dans [1], le sens de traduction est choisi dans [2]. La traduction est demandée par [3] et obtenue en [4].

Nous allons écrire une application windows cliente de l'application ci-dessus. Elle ne fera rien de plus que l'application du site [trans.voila.fr]. Son interface sera la suivante :



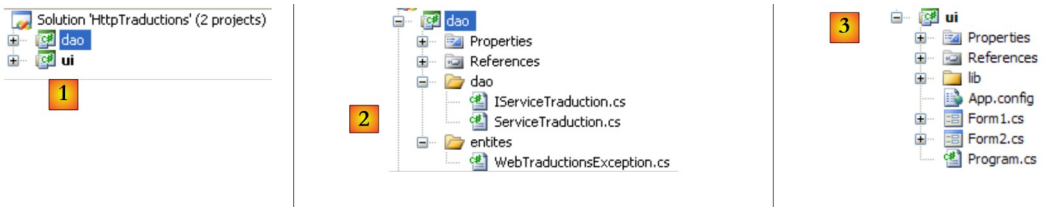
11.7.3.2 L'architecture de l'application

L'application aura l'architecture 2 couches suivante :



11.7.3.3 Le projet Visual studio

Le projet Visual studio sera le suivant :



- en [1], la solution est composée de deux projets,
  - [2] : l'un pour la couche [dao] et les entités utilisées par celle-ci,
  - [3] : l'autre pour l'interface windows

11.7.3.4 Le projet [dao]

Le projet [dao] est formé des éléments suivants :
- **IServiceTraduction.cs** : l'interface présentée à la couche [ui]
- **ServiceTraduction** : l'implémentation de cette interface
- **WebTraductionsException** : une exception spécifique à l'application

L'interface **IServiceTraduction** est la suivante :

```
1.  using System.Collections.Generic;
2.
3.  namespace dao {
4.   public interface IServiceTraduction {
5.      // langues utilisées
6.      IDictionary<string, string> LanguesTraduites { get; }
7.      // traduction
8.      string Traduire(string texte, string deQuoiVersQuoi);
9.   }
10. }
```

- ligne 6 : la propriété **LanguesTraduites** rend le dictionnaire des langues acceptées par le serveur de traduction. Ce dictionnaire a des entrées de la forme ["fe","Français-Anglais"] où la valeur désigne un sens de traduction, ici du Français vers l'Anglais, et la clé "fe" est un code utilisé par le serveur de traduction *trans.voila.fr*.
- ligne 8 : la méthode *Traduire* est la méthode de traduction :
  - *texte* est le texte à traduire
  - *deQuoiVersQuoi* est l'une des clés du dictionnaire des langues traduites
  - la méthode rend la traduction du texte

ServiceTraduction est une classe d'implémentation de l'interface **IServiceTraduction**. Nous la détaillons dans la section qui suit.

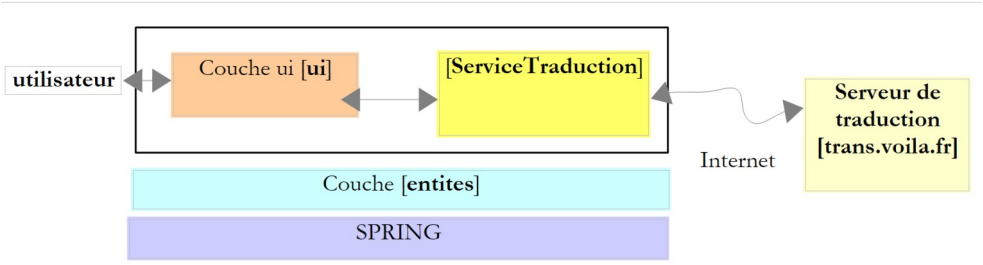
WebTraductionsException est la classe d'exception suivante :

```
1.  using System;
2.
3.  namespace entites {
4.   public class WebTraductionsException : Exception {
5.
6.      // code d'erreur
7.      public int Code { get; set; }
8.
9.      // constructeurs
10.     public WebTraductionsException() {
11.     }
12.     public WebTraductionsException(string message)
13.        : base(message) {
14.     }
15.     public WebTraductionsException(string message, Exception e)
16.        : base(message, e) {
17.     }
18.  }
19. }
```

- ligne 7 : un code d'erreur

11.7.3.5 Le client web [ServiceTraduction]

Revenons sur l'architecture de notre application :



La classe [ServiceTraduction] que nous devons écrire est un client du service web de traduction [trans.voila.fr]. Pour l'écrire, il nous faut comprendre

- ce qu'attend le serveur de traduction de son client
- ce qu'il renvoie en retour à son client

Voyons sur un exemple le dialogue client / serveur qui intervient dans une traduction. Reprenons l'exemple présenté en introduction de l'application :

Le texte à traduire est inséré dans [1], le sens de traduction est choisi dans [2]. La traduction est demandée par [3] et obtenue en [4].

Pour obtenir la traduction [4], le navigateur a envoyé la requête GET suivante (affichée dans son champ d'adresse) :

```
http://trans.voila.fr/traduction_voila.php?isText=1&translationDirection=fe&stext=ce+chien+est+malade
```

Elle est plutôt simple à comprendre :

- *http://trans.voila.fr/traduction_voila.php* est l'Url du service de traduction
- *isText=1* semble vouloir dire qu'on a affaire à du texte
- *translationDirection* désigne le sens de la traduction, ici *Français-Anglais*
- *stext* est le texte à traduire sous une forme qu'on appelle Url encodée. En effet, certains caractères ne peuvent apparaître dans une Url. C'est le cas par exemple de l'espace qui a été ici encodé par un +. Le framework .Net offre la méthode statique *System.Web.HttpUtility.UrlEncode* pour faire ce travail d'encodage.

On en conclut que pour interroger le serveur de traduction, notre classe [ServiceTraduction] pourra utiliser la chaîne

```
"http://trans.voila.fr/traduction_voila.php?isText=1&translationDirection={0}&stext={1}"
```

où les marqueurs {0} et {1} seront remplacés respectivement par le sens de traduction et le texte à traduire.

Comment connait-on les sens de traduction acceptés par le serveur ? Dans la copie d'écran ci-dessus, les langues traduites sont dans la liste déroulante. Si dans le navigateur on regarde (Affichage / source) le code Html de la page, on trouce ceci pour la liste déroulante :

```
1.  <select name="translationDirection" class="champs">
2.   <option selected value='fe'>Fran&ccedil;ais vers Anglais
3.   <option  value='ef'>Anglais vers Fran&ccedil;ais
4.   <option  value='fg'>Fran&ccedil;ais vers Allemand
5.   <option  value='gf'>Allemand vers Fran&ccedil;ais
6.   <option  value='fs'>Fran&ccedil;ais vers Espagnol
7.   <option  value='sf'>Espagnol vers Fran&ccedil;ais
8.   <option  value='fr'>Fran&ccedil;ais vers Russe
9.   <option  value='rf'>Russe vers Fran&ccedil;ais
10.  <option  value='es'>Anglais vers Espagnol
```

```
11.  <option  value='se'>Espagnol vers Anglais
12.  <option  value='eg'>Anglais vers Allemand
13.  <option  value='ge'>Allemand vers Anglais
14.  <option  value='ep'>Anglais vers Portugais
15.  <option  value='pe'>Portugais vers Anglais
16.  <option  value='ie'>Italien vers Anglais
17.  <option  value='gs'>Allemand vers Espagnol
18.  <option  value='sg'>Espagnol vers Allemand
19. </select>
```

Ce n'est pas un code Html très propre, dans la mesure où chaque balise <option> devrait être normalement fermée par une balise </option>. Ceci dit, les attributs **value** nous donnent la liste des codes de traduction qui doivent être envoyés au serveur. Dans le dictionnaire **LanguesTraduites** de l'interface **IServiceTraduction**, les clés seront les attributs **value** ci-dessus et les valeurs, les textes affichés par la liste déroulante.

Maintenant regardons (Affichage / source) où se trouve dans la page Html la traduction renvoyée par le serveur de traduction :

```
...
<strong>Texte traduit : </strong><div class="txtTrad">this dog is sick</div>
...
```

La traduction se trouve au beau milieu de la page Html renvoyée. Comment la retrouver ? On peut utiliser une expression régulière avec la séquence * ... * car la balise * * n'est présente qu'à cet endroit de la page Html. L'expression régulière C# permettant de récupérer le texte traduit est la suivante :

```
@"<div class=""txtTrad"">(.*?)</div>"
```

Nous avons désormais les éléments pour écrire la classe d'implémentation *ServiceTraduction* de l'interface *IServiceTraduction* :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.IO;
4.  using System.Net;
5.  using System.Text.RegularExpressions;
6.  using System.Web;
7.  using entites;
8.
9.  namespace dao {
10.  public class ServiceTraduction : IServiceTraduction {
11.     // propriétés automatiques de configuration du service
12.     public IDictionary<string, string> LanguesTraduites { get; set; }
13.     public string UrlServeurTraduction { get; set; }
14.     public string ProxyHttp { get; set; }
15.     public String RegexTraduction { get; set; }
16.
17.     // traduction
18.     public string Traduire(string texte, string deQuoiVersQuoi) {
19.        // la traduction demandée est-elle possible ?
20.        if (!LanguesTraduites.ContainsKey(deQuoiVersQuoi)) {
21.           throw new WebTraductionsException(String.Format("Le sens de traduction [{0}] n'est pas
    reconnu")) { Code = 10 };
22.        }
23.        // texte à traduire
24.        string texteATraduire = HttpUtility.UrlEncode(texte);
25.        // uri à demander
26.        string uri = string.Format(UrlServeurTraduction, deQuoiVersQuoi, texteATraduire);
27.        // expression régulière pour retrouver la traduction dans la réponse
28.        Regex patternTraduction = new Regex(RegexTraduction);
29.        // exception
30.        WebTraductionsException exception = null;
31.        // traduction
32.        string traduction = null;
33.        try {
34.           // on configure la requête
35.           HttpWebRequest httpWebRequest = WebRequest.Create(uri) as HttpWebRequest;
36.           httpWebRequest.Method = "GET";
37.           httpWebRequest.Proxy = ProxyHttp == null ? null : new WebProxy(ProxyHttp); ;
38.           // on l'exécute
39.           HttpWebResponse httpWebResponse = httpWebRequest.GetResponse() as HttpWebResponse;
40.           // document
41.           using (Stream stream = httpWebResponse.GetResponseStream()) {
42.              using (StreamReader reader = new StreamReader(stream)) {
43.                 bool traductionTrouvée = false;
44.                 string ligne = null;
45.                 while (!traductionTrouvée && (ligne = reader.ReadLine()) != null) {
46.                    // recherche traduction dans la ligne courante
47.                    MatchCollection résultats = patternTraduction.Matches(ligne);
```

```
48.                     // traduction trouvée ?
49.                     if (résultats.Count != 0) {
50.                        traduction = résultats[0].Groups[1].Value.Trim();
51.                        traductionTrouvée = true;
52.                     }
53.                  }
54.                  // traduction trouvée ?
55.                  if (!traductionTrouvée) {
56.                     exception = new WebTraductionsException("Le serveur n'a pas renvoyé de
    réponse") { Code = 12 };
57.                  }
58.               }
59.            }
60.         } catch (Exception e) {
61.            exception = new WebTraductionsException("Erreur rencontrée lors de la traduction", e)
    { Code = 11 };
62.         }
63.         // exception ?
64.         if (exception != null) {
65.            throw exception;
66.         } else {
67.            return traduction;
68.         }
69.      }
70.   }
71. }
```

- ligne 12 : la propriété *LanguesTraduites* de l'interface *IServiceTraduction* - initialisée de l'extérieur
- ligne 13 : la propriété *UrlServeurTraduction* est l'Url à demander au serveur de traduction : *http://trans.voila.fr/traduction_voila.php?isText=1&translationDirection={0}&stext={1}* où le marqueur {0} devra être remplacé par le sens de traduction et le marqueur {1} par le texte à traduire - initialisée de l'extérieur
- ligne 14 : la propriété *ProxyHttp* est l'éventuel proxy Http à utiliser, par exemple : *pproxy.istia.uang:3128* - initialisée de l'extérieur
- ligne 15 : la propriété *RegexTraduction* est l'expression régulière permettant de récupérer la traduction dans le flux Html renvoyé par le serveur de traduction, par exemple *@"<div class=""txtTrad"">(.*?) "* - initialisée de l'extérieur
- ces quatres propriétés seront, dans notre application, initialisées par Spring.

- lignes 20-22 : on vérifie que le sens de traduction demandé existe bien dans le dictionnaire des langues traduites. Si ce n'est pas le cas, une exception est lancée.
- ligne 24 : le texte à traduire est encodé pour pouvoir faire partie d'une Url
- ligne 26 : l'Uri du service de traduction est construite. Si la propriété *UrlServeurTraduction* est la chaîne *http://trans.voila.fr/traduction_voila.php?isText=1&translationDirection={0}&stext={1}*, le marqueur {0} est remplacé par le sens de traduction et le marqueur {1} par le texte à traduire.
- ligne 28 : le modèle de recherche de la traduction dans la réponse html du serveur de traduction est construit.
- lignes 33, 60 : l'opération d'interrogation du serveur de traduction se passe dans un try / catch
- ligne 35 : l'objet *HttpWebRequest* qui va être utilisé pour interroger le serveur de traduction est construit avec l'Uri du document demandé.
- ligne 36 : la méthode d'interrogation est GET. On pourrait se passer de cette instruction, car GET est probablement la méthode par défaut de l'objet *HttpWebRequest*.
- ligne 37 : on fixe la propriété *Proxy* de l'objet *HttpWebRequest*.
- ligne 39 : la requête au serveur de traduction est faite et on récupère sa réponse qui est de type *HttpWebResponse*.
- lignes 41-42 : on utilise un *StreamReader* pour lire chaque ligne de la réponse html du serveur.
- lignes 45-53 : dans chaque ligne de la réponse, on cherche la traduction. Lorsqu'on l'a trouvée, on arrête de lire la réponse Html et on ferme tous les flux qu'on a ouverts.
- lignes 55-57 : si on n'a pas trouvé de traduction dans la réponse html, on prépare une exception de type *WebTraductionsException* pour le dire.
- lignes 60-62 : si une exception s'est produite lors de l'écange client / serveur, on l'encapsule dans une exception de type *WebTraductionsException* pour le dire.
- lignes 64-68 : si une exception a été enregistrée, elle est lancée, sinon la traduction trouvée est rendue.

Notre exemple suppose que le proxy Http ne nécessite pas d'authentification. Si ce n'était pas le cas, on écrirait quelque chose comme :

```
httpWebRequest.Proxy = ProxyHttp == null ? null : new WebProxy(ProxyHttp); ;
httpWebRequest.Proxy.Credentials=new NetworkCredential("login","password");
```

Nous avons utilisé ici *WebRequest* / *WebResponse* plutôt que *WebClient* parce que nous n'avons pas à exploiter la totalité de la réponse Html du serveur de traduction. Une fois la traduction trouvée dans cette réponse, nous n'avons plus besoin du reste des lignes de la réponse. La classe *WebClient* ne permet pas de faire cela.

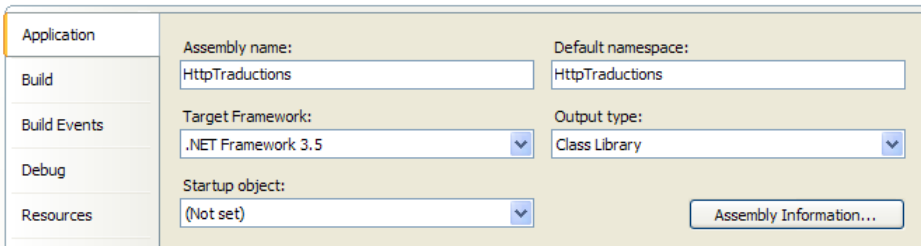
Voici un programme de test de la classe *ServiceTraduction* :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using dao;
4.  using entites;
5.
6.  namespace ui {
7.   class Program {
8.      static void Main(string[] args) {
9.         try {
10.           // création service traduction
11.           ServiceTraduction serviceTraduction = new ServiceTraduction();
12.           // expression régulière pour trouver la traduction
13.           serviceTraduction.RegexTraduction = @"<div class=""txtTrad"">(.*?)</div>";
14.           // url serveur de traduction
15.           serviceTraduction.UrlServeurTraduction = "http://trans.voila.fr/traduction_voila.php?
    isText=1&translationDirection={0}&stext={1}";
16.           // dictionnaire des langues traduites
17.           Dictionary<string, string> languesTraduites = new Dictionary<string, string>();
18.           languesTraduites["fe"]= "Français-Anglais";
19.           languesTraduites["fs"]= "Français-Espagnol";
20.           languesTraduites["ef"]= "Anglais-Français";
21.           serviceTraduction.LanguesTraduites = languesTraduites;
22.           // proxy
23.           //serviceTraduction.ProxyHttp = "pproxy.istia.uang:3128";
24.           // traduction
25.           string texte = "ce chien est perdu";
26.           string deQuoiVersQuoi = "fe";
27.           Console.WriteLine("Traduction [{0}] de [{1}] : [{2}]", languesTraduites[deQuoiVersQuoi],
    texte, serviceTraduction.Traduire(texte, deQuoiVersQuoi));
28.           texte = "l'été sera chaud";
29.           deQuoiVersQuoi = "fs";
30.           Console.WriteLine("Traduction [{0}] de [{1}] : [{2}]", languesTraduites[deQuoiVersQuoi],
    texte, serviceTraduction.Traduire(texte, deQuoiVersQuoi));
31.           texte = "my tailor is rich";
32.           deQuoiVersQuoi = "ef";
33.           Console.WriteLine("Traduction [{0}] de [{1}] : [{2}]", languesTraduites[deQuoiVersQuoi],
    texte, serviceTraduction.Traduire(texte, deQuoiVersQuoi));
34.           texte = "xx";
35.           deQuoiVersQuoi = "ef";
36.           Console.WriteLine("Traduction [{0}] de [{1}] : [{2}]", languesTraduites[deQuoiVersQuoi],
    texte, serviceTraduction.Traduire(texte, deQuoiVersQuoi));
37.        } catch (WebTraductionsException e) {
38.           // erreur
39.           Console.WriteLine("L'erreur suivante de code {1} s'est produite : {0}", e.Message,
    e.Code);
40.        }
41.     }
42.  }
43. }
```

Les résultats obtenus sont les suivants :

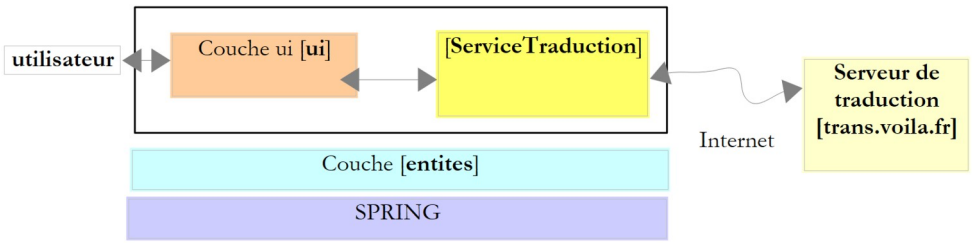
```
1.  Traduction [Français-Anglais] de [ce chien est perdu] : [this dog is lost]
2.  Traduction [Français-Espagnol] de [l'été sera chaud] : [el verano será caliente]
3.  Traduction [Anglais-Français] de [my tailor is rich] : [mon tailleur est riche]
4.  Traduction [Anglais-Français] de [xx] : [xx]
```

Le projet [dao] de la solution est compilée en une DLL *HttpTraductions.dll* :

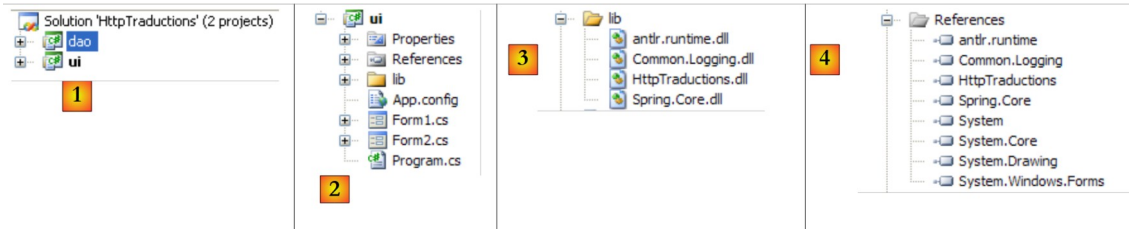


11.7.3.6 L'interface graphique de l'application

Revenons sur l'architecture de notre application :



Nous écrivons maintenant la couche [ui]. Celle-ci fait l'objet du projet [ui] de la solution en construction :



Le dossier [lib] [3] contient certaines des DLL référencées par le projet [4] :

- celles nécessaires à Spring : *Spring.Core*, *Common.Logging, antlr.runtime*
- celle de la couche [dao] : *HttpTraductions*

Le fichier [App.config] contient la configuration Spring :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.   </configSections>
10.
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <description>Traductions sur le web</description>
17.        <!-- le service de traduction -->
18.        <object name="ServiceTraduction" type="dao.ServiceTraduction, HttpTraductions">
19.           <property name="UrlServeurTraduction" value="http://trans.voila.fr/traduction_voila.php?
    isText=1&amp;translationDirection={0}&amp;stext={1}"/>
20.           <!--
21.           <property name="ProxyHttp" value="pproxy.istia.uang:3128"/>
22.           -->
23.           <property name="RegexTraduction" value="&lt;div class=&quot;txtTrad&quot;&gt;
    (.*?)&lt;/div&gt;"/>
24.           <property name="LanguesTraduites">
25.              <dictionary key-type="string" value-type="string">
26.                 <entry key="fe" value="Français-Anglais"/>
27.                 <entry key="ef" value="Anglais-Français"/>
28. ...
29.                 <entry key="ei" value="Anglais-Italien"/>
30.                 <entry key="ie" value="Italien-Anglais"/>
31.              </dictionary>
32.           </property>
33.        </object>
34.     </objects>
35.  </spring>
36. </configuration>
```

- ligne 15 : les objets à instancier par Spring. Il n'y en aura qu'un, celui de la ligne 18 qui instancie le service de traduction avec la classe *ServiceTraduction* trouvée dans la DLL *HttpTraductions*.
- ligne 19 : la propriété *UrlServeurTraduction* de la classe *ServiceTraduction*. Il y a une difficulté avec le caractère & de l'Url. ce caractère a une signification dans un fichier Xml. Il doit donc être protégé. C'est le cas également d'autres caractères que nous allons rencontrer dans la suite du fichier. Ils doivent être remplacés par une séquence [*&code;*] : & par [*&*], < par [*<*] > par [*>*], " par [*"*].
- ligne 21 : la propriété *ProxyHttp* de la classe *ServiceTraduction*. Une propriété non initialisée reste à *null*. Ne pas définir cette propriété revient à dire qu'il n'y a pas de proxy Http.
- ligne 23 : la propriété *RegexTraduction* de la classe *ServiceTraduction*. Dans l'expression régulière, il a fallu remplacer les caractères [< > "] par leurs équivalents protégés.
- lignes 24-33 : la propriété *LanguesTraduites* de la classe *ServiceTraduction*.

Le programme [Program.cs] est exécuté au démarrage de l'application. Son code est le suivant :

```
1.  using System;
2.  using System.Text;
3.  using System.Windows.Forms;
4.  using dao;
5.  using Spring.Context;
6.  using Spring.Context.Support;
7.
8.  namespace ui {
9.   static class Program {
10.     /// <summary>
11.     /// The main entry point for the application.
12.     /// </summary>
13.     [STAThread]
14.     static void Main() {
15.        Application.EnableVisualStyles();
16.        Application.SetCompatibleTextRenderingDefault(false);
17.
18.        // --------------- Code développeur
19.        // instanciation service de traduction
20.        IApplicationContext ctx = null;
21.        Exception ex = null;
22.        ServiceTraduction serviceTraduction = null;
23.        try {
24.           // contexte Spring
25.           ctx = ContextRegistry.GetContext();
26.           // on demande une référence sur le service de traduction
27.           serviceTraduction = ctx.GetObject("ServiceTraduction") as ServiceTraduction;
28.        } catch (Exception e1) {
29.           // mémorisation exception
30.           ex = e1;
31.        }
32.        // formulaire à afficher
33.        Form form = null;
34.        // y-a-t-il eu une exception ?
35.        if (ex != null) {
36.           // oui - on crée le message d'erreur à afficher
37.           StringBuilder msgErreur = new StringBuilder(String.Format("Chaîne des exceptions : {0}
    {1}", "".PadLeft(40, '-'), Environment.NewLine));
38.           Exception e = ex;
39.           while (e != null) {
40.              msgErreur.Append(String.Format("{0}: {1}{2}", e.GetType().FullName, e.Message,
    Environment.NewLine));
41.              msgErreur.Append(String.Format("{0}{1}", "".PadLeft(40, '-'), Environment.NewLine));
42.              e = e.InnerException;
43.           }
44.           // création fenêtre d'erreur à laquelle on passe le message d'erreur à afficher
45.           Form2 form2 = new Form2();
46.           form2.MsgErreur = msgErreur.ToString();
47.           // ce sera la fenêtre à afficher
48.           form = form2;
49.        } else {
50.           // tout s'est bien passé
51.           // création interface graphique [Form1] à laquelle on passe la référence sur le sevice
    de traduction
52.           Form1 form1 = new Form1();
53.           form1.ServiceTraduction = serviceTraduction;
54.           // ce sera la fenêtre à afficher
55.           form = form1;
56.        }
57.        // affichage fenêtre
58.        Application.Run(form);
59.     }
```

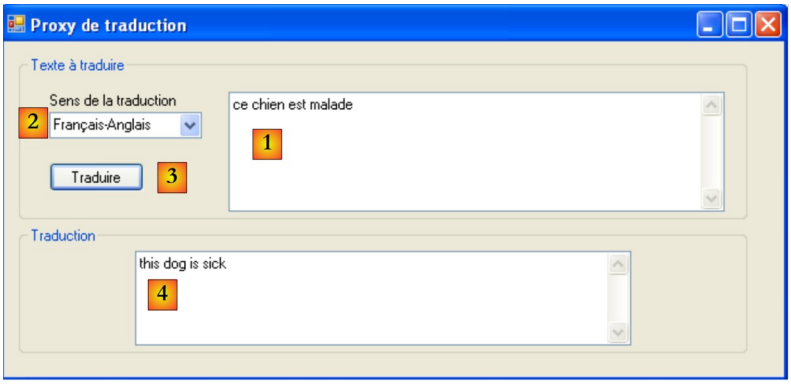
```
60.  }
61. }
```

Ce code a déjà été utilisé dans l'application Impôts version 6, au paragraphe 7.6.2, page 205.

- le service de traduction est créé ligne 27 par Spring. Si cette création s'est bien passée, le formulaire [Form1] sera affiché (lignes 52-55), sinon c'est le formulaire d'erreur [Form2] qui le sera (lignes 36-48).

Le formulaire [Form2] est celui utilisé dans l'application Impôts version 6 et a été expliqué au paragraphe 7.6.4, page 208.

Le formulaire [Form1] est le suivant :



n°	type	nom	rôle
1	TextBox	textBoxTexteATraduire	boîte de saisie du texte à traduire MultiLine=true
2	ComboBox	comboBoxLangues	la liste des sens de traduction
3	Button	buttonTraduire	pour demander la traduction du texte [1] dans le sens [2]
4	TextBox	textBoxTraduction	la traduction du texte [1]

Le code du formulaire [Form1] est le suivant :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Windows.Forms;
5.  using dao;
6.
7.  namespace ui {
8.   public partial class Form1 : Form {
9.      // service de traduction
10.     public ServiceTraduction ServiceTraduction { get; set; }
11.     // dictionnaire des langues
12.     Dictionary<string, string> languesInversées = new Dictionary<string, string>();
13.
14.     // constructeur
15.     public Form1() {
16.        InitializeComponent();
17.     }
18.
19.     // chargement initial du formulaire
20.     private void Form1_Load(object sender, EventArgs e) {
21.        // construction du dictionnaire inversé des langues
22.        foreach (string code in ServiceTraduction.LanguesTraduites.Keys) {
23.           // langues
24.           string langues = ServiceTraduction.LanguesTraduites[code];
25.           // ajout (langues, code) au dictionnaire inversé
26.           languesInversées[langues] = code;
27.        }
28.        // remplissage combo dans l'ordre alphabétique des langues
29.        string[] languesCombo = languesInversées.Keys.ToArray();
30.        Array.Sort<string>(languesCombo);
31.        foreach (string langue in languesCombo) {
32.           comboBoxLangues.Items.Add(langue);
33.        }
34.        // sélection 1ère langue
35.        if (comboBoxLangues.Items.Count != 0) {
```

```
36.           comboBoxLangues.SelectedIndex = 0;
37.        }
38.     }
39.
40.     private void buttonTraduire_Click(object sender, EventArgs e) {
41.        // qq chose a traduire ?
42.        string texte = textBoxTexteATraduire.Text.Trim();
43.        if (texte == "") return;
44.        // traduction
45.        try {
46.           textBoxTraduction.Text = ServiceTraduction.Traduire(texte,
    languesInversées[comboBoxLangues.SelectedItem.ToString()]);
47.        } catch (Exception ex) {
48.           textBoxTraduction.Text = ex.Message;
49.        }
50.     }
51.  }
52. }
```

- ligne 10 : une référence sur le service de traduction. Cette propriété publique a été initialisée par [Program.cs], ligne 53. Lorsque les méthodes *Form1_Load* (ligne 20) ou *buttonTraduire_Click* (ligne 40) s'exécutent, ce champ est donc déjà initialisé.
- ligne 12 : le dictionnaire des langues traduites avec des entrées de type ["Français-Anglais","fe"], c.a.d. l'inverse du dictionnaire *LanguesTraduites* rendu par le service de traduction.
- ligne 20 : la méthode *Form1_Load* s'exécute au chargement du formulaire.
- lignes 22-27 : on utilise le dictionnaire *serviceTraduction.LanguesTraduites* ["fe","Français-Anglais"] pour construire le dictionnaire *languesInversées* ["Français-Anglais", "fe"].
- ligne 29 : *languesCombo* est le tableau des clés du dictionnaire *languesInversées*, c.a.d. un tableau d'éléments ["Français-Anglais"]
- ligne 30 : ce tableau est trié afin de présenter dans le combo les sens de traduction par ordre alphabétique
- lignes 31-33 : le combo des langues est rempli.
- ligne 40 : la méthode exécutée lorsque l'utilisateur clique sur le bouton [Traduire]
- ligne 46 : il suffit d'appeler la méthode *serviceTraduction.Traduire* pour demander la traduction. Le 1er paramètre est le texte à traduire, le second le code du sens de traduction. Ce code est trouvé dans le dictionnaire *languesInversées* à partir de l'élément sélectionné dans le combo des langues.
- ligne 48 : s'il y a une exception, elle est affichée à la place de la traduction.

11.7.3.7 Conclusion

Cette application a montré que les clients web du framework .NET nous permettaient d'exploiter les ressources du web. La technique est à chaque fois similaire :

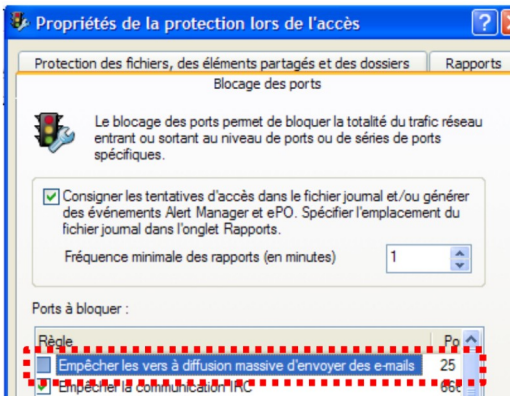
- déterminer l'Uri à interroger. Cette Uri est la plupart du temps paramétrée.
- l'interroger
- trouver dans la réponse du serveur ce qu'on cherche grâce à des expressions régulières

Cette technique est aléatoire. En effet au fil du temps, l'Uri interrogée ou l'expression régulière permettant de trouver le résultat attendu peuvent changer. On a donc intérêt à placer ces deux informations dans un fichier de configuration. Mais cela peut se révéler insuffisant. Nous verrons dans le chapitre suivant qu'il existe des ressources plus stables sur le web : les services web.

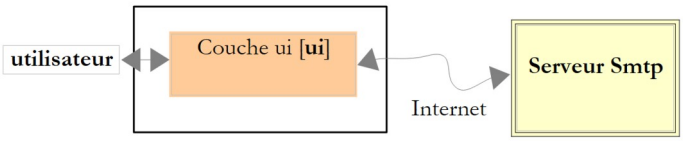
11.7.4 Un client SMTP (Simple Mail Tranport Protocol) avec la classe SmtpClient

Un client SMTP est un client d'un serveur SMTP, serveur d'envoi de courrier. La class .NET **SmtpClient** encapsule totalement les besoins d'un tel client. Le développeur n'a pas à connaître les détails du protocole SMTP. Nous connaissons ce dernier. Il a été présenté au paragraphe 11.4.3, page 323.

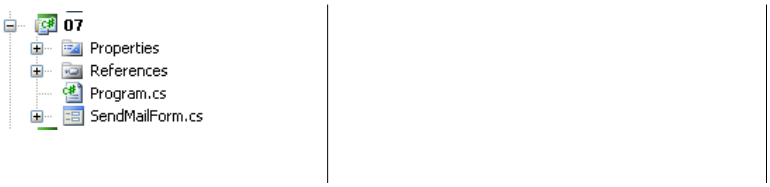
Nous présentons la classe **SmtpClient** dans le cadre d'une application windows basique qui permet d'envoyer des courriers électroniques avec pièces jointes. L'application va se connecter au port 25 d'un serveur SMTP. On rappelle que sur la plupart des PC windows, les pare-feu ou autres antivirus bloquent les connexions vers le port 25. Il est alors nécessaire de désactiver cette protection pour tester l'application :



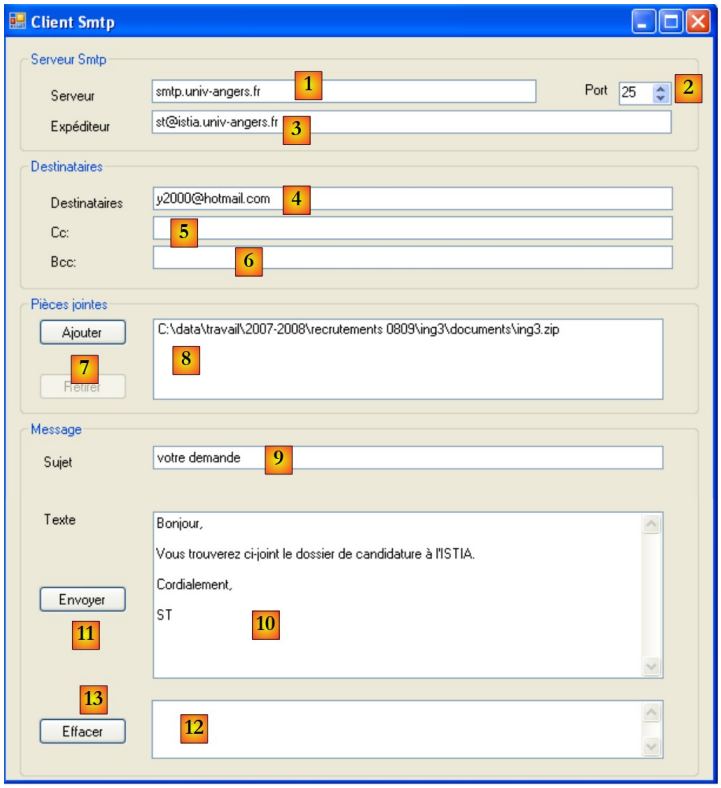
Le client Smtp aura une architecture mono-couche :



Le projet Visual studio est le suivant :



L'interface graphique [SendMailForm.cs] de l'application est la suivante :



n°	type	nom	rôle
1	TextBox	textBoxServeur	nom du serveur SMTP auquel se connecter
2	NumericUpDown	numericUpDownPort	le port sur lequel se connecter
3	TextBox	textBoxExpediteur	adresse de l'expéditeur du message
4	TextBox	textBoxTo	adresses des destinataires sous la forme : adresse1,adresse2, ...
5	TextBox	textBoxCc	adresses des destinataires en copie (CC=Carbon Copy) sous la forme : adresse1,adresse2, ...
6	TextBox	textBoxBcc	adresses des destinataires en copie aveugle (BCC=Blind Carbon Copy) sous la forme : adresse1,adresse2, ... Toutes les adresses de ces trois champs de saisie recevront le même message avec les mêmes attachements. Les personnes destinataires du message pourront connaître les adresses qui étaient dans les champs 4 et 5 mais pas celles du champ 6. Le Bcc est donc une façon de mettre quelqu'un en copie sans que les autres destinataires du message le sachent.
7	Button	buttonAjouter	pour ajouter une pièce jointe au courrier
8	ListBox	listBoxPiecesJointes	liste des pièces à joindre au courrier
9	TextBox	textBoxSujet	sujet du courrier
10	TextBox	textBoxMessage	le texte du message. *MultiLine=true*
11	Button	buttonEnvoyer	pour envoyer le message et les éventuelles pièces jointes
12	TextBox	textBoxRésultat	affiche un résumé du message envoyé ou bien un message d'erreur si un problème a été rencontré
13	Button	buttonEffacer	pour effacer [12]
	OpenfileDialog	openFileDialog1	contrôle non visuel qui permet le choix d'une pièce jointe dans le système de fichiers local

Dans l'exemple précédent, le résumé affiché en [12] est le suivant :

```
Envoi réussi...
Sujet : votre demande
Destinataires : y2000@hotmail.com
Cc :
Bcc :
Pièces jointes :
C:\data\travail\2007-2008\recrutements 0809\ing3\documents\ing3.zip
Texte : Bonjour,
```

```
Vous trouverez ci-joint le dossier de candidature à l'ISTIA.

Cordialement,

ST
```

Le code du formulaire [SendMailForm.cs] est le suivant :

```
1.  using System;
2.  using System.Windows.Forms;
3.  using System.Net.Mail;
4.  using System.Text.RegularExpressions;
5.  using System.Text;
6.
7.  namespace Chap9 {
8.   public partial class SendMailForm : Form {
9.      public SendMailForm() {
10.        InitializeComponent();
11.     }
12.
13.     // ajout d'une pièce jointe
14.     private void buttonAjouter_Click(object sender, EventArgs e) {
15.        // on paramètre la boîte de dialogue openfileDialog1
16.        openFileDialog1.InitialDirectory = Application.ExecutablePath;
17.        openFileDialog1.Filter = "Tous les fichiers (*.*)|*.*";
18.        openFileDialog1.FilterIndex = 0;
19.        openFileDialog1.FileName = "";
20.        // on affiche la boîte de dialogue et on récupère son résultat
21.        if (openFileDialog1.ShowDialog() == DialogResult.OK) {
22.           // on récupère le nom du fichier
23.           listBoxPiecesJointes.Items.Add(openFileDialog1.FileName);
24.        }
25.     }
26.
27.     private void textBoxServeur_TextChanged(object sender, EventArgs e) {
28.        setStatutEnvoyer();
29.     }
30.
31.     private void setStatutEnvoyer() {
32.        buttonEnvoyer.Enabled = textBoxServeur.Text.Trim() != "" && textBoxTo.Text.Trim() != "" &&
    textBoxSujet.Text.Trim() != "";
33.     }
34.
35.     // retirer une pièce jointe
36.     private void buttonRetirer_Click(object sender, EventArgs e) {
37.        // pièce jointe sélectionnée ?
38.        if (listBoxPiecesJointes.SelectedIndex != -1) {
39.           // on la retire
40.           listBoxPiecesJointes.Items.RemoveAt(listBoxPiecesJointes.SelectedIndex);
41.           // on met à jour le bouton Retirer
42.           buttonRetirer.Enabled = listBoxPiecesJointes.Items.Count != 0;
43.        }
44.     }
45.
46.     private void listBoxPiecesJointes_SelectedIndexChanged(object sender, EventArgs e) {
47.        // pièce jointe sélectionnée ?
48.        if (listBoxPiecesJointes.SelectedIndex != -1) {
49.           // on met à jour le bouton Retirer
50.           buttonRetirer.Enabled = true;
51.        }
52.     }
53.
54.     // envoi du message avec ses attachements
55.     private void buttonEnvoyer_Click(object sender, EventArgs e) {
56. ....
57.     }
58.
59.     private void textBoxTo_TextChanged(object sender, EventArgs e) {
60.        setStatutEnvoyer();
61.     }
62.
63.     private void textBoxSujet_TextChanged(object sender, EventArgs e) {
64.        setStatutEnvoyer();
65.     }
66.
67.     private void buttonEffacer_Click(object sender, EventArgs e) {
68.        textBoxResultat.Text = "";
69.     }
70.  }
71. }
```

Nous ne commenterons pas ce code qui ne présente pas de nouveautés. Pour comprendre la méthode *buttonAjouter_Click* de la ligne 14, le lecteur est invité à relire le paragraphe 7.5.1, de la page 197.

La méthode *buttonEnvoyer_Click* de la ligne 55, qui envoie le courrier est la suivante :

```
1.  private void buttonEnvoyer_Click(object sender, EventArgs e) {
2.           try {
3.              // sablier
4.              Cursor = Cursors.WaitCursor;
5.              // le client Smtp
6.              SmtpClient smtpClient = new SmtpClient(textBoxServeur.Text.Trim(),
    (int)numericUpDownPort.Value);
7.              // le message
8.              MailMessage message = new MailMessage();
9.              // expéditeur
10.             message.Sender = new MailAddress(textBoxExpéditeur.Text.Trim());
11.             message.From = message.Sender;
12.             // destinataires
13.             Regex marqueur = new Regex("\\s*,\\s*");
14.             string[] destinataires = marqueur.Split(textBoxTo.Text.Trim());
15.             foreach (string destinataire in destinataires) {
16.                if (destinataire.Trim() != "") {
17.                   message.To.Add(new MailAddress(destinataire));
18.                }
19.             }
20.             // CC
21.             string[] copies = marqueur.Split(textBoxCc.Text.Trim());
22.             foreach (string copie in copies) {
23.                if (copie.Trim() != "") {
24.                   message.CC.Add(new MailAddress(copie));
25.                }
26.             }
27.             // BCC
28.             string[] blindCopies = marqueur.Split(textBoxBcc.Text.Trim());
29.             foreach (string blindCopie in blindCopies) {
30.                if (blindCopie.Trim() != "") {
31.                   message.Bcc.Add(new MailAddress(blindCopie));
32.                }
33.             }
34.             // sujet
35.             message.Subject = textBoxSujet.Text.Trim();
36.             // texte du message
37.             message.Body = textBoxMessage.Text;
38.             // attachements
39.             foreach (string attachement in listBoxPiecesJointes.Items) {
40.                message.Attachments.Add(new Attachment(attachement));
41.             }
42.             // envoi du message
43.             smtpClient.Send(message);
44.             // Ok - on affiche un résumé
45.             StringBuilder msg = new StringBuilder(String.Format("Envoi réussi...{0}",
    Environment.NewLine));
46.             msg.Append(String.Format("Sujet : {0}{1}", textBoxSujet.Text.Trim(),
    Environment.NewLine));
47.             textBoxSujet.Clear();
48.             msg.Append(String.Format("Destinataires : {0}{1}", textBoxTo.Text.Trim(),
    Environment.NewLine));
49.             textBoxTo.Clear();
50.             msg.Append(String.Format("Cc : {0}{1}", textBoxCc.Text.Trim(), Environment.NewLine));
51.             textBoxCc.Clear();
52.             msg.Append(String.Format("Bcc : {0}{1}", textBoxBcc.Text.Trim(), Environment.NewLine));
53.             textBoxBcc.Clear();
54.             msg.Append(String.Format("Pièces jointes :{0}", Environment.NewLine));
55.             foreach (string attachement in listBoxPiecesJointes.Items) {
56.                msg.Append(String.Format("{0}{1}", attachement, Environment.NewLine));
57.             }
58.             msg.Append(String.Format("Texte : {0}{1}", textBoxMessage.Text, Environment.NewLine));
59.             listBoxPiecesJointes.Items.Clear();
60.             textBoxResultat.Text = msg.ToString();
61.          } catch (Exception ex) {
62.             // on affiche l'erreur
63.             textBoxResultat.Text = String.Format("L'erreur suivante s'est produite {0}", ex);
64.          }
65.          // curseur normal
66.          Cursor = Cursors.Arrow;
67.       }
```

- ligne 6 : le client Smtp est créé. Il a besoin de deux paramètres : le nom du serveur SMTP et le port sur lequel celui-ci opère
- ligne 8 : un message de type **MailMessage** est créé. C'est lui qui va encapsuler la totalité du message à envoyer.
- ligne 10 : l'adresse électronique **Sender** de l'expéditeur est renseignée. Une adresse électronique est une instance de type **MailAddress** construite à patir d'une chaîne de caractères "xx@yy.zz". Il faut que cette chaîne ait la forme attendue pour une adresse électronique, sinon une exception est lancée. Dans ce cas, elle sera afichée dans le champ *textBoxResultat* (ligne 63) sous une forme peu conviviale.
- lignes 13-19 : les adresses électroniques des destinataires sont placées dans la liste **To** du messsage. On récupère ces adresses dans le champ *textBoxTo*. L'expression régulière de la ligne 13 permet de récupérer les différentes adresses qui sont séparées par une virgule.
- lignes 21-26 : on répète le même processus pour initialiser le champ **CC** du messsage avec les adresses en copie du champ *textBoxCc*.
- lignes 28-33 : on répète le même processus pour initialiser le champ **Bcc** du messsage avec les adresses en copie aveugle du champ *textBoxBcc*.
- ligne 35 : le champ **Subject** du messsage est initialisé avec le sujet du champ *textBoxSujet*.
- ligne 37 : le champ **Body** du messsage est initialisé avec le texte du message *textBoxMessage*.
- lignes 39-41 : les pièces jointes sont attachées au message. Chaque pièce jointe est ajoutée sous forme d'un objet **Attachment** au champ **Attachments** du message. Un objet **Attachment** est instancié à partir du chemin complet de la pièce à attacher dans le système de fichiers local.
- ligne 43 : le message est envoyé à l'aide de la méthode **Send** du client Smtp.
- lignes 45-60 : écriture du résumé de l'envoi dans le champ *textBoxResultat* et réinitialisation du formulaire.
- ligne 63 : affichage d'une éventuelle erreur

11.8 Un client Tcp générique asynchrone

11.8.1 Présentation

Dans tous les exemples de ce chapitre, la communication client / serveur se faisait en mode bloquant qu'on appelle également mode synchrone :

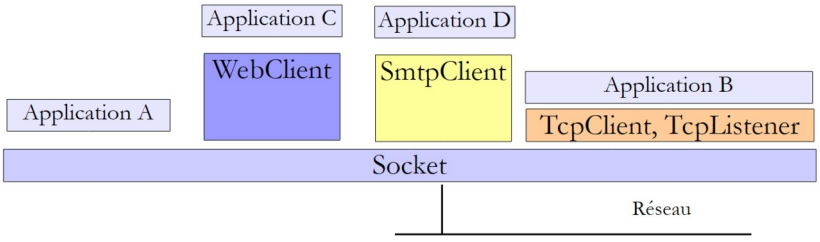
- lorsqu'un client se connecte à un serveur, il attend la réponse du serveur à cette demande avant de continuer.
- lorsqu'un client lit une ligne de texte envoyée par le serveur, il est bloqué tant que le serveur n'a pas envoyé celle-ci.
- côté serveur, les threads de service qui assurent le service au client fonctionnent de la même façon que ci-dessus.

Dans les interfaces graphiques, il est souvent nécessaire de ne pas bloquer l'utilisateur sur des opérations longues. Le cas souvent cité est celui du téléchargement d'un gros fichier. Pendant ce téléchargement, il faut laisser l'utilisateur libre de continuer à interagir avec l'interface graphique.

Nous nous proposons ici de réécrire le client Tcp générique du paragraphe 11.6.3, page 333 en y apportant les changements suivants :

- l'interface sera graphique
- l'outil de communication avec le serveur sera un objet **Socket**
- le mode de communication sera asynchrone :
    - le client initiera une connexion au serveur mais ne restera pas bloqué à attendre qu'elle soit établie
    - le client initiera un envoi au serveur mais ne restera pas bloqué à attendre qu'il soit terminé
    - le client initiera la réception de données provenant du serveur mais ne restera pas bloqué à attendre la fin de celle-ci.

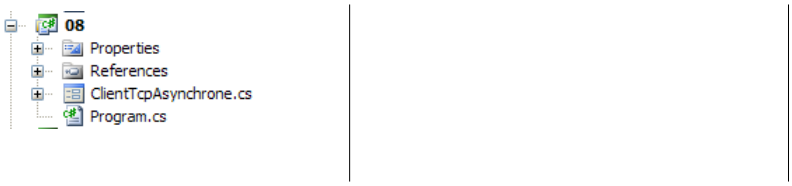
Rappelons à quel niveau se situe l'objet **Socket** dans la communication client / serveur Tcp :



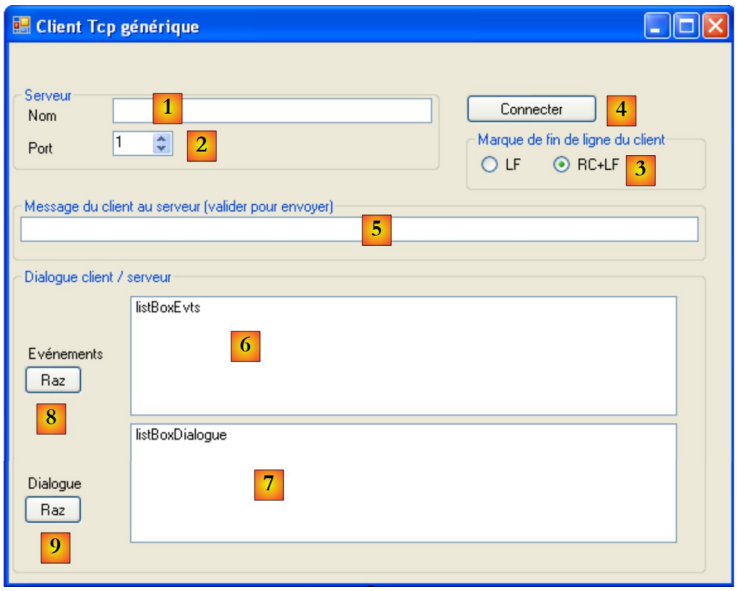
La classe **Socket** est celle qui opère le plus près du réseau. Elle permet de gérer finement la connexion réseau. Le terme *socket* désigne une prise de courant. Le terme a été étendu pour désigner une prise de réseau logicielle. Dans une communication TCP-IP entre deux machines A et B, ce sont deux *sockets* qui communiquent entre-eux. Une application peut travailler directement avec les *sockets*. C'est le cas de l'application A ci-dessus. Un socket peut être un socket *client* ou *serveur*.

11.8.2 L'interface graphique du client Tcp asynchrone

L'application Visual studio est la suivante :



[ClientTcpAsynchrone.cs] est l'interface graphique. Celle-ci est la suivante :



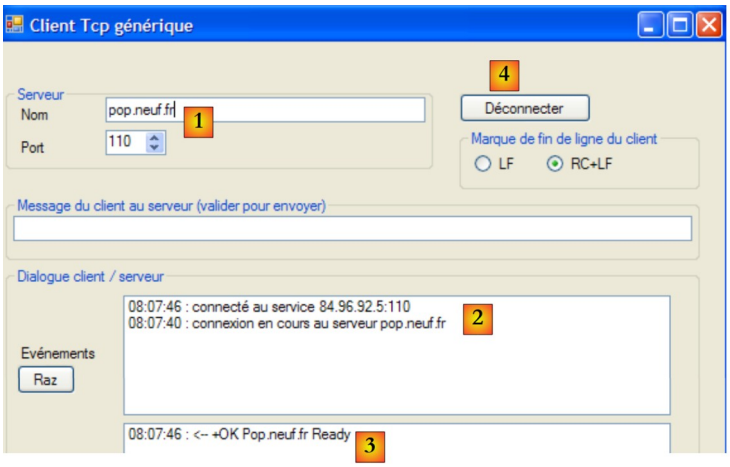
n°	type	nom	rôle
1	TextBox	textBoxNomServeur	nom du serveur Tcp auquel se connecter
2	NumericUpDown	numericUpDownPortServeur	le port sur lequel se connecter
3	RadioButton	radioButtonLF radioButtonRCLF	pour indiquer la marque de fin de ligne que le client doit utiliser : LF "\n" ou RCLF "\r\n"
4	Button	buttonConnexion	pour se connecter sur le port [2] du serveur [1]. Le bouton a le libellé [Connecter] lorsque le client n'est pas connecté à un serveur, [Déconnecter] lorsqu'il est connecté.
5	TextBox	textBoxMsgToServeur	message à envoyer au serveur une fois la connexion faite. Lorsque l'utilisateur tape sur la touche [Entrée], le message est envoyé avec la marque de fin de ligne choisie en [3]
6	ListBox	listBoxEvts	liste dans laquelle sont affichés les principaux événements de la liaison client / serveur : connexion, déconnexion, fermeture de flux, erreurs de communication
7	ListBox	listBoxDialogue	liste dans laquelle sont affichés les messages du dialogue client / serveur
8	Button	buttonRazEvts	pour effacer la liste [6]
4	Button	buttonRazDialogue	pour effacer la liste [7]

Les principes de fonctionnent de cette interface sont les suivants :

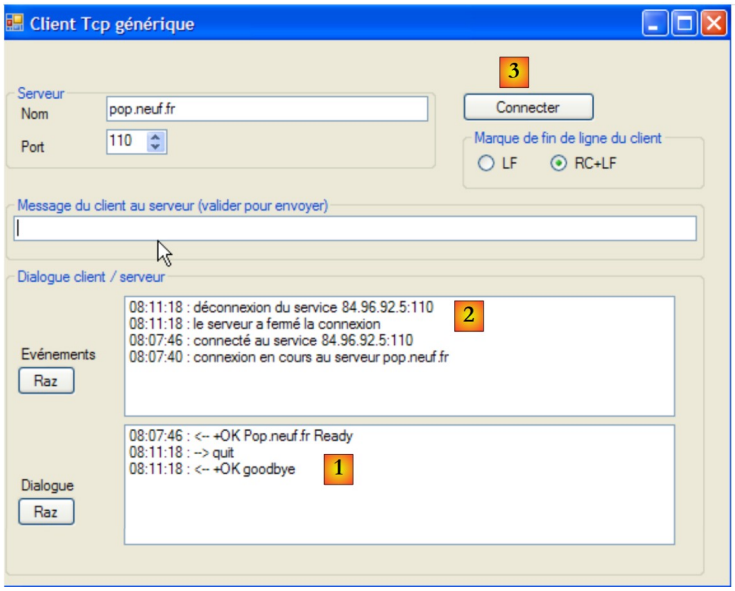
- l'utilisateur connecte son client Tcp graphique à un service Tcp grâce à [1, 2, 3, 4].

- un thread asynchrone accepte en continu toutes les données envoyées par le serveur Tcp et les affiche dans la liste [7]. Ce thread est dissocié des autres activités de l'interface.
- l'utilisateur peut envoyer à son rythme des messages au serveur grâce à [5]. Chaque message est envoyé par un thread asynchrone. A la différence du thread de réception qui ne s'arrête jamais, le thread d'émission est lui terminé dès que le message a été envoyé. Un nouveau thread asynchrone sera utilisé pour le message suivant.
- la communication client / serveur se termine lorsque l'un des partenaires clôt la connexion. L'utilisateur peut prendre cette initiative avec le bouton [4] qui une fois la connexion établie a le libellé [Déconnecter].

Voici une copie d'écran d'une exécution :



- en [1] : connexion à un service POP
- en [2] : affichage des événements ayant eu lieu lors de la connexion
- en [3] : le message envoyé par le serveur POP à l'issue de la connexion
- en [4] : le bouton [Connecter] est devenu le bouton [Déconnecter]



- en [1], on a envoyé la commande *quit* au serveur POP. Le serveur a répondu *+OK goodbye* et a fermé la connexion
- en [2], cette fermeture côté serveur a été détectée. Le client a alors fermé la connexion de son côté.
- en [3], le bouton [Déconnecter] est redevenu un bouton [Connecter]

11.8.3 Connexion asynchrone au serveur

L'appui sur le bouton [Connecter] provoque l'exécution de la méthode suivante :

```
1.        private void buttonConnexion_Click(object sender, EventArgs e) {
2.           // connexion ou déconnexion ?
3.           if (buttonConnexion.Text == "Déconnecter")
4.              déconnexion();
5.           else
6.              connexion();
7.  }
```

- ligne 3 : le bouton peut avoir le libellé [Connecter] ou [Déconnecter].

La méthode de connexion est la suivante :

```
1.  using System.Net.Sockets;
2.  ...
3.
4.  namespace Chap9 {
5.   public partial class ClientTcp : Form {
6.      const int tailleBuffer = 1024;
7.      private Socket client = null;
8.      private byte[] data = new byte[tailleBuffer];
9.      private string réponse = null;
10.     private string finLigne = "\r\n";
11.
12.     // délégués
13.     public delegate void writeLog(string log);
14.
15.     public ClientTcp() {
16.        InitializeComponent();
17.     }
18. ..................................
19.  private void connexion() {
20.        // vérifications données
21.        string nomServeur = textBoxNomServeur.Text.Trim();
22.        if (nomServeur == "") {
23.           logEvent("indiquez le nom du serveur");
24.           return;
25.        }
26.        // suivi
27.        logEvent(String.Format("connexion en cours au serveur {0}", nomServeur));
28.        try {
29.           // création socket
30.           client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
31.           // connexion asynchrone
32.           client.BeginConnect(Dns.GetHostEntry(nomServeur).AddressList[0],
    (int)numericUpDownPortServeur.Value, connecté, client);
33.
34.        } catch (Exception ex) {
35.           logEvent(String.Format("erreur de connexion : {0}", ex.Message));
36.           return;
37.        }
38.     }
39.
40.     // la connexion a eu lieu
41.     private void connecté(IAsyncResult résultat) {
42.        // on récupère le socket du client
43.        Socket client = résultat.AsyncState as Socket;
44.  ...
45.     }
46.
47.
48.     // suivi du processus
49.     private void logEvent(string msg) {
50. ....
51.     }
52.  }
53. }
```

- ligne 1 : la classe **Socket** fait partie de l'espace de noms *System.Net.Sockets*.

Un certain nombre de données doivent être partagées entre plusieurs méthodes du formulaire. Ce sont les suivantes :

- ligne 7 : **client** est le socket de communication avec le serveur
- lignes 6 et 8 : le client va recevoir ses messages dans un tableau d'octets **data**.
- ligne 9 : **réponse** est la réponse envoyée par le serveur.
- ligne 10 : **finLigne** est la marque de fin de ligne utilisée par le client Tcp - est initialisée par défaut à RCLF mais peut être modifiée par l'utilisateur avec les boutons radio [3].

La procédure *connexion* de la ligne 19 réalise la connexion au serveur Tcp :

- lignes 21-25 : on vérifie que le nom du serveur est non vide. Si ce n'est pas le cas, l'événement est logué dans *listBoxEvts* par la méthode **logEvent** de la ligne 49.
- ligne 27 : on signale que la connexion va avoir lieu
- ligne 30 : on crée l'objet **Socket** nécessaire à la communication Tcp-Ip. Le constructeur admet trois paramètres :
  - **AddressFamily** *addressFamily* : la famille des adresses IP du client et du serveur, ici des adresses IPv4 (*AddressFamily.InterNetwork*)
  - **SocketType** *socketType* : le type du socket. Le type *SocketType.Stream* est adapté aux connexions Tcp-Ip
  - **ProtocolType** *protocolType* : le type de protocole internet utilisé, ici le protocole Tcp
- ligne 32 : la connexion est faite de façon asynchrone. La connexion est lancée mais l'exécution continue sans en attendre la fin. La méthode *[Socket].BeginConnect* admet quatre paramètres :
  - **IPAddress** *ipAddress* : l'adresse Ip de la machine sur laquelle s'exécute le service auquel il faut se connecter
  - **Int32** *port* : le port du service
  - **AsyncCallBack** *asyncCallBack* : *AsyncCallBack* est un type **delegate** :

```
public void AsyncCallBack(IAsyncResult ar);
```

    La méthode *asyncCallBack* passée en 3ième paramètre de la méthode *BeginConnect* doit être une méthode acceptant un type *IAsyncCallBack* et ne rendant aucun résultat. C'est la méthode qui sera appelée lorsque la connexion aura été faite. Nous passons ici comme 3ième paramètre, la méthode *connecté* de la ligne 41.
  - **Object** *state* : un objet à passer à la méthode *asyncCallBack*. Cette méthode reçoit (cf délégué ci-dessus) un paramètre **ar** de type **IAsyncResult**. L'objet *state* pourra être récupéré dans **ar.AsyncState** (ligne 43). Nous passons ici comme 4ième paramètre le socket du client.
- ligne 38 : la méthode est terminée. L'utilisateur peut de nouveau interagir avec l'interface graphique. La connexion se passe en tâche de fond, en parallèle de la gestion des événements de l'interface graphique. Toujours en parallèle, la méthode *connecté* de la ligne 41 va être appelée à la fin de la connexion, que celle-ci se termine bien ou mal.

Le code de la méthode *connecté* est le suivant :

```
1.  // la connexion a eu lieu
2.        private void connecté(IAsyncResult résultat) {
3.           // on récupère la socket du client
4.           Socket client = résultat.AsyncState as Socket;
5.           try {
6.              // on termine l'opération asynchrone
7.              client.EndConnect(résultat);
8.              // suivi
9.              logEvent(String.Format("connecté au service {0}", client.RemoteEndPoint));
10.             // formulaire
11.             buttonConnexion.Text = "Déconnecter";
12.             // lecture asynchrone des données en provenance du serveur
13.             réponse = "";
14.             client.BeginReceive(data, 0, tailleBuffer, SocketFlags.None, lecture, client);
15.          } catch (SocketException e) {
16.             logEvent(String.Format("erreur de connexion : {0}", e.Message));
17.             return;
18.          }
19. }
20.
21.       // réception de données
22.       private void lecture(IAsyncResult résultat) {
23.          // on récupère la socket du client
24.          Socket client = résultat.AsyncState as Socket;
25. ...
26.       }
27.
```

- ligne 4 : le socket du client est récupéré dans le paramètre *résultat* reçu par la méthode. On rappelle que cet objet est celui passé comme 4ième paramètre de la méthode *BeginConnect*.
- ligne 7 : la tentative de connexion est terminée par la méthode *EndConnect* à qui on doit passer le paramètre *résultat* reçu par la méthode.
- ligne 9 : l'événement est logué dans la liste des événements
- ligne 11 : le bouton [Connecter] devient un bouton [Déconnecter] pour que l'utilisateur puisse demander la déconnexion.
- ligne 13 : la réponse du serveur est initialisée. Elle va être mise à jour par des appels répétés à la méthode asynchrone *BeginReceive*.
- ligne 14 : 1er appel à la méthode asynchrone *BeginReceive*. Celle-ci est appelée avec les paramètres suivants :
  - **byte[]** *buffer* : le tampon dans lequel placer les données qui vont être reçues - ici le buffer est *data*
  - **int** *offset* : à partir de quelle position du tampon placer les données les données qui vont être reçues - ici l'offset est 0, c.a.d. que les données sont placées dès le 1er octet du tampon.

- **int** *size* : la taille en octets du tampon - ici la taille est *tailleBuffer*.
- **SocketFlags** *socketFlags* : configuration du socket - ici aucune configuration
- **AsyncCallBack** *asyncCallBack* : la méthode à rappeler lorsque la réception sera terminée. Ce sera la cas soit parce que le buffer a reçu des données soit parce que la connexion a été fermée. Ici, la méthode de rappel est la méthode *lecture* de la ligne 22.
- **Object** *state* : l'objet à passer à la méthode de rappel *asyncCallBack*. Ici, on passe de nouveau le socket du client.

On notera que tout ceci se passe sans action de l'utilisateur, autre que la demande initiale de connexion avec le bouton [Connecter]. A la fin de la méthode *connecté*, une autre méthode est exécutée en tâche de fond : la méthode *lecture* que nous examinons maintenant.

```
1.  // réception de données
2.        private void lecture(IAsyncResult résultat) {
3.           // on récupère la socket du client
4.           Socket client = résultat.AsyncState as Socket;
5.           int nbOctetsReçus = 0;
6.           bool erreur = false;
7.           try {
8.              // nbre d'octets reçus
9.              nbOctetsReçus = client.EndReceive(résultat);
10.             if (nbOctetsReçus == 0) {
11.                // le serveur ne répond plus
12.                logEvent("le serveur a fermé la connexion");
13.             }
14.          } catch (Exception e) {
15.             // on a eu un pb de réception
16.             logEvent(String.Format("erreur de réception : {0}", e.Message));
17.             erreur = true;
18.          }
19.          // terminé ?
20.          if (nbOctetsReçus == 0 || erreur) {
21.             // on déconnecte au besoin le client
22.             déconnexion();
23.             // on affiche la fin de la réponse
24.             afficherRéponseServeur(réponse, true);
25.             // fin lecture
26.             return;
27.          }
28.          // on récupère les données reçues
29.          string données = Encoding.UTF8.GetString(data, 0, nbOctetsReçus);
30.          // on les ajoute aux données déjà recues
31.          réponse += données;
32.          // on affiche la réponse
33.          afficherRéponseServeur(réponse, false);
34.          // on continue à lire
35.          client.BeginReceive(data, 0, tailleBuffer, SocketFlags.None, lecture, client);
36.       }
```

- ligne 2 : la méthode *lecture* se déclenche en tâche de fond lorsque le buffer *data* a reçu des données ou que la connexion a été fermée par le serveur.
- ligne 9 : la demande asynchrone de lecture est terminée par *EndReceive*. Là encore cette méthode doit être appelée avec le paramètre reçu par la fonction de rappel. La méthode *EndReceive* rend le nombre d'octets reçus dans le buffer de lecture.
- ligne 10 : si le nombre d'octets est nul c'est que la connexion a été fermée par le serveur.
- ligne 12 : on note l'événement dans la liste des événements
- ligne 14 : on traite une éventuelle exception
- lignes 16-17 : on note l'événement dans la liste des événements et on note l'erreur
- ligne 20 : on regarde si on doit fermer la connexion
- ligne 22 : on ferme la connexion côté client avec une méthode *déconnexion* que nous verrons ultérieurement.
- ligne 24 : la réponse du serveur, c.a.d. la variable globale *réponse* est affichée dans la liste de dialogue *listBoxDialogue* au moyen d'une méthode privée *afficherRéponseServeur*.
- ligne 26 : fin de la méthode asynchrone *lecture*
- ligne 29 : les octets reçus sont mis dans une chaîne de caractères au format UTF8.
- ligne 31 : ils sont ajoutés à la réponse en cours de construction
- ligne 33 : la réponse est affichée dans la liste *listBoxDialogue*.
- ligne 35 : on se remet à attendre des données en provenance du serveur

En définitive, la méthode asynchrone *lecture* ne s'arrête jamais. De façon continue, elle lit les données en provenance du serveur et les fait afficher dans la liste *listBoxDialogue*. Elle ne s'arrête que lorsque la connexion est fermée soit par le serveur soit par l'utilisateur lui-même.

11.8.4 Déconnexion du serveur

L'appui sur le bouton [Déconnecter] provoque l'exécution de la méthode suivante :

```
8.        private void buttonConnexion_Click(object sender, EventArgs e) {
9.           // connexion ou déconnexion ?
10.          if (buttonConnexion.Text == "Déconnecter")
11.             déconnexion();
12.          else
13.             connexion();
14. }
```

- ligne 3 : le bouton peut avoir le libellé [Connecter] ou [Déconnecter].

La méthode *déconnexion* assure la déconnexion du client :

```
1.  private void déconnexion() {
2.           // fermeture socket
3.           if (client != null && client.Connected) {
4.              try {
5.                 // suivi
6.                 logEvent(String.Format("déconnexion du service {0}", client.RemoteEndPoint));
7.                 // déconnexion
8.                 client.Shutdown(SocketShutdown.Both);
9.                 client.Close();
10.                // formulaire
11.                buttonConnexion.Text = "Connecter";
12.             } catch (Exception ex) {
13.                // suivi
14.                logEvent(String.Format("erreur de lors de la déconnexion : {0}", ex.Message));
15.             }
16.          }
17.       }
```

- ligne 3 : si le client existe et est connecté
- ligne 6 : on annonce la déconnexion dans *listBoxEvts*. La propriété *client.RemoteEndPoint* donne le couple (Adresse Ip, port) de l'autre extrémité de la connexion, c.a.d ici du serveur.
- ligne 8 : le flux de données du socket est fermé avec la méthode *ShutDown*. Le flux de données d'un socket est bidirectionnel : le socket émet et reçoit des données. Le paramètre de la méthode *ShutDown* peut être : *ShutDown.Receive* pour fermer le flux de réception, *Shutdonw.Send* pour fermer le flux d'émission ou *ShutDown.Both* pour fermer les deux flux.
- ligne 9 : on libère les ressources associées au socket
- ligne 11 : le bouton [Déconnecter] devient le bouton [Connecter]
- lignes 12-15 : gestion d'une éventuelle exception

11.8.5 Envoi asynchrone de données au serveur

Lorsque l'utilisateur valide le message du champ *textBoxMsgToServeur*, la méthode suivante est exécutée :

```
1.        private void textBoxMsgToServeur_KeyPress(object sender, KeyPressEventArgs e) {
2.           // touche [Entrée] ?
3.           if (e.KeyChar == 13 && client.Connected) {
4.              envoyerMessage();
5.           }
6.  }
```

- lignes 3-5 : si l'utilisateur a enfoncé la touche [Entrée] et si le socket du client est connecté, alors le message du champ *textBoxMsgToServeur* est envoyé avec la méthode *envoyerMessage*.

La méthode *envoyerMessage* est la suivante :

```
1.        private void envoyerMessage() {
2.           // envoyer un message de façon asynchrone
3.           // le message
4.           byte[] message = Encoding.UTF8.GetBytes(textBoxMsgToServeur.Text.Trim() + finLigne);
5.           // il est envoyé
6.           client.BeginSend(message, 0, message.Length, SocketFlags.None, écriture, client);
7.           // dialogue
8.           logDialogue("--> " + textBoxMsgToServeur.Text.Trim());
9.           // raz message
10.          textBoxMsgToServeur.Clear();
11. }
```

- ligne 4 : on ajoute au message la marque de fin de ligne du client et on le met dans le tableau d'octets *message*.
- ligne 6 : une émission asynchrone est commencée avec la méthode *BeginSend*. Les paramètres de *BeginSend* sont identiques à ceux de la méthode *BeginReceive*. A la fin de l'opération d'émission asynchrone du message la méthode *écriture* sera appelée.
- ligne 8 : le message envoyé est ajouté à la liste *listBoxDialogue* afin d'avoir un suivi du dialogue client / serveur
- ligne 10 : le message envoyé est effacé de l'interface graphique

La méthode de rappel *écriture* est la suivante :

```
1.        private void écriture(IAsyncResult résultat) {
2.           // résultat de l'émission d'un message
3.           Socket client = résultat.AsyncState as Socket;
4.           try {
5.              client.EndSend(résultat);
6.           } catch (Exception e) {
7.              // on a eu un pb d'émission
8.              logEvent(String.Format("erreur d'émission : {0}", e.Message));
9.           }
10. }
```

- ligne 4 : la méthode de rappel *écriture* reçoit un paramètre résultat de type *IAsyncResult*.
- ligne 3 : dans le paramètre *résultat*, on récupère le socket du client. Ce socket était le 5ième paramètre de la méthode *BeginSend*.
- ligne 5 : on termine l'opération asynchrone d'émission.

On n'attend pas la fin de l'émission d'un message pour redonner la main à l'utilisateur. Celui-ci peut ainsi émettre un second message alors que l'émission du premier n'est pas terminé.

11.8.6 Affichage des événements et du dialogue client / serveur

Les événements sont affichés par la méthode *logEvents* :

```
1.        // suivi du processus
2.        private void logEvent(string msg) {
3.           listBoxEvts.Invoke(new writeLog(logEventCallBack), msg);
4.        }
5.
6.        private void logEventCallBack(string msg) {
7.           // affichage message
8.           msg = msg.Replace(finLigne, " ");
9.           listBoxEvts.Items.Insert(0, String.Format("{0:hh:mm:ss} : {1}", DateTime.Now, msg));
10. }
```

- ligne 2 : la méthode *logEvents* reçoit en paramètre le message à ajouter dans la liste *listBoxEvts*.
- ligne 3 : on ne peut utiliser directement le composant *listBoxEvents*. En effet, la méthode *logEvents* est appelée par deux types de threads :
  - le thread principal propriétaire de l'interface graphique, par exemple lorsqu'il signale qu'une tentative de connexion est en cours
  - un thread secondaire assurant une opération asynchrone. Ce type de thread n'est pas propriétaire des composants et son accès à un composant C doit être contrôlé par une opération *C.Invoke*. Cette opération indique au contrôle C qu'un thread veut faire une opération sur lui. La méthode *Invoke* admet deux paramètres :
    - une fonction de rappel de type *delegate*. Cette fonction de rappel sera exécutée par le thread propriétaire de l'interface graphique et non par le thread exécutant la méthode *C.Invoke*.
    - un objet qui sera passé à la fonction de rappel.

    Ici le premier paramètre passé à la méthode *Invoke* est une instance du délégué suivant :

    ```
        public delegate void writeLog(string log);
    ```

    Le délégué *writeLog* a un paramètre de type *string* et ne rend aucun résultat. Le paramètre sera le message à enegistrer dans *listBoxEvts*.
    Ligne 3, le premier paramètre passé à la méthode *Invoke* est la méthode *logEventCallBack* de la ligne 6. Elle correspond bien à la signature du délégué *writeLog*. Le second paramètre passé à la méthode *Invoke* est le message qui sera passé en paramètre à la méthode *logEventCallBack*.

    L'opération *Invoke* est une opération synchrone. L'exécution du thread secondaire est bloquée jusqu'à ce que le thread propriétaire du contrôle exécute la méthode de rappel.
- ligne 6 : la méthode de rappel exécutée par le thread de l'interface graphique reçoit le message à afficher dans le contrôle *listBoxEvts*.

- ligne 9 : l'événement est logué en 1ère position de la liste afin d'avoir les événements les plus récents en haut de la liste.

Les messages du dialogue client / serveur sont affichés par la méthode *logDialogue* :

```
1.        // suivi du dialogue
2.        private void logDialogue(string msg) {
3.           listBoxDialogue.Invoke(new writeLog(logDialogueCallBack), msg);
4.        }
5.        private void logDialogueCallBack(string msg) {
6.           // affichage message
7.           msg = msg.Replace(finLigne, " ");
8.           listBoxDialogue.Items.Add(String.Format("{0:hh:mm:ss} : {1}", DateTime.Now, msg));
9.  }
```

Le principe est le même que dans la méthode *logEvent*.

Les messages reçus par le client sont affichés par la méthode *afficherRéponseServeur* :

```
1.        private void afficherRéponseServeur(String msg, bool dernièreLigne) {
2.  ...
3.  }
```

Le permier paramètre est le message à afficher. Ce message peut être une suite de lignes. En effet le client lit les données en provenance du serveur par bloc de *tailleBuffer* (1024) octets. Dans ces 1024 octets, on peut trouver diverses lignes que l'on reconnaît à leur marque de fin de ligne "\n". La dernière ligne peut être incomplète, sa marque de fin de ligne étant dans les 1024 octets qui vont suivre. La méthode retrouve dans le message les lignes terminées par "\n" et demande ensuite à *logDialogue* de les afficher. Le second paramètre de la méthode indique s'il faut afficher la dernière ligne trouvée ou la laisser dans le buffer pour être complétée par le message suivant. Le code est assez complexe et ne présente pas d'intérêt ici. Aussi ne sera-t-il pas commenté.

11.8.7 Conclusion

Le même exemple pourrait être traité avec des opérations synchrones. Ici l'aspect asynchrone de l'interface graphique apporte peu à l'utilisateur. Néanmoins, s'il se connecte et qu'ensuite il se rend compte que le serveur "ne répond plus", il a la possibilité de se déconnecter grâce au fait que l'interface graphique continue à répondre aux événements pendant l'exécution des opérations asynchrones. Cet exemple plutôt complexe nous a permis de présenter de nouvelles notions :

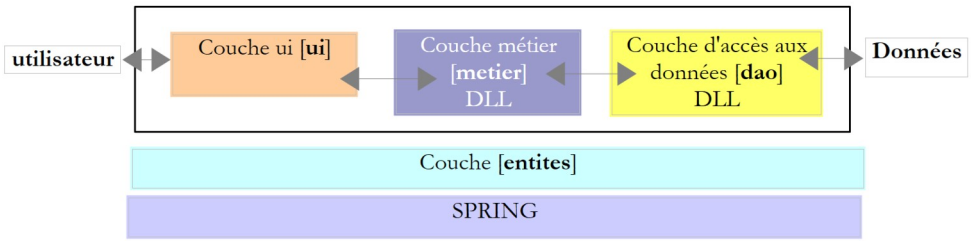
- l'utilisation de sockets
- l'utilisation de méthodes asynchrones. Ce qui a été vu fait partie d'un standard. D'autres méthodes asynchrones existent et fonctionnent sur le même modèle.
- la mise à jour de contrôles d'une interface graphique par des threads secondaires.

La communication Tcp / Ip asynchrone présente des avantages plus sérieux pour un serveur que ceux affichés par l'exemple précédent. On sait que le serveur sert ses clients à l'aide de threads secondaires. Si son pool de threads a N threads, cela signifie qu'il ne peut servir que N clients simultanément. Si les N threads font tous une opération bloquante (synchrone), il n'y a plus de threads disponibles pour un nouveau client jusqu'à ce que l'une des opérations bloquantes s'achève et libère un thread. Si sur les threads, on fait des opérations asynchrones plutôt que synchrones, un thread n'est jamais bloqué et peut être rapidement recyclé pour de nouveaux clients.

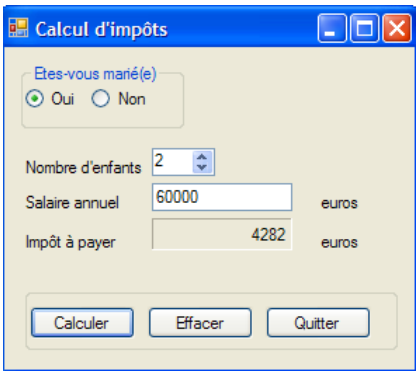
11.9 Application exemple, version 8 : Serveur de calcul d'impôts

11.9.1 L'architecture de la nouvelle version

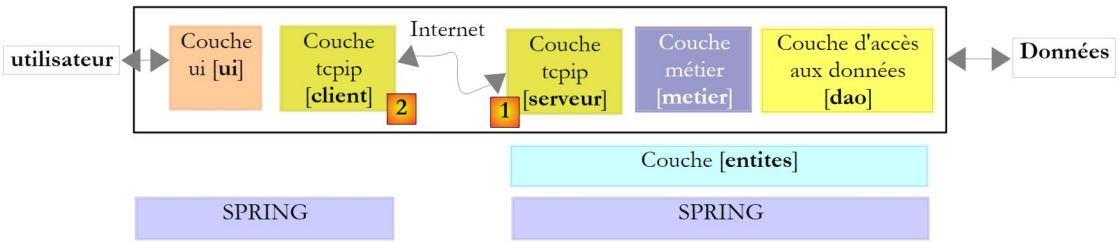
Nous reprenons l'application de calcul d'impôt déjà traitée sous diverses formes. Rappelons sa dernière mouture, celle de la version 7 du paragraphe 9.8, page 258.



Les données étaient dans une base de données et la couche [ui] était une interface graphique :



Nous allons rependre cette architecture et la distribuer sur deux machines :



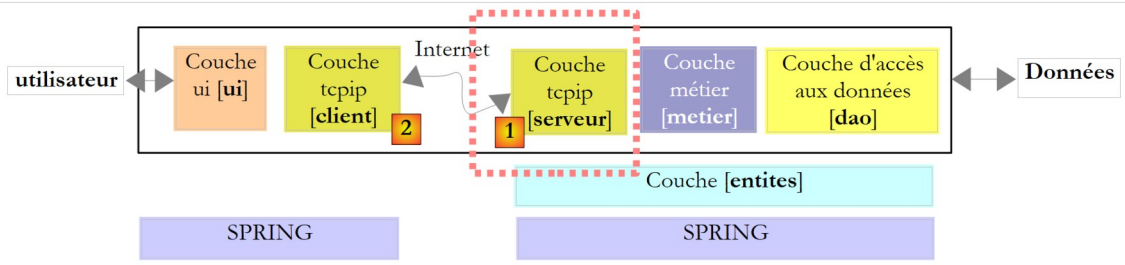
- une machine [serveur] hébergera les couches [metier] et [dao] de la version 7. Une couche Tcp/Ip [serveur] [1] sera construite afin de permettre à des clients de l'internet d'interroger le service de calcul de l'impôt.
- une machine [client] hébergera la couche [ui] de la version 7. Une couche Tcp/Ip [client] [2] sera construite afin de permettre à la couche [ui] d'interroger le service de calcul de l'impôt.

L'architecture change profondément ici. La version 7 était une application windows monoposte. La version 8 devient une application client / serveur de l'internet. Le serveur pourra servir plusieurs clients simultanément.

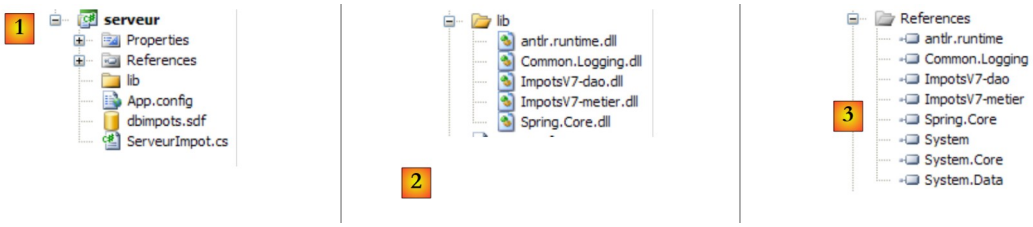
Nous allons tout d'abord écrire la partie [serveur] de l'application.

11.9.2 Le serveur de calcul d'impôt

11.9.2.1 Le projet Visual Studio



Le projet Visual studio sera le suivant :



- en [1], le projet. On y trouve les éléments suivants :
- [ServeurImpot.cs] : le serveur Tcp/Ip de calcul de l'impôt sous la forme d'une application console.
- [dbimpots.sdf] : la base de données SQL Server compact de la version 7 décrite au paragraphe 9.8.5, page 267.
- [App.config] : le fichier de configuration de l'application.
- en [2], le dossier [lib] contient les DLL nécessaires au projet :
  - [ImpotsV7-dao] : la couche [dao] de la version 7
  - [ImpotsV7-metier] : la couche [metier] de la version 7
  - [antlr.runtime, CommonLogging, Spring.Core] pour Spring
- en [3], les références du projet

11.9.2.2 Configuration de l'application

Le fichier [App.config] est exploité par Spring. Son contenu est le suivant :

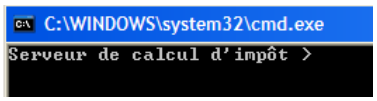
```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.   </configSections>
10.
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <object name="dao" type="Dao.DataBaseImpot, ImpotsV7-dao">
17.           <constructor-arg index="0" value="System.Data.SqlServerCe.3.5"/>
18.           <constructor-arg index="1" value="Data Source=|DataDirectory|\dbimpots.sdf;" />
19.           <constructor-arg index="2" value="select data1, data2, data3 from data"/>
20.        </object>
21.        <object name="metier" type="Metier.ImpotMetier, ImpotsV7-metier">
22.           <constructor-arg index="0" ref="dao"/>
23.        </object>
24.     </objects>
25.  </spring>
26. </configuration>
```

- lignes 16-20 : configuration de la couche [dao] associée à la base SQL Server compact
- lignes 21-23 : configuration de la couche [metier].

C'est le fichier de configuration utilisée dans la couche [ui] de la version 7. Il a été présenté au paragraphe 9.8.4, page 265.

11.9.2.3 Fonctionnement du serveur

Au démarrage du serveur, l'application serveur instancie les couches [metier] et [dao] puis affiche une interface console d'administration :



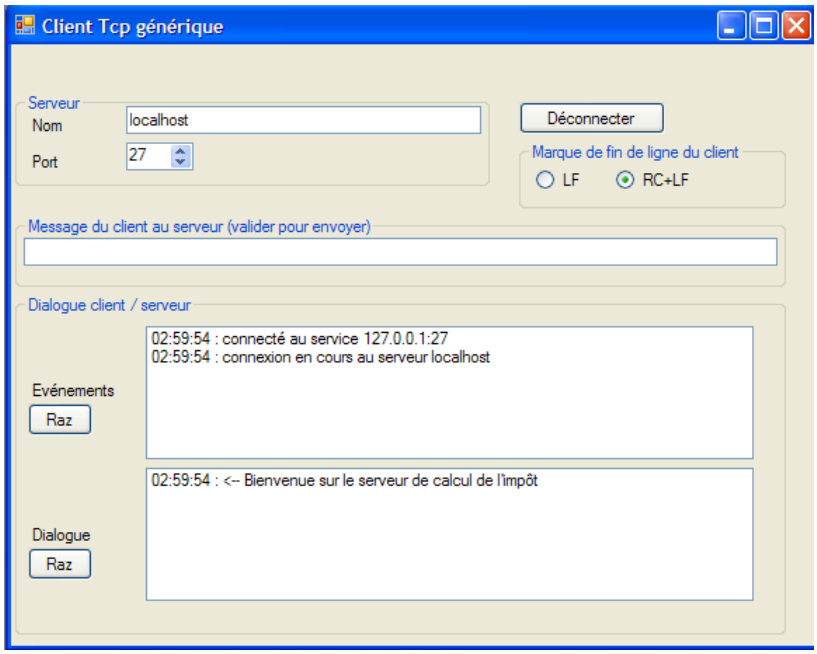
La console d'administration accepte les commandes suivantes :

`start port`	pour lancer le service sur un port donné
`stop`	pour arrêter le service. Il peut être ensuite relancé sur le même port ou un autre.
`echo start`	pour activer l'écho du dialogue client / serveur sur la console
`echo stop`	pour désactiver l'écho
`status`	pour faire afficher l'état actif / inactif du service
`quit`	pour quitter l'application

Lançons le serveur :

```
1.  Serveur de calcul d'impôt >start 27
2.  Serveur de calcul d'impôt lancé sur le port 27
3.  Serveur de calcul d'impôt >
```

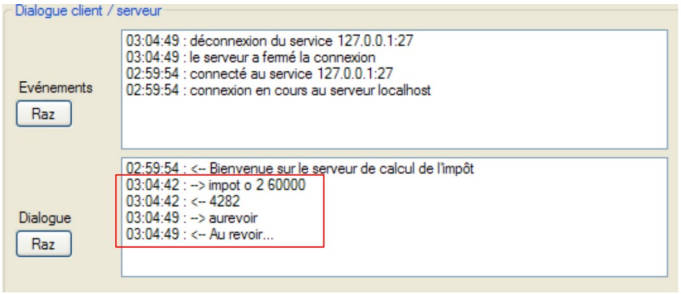
Lançons maintenant le client Tcp graphique asynchrone étudié précédemment au paragraphe 11.8, page 365.



Le client est connecté. Il peut envoyer les commandes suivantes au serveur de calcul d'impôt :

`aide`	pour avoir la liste des commandes autorisées
`impot marié nbEnfants salaireAnnuel`	pour calculer l'impôt de quelqu'un ayant *nbEnfants* enfants et un salaire de *salaireAnnuel* euros. *marié* vaut **o** si la personne est mariée, **n** sinon.
`aurevoir`	pour clôre la connexion avec le serveur

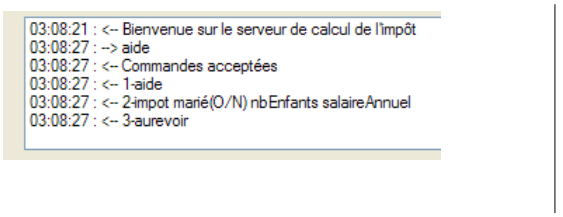
Voici un exemple de dialogue :



Côté serveur, la console affiche la chose suivante :

```
1.  Serveur de calcul d'impôt >start 27
2.  Serveur de calcul d'impôt >Serveur de calcul d'impôt lancé sur le port 27
3.  Début du service au client 0
4.  Fin du service au client 0
```

Mettons l'écho en route et recommençons un nouveau dialogue à partir du client graphique :



La console d'administration affiche alors la chose suivante :

```
1.  echo start
2.  Serveur de calcul d'impôt >Début du service au client 1
3.  <--- Client 1 : aide
4.  ---> Client 1 : Commandes acceptées
5.  1-aide
6.  2-impot marié(O/N) nbEnfants salaireAnnuel
7.  3-aurevoir
```

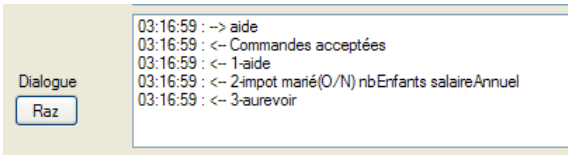
- ligne 1 : l'écho du dialogue client / serveur est activé
- ligne 2 : un client est arrivé
- ligne 3 : il a envoyé la commande [aide]
- lignes 4-7 : la réponse du serveur sur 4 lignes.

Arrêtons le service :

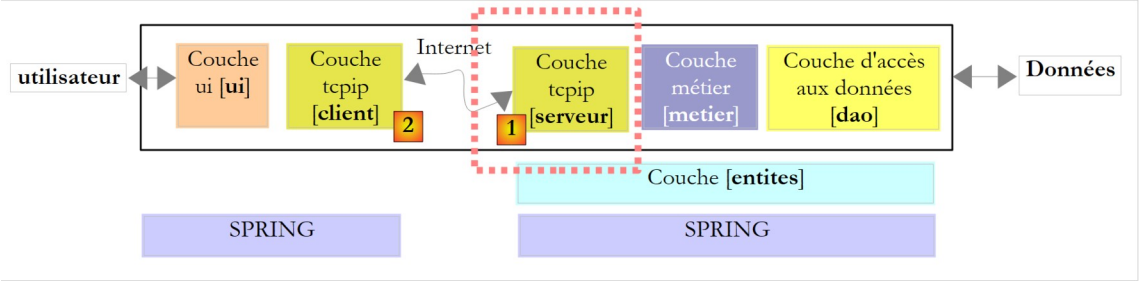
```
1.  stop
2.  L'erreur suivante s'est produite sur le serveur : Une opération de blocage a été interrompue par
    un appel à WSACancelBlockingCall
3.  Serveur de calcul d'impôt >
```

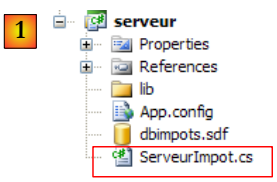
- ligne 1 : on demande l'arrêt du service (pas de l'application elle-même)
- ligne 2 : une exception due au fait que le serveur bloqué sur une attente de client a été brutalement interrompu à cause de la fermeture du service d'écoute.
- ligne 3 : le service peut être maintenant relancé par *start port* ou arrêté par *quit*.

Avant que le service d'écoute ne soit arrêté, un client était servi sur une autre connexion. Cette connexion n'est pas fermée par la fermeture de la socket d'écoute. Le client peut continuer à émettre des commandes : le thread de service qui lui avait été associé avant la fermeture du service d'écoute, continue à lui répondre :



11.9.3 Le code du serveur Tcp de calcul d'impôt





Le code du serveur [ServeurImpot.cs] est le suivant :

```
1.  ...
2.  namespace Chap9 {
3.   public class ServeurImpot {
4.
5.      // données partagées entre les threads et méthodes
6.      private static IImpotMetier metier = null;
7.      private static int port;
8.      private static TcpListener service;
9.      private static bool actif = false;
10.     private static bool echo = false;
11.
12.     // programme principal
13.     public static void Main(string[] args) {
14.        // instanciations couches [metier] et [dao]
15.        IApplicationContext ctx = null;
16.        metier = null;
17.        try {
18.           // contexte Spring
19.           ctx = ContextRegistry.GetContext();
20.           // on demande une référence sur la couche [metier]
21.           metier = (IImpotMetier)ctx.GetObject("metier");
22.
23.           // configuration pool de threads
24.           ThreadPool.SetMinThreads(10, 10);
25.           ThreadPool.SetMaxThreads(10, 10);
26.
27.           // lit les commandes d'administration du serveur tapées au clavier dans une boucle sans
    fin
28.           string commande = null;
29.           string[] champs = null;
30.           while (true) {
31.              // invite
32.              Console.Write("Serveur de calcul d'impôt >");
33.              // lecture commande
34.              commande = Console.ReadLine().Trim().ToLower();
```

```
35.             champs = Regex.Split(commande, @"\s+");
36.             // exécution commande
37.             switch (champs[0]) {
38.                case "start":
39.                   // actif ?
40.                   if (actif) {
41.                      //erreur
42.                      Console.WriteLine("Le serveur est déjà actif");
43.                   } else {
44.                      // vérification port
45.                      if (champs.Length != 2 || !int.TryParse(champs[1], out port) || port <= 0) {
46.                         Console.WriteLine("Syntaxe : start port. Port incorrect");
47.                      } else {
48.                         // on lance le service d'écoute
49.                         ThreadPool.QueueUserWorkItem(doEcoute, null);
50.                      }
51.                   }
52.                   break;
53.                case "echo":
54.                   // echo start / stop
55.                   if (champs.Length != 2 || (champs[1] != "start" && champs[1] != "stop")) {
56.                      Console.WriteLine("Syntaxe : echo start / stop");
57.                   } else {
58.                      echo = champs[1] == "start";
59.                   }
60.                   break;
61.                case "stop":
62.                   // fin du service
63.                   if (actif) {
64.                      service.Stop();
65.                      actif = false;
66.                   }
67.                   break;
68.                case "status":
69.                   // état du serveur
70.                   if (actif) {
71.                      Console.WriteLine("Le service est lancé sur le port {0}", port);
72.                   } else {
73.                      Console.WriteLine("Le service n'est pas lancé}");
74.                   }
75.                   break;
76.                case "quit":
77.                   // on quitte l'application
78.                   Console.WriteLine("Fin du service");
79.                   Environment.Exit(0);
80.                   break;
81.                default:
82.                   // commande incorrecte
83.                   Console.WriteLine("Commande incorrecte. Utilisez (start,stop,echo, status,
    quit)");
84.                   break;
85.             }
86.          }
87.       } catch (Exception e1) {
88.          // affichage exception
89.          Console.WriteLine("L'erreur suivante s'est produite à l'initialisation de
    l'application : {0}", e1.Message);
90.          return;
91.       }
92.    }
93.
94.
95.    private static void doEcoute(Object data) {
96. ...
97.    }
98.
99. ....
100. }
101.}
```

- lignes 18-21 : les couches [metier] et [dao] sont instanciées par Spring configuré par [App.config]. La variable globale *metier* de la ligne 6 est alors initialisée.
- lignes 24-25 : on configure le pool de threads de l'application avec 10 threads minimum et maximum.
- lignes 30-86 : la boucle de saisie des commandes d'administration du service (start, stop, quit, echo, status).
- ligne 32 : invite du serveur pour chaque nouvelle commande
- ligne 34 : lecture commande administrateur
- ligne 35 : la commande est découpée en champs afin d'être analysée
- lignes 38-52 : la commande *start port* qui a pour but de lancer le service d'écoute

    - ligne 40 : si le service est déjà actif, il n'y a rien à faire
    - ligne 45 : on vérifie que le port est bien présent et correct. Si oui, la variable globale *port* de la ligne 7 est positionnée.
    - ligne 49 : le service d'écoute va être géré par un thread secondaire afin que le thread principal puisse continuer à exécuter les commandes de la console. Si la méthode *doEcoute* réussit la connexion, les variables globales *service* de la ligne 8 et *actif* de la ligne 9 sont initialisées.
- lignes 53-60 : la commande *echo start / stop* qui active / désactive l'écho du dialogue client / serveur sur la console
    - ligne 58 : la variable globale *echo* de la ligne 7 est positionnée
- lignes 61-67 : la commande *stop* qui arrête le service d'écoute.
    - ligne 64 : arrêt du service d'écoute
- lignes 68-75 : la commande *status* qui affiche l'état actif / inactif du service
- lignes 76-80 : la commande *quit* qui arrête tout.

Le thread chargé d'écouter les demandes des clients exécute la méthode *doEcoute* suivante :

```
1.        private static void doEcoute(Object data) {
2.           // thread d'écoute des demandes des clients
3.           try {
4.              // on crée le service
5.              service = new TcpListener(IPAddress.Any, port);
6.              // on le lance
7.              service.Start();
8.              // le serveur est actif
9.              actif = true;
10.             // suivi
11.             Console.WriteLine("Serveur de calcul d'impôt lancé sur le port {0}", port);
12.             // boucle de service aux clients
13.             TcpClient tcpClient = null;
14.             // n° client
15.             int numClient = 0;
16.             // boucle sans fin
17.             while (true) {
18.                // attente d'un client
19.                tcpClient = service.AcceptTcpClient();
20.                // le service est assuré par une autre tâche
21.                ThreadPool.QueueUserWorkItem(doService, new Client() { CanalTcp = tcpClient,
    NumClient = numClient });
22.                // client suivant
23.                numClient++;
24.             }
25.          } catch (Exception ex) {
26.             // on signale l'erreur
27.             Console.WriteLine("L'erreur suivante s'est produite sur le serveur : {0}", ex.Message);
28.          }
29.       }
30.
31.       // infos client
32.       internal class Client {
33.          public TcpClient CanalTcp { get; set; }      // liaison avec le client
34.          public int NumClient { get; set; }        // n° de client
35. }
```

On a là un code similaire à celui du serveur d'écho étudié au paragraphe 11.6.1, page 329. Nous ne commentons que ce qui est différent :

- ligne 7 : le service d'écoute est lancé
- ligne 9 : on note le fait que le service est désormais actif

Ligne 21, les clients sont servis par des threads de service exécutant la méthode *doService* suivante :

```
1.  private static void doService(Object infos) {
2.           // on récupère le client qu'il faut servir
3.           Client client = infos as Client;
4.           // rend le service au client
5.           Console.WriteLine("Début du service au client {0}", client.NumClient);
6.           // exploitation liaison TcpClient
7.           try {
8.              using (TcpClient tcpClient = client.CanalTcp) {
9.                 using (NetworkStream networkStream = tcpClient.GetStream()) {
10.                   using (StreamReader reader = new StreamReader(networkStream)) {
11.                      using (StreamWriter writer = new StreamWriter(networkStream)) {
12.                         // flux de sortie non bufferisé
13.                         writer.AutoFlush = true;
14.                         // envoi d'un msg de bienvenue au client
15.                         writer.WriteLine("Bienvenue sur le serveur de calcul de l'impôt");
```

```
16.                         // boucle lecture demande/écriture réponse
17.                         string demande = null;
18.                         bool serviceFini = false;
19.                         while (!serviceFini && (demande = reader.ReadLine()) != null) {
20.                            // suivi console
21.                            if (echo) {
22.                               Console.WriteLine("<--- Client {0} : {1}", client.NumClient, demande);
23.                            }
24.                            // analyse demande
25.                            demande = demande.Trim().ToLower();
26.                            // demande vide ?
27.                            if (demande.Length == 0) {
28.                               // demande erronée
29.                               writeClient(writer,client.NumClient,"Commande non reconnue. Utilisez
   la commande aide.");
30.                               return;
31.                            }
32.
33.                            // on décompose la demande en champs
34.                            string[] champs = Regex.Split(demande, @"\s+");
35.                            // analyse
36.                            switch (champs[0].ToLower()) {
37.                               case "aide":
38.                                  writeClient(writer, client.NumClient, "Commandes acceptées\n1-
   aide\n2-impot marié(O/N) nbEnfants salaireAnnuel\n3-aurevoir");
39.                                  break;
40.                               case "impot":
41.                                  // on calcule l'impôt
42.                                  writeClient(writer, client.NumClient, calculImpot(writer,
   client.NumClient, champs));
43.                                  break;
44.                               case "aurevoir":
45.                                  serviceFini = true;
46.                                  writeClient(writer, client.NumClient, "Au revoir...");
47.                                  break;
48.                               default:
49.                                  writeClient(writer, client.NumClient, "Commande non reconnue.
   Utilisez la commande aide.");
50.                                  break;
51.                            }
52.                         }
53.                      }
54.                   }
55.                }
56.             }
57.          } catch (Exception e) {
58.             // erreur
59.             Console.WriteLine("L'erreur suivante s'est produite lors du service au client {0} :
   {1}", client.NumClient, e.Message);
60.          } finally {
61.             Console.WriteLine("Fin du service au client {0}", client.NumClient);
62.          }
63.       }
64.
65.       private static void writeClient(StreamWriter writer, int numClient, string message) {
66.          // echo console ?
67.          if (echo) {
68.             Console.WriteLine("---> Client {0} : {1}", numClient, message);
69.          }
70.          // envoi msg au client
71.          writer.WriteLine(message);
72. }
```

De nouveau, on a là un code similaire à celui du serveur d'écho étudié au paragraphe 11.6.1, page 329. Nous ne commentons que ce qui est différent :

- ligne 15 : une fois le client connecté, le serveur lui envoie un message de bienvenue.
- lignes 19-52 : la boucle de lecture des commandes du client. La boucle s'arrête lorsque le client envoie la commande "*aurevoir*".
- ligne 27 : cas d'une commande vide
- ligne 34 : la demande est décomposée en champs pour être analysée
- ligne 37 : commande *aide* : le client demande la liste des commandes autorisées
- ligne 40 : commande *impot* : le client demande un calcul d'impôt. On répond avec le message retourné par la méthode *calculImpot* que nous allons détailler prochainement.
- ligne 44 : commande *aurevoir* : le client indique qu'il a terminé.
    - ligne 45 : on se prépare à sortir de la boucle de lecture des demandes du clients (lignes 19-52)
    - ligne 46 : on répond au client par un message d'au revoir

- ligne 48 : une commande incorrecte. On envoie au client un message d'erreur.

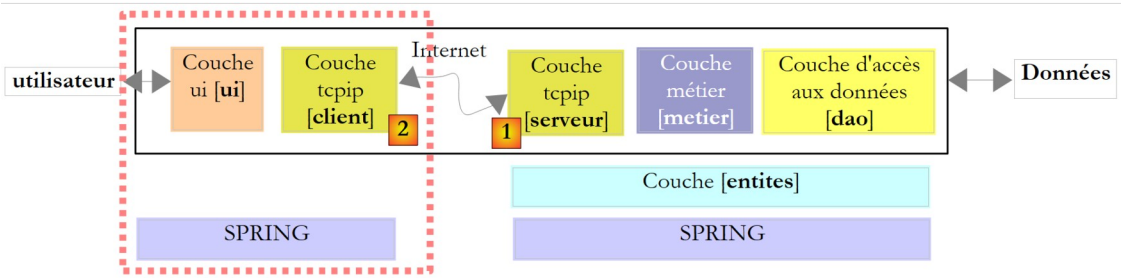
Le traitement de la commande *impot* est assurée par la méthode *calculImpot* suivante :

```
1.  private static string calculImpot(StreamWriter writer, int numClient, string[] champs) {
2.          // demande calcul marié(O/N) nbEnfants salaireAnnuel
3.          // il faut 4 champs
4.          if (champs.Length != 4) {
5.             return "Commande calcul incorrecte. Utilisez la commande aide.";
6.          }
7.          // champs [1]
8.          string marié = champs[1];
9.          if (marié != "o" && marié != "n") {
10.            return "Commande calcul incorrecte. Utilisez la commande aide.";
11.         }
12.         // champs [2]
13.         int nbEnfants;
14.         if (!int.TryParse(champs[2], out nbEnfants)) {
15.            return "Commande calcul incorrecte. Utilisez la commande aide.";
16.         }
17.         // champs [3]
18.         int salaireAnnuel;
19.         if (!int.TryParse(champs[3], out salaireAnnuel)) {
20.            return "Commande calcul incorrecte. Utilisez la commande aide.";
21.         }
22.         // c'est bon - on calcule l'impôt
23.         int impot = 0;
24.         try {
25.            impot = metier.CalculerImpot(marié == "o", nbEnfants, salaireAnnuel);
26.            return impot.ToString();
27.         } catch (Exception ex) {
28.            return ex.Message;
29.         }
30.      }
```

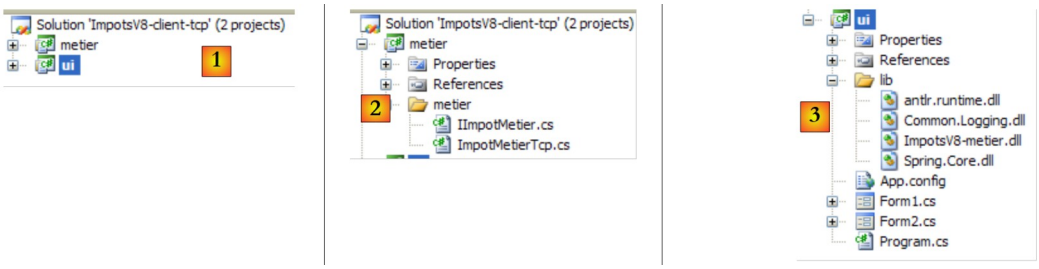
- ligne 1 : la méthode reçoit comme 3ième paramètre le tableau des champs de la commande *impot*. Si celle-ci a été correctement formulée, elle est de la forme *impot marié nbEnfants salaireAnnuel*. La méthode rend comme résultat la réponse à envoyer au client.
- ligne 4 : on vérifie que la commande a 4 champs
- ligne 8 : on vérifie que le champ *marié* est valide
- ligne 14 : on vérifie que le champ *nbEnfants* est valide
- ligne 19 : on vérifie que le champ *salaireAnnuel* est valide
- ligne 25 : l'impôt est calculé à l'aide de la méthode *CalculerImpot* de la couche [metier]. On rappelle que cette couche est encapsulée dans une DLL.
- ligne 26 : si la couche [metier] a rendu un résultat, celui-ci est rendu au client.
- ligne 28 : si la couche [metier] a lancé une exception, le message de celle-ci est rendu au client.

11.9.4 Le client graphique du serveur Tcp de calcul d'impôt

11.9.4.1 Le projet Visual Studio



Le projet Visual studio du client graphique sera le suivant :



- en [1], les deux projets de la solution, un pour chacune des deux couches de l'application
- en [2], le client Tcp qui joue le rôle de couche [metier] pour la couche [ui]. Aussi utiliserons-nous les deux termes.
- en [3], la couche [ui] de la version 7, à un détail près dont nous parlerons

11.9.4.2 La couche [metier]

L'interface *IImpotMetier* n'a pas changé. C'est toujours celle de la version 7 :

```
1.  namespace Metier {
2.   public interface IImpotMetier {
3.      int CalculerImpot(bool marié, int nbEnfants, int salaire);
4.   }
5.  }
```

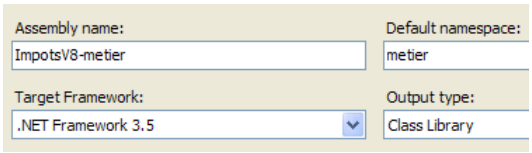
L'implémentation de cette interface est la classe [ImpotMetierTcp] suivante :

```
1.  using System.Net.Sockets;
2.  using System.IO;
3.  namespace Metier {
4.   public class ImpotMetierTcp : IImpotMetier {
5.
6.      // informations [serveur]
7.      private string Serveur { get; set; }
8.      private int Port { get; set; }
9.
10.     // calcul de l'impôt
11.     public int CalculerImpot(bool marié, int nbEnfants, int salaire) {
12.           // on se connecte au service
13.           using (TcpClient tcpClient = new TcpClient(Serveur, Port)) {
14.              using (NetworkStream networkStream = tcpClient.GetStream()) {
15.                 using (StreamReader reader = new StreamReader(networkStream)) {
16.                    using (StreamWriter writer = new StreamWriter(networkStream)) {
17.                       // flux de sortie non bufferisé
18.                       writer.AutoFlush = true;
19.                       // on saute le msg de bienvenue
20.                       reader.ReadLine();
21.                       // demande
22.                       writer.WriteLine(string.Format("impot {0} {1} {2}",marié ? "o" :
    "n",nbEnfants, salaire));
23.                       // réponse
24.                       return int.Parse(reader.ReadLine());
25.                    }
26.                 }
27.              }
28.           }
29.        }
30.     }
31.  }
```

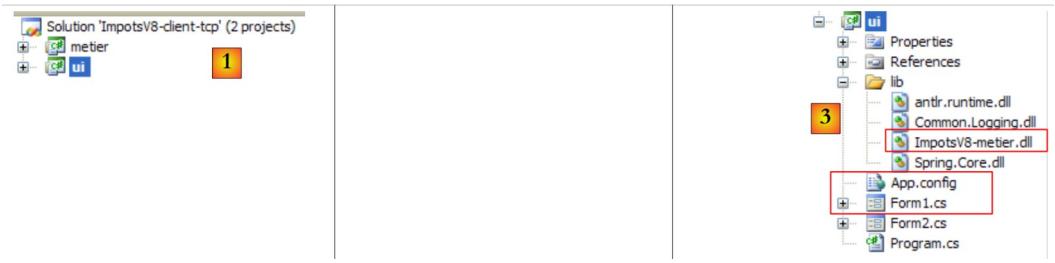
- ligne 7 : le nom ou l'adresse Ip du serveur Tcp de calcul d'impôt
- ligne 8 : le port d'écoute de ce serveur
- ces deux propriétés seront initialisées par Spring lors de l'instanciation de la classe [ImpotMetierTcp].
- ligne 11 : la méthode de calcul de l'impôt. Lorsqu'elle s'exécute, les propriétés *Serveur* et *Port* sont déjà initialisées. On retrouve dans le code la démarche classique d'un client Tcp
- ligne 13 : la connexion avec le serveur est ouverte
- lignes 14-16 : on récupère (ligne 14) le flux réseau associé à cette connexion duquel on tire un flux de lecture (ligne 15) et un flux d'écriture (ligne 16).
- ligne 18 : le flux d'écriture doit être non bufferisé

- ligne 20 : ici, il faut se rappeler qu'à l'ouverture de la connexion, le serveur envoie au client une 1ère ligne qui est le message de bienvenue "*Bienvenue sur le serveur de calcul de l'impôt*". Ce message est lu et ignoré.
- ligne 22 : on envoie au serveur la commande du type : *impot o 2 60000* pour lui demander de calculer l'impôt d'une personne mariée ayant 2 enfants et un salaire annuel de 60000 euros.
- ligne 24 : le serveur répond par le montant de l'impôt sous la forme "4282" ou bien avec un message d'erreur si la commande était mal formée (ça n'arrivera pas ici) ou si le calcul de l'impôt a rencontré un problème. Ici, ce dernier cas n'est pas géré mais il aurait été certainement plus "propre" de le faire. En effet, si la ligne lue est un message d'erreur, une exception sera lancée parce que la conversion vers un entier va échouer. L'exception récupérée par l'interface graphique va être une erreur de conversion alors que l'exception originelle est d'une tout autre nature. Le lecteur est invité à améliorer ce code.
- lignes 25-28 : libération de toutes les ressources utilisées avec une clause "using".

La couche [metier] est compilée dans la DLL ImpotsV8-metier.dll :



11.9.4.3 La couche [ui]



La couche [ui] [1,3] est celle étudiée dans la version 7 au paragraphe 9.8.4, page 265 à trois détails près :

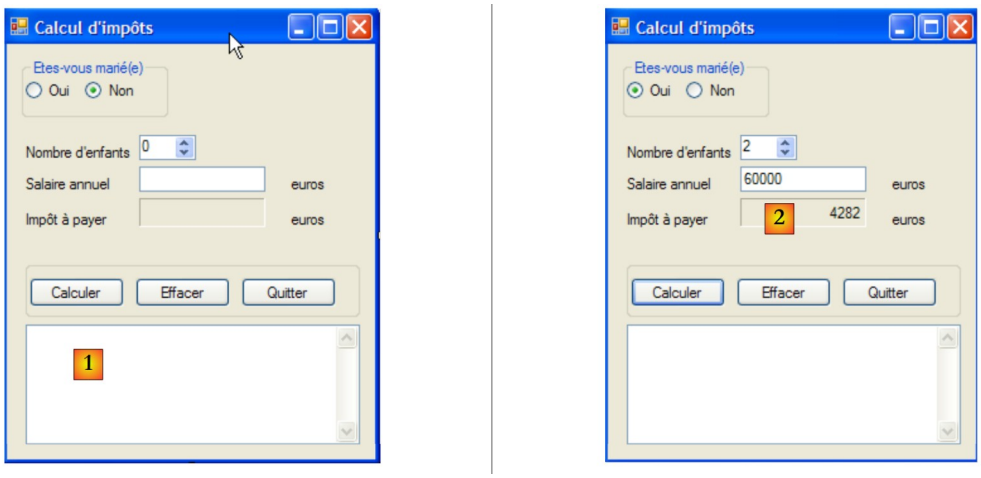
- la configuration de la couche [metier] dans [App.config] est différente parce que l'implémentation de celle-ci a changé
- l'interface graphique [Form1.cs] a été modifiée pour afficher une éventuelle exception
- la couche [metier] est dans la DLL [ImpotsV8-metier.dll].

Le fichier [App.config] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.   </configSections>
10.
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <object name="metier" type="Metier.ImpotMetierTcp, ImpotsV8-metier">
17.           <property name="Serveur" value="localhost"/>
18.           <property name="Port" value="27"/>
19.        </object>
20.     </objects>
21.  </spring>
22. </configuration>
```

- ligne 16 : instanciation de la couche [metier] avec la classe *Metier.ImpotMetierTcp* de la DLL *ImpotsV8-metier.dll*
- lignes 17-18 : les propriétés **Serveur** et **Port** de la classe *Metier.ImpotMetierTcp* sont initialisées. Le serveur sera sur la machine *localhost* et opèrera sur le port 27.

L'interface graphique présentée à l'utilisateur est la suivante :



- en [1], on a ajouté un *TextBox* pour afficher une éventuelle exception. Ce champ n'existait pas dans la version précédente.

A part ce détail, le code du formulaire est celui déjà étudié au paragraphe 6.4.3, page 149. Le lecteur est invité à s'y reporter. En [2], on voit un exemple d'exécution obtenu avec un serveur lancé de la façon suivante :

```
1.  Serveur de calcul d'impôt >start 27
2.  Serveur de calcul d'impôt lancé sur le port 27
3.  Serveur de calcul d'impôt >echo start
4.  Serveur de calcul d'impôt >
5.  ...
6.  Début du service au client 9
7.  <--- Client 9 : impot o 2 60000
8.  ---> Client 9 : 4282
9.  Fin du service au client 9
```

La copie d'écran [2] du client correspond aux lignes du client 9 ci-dessus.

11.9.5 Conclusion

De nouveau, nous avons pu réutiliser du code existant, sans modifications (couches [metier], [dao] du serveur) ou avec très peu de modifications (couche [ui] du client). Cela a été rendu possible par notre utilisation systématique d'interfaces et l'instanciation de celles-ci avec Spring. Si dans la version 7, nous avions mis le code métier directement dans les gestionnaires d'événements de l'interface graphique, ce code métier n'aurait pas été réutilisable. C'est l'inconvénient majeur des architectures 1 couche.

On notera enfin que la couche [ui] n'a aucune connaissance du fait que c'est un serveur distant qui lui calcule le montant de l'impôt.

12 Services Web

12.1 Introduction

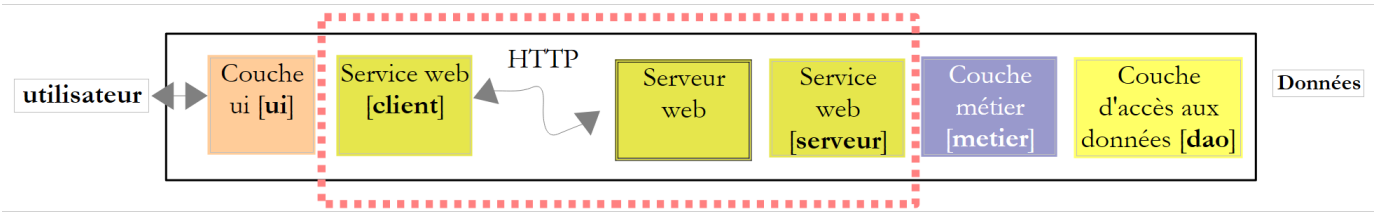
Nous avons présenté dans le chapitre précédent plusieurs applications client-serveur Tcp-Ip. Dans la mesure où les clients et le serveur échangent des lignes de texte, ils peuvent être écrits en n'importe quel langage. Le client doit simplement connaître le protocole de dialogue attendu par le serveur.

Les services Web sont également des applications serveur Tcp-Ip. Ils présentent les caractéristiques suivantes :

- Ils sont hébergés par des serveurs web et le protocole d'échanges client-serveur est 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.
- 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.

L'architecture d'une application client / serveur à base d'un service web est la suivante :

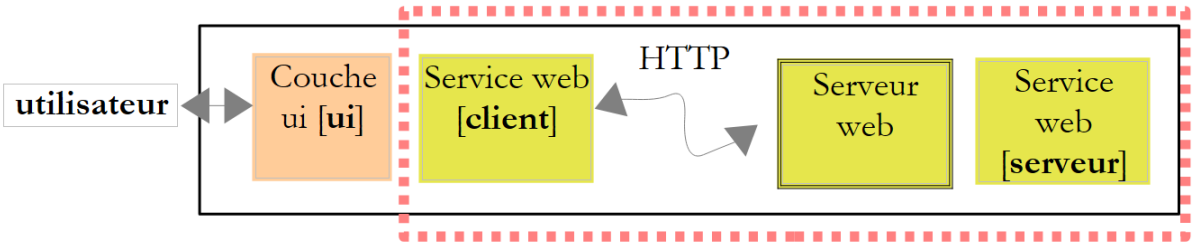


C'est une extension de l'architecture 3 couches à laquelle on ajoute des classes de communication réseau spécialisées. On a déjà rencontré une architecture similaire avec l'application client graphique windows / serveur Tcp d'impôts au paragraphe 11.9.1, page 373.

Explicitons ces généralités avec un premier exemple.

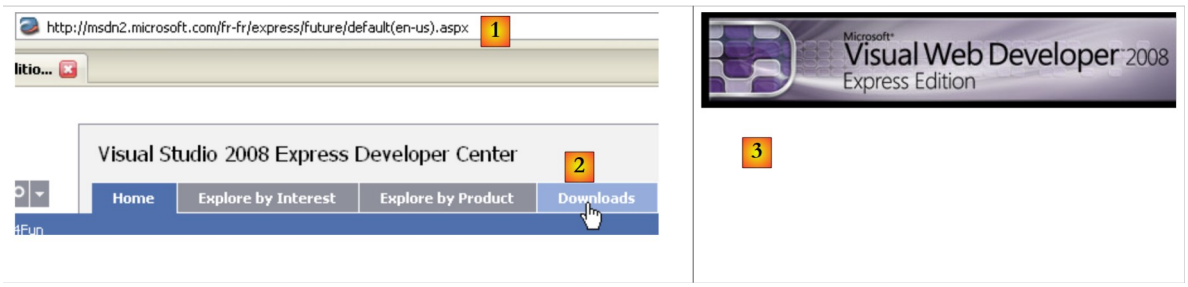
12.2 Un premier service Web avec Visual Web Developer

Nous allons construire une première application client / serveur ayant l'architecture simplifiée suivante :



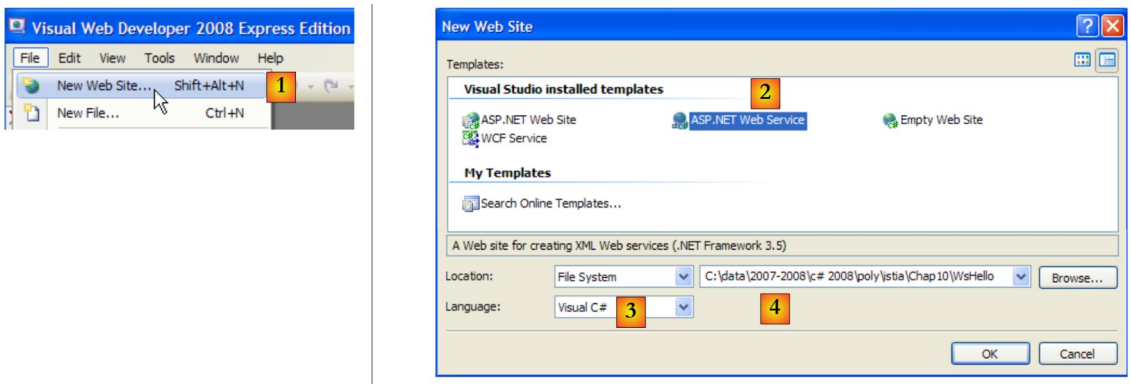
12.2.1 La partie serveur

Nous avons indiqué qu'un service web était hébergé par un serveur web. L'écriture d'un service web entre dans le cadre général de la programmation web, côté serveur. Nous avons eu précédemment l'occasion d'écrire des clients web, ce qui est aussi de la programmation web mais côté client cette fois. Le terme programmation web désigne le plus souvent la programmation côté serveur plutôt que celle côté client. Pour développer des services web ou plus généralement des applications web, Visual C# n'est pas l'outil approprié. Nous allons utiliser Visual Developer l'une des versions Express de Visual Studio 2008 téléchargeables [2] à l'adresse [1] : [http://msdn.microsoft.com/fr-fr/express/future/bb421473(en-us).aspx] (mai 2008) :

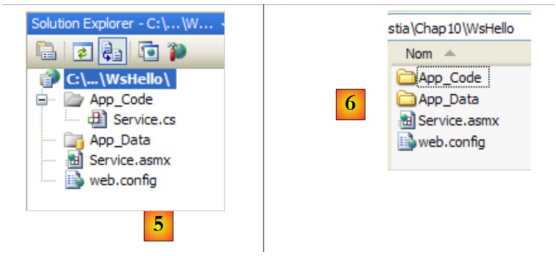


- [1] : l'adresse de téléchargement
- [2] : l'onglet des téléchargements
- [3] : télécharger Visual Developer 2008

Pour créer un premier service web, on pourra procéder comme suit, après avoir lancé Visual Developer :



- [1] : prendre l'option **File / New Web Site**
- [2] : choisir une application de type **ASP.NET Web Service**
- [3] : choisir le langage de développement : **C#**
- [4] : indiquer le dossier où créer le projet



- [5] : le projet créé dans Visual Web Developer
- [6] : le dossier du projet sur le disque

Une application web est structurée de la façon suivante dans Web Developer :

- une racine dans laquelle on trouve les documents du site web (pages web statiques Html, images, pages web dynamiques .aspx, services web .asmx, ...). On y trouve également le fichier [web.config] qui est le fichier de configuration de l'application web. Il joue le même rôle que le fichier [App.config] des applications windows et est structuré de la même façon.
- un dossier [App_Code] dans lequel on trouve les classes et interfaces du site web destinées à être compilées.
- un dossier [App_Data] dans lequel on mettra des données exploitées par les classes de [App_Code]. On pourra par exemple y trouver une base SQL Server *.mdf.

[Service.asmx] est le service web dont nous avons demandé la création. Il ne contient que la ligne suivante :

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Service.cs" Class="Service" %>
```

Le code-source ci-dessus est destiné au serveur Web qui hébergera l'application. En mode production, ce serveur est en général IIS (Internet Information Server), le serveur web de Microsoft. Visual Web Developer embarque un serveur web léger qui est utilisé en mode développement. La directive précédente indique au serveur web :

- [Service.asmx] est un service Web (directive *WebService*)
- écrit en C# (attribut *Language*)
- que le code C# du service web se trouve dans le fichier [~/App_Code/Service.cs] (attribut *CodeBehind*). C'est là qu'ira le chercher le serveur web pour le compiler.
- que la classe implémentant le service web s'appelle *Service* (attribut *Class*)

Le code C# [Service.cs] du service web généré par Visual Developer est le suivant :

```
1.  using System.Web.Services;
2.
3.  [WebService(Namespace = "http://tempuri.org/")]
4.  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
5.  // To allow this Web Service to be called from script, using ASP.NET AJAX, uncomment the following
    line.
6.  // [System.Web.Script.Services.ScriptService]
7.  public class Service : System.Web.Services.WebService
8.  {
9.      public Service () {
10.
11.         //Uncomment the following line if using designed components
12.         //InitializeComponent();
13.     }
14.
15.     [WebMethod]
16.     public string HelloWorld() {
17.         return "Hello World";
18.     }
19.
20. }
```

La classe *Service* ressemble à une classe C# classique avec cependant quelques points à noter :

- ligne 7 : 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.
- ligne 3 : la classe elle-même est précédée d'un attribut **[WebService(Namespace="http://tempuri.org/")]** 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 **http://tempuri.org/**. Ce nom peut être quelconque. Il n'a pas forcément la forme d'une Uri Http.

- ligne 15 : la méthode *HelloWorld* est précédée d'un attribut **[WebMethod]** qui indique au compilateur que la méthode doit être rendue visible aux clients distants du service web. Une méthode non précédée de cet attribut n'est pas visible par les clients du service web. Ce pourrait être une méthode interne utilisée par d'autres méthodes mais pas destinée à être publiée.
- ligne 9 : le constructeur du service web. Il est inutile dans notre application.

La classe [Service.cs] générée est transformée de la façon suivante :

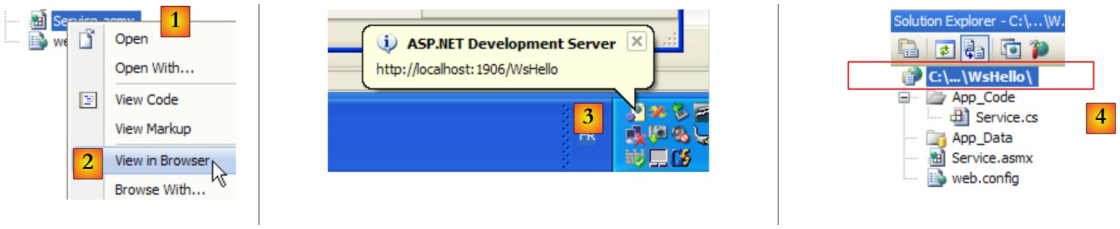
```
1.  using System.Web.Services;
2.
3.  [WebService(Namespace = "http://st.istia.univ-angers.fr")]
4.  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
5.  public class Service : System.Web.Services.WebService
6.  {
7.      [WebMethod]
8.      public string DisBonjourALaDame(string nomDeLaDame) {
9.          return string.Format("Bonjour Mme {0}", nomDeLaDame);
10.     }
11.
12. }
```

Le fichier de configuration [web.config] généré pour l'application web est le suivant :

```
1.  <?xml version="1.0"?>
2.  <!--
3.      Note: As an alternative to hand editing this file you can use the
4.      web admin tool to configure settings for your application. Use
5.      the Website->Asp.Net Configuration option in Visual Studio.
6.      A full list of settings and comments can be found in
7.      machine.config.comments usually located in
8.      \Windows\Microsoft.Net\Framework\v2.x\Config
9.  -->
10. <configuration>
11.
12.
13.     <configSections>
14.       <sectionGroup name="system.web.extensions"
    type="System.Web.Configuration.SystemWebExtensionsSectionGroup, System.Web.Extensions,
    Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
15. ...
16.       </sectionGroup>
17.     </configSections>
18.
19.
20.     <appSettings/>
21.     <connectionStrings/>
22. ...
23. </configuration>
```

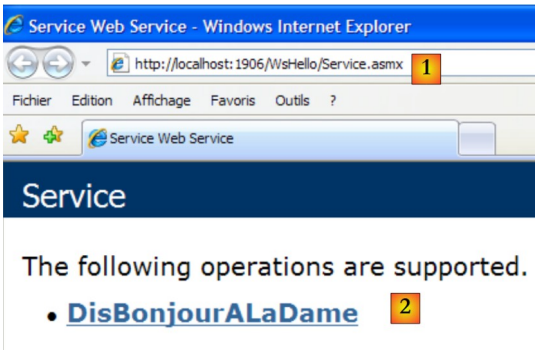
Le fichier fait 140 lignes. Il est complexe et nous ne le commenterons pas. Nous le garderons tel quel. Ci-dessus, on retrouve les balises <*configuration*>, <*configSections*>, <*sectionGroup*>, <*appSettings*>, <*connectionString*> que nous avons rencontrées dans le fichier [App.config] des applications windows.

Nous avons un service web opérationnel qui peut être exécuté :



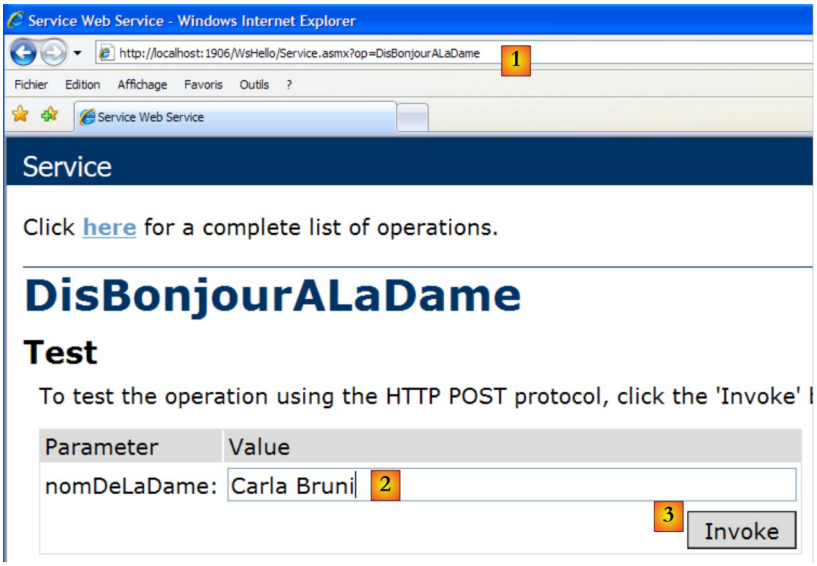
- [1,2] : on clique droit sur [Service.asmx] et on demande à voir la page dans un navigateur
- [3] : Visual Web Developer lance son serveur web intégré et met l'icône de celui-ci en bas à droite dans la barre des tâches. Le serveur web est lancé sur un port aléatoire, ici 1906. L'Uri affichée /WsHello est le nom du site web [4].

Visual Web Developer a également lancé un navigateur pour afficher la page demandée à savoir [Service.asmx] :



- en [1], l'Uri de la page. On retrouve l'Uri du site [http://localhost:1906/WsHello] suivi de celle de la page /Service.asmx.
- en [2], le suffixe .asmx a indiqué au serveur web qu'il s'agissait non pas d'une page web normale (suffixe .aspx) donnant naissance à une page Html, mais de la page d'un service web. Il génère alors de façon automatique une page web présentant un lien pour chacune des méthodes du service web ayant l'attribut [WebMethod]. Ces liens sont des liens permettant de tester les méthodes.

Un clic sur le lien [2] ci-dessus nous amène à la page suivante :



- en [1], on notera l'Uri [http://localhost:1906/WsHello/Service.asmx?op=DisBonjourALaDame] de la nouvelle page. C'est l'Uri du service web avec un paramètre *op=M*, où *M* est le nom d'une des méthodes du service web.
- Rappelons la signature de la méthode [DisBonjourALaDame] :

```
    public string DisBonjourALaDame(string nomDeLaDame) ;
```

La méthode admet un paramètre de type *string* et rend un résultat de type *string* également. La page nous permet d'exécuter la méthode [DisBonjourALaDame] : en [2] on met la valeur du paramètre *nomDeLaDame* et en [3], on demande l'exécution de la méthode. Nous obtenons le résultat suivant :

- en [1], on notera que l'Uri de la réponse n'est pas identique à celle de la demande. Elle a changé.
- en [2], la réponse du serveur web. On notera les points suivants :
  - c'est une réponse XML et non HTML
  - le résultat de la méthode [DisBonjourALaDame] est encapsulé dans une balise <string> représentant son type.
  - la balise <string> a un attribut *xmlns* (*xml name space*) qui est l'espace de noms que nous avons donné à notre service web (ligne 1 ci-dessous).

```
1.  [WebService(Namespace = "http://st.istia.univ-angers.fr")]
2.  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
3.  public class Service : System.Web.Services.WebService
```

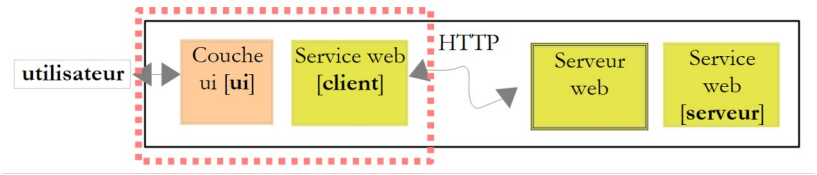
Pour savoir comment le navigateur web a fait sa demande, il faut regarder le code Html du formulaire de test :

```
1.  ...
2.  <span>
3.    <p class="intro">Click <a href="Service.asmx">here</a> for a complete list of operations.</p>
4.    <h2>DisBonjourALaDame</h2>
5.    <p class="intro"></p>
6.
7.    <h3>Test</h3>
8.
9.          To test the operation using the HTTP POST protocol, click the 'Invoke' button.
10.
11.    <form action='http://localhost:1906/WsHello/Service.asmx/DisBonjourALaDame' method="POST">
12.       <table>
13.          <tr>
14.             <td>Parameter</td>
15.             <td>Value</td>
16.          </tr>
17.          <tr>
18.               <td>nomDeLaDame:</td>
19.               <td><input type="text" size="50" name="nomDeLaDame"></td>
20.            </tr>
21.          <tr>
22.                <td></td>
23.                <td align="right"> <input type="submit" value="Invoke" class="button"></td>
24.            </tr>
25.       </table>
26.  </form>
27. <span>
28. ...
```

- ligne 11 : les valeurs du formulaire (balise **form**) seront postées (attribut **method**) à l'Url [http://localhost:1906/WsHello/Service.asmx/DisBonjourALaDame] (attribut **action**).
- ligne 19 : le champ de saisie s'appelle *nomDeLaDame* (attribut **name**).

Demander l'exécution du service web [/Service.asmx] nous a permis de tester ses méthodes et d'avoir un minimum de compréhension des échanges client serveur.

12.2.2 La partie client



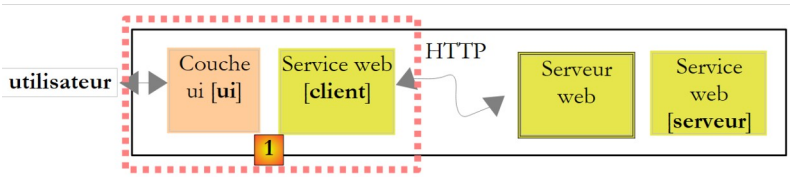
Il est possible d'implémenter le client du service web distant ci-dessus avec un client Tcp-Ip basique. Voici par exemple le dialogue client / serveur réalisé avec un client *putty* connecté au service web distant (localhost,1906) :

```
1.  POST /WsHello/Service.asmx/DisBonjourALaDame HTTP/1.1
2.  Host: localhost
3.  Content-Type: application/x-www-form-urlencoded
4.  Content-Length: 23
5.
6.  HTTP/1.1 100 Continue
```

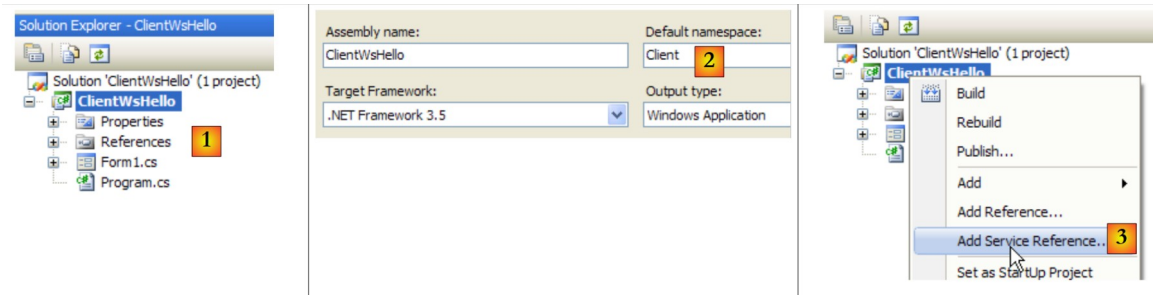
```
7.  Server: ASP.NET Development Server/9.0.0.0
8.  Date: Sat, 10 May 2008 08:36:41 GMT
9.  Content-Length: 0
10.
11. nomDeLaDame=Carla+Bruni
12. HTTP/1.1 200 OK
13. Server: ASP.NET Development Server/9.0.0.0
14. Date: Sat, 10 May 2008 08:36:47 GMT
15. X-AspNet-Version: 2.0.50727
16. Cache-Control: private, max-age=0
17. Content-Type: text/xml; charset=utf-8
18. Content-Length: 119
19. Connection: Close
20.
21. <?xml version="1.0" encoding="utf-8"?>
22. <string xmlns="http://st.istia.univ-angers.fr">Bonjour Mme Carla Bruni</string>
```

- lignes 1-5 : messages envoyés par le client *putty*
- ligne 1 : commande POST
- lignes 6-10 : réponse du serveur. Elle signifie que le client peut envoyer les valeurs du POST.
- ligne 11 : les valeurs postées sous la forme *param1=val1&param2=val2&* .... Certains caractères doivent être en caractères acceptables dans une Url. C'est ce qu'on a appelé précédemment une Url encodée. Ici le formulaire n'a qu'un unique paramètre nommé *nomDeLaDame*. La valeur postée a au total 23 caractères. Cette taille doit être déclarée dans l'entête Http de la ligne 4.
- lignes 12-22 : la réponse du serveur
- ligne 22 : le résultat de la méthode web [DisBonjourALaDame].

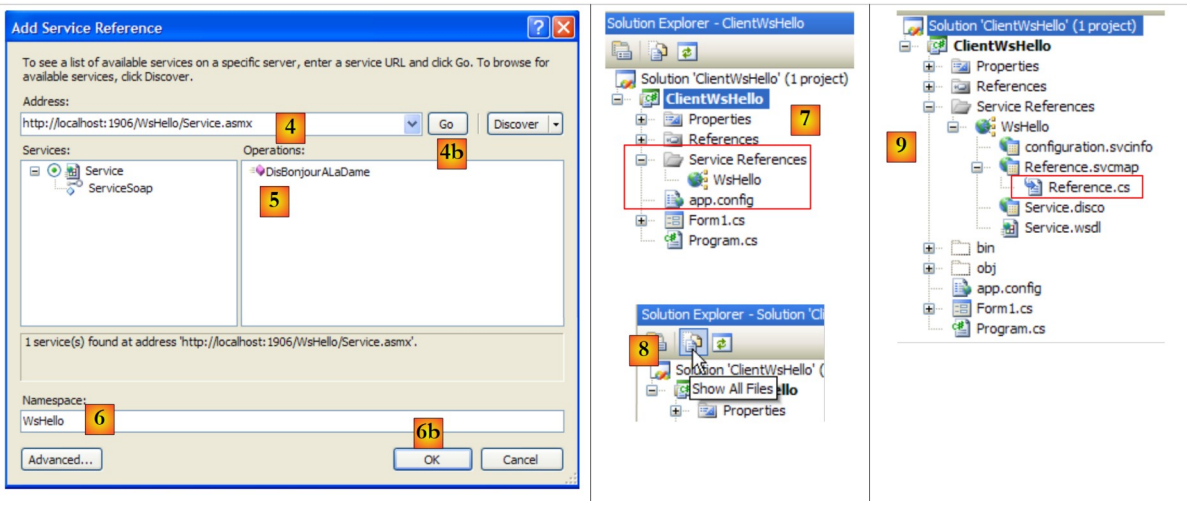
Avec Visual C#, il est possible de générer à l'aide d'un assistant le client d'un service web distant. C'est ce que nous voyons maintenant.



La couche [1] ci-dessus est implémentée par un projet Visual studio C# de type *Application Windows* et nommé **ClientWsHello** :



- en [1], le projet *ClientWsHello* dans Visual C#
- en [2], l'espace de noms par défaut du projet sera *Client* (clic droit sur le projet / Properties / Application). Cet espace de noms va servir à construire l'espace de noms du client qui va être généré.
- en [3], clic droit sur le projet pour lui ajouter une référence à un service web distant



- en [4], mettre l'Uri du service web construit précédemment
- en [4b], connecter Visual C# au service web désigné par [4]. Visual C# va récupérer la description du service web et grâce à cette description va pouvoir en générer un client.
- en [5], une fois la description du service web récupérée, Visual C# peut en afficher les méthodes publiques
- en [6], indiquer un espace de noms pour le client qui va être généré. Celui-ci sera ajouté à l'espace de noms défini en [2]. Ainsi l'espace de noms du client sera *Client.WsHello*.
- en [6b] valider l'assistant.
- en [7], la référence au service web *WsHello* apparaît dans le projet. Par ailleurs, un fichier de configuration [app.config] a été créé.
- en [8], visualiser tous les fichiers du projet.
- en [9], la référence au service web *WsHello* contient divers fichiers que nous n'expliciterons pas. Nous jetterons cependant un coup d'oeil sur le fichier [Reference.cs] qui est le code C# du client généré :

```
1.  namespace Client.WsHello {
2.  ...
3.      public partial class ServiceSoapClient :
    System.ServiceModel.ClientBase<Client.WsHello.ServiceSoap>, Client.WsHello.ServiceSoap {
4.
5.          public ServiceSoapClient() {
6.          }
7.  ...
8.          public string DisBonjourALaDame(string nomDeLaDame) {
9.              Client.WsHello.DisBonjourALaDameRequest inValue = new
    Client.WsHello.DisBonjourALaDameRequest();
10.             inValue.Body = new Client.WsHello.DisBonjourALaDameRequestBody();
11.             inValue.Body.nomDeLaDame = nomDeLaDame;
12.             Client.WsHello.DisBonjourALaDameResponse retVal = ((Client.WsHello.ServiceSoap)
    (this)).DisBonjourALaDame(inValue);
13.             return retVal.Body.DisBonjourALaDameResult;
14.         }
15.     }
16. }
```

- ligne 1 : l'espace de nom du client généré est *Client.WsHello*. Si on souhaite changer cet espace de noms, c'est là qu'il faut le faire.
- ligne 3 : la classe *ServiceSoapClient* est la classe du client généré. C'est une classe proxy dans le sens où elle va cacher à l'application windows le fait qu'un service web distant est utilisé. L'application windows va utiliser la classe *WsHello* distante via la classe locale *Client.WsHello.ServiceSoapClient*. Pour créer une instance du client, on utilisera le constructeur de la ligne 5 :

```
Client.WsHello.ServiceSoapClient client=new Client.WsHello.ServiceSoapClient();
```

- ligne 8 : la méthode *DisBonjourALaDame* est le pendant, côté client, de la méthode *DisBonjourALaDame* du service web. L'application windows utilisera la méthode distante *DisBonjourALaDame* via la méthode locale *Client.WsHello.ServiceSoapClient.DisBonjourALaDame* sous la forme suivante :

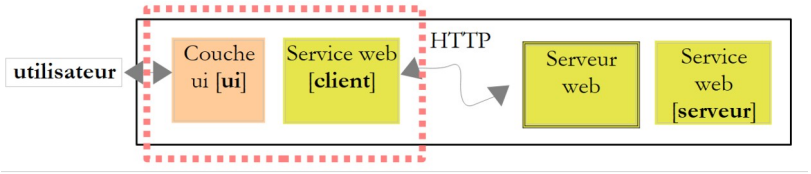
```
string bonjour=client.DisBonjourALaDame("Carla Bruni");
```

Le fichier [app.config] généré est le suivant :

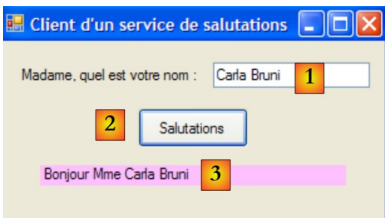
```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.      <system.serviceModel>
4.          <bindings>
5.           ....
6.          </bindings>
7.          <client>
8.              <endpoint address="http://localhost:1906/WsHello/Service.asmx"... />
9.          </client>
10.     </system.serviceModel>
11. </configuration>
```

De ce fichier, nous ne retiendrons que la ligne 8 qui contient l'Uri du service web. Si celui-ci change d'Uri, le client windows n'a pas à être reconstruit. Il suffit de changer l'Uri du fichier [app.config].

Revenons à l'architecture de l'application windows que nous voulons construire :



Nous avons construit la couche [client] du service web. La couche [ui] sera la suivante :



n°	type	nom	rôle
1	TextBox	textBoxNomDame	nom de la dame
2	Button	buttonSalutations	pour se connecter au service web *WsHello* distant et interroger la méthode *DisBonjourALaDame*.
3	Label	labelBonjour	le résultat renvoyé par le service web

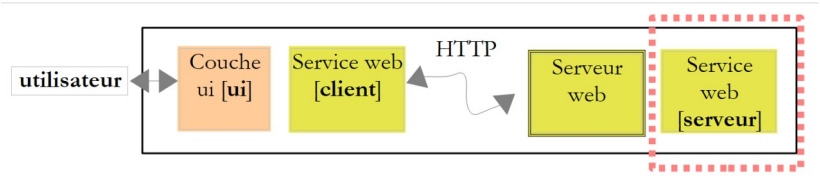
Le code du formulaire [Form1.cs] est le suivant :

```
1.  using System;
2.  using System.Windows.Forms;
3.  using Client.WsHello;
4.
5.  namespace ClientSalutations {
6.   public partial class Form1 : Form {
7.      public Form1() {
8.          InitializeComponent();
9.      }
10.
11.     private void buttonSalutations_Click(object sender, EventArgs e) {
12.         // sablier
13.         Cursor=Cursors.WaitCursor;
14.         // interrogation service web
15.         labelBonjour.Text = new ServiceSoapClient().DisBonjourALaDame(textBoxNomDame.Text.Trim());
16.         // curseur normal
17.         Cursor = Cursors.Arrow;
18.     }
19.  }
20. }
```

- ligne 15 : le client du service web est instancié. Il est de type *Client.WsHello.ServiceSoapClient*. L'espace de noms *Client.WsHello* est déclaré ligne 3. La méthode locale *ServiceSoapClient().DisBonjourALaDame* est appelée. On sait qu'elle même interroge la méthode distante de même nom du service web.

12.3 Un service Web d'opérations arithmétiques

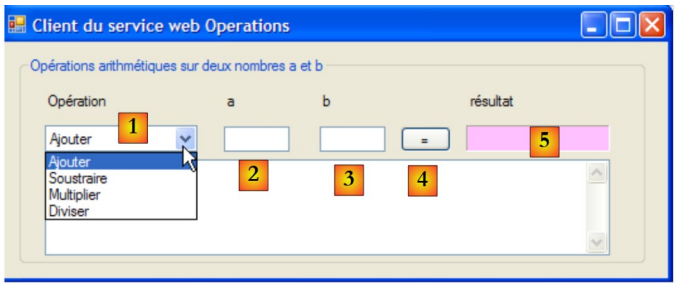
Nous allons construire une seconde application client / serveur ayant de nouveau l'architecture simplifiée suivante :



Le service web précédent offrait une unique méthode. Nous considérons un service Web qui offrira les 4 opérations arithmétiques :

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

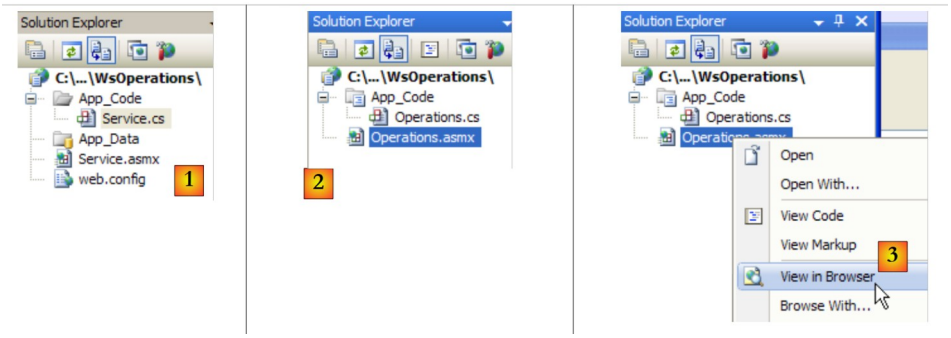
et qui sera interrogé par l'interface graphique suivante :



- en [1], l'opération à faire
- en [2,3] : les opérandes
- en [4], le bouton d'appel du service web
- en [5], le résultat rendu par le service web

12.3.1 La partie serveur

Nous construisons un projet de type service web avec Visual Web Developer :



- en [1], l'application web *WsOperations* générée
- en [2], l'application web *WsOperations* relookée de la façon suivante :
  - la page web [Service.asmx] a été renommée [Operations.asmx]
  - la classe [Service.cs] a été renommée [Operations.cs]
  - le fichier [web.config] a été supprimé afin de montrer qu'il n'est pas indispensable.

La page web [Service.asmx] contient la ligne suivante :

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Operations.cs" Class="Operations" %>
```

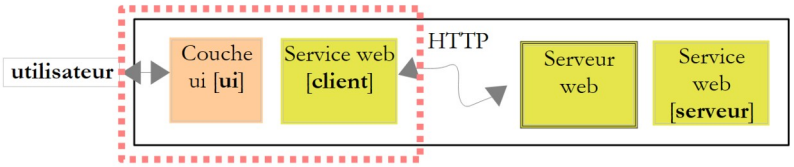
Le service web est assuré par classe [Operations.cs] suivante :

```
1.  using System.Web.Services;
2.
3.  [WebService(Namespace = "http://st.istia.univ-angers.fr/")]
4.  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
5.  public class Operations : System.Web.Services.WebService
6.  {
7.      [WebMethod]
8.      public double Ajouter(double a, double b)
9.      {
10.         return a + b;
11.     }
12.
13.     [WebMethod]
14.     public double Soustraire(double a, double b)
15.     {
16.         return a - b;
17.     }
18.
19.     [WebMethod]
20.     public double Multiplier(double a, double b)
21.     {
22.         return a * b;
23.     }
24.
25.     [WebMethod]
26.     public double Diviser(double a, double b)
27.     {
28.         return a / b;
29.     }
30.
31. }
```

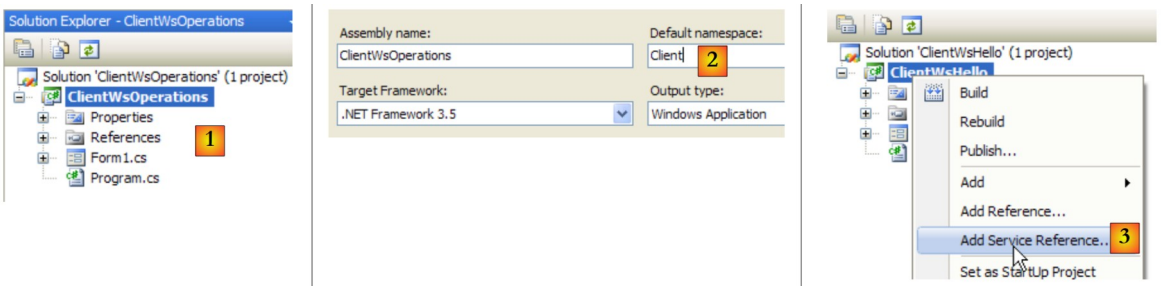
Pour mettre en ligne le service web, nous procédons comme indiqué en [3]. Nous obtenons alors la page de test des 4 méthodes du service web *WsOperations* :

Le lecteur est invité à tester les 4 méthodes.

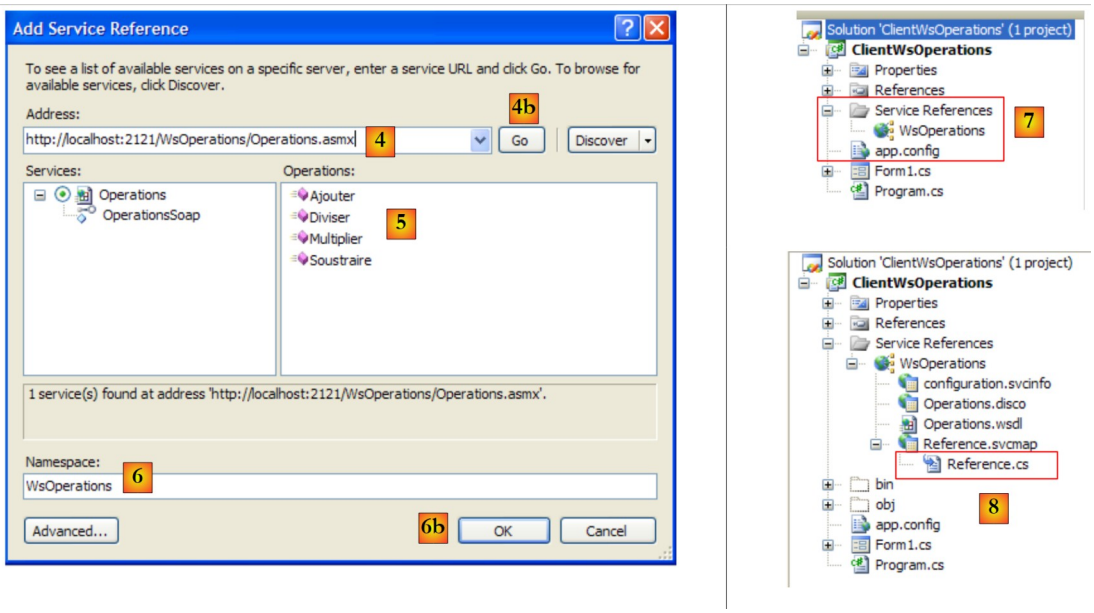
12.3.2 La partie client



Avec Visual C# nous créons une application Windows *ClientWsOperations* :



- en [1], le projet *ClientWsOperations* dans Visual C#
- en [2], l'espace de noms par défaut du projet sera *Client* (clic droit sur le projet / Properties / Application). Cet espace de noms va servir à construire l'espace de noms du client qui va être généré.
- en [3], clic droit sur le projet pour lui ajouter une référence à un service web existant



- en [4], mettre l'Uri du service web construit précédemment. Il faut pour cela regarder ce qui est affiché dans le champ adresse du navigateur qui affiche la page de test du service web.
- en [4b], connecter Visual C# au service web désigné par [4]. Visual C# va récupérer la description du service web et grâce à cette description va pouvoir en générer un client.
- en [5], une fois la description du service web récupérée, Visual C# peut en afficher les méthodes publiques
- en [6], indiquer un espace de noms pour le client qui va être généré. Celui-ci sera ajouté à l'espace de noms défini en [2]. Ainsi l'espace de noms du client sera *Client.WsOperations*.
- en [6b] valider l'assistant.
- en [7], la référence au service web *WsOperations* apparaît dans le projet. Par ailleurs, un fichier de configuration [app.config] a été créé.

On rappelle que le client généré est de type *Client.WsOperations.OperationsSoapClient* où

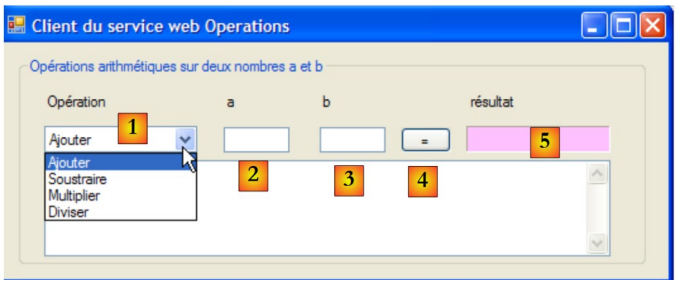
- *Client.WsOperations* est l'espace de noms du client du service web
- *Operations* est la classe du service web distant.

Même s'il y a une façon logique de construire ce nom, il est souvent plus simple de le retrouver dans le fichier [Reference.cs] qui est un fichier caché par défaut. Son contenu est le suivant :

```
1.  namespace Client.WsOperations {
2.   ...
3.      public partial class OperationsSoapClient :
    System.ServiceModel.ClientBase<Client.WsOperations.OperationsSoap>,
    Client.WsOperations.OperationsSoap {
4.
5.          public OperationsSoapClient() {
6.          }
7.  ...
8.          public double Ajouter(double a, double b) {
9.              ...
10.         }
11.
12.         public double Soustraire(double a, double b) {
13.             ...
14.         }
15.
16.         public double Multiplier(double a, double b) {
17.             ...
18.         }
19.
20.         public double Diviser(double a, double b) {
21.             ...
22.         }
23.     }
24. }
```

Les méthodes *Ajouter*, *Soustraire*, *Multiplier*, *Diviser* du service web distant seront accédées via les méthodes proxy de même nom (lignes 8, 12, 16, 20) du client de type *Client.WsOperations.OperationsSoapClient* (ligne 3).

Il nous reste à construire l'interface graphique :



n°	type	nom	rôle
1	ComboBox	comboBoxOperations	liste des opérations arithmétiques
2	TextBox	textBoxA	nombre a
3	TextBox	textBoxB	nombre b
4	Button	buttonExécuter	interroge le service web distant
5	Label	labelRésultat	le résultat de l'opération

Le code de [Form1.cs] est le suivant :

```
1.  using System;
2.  using System.Windows.Forms;
3.  using Client.WsOperations;
4.
5.  namespace ClientWsOperations {
6.   public partial class Form1 : Form {
7.      // tableau des opérations
8.      private string[] opérations = { "Ajouter", "Soustraire", "Multiplier", "Diviser" };
9.      // service web à contacter
10.     private OperationsSoapClient opérateur = new OperationsSoapClient();
11.
12.     // constructeur
13.     public Form1() {
14.         InitializeComponent();
15.     }
```

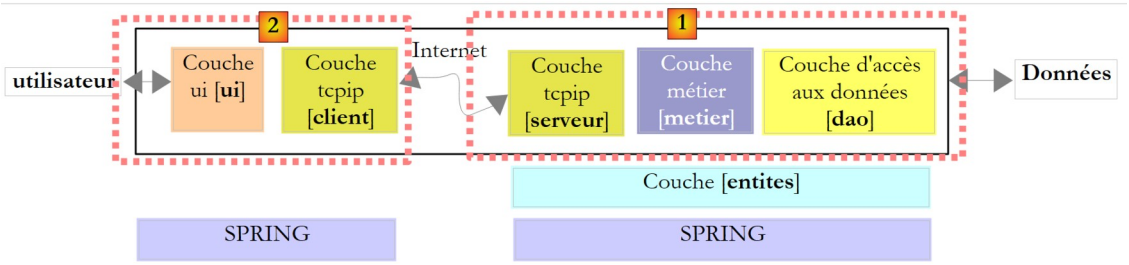
```
16.
17.     private void Form1_Load(object sender, EventArgs e) {
18.         // remplissage combo des opérations
19.         comboBoxOperations.Items.AddRange(opérations);
20.         comboBoxOperations.SelectedIndex = 0;
21.     }
22.
23.     private void buttonExécuter_Click(object sender, EventArgs e) {
24.         // vérification des paramètres a et b de l'opération
25.         textBoxMessage.Text = "";
26.         bool erreur = false;
27.         Double a = 0;
28.         if (!Double.TryParse(textBoxA.Text, out a)) {
29.             textBoxMessage.Text += "Nombre a erroné...";
30.         }
31.         Double b = 0;
32.         if (!Double.TryParse(textBoxB.Text, out b)) {
33.             textBoxMessage.Text += String.Format("{0}Nombre b erroné...", Environment.NewLine);
34.         }
35.         if (erreur) {
36.             return;
37.         }
38.         // exécution de l'opération
39.         Double c=0;
40.         try {
41.             switch (comboBoxOperations.SelectedItem.ToString()) {
42.                 case "Ajouter":
43.                     c=opérateur.Ajouter(a, b);
44.                     break;
45.                 case "Soustraire":
46.                     c=opérateur.Soustraire(a, b);
47.                     break;
48.                 case "Multiplier":
49.                     c=opérateur.Multiplier(a, b);
50.                     break;
51.                 case "Diviser":
52.                     c=opérateur.Diviser(a, b);
53.                     break;
54.             }
55.             // affichage résultat
56.             labelRésultat.Text = c.ToString();
57.         } catch (Exception ex) {
58.             textBoxMessage.Text = ex.Message;
59.         }
60.     }
61. }
62. }
```

- ligne 3 : l'espace de noms du client du service web distant
- ligne 10 : le client du service web distant est instancié en même temps que le formulaire
- lignes 17-21 : le combo des opérations est rempli lors du chargement initial du formulaire
- ligne 23 : exécution de l'opération demandée par l'utilisateur
- lignes 25-37 : on vérifie que les saisies a et b sont bien des nombres réels
- lignes 41-54 : un switch pour exécuter l'opération distante demandée par l'utilisateur
- lignes 43, 46, 49, 52 : c'est le client local qui est interrogé. De façon transparente, ce dernier interroge le service web distant.

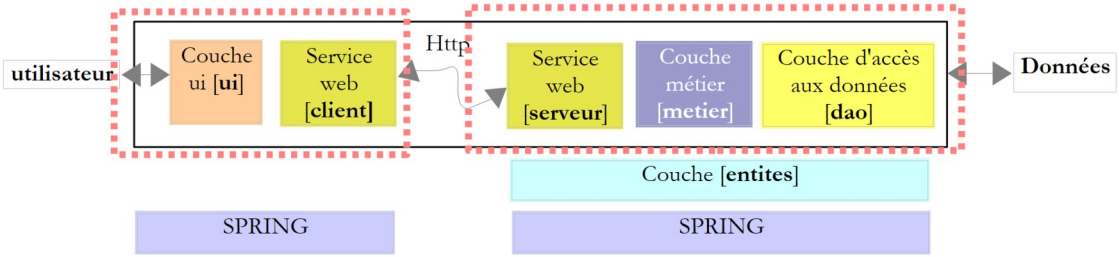
12.4 Un service web de calcul d'impôt

Nous reprenons l'application de calcul d'impôt maintenant bien connue. La dernière fois que nous avons travaillé avec, nous en avions fait un serveur Tcp distant qu'on pouvait appeler sur l'internet. Nous en faisons maintenant un service web.

L'architecture de la version 8 était la suivante :



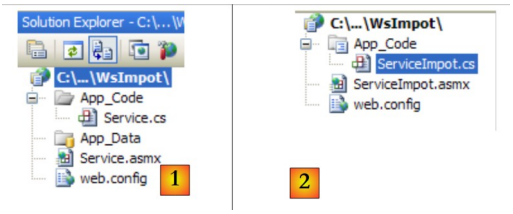
L'architecture de la version 9 sera similaire :



C'est une architecture similaire à celle de la version 8 étudiée au paragraphe 11.9.1, page 373 mais où le serveur et le client Tcp sont remplacés par un service web et son client proxy. Nous allons reprendre intégralement les couches [ui], [metier] et [dao] de la version 8.

12.4.1 La partie serveur

Nous construisons un projet de type service web avec Visual Web Developer :



- en [1], l'application web *WsImpot* générée
- en [2], l'application web *WsImpot* relookée de la façon suivante :
  - la page web [Service.asmx] a été renommée [ServiceImpot.asmx]
  - la classe [Service.cs] a été renommée [ServiceImpot .cs]

La page web [ServiceImpot.asmx] contient la ligne suivante :

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/ServiceImpot.cs" Class="ServiceImpot" %>
```

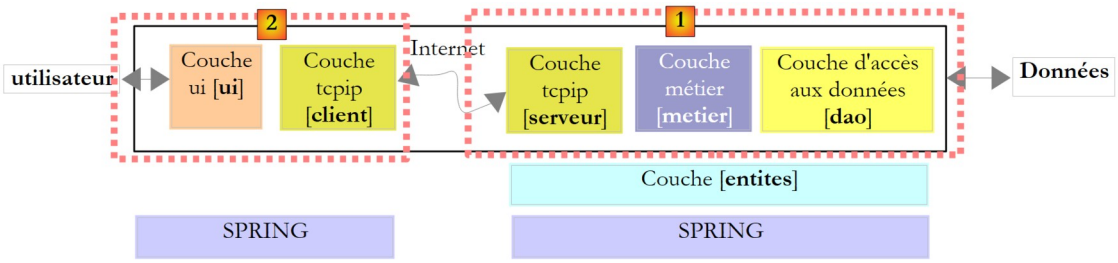
Le service web est assuré par classe [ServiceImpot.cs] suivante :

```
1.  using System.Web.Services;
2.
3.  [WebService(Namespace = "http://st.istia.univ-angers.fr/")]
4.  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
5.  public class ServiceImpot : System.Web.Services.WebService
6.  {
7.
8.      [WebMethod]
9.      public int CalculerImpot(bool marié, int nbEnfants, int salaire)
10.     {
11.         return 0;
12.     }
13.
```

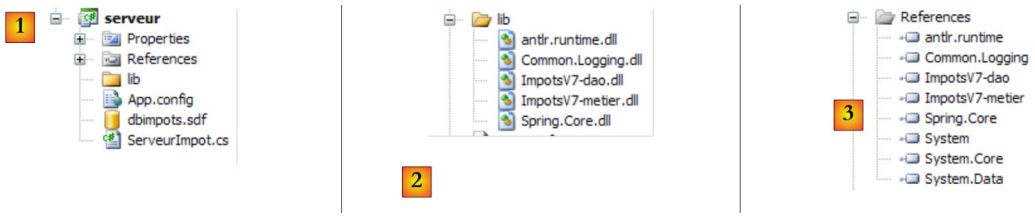
```
14. }
```

Le service web n'exposera que la méthode *CalculerImpot* de la ligne 9.

Revenons à l'architecture client / serveur de la version 8 :

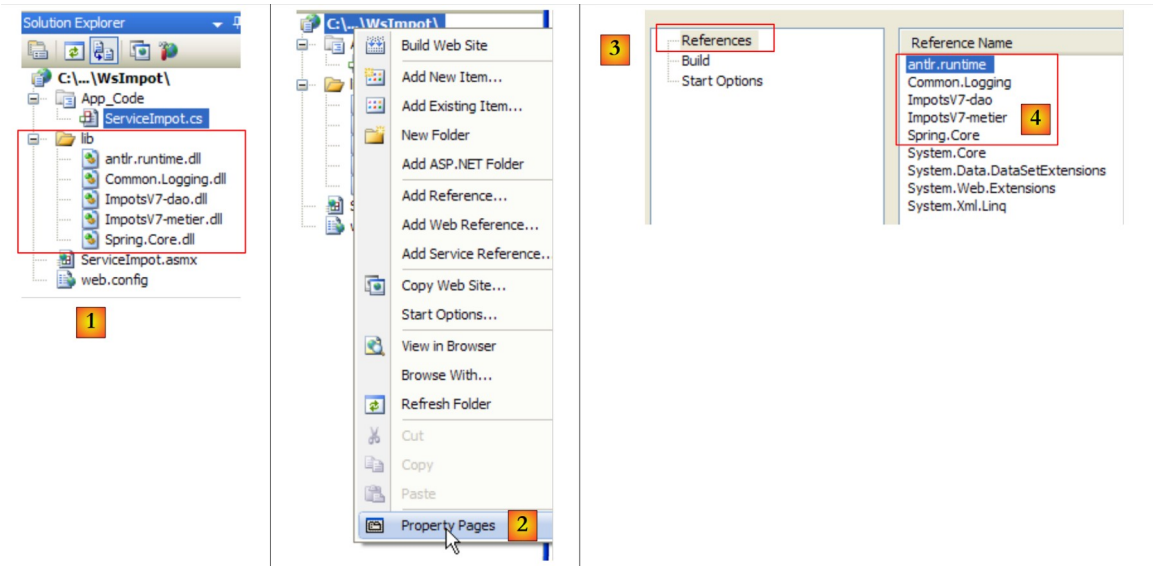


Le projet Visual studio du serveur [1] était le suivant :



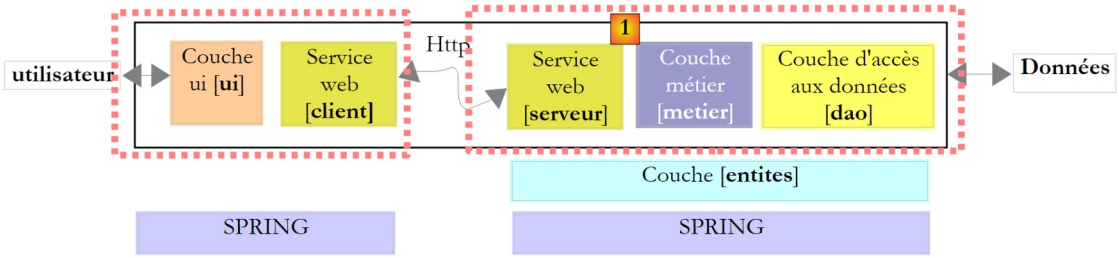
- en [1], le projet. On y trouvait les éléments suivants :
- [ServeurImpot.cs] : le serveur Tcp/Ip de calcul de l'impôt sous la forme d'une application console.
- [dbimpots.sdf] : la base de données SQL Server compact de la version 7 décrite au paragraphe 9.8.5, page 267.
- [App.config] : le fichier de configuration de l'application.
- en [2], le dossier [lib] contient les DLL nécessaires au projet :
- [ImpotsV7-dao] : la couche [dao] de la version 7
- [ImpotsV7-metier] : la couche [metier] de la version 7
- [antlr.runtime, CommonLogging, Spring.Core] pour Spring
- en [3], les références du projet

Les couches [metier] et [dao] de la version existent déjà : ce sont celles utilisées dans les versions 7 et 8. Elles sont sous forme de DLL que nous intégrons au projet de la façon suivante :

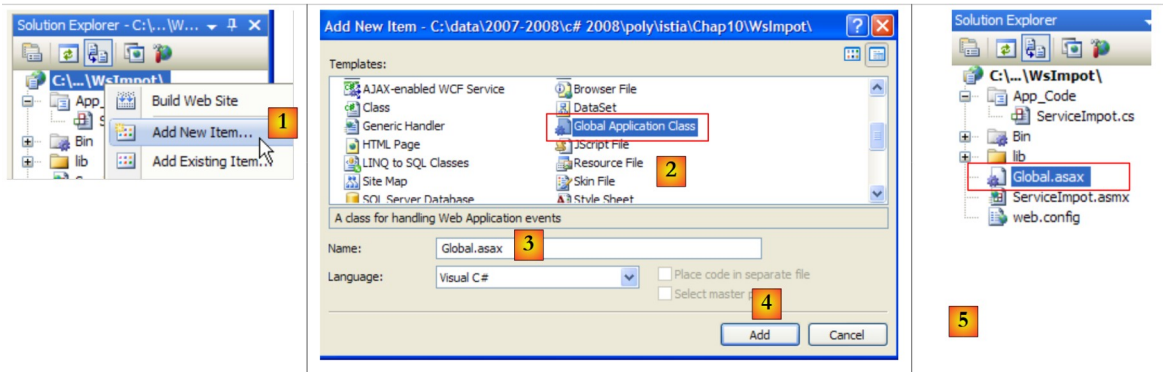


- en [1] le dossier [lib] du serveur de la version 8 a été recopié dans le projet du service web de la version 9.
- en [2] on modifie les propriétés de la page pour ajouter les DLL du dossier [lib] [4] aux références du projet [3].

Après cette opération, nous avons la totalité des couches nécessaires au serveur [1] ci-dessous :



Si les éléments du serveur [1], [serveur], [metier], [dao], [entites], [spring] sont bien tous présents dans le projet Visual Studio, il nous manque l'élément qui va les instancier au démarrage de l'application web. Dans la version 8, une classe principale ayant la méthode statique [Main] faisait le travail d'instancier les couches avec l'aide de Spring. Dans une application web, la classe capable de faire un travail analogue est la classe associée au fichier [Global.asax] :



- en [1], on ajoute un nouvel élément au projet web
- en [2], on choisit le type *Global Application Class*
- en [3], le nom proposé par défaut pour cet élément
- en [4], on valide l'ajout
- en [5], le nouvel élément a été intégré dans le projet

Regardons le contenu du fichier [Global.asax] :

```
1.  <%@ Application Language="C#" %>
2.
3.  <script runat="server">
4.
5.      void Application_Start(object sender, EventArgs e)
6.      {
7.          // Code that runs on application startup
8.      }
9.
10.     void Application_End(object sender, EventArgs e)
11.     {
12.         //  Code that runs on application shutdown
13.     }
14.
15.     void Application_Error(object sender, EventArgs e)
16.     {
17.         // Code that runs when an unhandled error occurs
18.     }
19.
20.     void Session_Start(object sender, EventArgs e)
21.     {
22.         // Code that runs when a new session is started
23.     }
24.
25.     void Session_End(object sender, EventArgs e)
26.     {
27.         // Code that runs when a session ends.
28.     }
```

```
29.
30. </script>
```

Le fichier est un mélange de balises à destination du serveur web (lignes 1, 3, 30) et de code C#. Cette méthode était la seule utilisée avec ASP, l'ancêtre de ASP.NET qui est la technologie actuelle de Microsoft pour la programmation web. Avec ASP.NET, cette méthode est toujours utilisable mais n'est pas la méthode par défaut. La méthode par défaut est la méthode dite du "CodeBehind" et qu'on a rencontrée dans les pages des services web, par exemple ici dans [ServiceImpot.asmx] :

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/ServiceImpot.cs" Class="ServiceImpot" %>
```

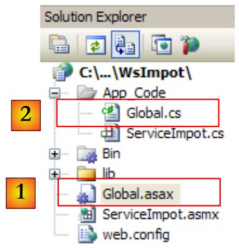
L'attribut *CodeBehind* précise où se trouve le code source de la page [ServiceImpot.asmx]. Sans cet attribut, le code source serait dans la page [ServiceImpot.asmx] avec une syntaxe analogue à celle trouvée dans [Global.asax]. Nous ne garderons pas le fichier [Global.asax] tel qu'il a été généré mais son code nous permet de comprendre à quoi il sert :

- la classe associée à *Global.asax* est instanciée au démarrage de l'application. Sa durée de vie est celle de l'application tout entière. Concrètement, elle ne disparaît que lorsque le serveur web est arrêté.
- la méthode *Application_Start* est exécutée ensuite. C'est la seule fois où elle le sera. Aussi l'utilise-t-on pour instancier des objets partagés entre tous les utilisateurs. Ces objets sont placés :
  - soit dans des champs statiques de la classe associée à *Global.asax*. Cette classe étant présente de façon permanente, toute requête de tout utilisateur peut y lire des informations.
  - soit le conteneur **Application**. Ce conteneur est lui aussi créé au démarrage de l'application et sa durée de vie est celle de l'application.
    - pour mettre une donnée dans ce conteneur, on écrit *Application["clé"]=valeur;*
    - pour la récupérer, on écrit *T valeur=(T)Application["clé"];* où T est le type de *valeur*.
- la méthode *Session_Start* est exécutée à chaque fois qu'un nouvel utilisateur fait une requête. Comment reconnaît-on un nouvel utilisateur ? Chaque utilisateur (un navigateur le plus souvent) reçoit à l'issue de sa première requête, un jeton de session qui est une chaîne de caractères unique pour chaque utilisateur. Ensuite, l'utilisateur renvoie le jeton de session qu'il a reçu à chaque nouvelle requête qu'il fait. Ceci permet au serveur web de le reconnaître. Au fil des différentes requêtes d'un même utilisateur, des données qui lui sont propres peuvent être mémorisées dans le conteneur **Session** :
  - pour mettre une donnée dans ce conteneur, on écrit *Session["clé"]=valeur;*
  - pour la récupérer, on écrit *T valeur=(T)Session["clé"];* où T est le type de *valeur*.

  La durée de vie d'une session est limitée par défaut à 20 mn d'inactivité de l'utilisateur (c.a.d. qu'il n'a pas renvoyé son jeton de session depuis 20 mn).
- la méthode *Application_Error* est exécutée lorsqu'une exception non gérée par l'application web remonte jusqu'au serveur web.
- les autres méthodes sont plus rarement utilisées.

Après ces généralités, à quoi peut nous servir *Global.asax* ? Nous allons utiliser sa méthode *Application_Start* pour initialiser les couches [metier], [dao] et [entites] contenues dans les DLL [ImpotsV7-metier, ImpotsV7-dao]. Nous utiliserons Spring pour les instancier. Les références des couches ainsi créées seront ensuite mémorisées dans des champs statiques de la classe associée à *Global.asax*.

Première étape, nous déportons le code C# de *Global.asax* dans une classe à part. Le projet évolue de la façon suivante :



En [1], le fichier [Global.asax] va être associé à la classe [Global.cs] [2] en ayant l'unique ligne suivante :

```
<%@ Application Language="C#" Inherits="WsImpot.Global"%>
```

L'attribut *Inherits="WsImpot.Global"* indique que la classe associée à *Global.asax* hérite de la classe *WsImpot.Global*. Cette classe est définie dans [Global.cs] de la façon suivante :

```
1.  using System;
2.  using Metier;
3.  using Spring.Context.Support;
4.  namespace WsImpot
```

```
5.  {
6.      public class Global : System.Web.HttpApplication
7.      {
8.          // couche métier
9.          public static IImpotMetier Metier;
10.
11.         // méthode exécutée au démarrage de l'application
12.         private void Application_Start(object sender, EventArgs e)
13.         {
14.             // instanciations couches [metier] et [dao]
15.             Metier = ContextRegistry.GetContext().GetObject("metier") as IImpotMetier;
16.         }
17.     }
18. }
```

- ligne 4 : l'espace de noms de la classe
- ligne 6 : la classe *Global*. On peut lui donner le nom qu'on veut. L'important est qu'elle dérive de la classe **System.Web.HttpApplication**.
- ligne 9 : un champ statique public qui va contenir une référence sur la couche [metier].
- ligne 12 : la méthode *Application_Start* qui va être exécutée au démarrage de l'application.
- ligne 15 : Spring est utilisé pour exploiter le fichier [web.config] dans lequel il va trouver les objets à instancier pour créer les couches [metier] et [dao]. Il n'y a aucune différence entre l'utilisation de Spring avec [App.config] dans une application windows, et celle de Spring avec [web.config] dans une application web. [web.config] et [App.config] ont par ailleurs la même structure. La ligne 15 range la référence de la couche [metier] dans le champ statique de la ligne 9, afin que cette référence soit disponible pour toutes les requêtes de tous les utilisateurs.

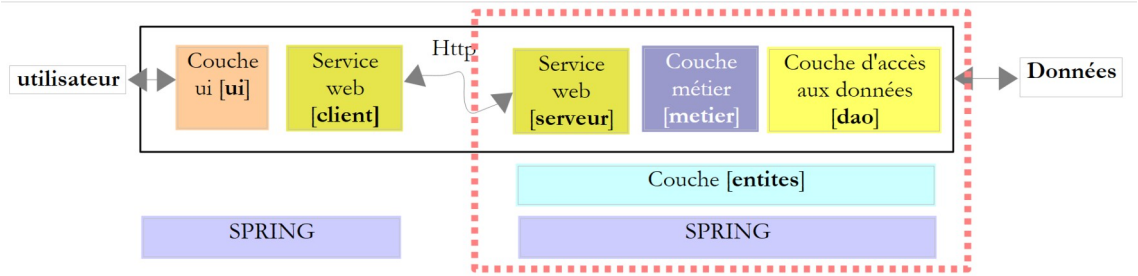
Le fichier [web.config] sera le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.    <configSections>
5.      <sectionGroup name="spring">
6.        <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.        <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
8.      </sectionGroup>
9.    </configSections>
10.
11.   <spring>
12.     <context>
13.       <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.       <object name="dao" type="Dao.DataBaseImpot, ImpotsV7-dao">
17.         <constructor-arg index="0" value="MySql.Data.MySqlClient"/>
18.         <constructor-arg index="1"
    value="Server=localhost;Database=bdimpots;Uid=admimpots;Pwd=mdpimpots;"/>
19.         <constructor-arg index="2" value="select limite, coeffr, coeffn from tranches"/>
20.       </object>
21.       <object name="metier" type="Metier.ImpotMetier, ImpotsV7-metier">
22.         <constructor-arg index="0" ref="dao"/>
23.       </object>
24.     </objects>
25.   </spring>
26. </configuration>
```

C'est le fichier [App.config] utilisée dans la version 7 de l'application et étudiée au paragraphe 9.8.4, page 265.

- lignes 16-20 : définissent une couche [dao] travaillant avec une base de données MySQL5. Cette base de données a été décrite au paragraphe 9.8.1, page 258.
- lignes 21-23 : définissent la couche [metier]

Revenons au puzzle du serveur :



Au démarrage de l'application, les couches [metier] et [dao] ont été instanciées. La durée de vie des couches est celle de l'application elle-même. Quand est instancié le service web ? En fait à chaque requête qu'on lui fait. A la fin de la requête, l'objet qui l'a servie est supprimé. Un service web est donc à première vue **sans état**. Il ne peut mémoriser des informations entre deux requêtes dans des champs qui lui appartiendraient. Il peut en mémoriser dans la session de l'utilisateur. Pour cela, les méthodes qu'il expose doivent être taguées avec un attribut spécial :

```
1.      [WebMethod(EnableSession=true)]
2.      public int CalculerImpot(bool marié, int nbEnfants, int salaire)
3.  ....
```

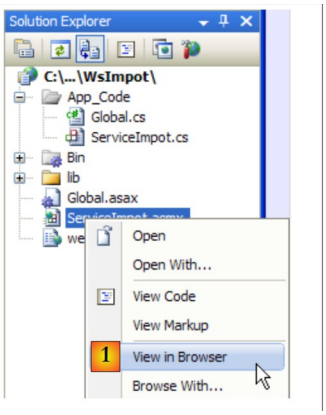
Ci-dessus, la ligne 1 autorise la méthode *CalculerImpot* à avoir accès au conteneur *Session* dont nous avons parlé précédemment. Nous n'aurons pas à utiliser cet attribut dans notre application. Le service web *WsImpot* sera donc instancié à chaque requête et sera sans état.

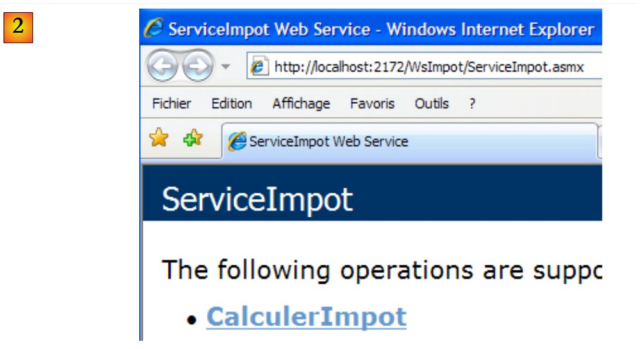
Nous pouvons maintenant écrire le code [ServiceImpot.cs] qui implémente le service web :

```
1.  using System.Web.Services;
2.  using WsImpot;
3.
4.  [WebService(Namespace = "http://st.istia.univ-angers.fr/")]
5.  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
6.  public class ServiceImpot : System.Web.Services.WebService
7.  {
8.
9.      [WebMethod]
10.     public int CalculerImpot(bool marié, int nbEnfants, int salaire)
11.     {
12.         return Global.Metier.CalculerImpot(marié, nbEnfants, salaire);
13.     }
14.
15. }
```

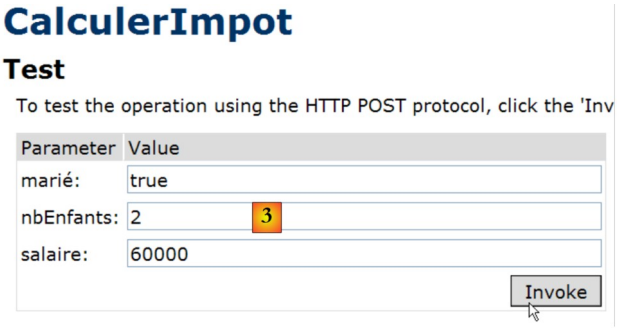
- ligne 10 : l'unique méthode du service web
- ligne 12 : on utilise la méthode *CalculerImpot* de la couche [metier]. Une référence de cette couche est trouvée dans le champ statique *Metier* de la class *Global*. Celle-ci appartient à l'espace de noms *WsImpot* (ligne 2).

Nous sommes prêts pour lancer le service web. Auparavant il faut lancer le SGBD MySQL5 afin que la base de données *bdimpots* soit accessible. Ceci fait, nous lançons [1] le service web :

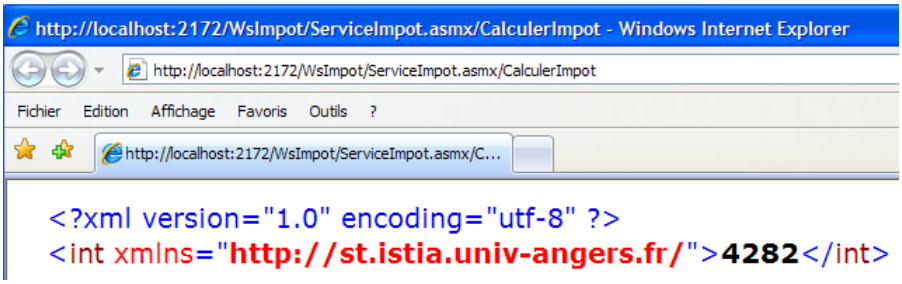




Le navigateur affiche alors la page [2]. Nous suivons le lien :

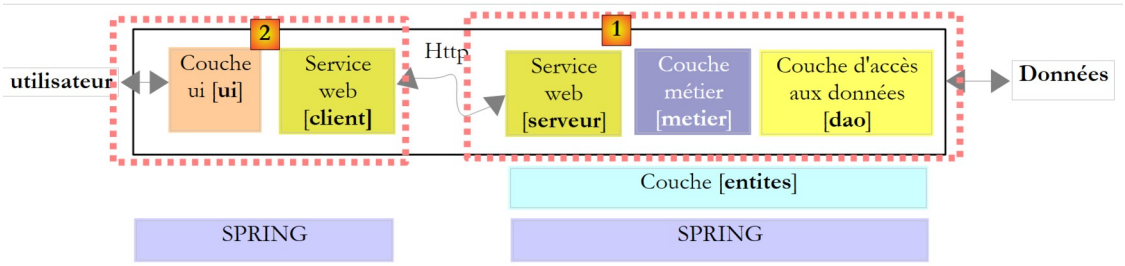


Nous donnons une valeur à chacun des trois paramètres de la méthode *CalculerImpot* et nous demandons l'exécution de la méthode. Nous obtenons le résultat suivant qui est correct :

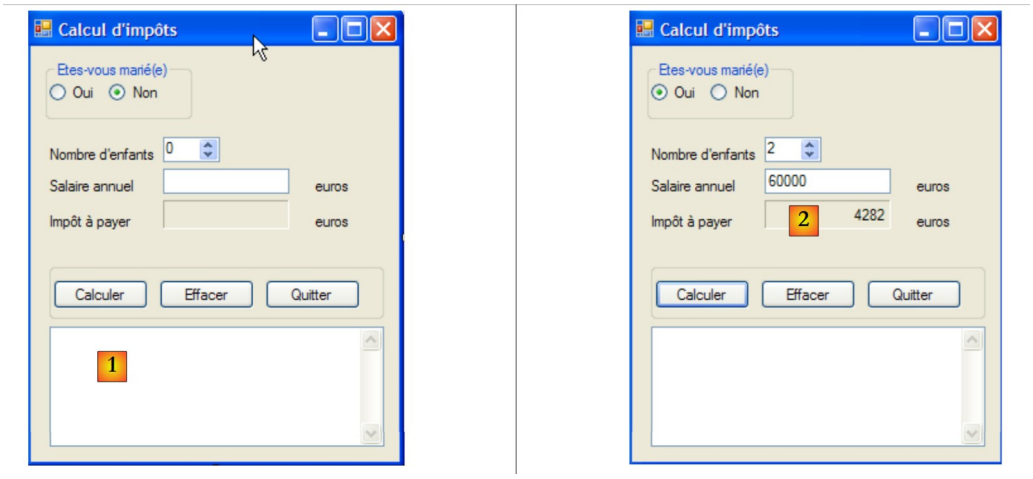


12.4.2 Un client graphique windows pour le service web distant

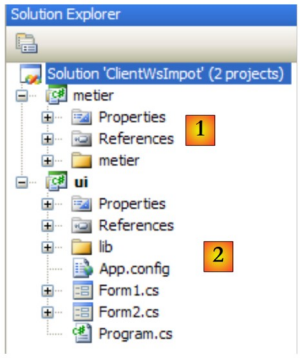
Maintenant que le service web a été écrit, nous passons au client. Revenons sur l'architecture de l'application client / serveur :



Il nous faut écrire le client [2]. L'interface graphique sera identique à celle de la version 8 :



Pour écrire la partie [client] de la version 9, nous allons partir de la partie [client] de la version 8, puis nous apporterons les modifications nécessaires. Nous dupliquons le projet Visual studio étudié au paragraphe 11.9.4.1, page 382, le renommons *ClientWsImpot* et le chargeons dans Visual Studio :



La solution Visual Studio de la version 8 était constituée de 2 projets :

- le projet [metier] [1] qui était un client Tcp du serveur Tcp de calcul d'impôt
- le projet [ui] [2] de l'interface graphique.

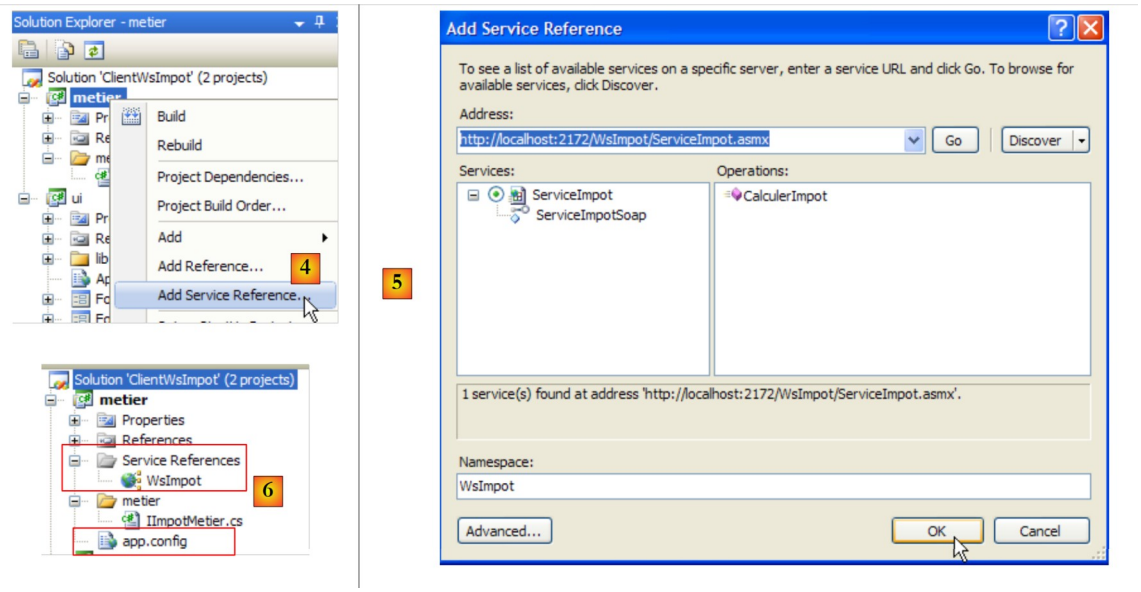
Les modifications à faire sont les suivantes :

- le projet [metier] doit être désormais le client d'un service web
- le projet [ui] doit référencer la DLL de la nouvelle couche [metier]
- la configuration de la couche [metier] dans [App.config] doit changer.

12.4.2.1 La nouvelle couche [metier]

- en [1], *IImpotMetier* est l'interface de la couche [metier] et *ImpotMetierTcp* son implémentation par un client Tcp
- en [2], nous supprimons l'implémentation *ImpotMetierTcp*. Nous devons créer une autre implémentation de l'interface *IImpotMetier* qui sera cliente d'un service web.

- en [3], nous nommons *Client* l'espace de noms par défaut du projet [metier]. La DLL qui sera générée s'appellera [ImpotsV9-metier.dll].



- en [4], nous créons une référence au service web *WsImpot*.
- en [5], nous le configurons et le validons.
- en [6], la référence au service web *WsImpot* a été créée et un fichier [app.config] a été généré.

Dans le fichier caché [Reference.cs] :

- l'espace de noms est *Client.WsImpot*
- la classe cliente s'appelle *ServiceImpotSoapClient*
- elle a une unique méthode de signature :

```
        public int CalculerImpot(bool marié, int nbEnfants, int salaire) ;
```

Il nous reste à implémenter l'interface *IImpotMetier* :

```
1.  namespace Metier {
2.   public interface IImpotMetier {
3.      int CalculerImpot(bool marié, int nbEnfants, int salaire);
4.   }
5.  }
```

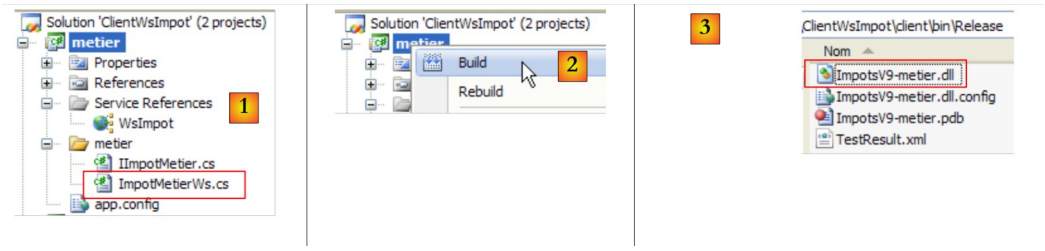
Nous l'implémentons avec la classe *ImpotMetierWs* suivante :

```
1.  using System.Net.Sockets;
2.  using System.IO;
3.  using Client.WsImpot;
4.
5.  namespace Metier {
6.   public class ImpotMetierWs : IImpotMetier {
7.
8.      // client du service web distant
9.      private ServiceImpotSoapClient client = new ServiceImpotSoapClient();
10.
11.     // calcul de l'impôt
12.     public int CalculerImpot(bool marié, int nbEnfants, int salaire) {
13.        return client.CalculerImpot(marié, nbEnfants, salaire);
14.     }
15.
16.  }
17. }
```

- ligne 6 : la classe *ImpotMetierWs* implémente l'interface *IImpotMetier*.
- ligne 9 : à la création d'une instance *ImpotMetierWs*, le champ *client* est initialisé avec une instance d'un client du service web de calcul d'impôt.
- ligne 12 : l'unique méthode de l'interface *IImpotMetier* à implémenter.

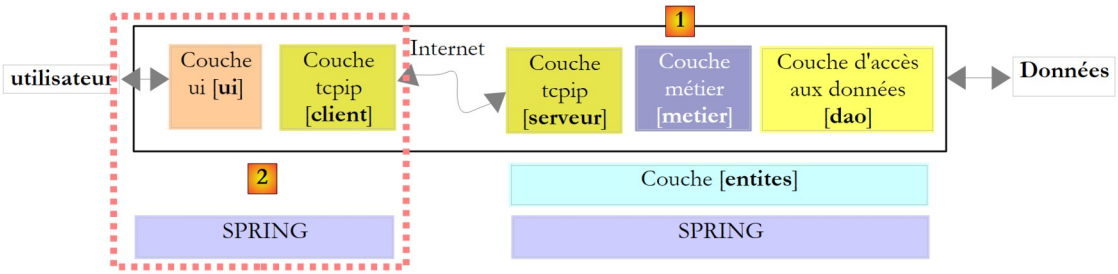
- ligne 13 : on utilise la méthode *CalculerImpot* du client du service web distant de calcul d'impôt. Au final, c'est la méthode *CalculerImpot* du service web distant qui sera interrogée.

On peut générer la DLL du projet :

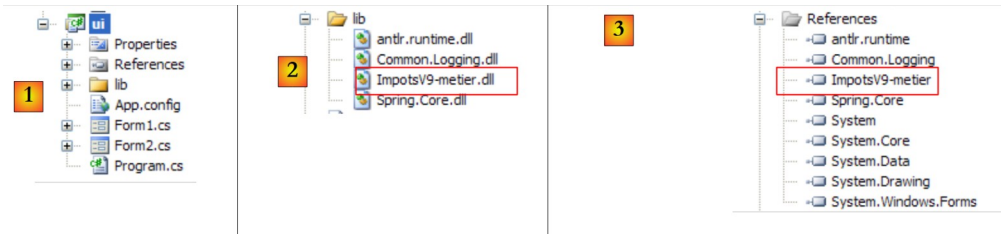


- en [1], le projet [client] dans son état final
- en [2], génération de la DLL du projet
- en [3], la DLL *ImpotsV9-metier.dll* est dans le dossier /bin/Release du projet.

12.4.2.2 La nouvelle couche [ui]



La couche [client] du client a été écrite. Il nous reste à écrire la couche [ui]. Revenons au projet Visual studio :



- en [1], le projet [ui] issu de la version 8
- en [2], la DLL *ImpotsV8-metier* de l'ancienne couche [metier] est remplacée par la DLL ImpotsV9-metier de la nouvelle couche
- en [3], la DLL *ImpotsV9-metier* est ajoutée aux références du projet.

Le second changement intervient dans [App.config]. Il faut se rappeler que ce fichier est utilisé par Spring pour instancier la couche [metier]. Comme celle-ci a changé, la configuration de [App.config] doit changer. D'autre part, [App.config] doit avoir la configuration permettant de joindre le service web distant de calcul d'impôt. Cette configuration a été générée dans le fichier [app.config] du projet [metier] lorsque la référence au service web distant y a été ajoutée.

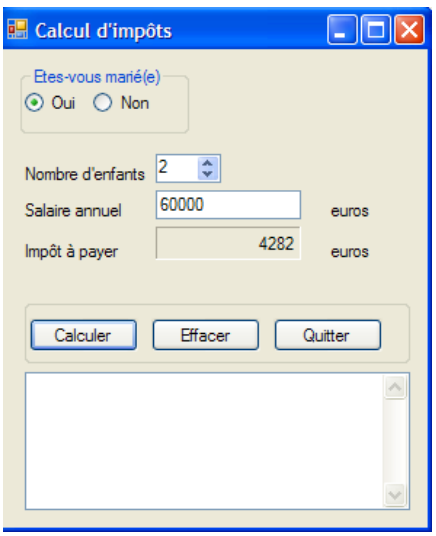
Le fichier [App.config] devient donc le suivant :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <configuration>
3.
4.   <configSections>
5.      <sectionGroup name="spring">
6.         <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
7.         <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
```

```
8.      </sectionGroup>
9.   </configSections>
10.
11.  <spring>
12.     <context>
13.        <resource uri="config://spring/objects" />
14.     </context>
15.     <objects xmlns="http://www.springframework.net">
16.        <object name="metier" type="Metier.ImpotMetierWs, ImpotsV9-metier">
17.        </object>
18.     </objects>
19.  </spring>
20.
21.  <!-- service web -->
22.  <system.serviceModel>
23.     <bindings>
24.        <basicHttpBinding>
25.           <binding name="ServiceImpotSoap" closeTimeout="00:01:00" openTimeout="00:01:00"
26.                 receiveTimeout="00:10:00" sendTimeout="00:01:00" allowCookies="false"
27.                 bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"
28.                 maxBufferSize="65536" maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
29.                 messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
30.                 useDefaultWebProxy="true">
31.              <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384"
32.                    maxBytesPerRead="4096" maxNameTableCharCount="16384" />
33.              <security mode="None">
34.                 <transport clientCredentialType="None" proxyCredentialType="None"
35.                       realm="" />
36.                 <message clientCredentialType="UserName" algorithmSuite="Default" />
37.              </security>
38.           </binding>
39.        </basicHttpBinding>
40.     </bindings>
41.     <client>
42.        <endpoint address="http://localhost:2172/WsImpot/ServiceImpot.asmx"
43.              binding="basicHttpBinding" bindingConfiguration="ServiceImpotSoap"
44.              contract="WsImpot.ServiceImpotSoap" name="ServiceImpotSoap" />
45.     </client>
46.  </system.serviceModel>
47. </configuration>
```

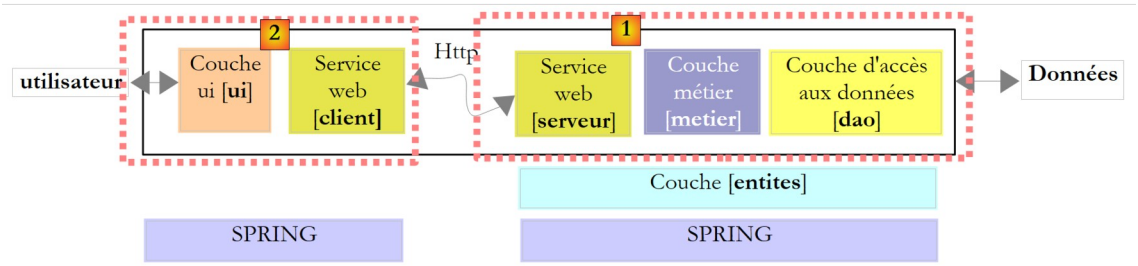
- lignes 15-18 : Spring n'instancie qu'un objet, la couche [metier]
- ligne 16 : la couche [metier] est instanciée par la classe [Metier.ImpotMetierWs] qui se trouve dans la DLL *ImpotsV9-metier*.
- lignes 22-46 : la configuration du client du service web distant. C'est un copier / coller du contenu du fichier [app.config] du projet [metier].

On est prêts. On exécute l'application par Ctrl-F5 (le service web doit être lancé, le SGBD MySQL5 doit être lancé, le port ligne 42 ci-dessus doit être le bon) :

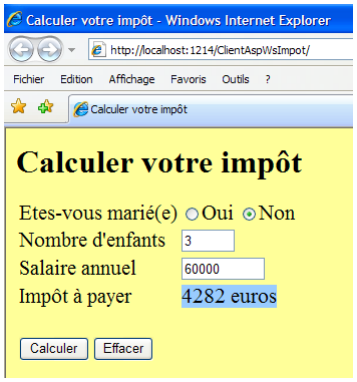


12.5 Un client web pour le service web de calcul d'impôt

Reprenons l'architecture de l'application client / serveur qui vient d'être écrite :



La couche [ui] ci-dessus était implémentée par un client graphique windows. Nous l'implémentons maintenant avec une interface web :



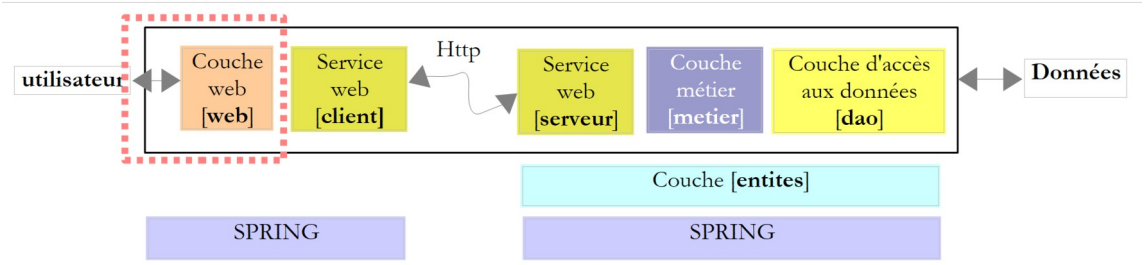
C'est un changement important pour les utilisateurs. Actuellement notre application client / serveur, version 9, peut servir plusieurs clients simultanément. C'est une amélioration vis à vis de la version 8 où elle ne servait qu'un client à la fois. La contrainte est que les utilisateurs désireux d'utiliser le service web de calcul d'impôt doivent disposer sur leur poste du client windows que nous avons écrit. Dans cette nouvelle version, que nous appellerons version 10, les utilisateurs pourront avoir accès, avec leur navigateur, au service web de calcul d'impôt.

Dans l'architecture ci-dessus :

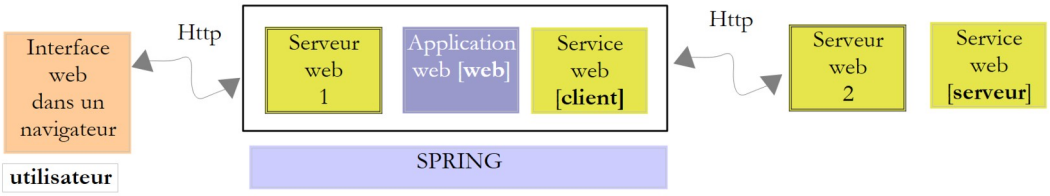
- le côté serveur ne bouge pas. Il reste ce qu'il est dans la version 9.
- côté client, la couche [client du service web] ne bouge pas. Elle a été encapsulée dans la DLL [ImpotsV9-metier]. Nous allons réutiliser cette DLL.
- au final le seul changement consiste à remplacer une interface graphique windows par une interface web.

Nous allons aborder de nouvelles notions de programmation web côté serveur. Le but de ce document n'étant pas d'enseigner la programmation web, nous essaierons d'expliquer la démarche qui va être suivie mais sans entrer dans les détails. Il y aura donc un côté un peu "magique" dans cette section. Il nous semble cependant intéressant de faire cette démarche pour montrer un nouvel exemple d'architecture multi-couches où l'une des couches est changée.

L'architecture de la version 10 est donc la suivante :



Nous avons déjà toutes les couches, sauf celle de la couche [web]. Pour mieux comprendre ce qui va être fait, nous avons besoin d'être plus précis sur l'architecture du client. Elle sera la suivante :

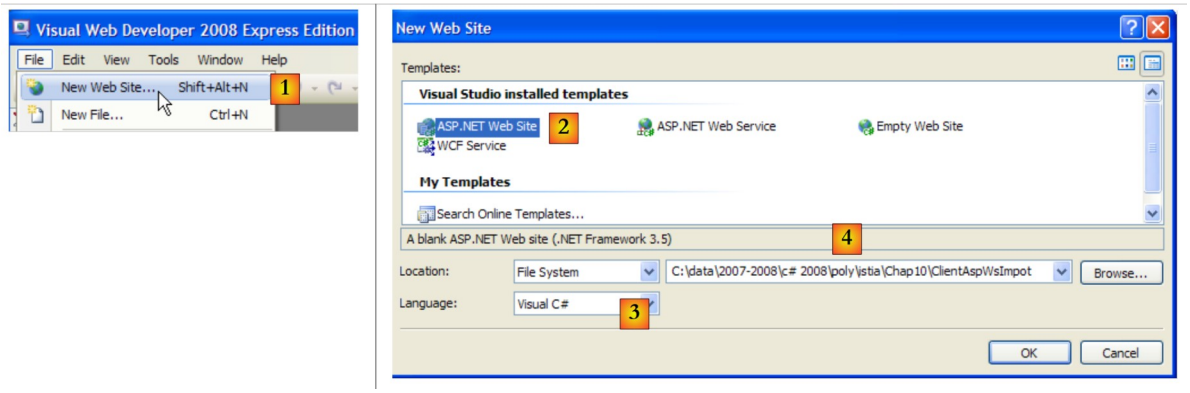


- l'utilisateur web a dans son navigateur un formulaire web
- ce formulaire est posté au serveur web 1 qui le fait traiter par la couche [web]
- la couche [web] aura besoin des services du client du service web distant, encapsulé dans [ImpotsV9-metier.dll].
- le client du service web distant communiquera avec le serveur web 2 qui héberge le service web distant.
- la réponse du service web distant va remonter jusqu'à la couche web du client qui va la mettre en forme dans une page qu'il va envoyer à l'utilisateur.

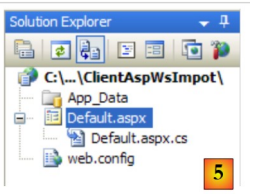
Notre travail ici est donc :

- de construire le formulaire web que verra l'utilisateur dans son navigateur
- d'écrire l'application web qui va traiter la demande de l'utilisateur et lui envoyer une réponse sous la forme d'une nouvelle page web. Celle-ci sera en fait la même que le formulaire dans lequel on aura rajouté le montant de l'impôt à payer
- d'écrire la "glue" qui fait que tout ça marche ensemble.

Tout ceci sera fait à l'aide d'un nouveau site web créé avec Visual Web Developer :

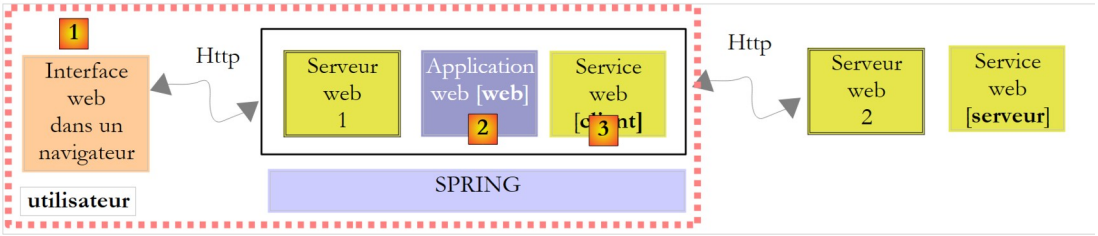


- [1] : prendre l'option **File / New Web Site**
- [2] : choisir une application de type **ASP.NET Web Site**
- [3] : choisir le langage de développement : **C#**
- [4] : indiquer le dossier où créer le projet



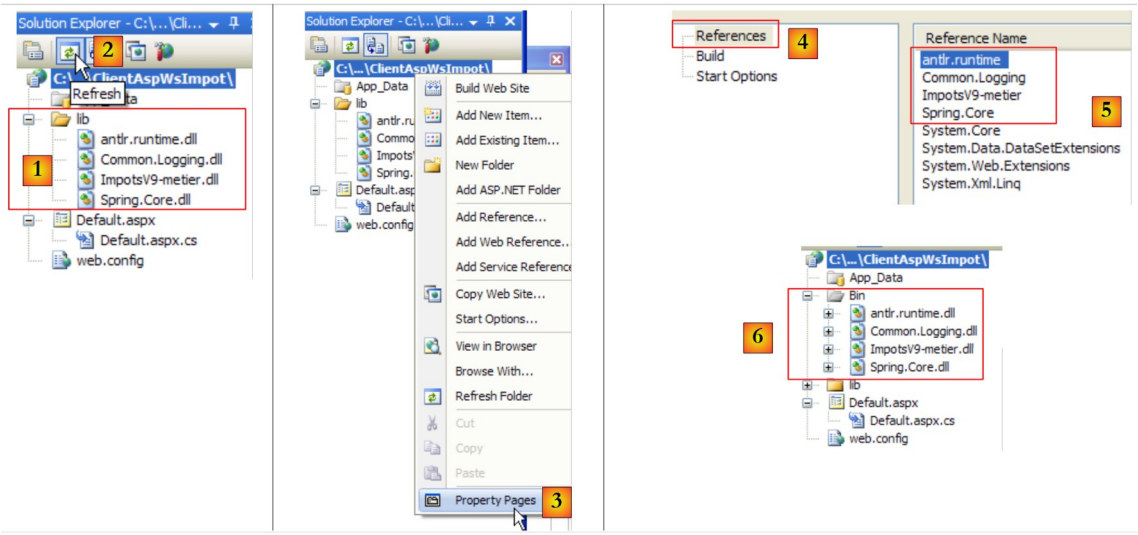
- [5] : le projet créé dans Visual Web Developer
  - [Default.aspx] est une page web appelée la page par défaut. C'est celle qui sera délivrée si on demande l'Url *http://.../ClientAspImpot* sans préciser de document. C'est cette page qui contiendra le formulaire de calcul de l'impôt que l'utilisateur verra dans son navigateur.
  - [Default.aspx.cs] est la classe associée à la page, celle qui va générer le formulaire envoyé à l'utilisateur puis le traiter lorsque celui-ci l'aura rempli et validé.
  - [web.config] est le fichier de configuration de l'application. Contrairement aux fois précédentes nous allons le garder.

Si nous revenons à l'architecture que nous devons construire :



- [1] va être implémentée par [Default.aspx]
- [2] va être implémentée par [Default.aspx.cs]
- [3] va être implémentée par la DLL [ImpotV9-metier]

Commençons par implémenter la couche [3]. Il y a plusieurs étapes :



- en [1], le dossier [lib] du client graphique windows version 9 est copié dans le dossier du projet web [ClientAspWsImpot]. Cela se fait avec l'explorateur windows. Pour faire apparaître ce dossier dans la solution Web Developer, il faut rafraîchir la solution avec le bouton [2].
- puis les ajouter aux références du projet [3,4,5]. Les Dll référencées sont automatiquement recopiées dans le dossier /bin du projet [6].

On a désormais les Dll nécessaires au fonctionnement de Spring et la couche client du service web distant est également implémentée. Si le code de celui-ci est bien présent, sa configuration reste à faire. Dans la version 9, il était configuré par le fichier [App.config] suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

 <configSections>
    <sectionGroup name="spring">
       <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
       <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
 </configSections>

 <spring>
    <context>
       <resource uri="config://spring/objects" />
    </context>
    <objects xmlns="http://www.springframework.net">
       <object name="metier" type="Metier.ImpotMetierWs, ImpotsV9-metier">
       </object>
    </objects>
 </spring>

 <!-- service web -->
 <system.serviceModel>
    <bindings>
       <basicHttpBinding>
...
       </basicHttpBinding>
    </bindings>
    <client>
       <endpoint address="http://localhost:2172/WsImpot/ServiceImpot.asmx"
             binding="basicHttpBinding" bindingConfiguration="ServiceImpotSoap"
             contract="WsImpot.ServiceImpotSoap" name="ServiceImpotSoap" />
    </client>
 </system.serviceModel>
</configuration>
```

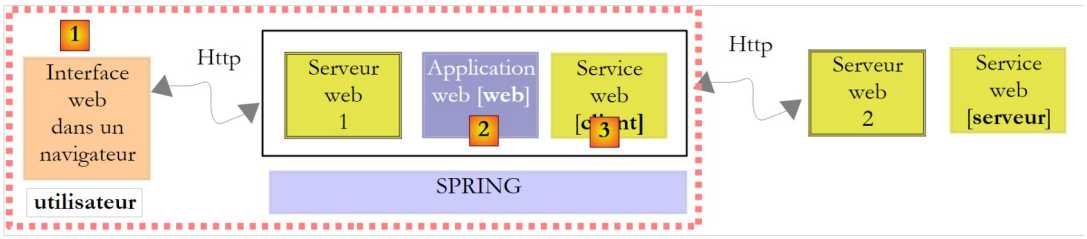
Nous reprenons cette configuration à l'identique et l'intégrons au fichier [web.config] de la façon suivante :

```
<?xml version="1.0"?>
<configuration>


    <configSections>
      <sectionGroup name="system.web.extensions"...>
...
      </sectionGroup>
      <!-- début section Spring -->
        <sectionGroup name="spring">
          <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
          <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
      <!-- fin section Spring -->
    </configSections>

  <!-- début configuration Spring -->
  <spring>
    <context>
      <resource uri="config://spring/objects" />
    </context>
    <objects xmlns="http://www.springframework.net">
      <object name="metier" type="Metier.ImpotMetierWs, ImpotsV9-metier">
      </object>
    </objects>
  </spring>
  <!-- fin configuration Spring -->

  <!-- début configuration client du service web distant -->
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
...
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:2172/WsImpot/ServiceImpot.asmx"
              binding="basicHttpBinding" bindingConfiguration="ServiceImpotSoap"
              contract="WsImpot.ServiceImpotSoap" name="ServiceImpotSoap" />
    </client>
  </system.serviceModel>
  <!-- fin configuration client du service web distant -->
```

```
43.
44.    <!-- autres configurations déjà présentes dans le web.config généré -->
45.  ...
46.  </configuration>
```

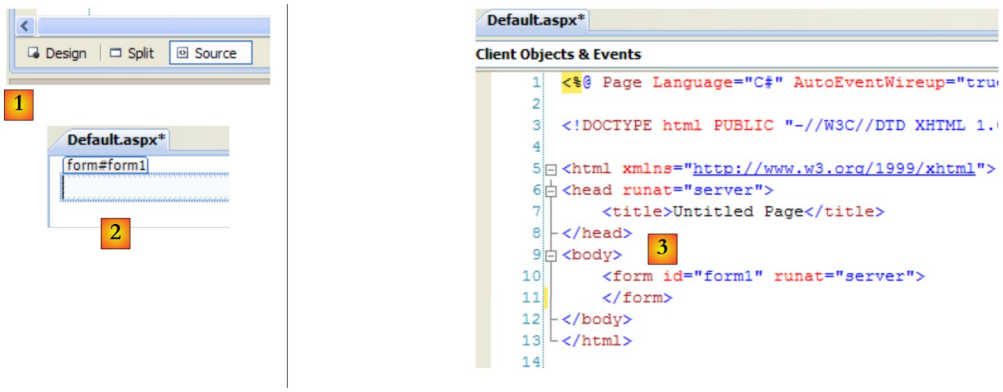
On notera que la ligne 37 référence le port du service web distant. Ce port peut changer puisque Visual Developer lance le service web sur un port aléatoire.

Revenons à l'architecture du client web que nous devons construire :



- [1] va être implémentée par [Default.aspx]
- [2] va être implémentée par [Default.aspx.cs]
- [3] a été implémentée par la DLL [ImpotV9-metier]

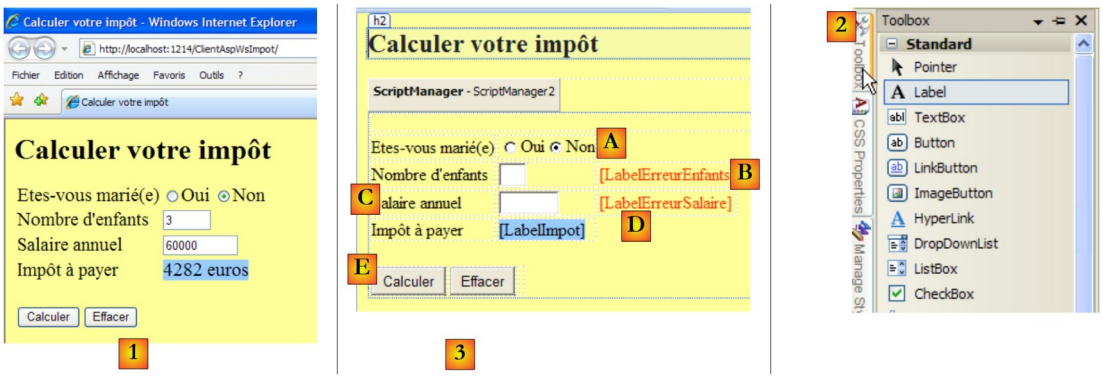
Nous venons d'implémenter la couche [3]. Nous passons à l'interface web [1] implémentée par la page [Default.aspx]. Double-cliquons sur la page [Default.aspx] pour passer en mode conception.



Il y a deux façons de construire une page web :

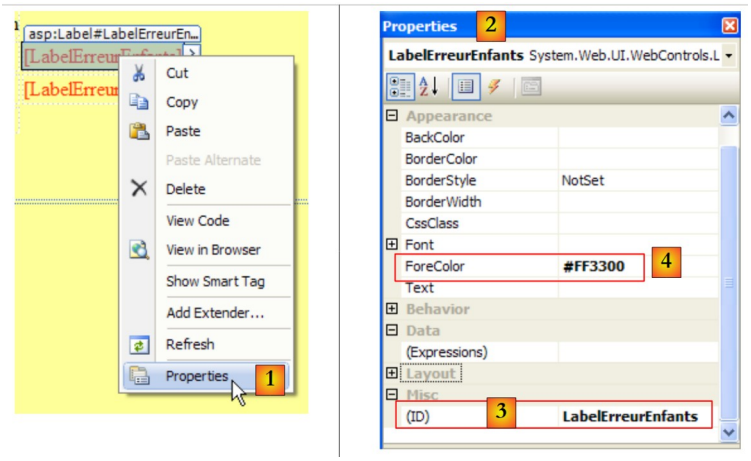
- graphiquement comme en [2]. Il faut alors choisir le mode [Design] en [1]. On trouvera cette barre de boutons en bas dans la barre d'état de l'éditeur de la page web.
- avec un langage de balises comme en [3]. Il faut alors choisir le mode [Source] en [1].

Les modes [Design] et [Source] sont bidirectionnels : une modification faite en mode [Design] se traduit par une modification en mode [Source] et vice-versa. Rappelons que le formulaire web à présenter dans le navigateur est le suivant :



- en [1], le formulaire affiché dans un navigateur
- en [2], les composants utilisés pour le construire
- en [3], la page de conception du formulaire. Il comprend les éléments suivants :
  - ligne A, deux boutons radio nommés *RadioButtonOui* et *RadioButtonNon*
  - ligne B, une zone de saisie nommée *TextBoxEnfants* et un label nommé *LabelErreurEnfants*
  - ligne C, une zone de saisie nommée *TextBoxSalaire* et un label nommé *LabelErreurSalaire*
  - ligne D, un label nommé *LabelImpot*
  - ligne E, deux boutons nommés *ButtonCalculer* et *ButtonEffacer*

Une fois un composant déposé sur la surface de conception, on a accès à ses propriétés :



- en [1], accès aux propriétés d'un composant
- en [2], la fiche des propriétés du composant [LabelErreurEnfants]
- en [3], (ID) est le nom du composant
- en [4], nous avons donné la couleur rouge aux caractères du label.

Il n'est pas suffisant de déposer des composants sur le formulaire puis de fixer leurs propriétés. Il faut également organiser leur disposition. Dans une interface graphique windows, cette disposition est absolue. On fait glisser le composant là où on veut qu'il soit. Dans une page web, c'est différent, plus complexe mais aussi plus puissant. Cet aspect ne sera pas abordé ici.

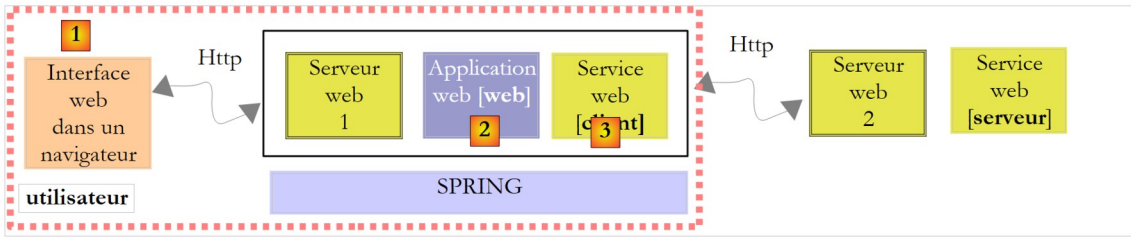
Le code source [Default.aspx] généré par cette conception est le suivant :

```
1.  <%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.    <title>Calculer votre impôt</title>
7.  </head>
8.  <body bgcolor="#ffff99">
9.    <h2>
10.     Calculer votre impôt</h2>
11.   <form id="form1" runat="server">
12.   <asp:ScriptManager ID="ScriptManager2" runat="server" EnablePartialRendering="true" />
13.   <asp:UpdatePanel runat="server" ID="UpdatePanelPam">
14.     <ContentTemplate>
15.       <div>
16.       </div>
17.       <table>
18.         <tr>
19.           <td>
20.             Etes-vous marié(e)
21.           </td>
22.           <td>
23.             <asp:RadioButton ID="RadioButtonOui" runat="server" GroupName="statut" Text="Oui" />
24.             <asp:RadioButton ID="RadioButtonNon" runat="server" GroupName="statut" Text="Non"
25.               Checked="True" />
26.           </td>
27.         </tr>
28.         <tr>
29.           <td>
30.             Nombre d&#39;enfants
31.           </td>
32.           <td>
```

```
33.             <asp:TextBox ID="TextBoxEnfants" runat="server" Columns="3"></asp:TextBox>
34.           </td>
35.           <td>
36.             <asp:Label ID="LabelErreurEnfants" runat="server" ForeColor="#FF3300"></asp:Label>
37.           </td>
38.         </tr>
39.         <tr>
40.           <td>
41.             Salaire annuel
42.           </td>
43.           <td>
44.             <asp:TextBox ID="TextBoxSalaire" runat="server" Columns="8"></asp:TextBox>
45.           </td>
46.           <td>
47.             <asp:Label ID="LabelErreurSalaire" runat="server" ForeColor="#FF3300"></asp:Label>
48.           </td>
49.         </tr>
50.         <tr>
51.           <td>
52.             Impôt à payer
53.           </td>
54.           <td>
55.             <asp:Label ID="LabelImpot" runat="server" BackColor="#99CCFF"></asp:Label>
56.           </td>
57.         </tr>
58.       </table>
59.       <br />
60.       <table>
61.         <tr>
62.           <td>
63.             <asp:Button ID="ButtonCalculer" runat="server" Text="Calculer" OnClick="ButtonCalculer_Click" />
64.           </td>
65.           <td>
66.             <asp:Button ID="ButtonEffacer" runat="server" Text="Effacer" OnClick="ButtonEffacer_Click" />
67.           </td>
68.           <td>
69.              
70.           </td>
71.         </tr>
72.       </table>
73.       </div>
74.     </ContentTemplate>
75.   </asp:UpdatePanel>
76.   </form>
77. </body>
78. </html>
```

On reconnaît les composants du formulaire aux lignes 23, 24, 33, 36, 44, 47, 55, 63 et 66. Le reste est essentiellement de la mise en forme.

Revenons à l'architecture que nous devons construire :



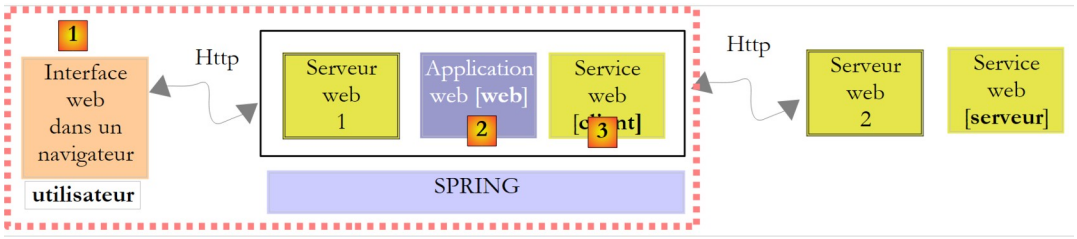
- [1] a été implémentée par [Default.aspx]
- [2] va être implémentée par [Default.aspx.cs]
- [3] a été implémentée par la DLL [ImpotV9-metier]

Les couches [1] et [3] sont désormais implémentées. Il nous reste à écrire la couche [2], celle qui génère le formulaire, l'envoie à l'utilisateur, le traite lorsque celui-ci le lui renvoie rempli, utilise la couche [3] pour le calcul de l'impôt, génère la page web de réponse à l'utilisateur et la lui renvoie. C'est le code [Default.aspx.cs] qui fait tout ce travail :

```
1.  using System;
2.  using WsImpot;
3.
4.  public partial class _Default : System.Web.UI.Page
5.  {
6.      protected void ButtonCalculer_Click(object sender, EventArgs e)
7.      {
8.    ...
9.      }
```

```
10.     protected void ButtonEffacer_Click(object sender, EventArgs e)
11.     {
12. ...
13.     }
14. }
```

C'est un code très proche de celui d'un formulaire windows classique. C'est l'avantage principal de la technologie ASP.NET : il n'y a pas de rupture entre le modèle de programmation windows et celui de la programmation web ASP.NET. Il faut simplement toujours se souvenir du schéma suivant :



Lorsque qu'en [1], l'utilisateur va cliquer sur le bouton [Calculer], la procédure *ButtonCalculer_Click* de la ligne 6 de [Default.aspx.cs] va être exécutée. Mais entre-temps :

- les valeurs du formulaire rempli vont transiter du navigateur au serveur web via le protocole Http
- le serveur ASP.NET va analyser la demande et la transférer à la page [Default.aspx]
- la page [Default.aspx] va être instanciée.
- ses composants (*RadioButtonOui*, *RadioButtonNon*, *TextBoxEnfants*, *TextBoxSalaire*, *LabelErreurEnfants*, *LabelErreurSalaire*, *LabelImpot*) vont être initialisés avec la valeur qu'ils avaient lorsque le formulaire a été envoyé initialement au navigateur grâce à un mécanisme appelé "ViewState".
- les valeurs postées vont être affectées à leurs composants (*RadioButtonOui*, *RadioButtonNon*, *TextBoxEnfants*, *TextBoxSalaire*). Ainsi si l'utilisateur a mis 2 comme nombre d'enfants, on aura *TextBoxEnfants.Text="2"*.
- si la page [Default.aspx] a une méthode [Page_Load], celle-ci sera exécutée
- la méthode [ButtonCalculer_Click] de la ligne 6 sera exécutée si c'est le bouton [Calculer] qui a été cliqué
- la méthode [ButtonEffacer_Click] de la ligne 10 sera exécutée si c'est le bouton [Effacer] qui a été cliqué

Entre le moment où l'utilisateur crée un événement dans son navigateur et celui où il est traité dans [Default.aspx.cs], il y a une grande complexité. Celle-ci est cachée et on peut faire comme si elle n'existait pas lorsqu'on écrit les gestionnaires d'événements de la page web. Mais on ne doit jamais oublier qu'il y a le réseau entre l'événement et son gestionnaire et qu'il n'est donc pas question de gérer des événements souris tels *Mouse_Move* qui provoqueraient des aller /retour client / serveur coûteux ...

Le code des gestionnaires des clics sur les boutons [Calculer] et [Effacer] est celui qu'on aurait écrit pour une application windows classique :

```
1.  protected void ButtonCalculer_Click(object sender, EventArgs e)
2.      {
3.          // vérification données
4.          int nbEnfants;
5.          bool erreur = false;
6.          if (!int.TryParse(TextBoxEnfants.Text.Trim(), out nbEnfants) || nbEnfants < 0)
7.          {
8.              LabelErreurEnfants.Text = "Valeur incorrecte...";
9.              erreur = true;
10.         }
11.         int salaire;
12.         if (!int.TryParse(TextBoxSalaire.Text.Trim(), out salaire) || salaire < 0)
13.         {
14.             LabelErreurSalaire.Text = "Valeur incorrecte...";
15.             erreur = true;
16.         }
17.         // erreur ?
18.         if (erreur) return;
19.         // on efface les éventuelles erreurs
20.         LabelErreurEnfants.Text = "";
21.         LabelErreurSalaire.Text = "";
22.         // état marital
23.         bool marié = RadioButtonOui.Checked;
24.         // calcul de l'impôt
25.         try
26.         {
27.             LabelImpot.Text = String.Format("{0} euros",Global.Metier.CalculerImpot(marié,
    nbEnfants, salaire));
28.         }
```

```
29.          catch (Exception ex)
30.          {
31.              LabelImpot.Text = ex.Message;
32.          }
33.      }
```

- pour comprendre ce code, il faut savoir
  - qu'au début de son exécution, le formulaire [Default.aspx] est tel que l'utilisateur l'a rempli. Ainsi les champs (*RadioButtonOui*, *RadioButtonNon*, *TextBoxEnfants*, *TextBoxSalaire*) ont les valeurs saisies par l'utilisateur.
  - qu'à l'issue de son exécution, la même page [Default.aspx] va être renvoyée à l'utilisateur. Cela est fait de façon automatique.

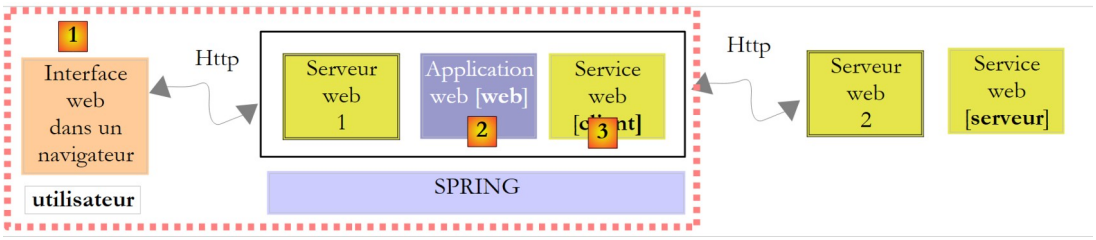
  La procédure *ButtonCalculer_Click* doit donc à partir des valeurs actuelles des champs (*RadioButtonOui*, *RadioButtonNon*, *TextBoxEnfants*, *TextBoxSalaire*) fixer la valeur de tous les champs (*RadioButtonOui*, *RadioButtonNon*, *TextBoxEnfants*, *TextBoxSalaire*, *LabelErreurEnfants*, *LabelErreurSalaire*, *LabelImpot*) de la nouvelle page [Default.aspx] qui va être renvoyée à l'utilisateur.

Il n'y a pas de difficulté particulière à ce code. Seule la ligne 27 mérite d'être expliquée. Elle utilise la méthode *CalculerImpot* d'un champ *Global.Metier* qui n'a pas été rencontré. Nous allons y revenir prochainement.

La méthode *ButtonEffacer_Click* est la suivante :

```
1.      protected void ButtonEffacer_Click(object sender, EventArgs e)
2.      {
3.          // raz formulaire
4.          TextBoxEnfants.Text = "";
5.          TextBoxSalaire.Text = "";
6.          LabelImpot.Text = "";
7.          LabelErreurEnfants.Text = "";
8.          LabelErreurSalaire.Text = "";
9.  }
```

Revenons à l'architecture que nous devons construire :



- [1] a été implémentée par [Default.aspx]
- [2] a été implémentée par [Default.aspx.cs]
- [3] a été implémentée par la DLL [ImpotV9-metier]

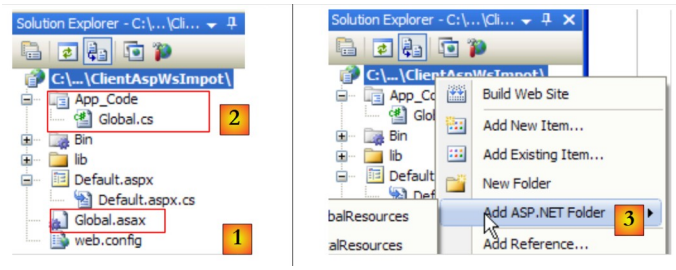
Il nous reste à mettre la "glue" autour de ces trois couches. Il s'agit essentiellement :

- d'instancier la couche [3] au démarrage de l'application
- d'en mettre une référence dans un endroit où la page web [Default.aspx.cs] pourra aller la chercher à chaque fois qu'elle sera instanciée et qu'on lui demandera de calculer l'impôt.

Ce n'est pas un problème nouveau. Il a déjà été rencontré dans la construction du service web distant et étudié au paragraphe 12.4.1, page 402. On sait que la solution consiste :

- à créer un fichier [Global.asax] associé à une classe [Global.cs]
- à instancier la couche [3] dans la méthode *Application_Start* de [Global.cs]
- à mettre la référence de la couche [3] dans un champ statique de la classe [Global.cs], car la durée de vie de cette classe est celle de l'application.

Aussi notre projet web évolue-t-il de la façon suivante :



- en [1], le fichier [Global.asax].
- en [2], le code [Global.cs] associé. Le dossier [App_Code] dans lequel se trouve ce fichier n'est pas présent par défaut dans la solution web. Utiliser [3] pour le créer.

Le fichier *Global.asax* est le suivant :

```
<%@ Application Language="C#" Inherits="WsImpot.Global"%>
```

Le code [Global.cs] est le suivant :

```
1.  using System;
2.  using Metier;
3.  using Spring.Context.Support;
4.  namespace WsImpot
5.  {
6.      public class Global : System.Web.HttpApplication
7.      {
8.          // couche métier
9.          public static IImpotMetier Metier;
10.
11.         // méthode exécutée au démarrage de l'application
12.         private void Application_Start(object sender, EventArgs e)
13.         {
14.             // instanciations couches [metier] et [dao]
15.             Metier = ContextRegistry.GetContext().GetObject("metier") as IImpotMetier;
16.         }
17.     }
18. }
```

- ligne 6 : la classe s'appelle *Global* et fait partie de l'espace de noms *WsImpot* (ligne 4). Aussi son nom complet est-il *WsImpot.Global* et c'est ce nom qu'il faut mettre dans l'attribut *Inherits* de *Global.asax*.
- ligne 6 : on sait que la classe associée à *Global.asax* doit obligatoirement dériver de la classe *System.Web.HttpApplication*.
- ligne 12 : la méthode *Application_Start* exécutée au démarrage de l'application web.
- ligne 15 : on intancie la couche [metier] (couche [3] de l'application en cours de construction) à l'aide de Spring et de la configuration suivante dans [web.config] :

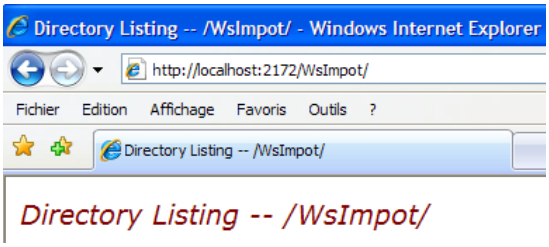
```
(a)    <!-- objets Spring -->
(b)   <spring>
(c)     <context>
(d)       <resource uri="config://spring/objects" />
(e)     </context>
(f)     <objects xmlns="http://www.springframework.net">
(g)       <object name="metier" type="Metier.ImpotMetierWS, ImpotsV9-metier">
(h)       </object>
(i)     </objects>
(j) </spring>
```

La classe [Metier.ImpotMetierWS] de la ligne (g) ci-dessus se trouve dans [ImpotsV9-metier.dll].

La référence de la couche [metier] créée est mise dans le champ statique de la ligne 9. C'est ce champ qui est utilisé dans la ligne 27 de la procédure *ButtonCalculer_Click* :

```
LabelImpot.Text = String.Format("{0} euros",Global.Metier.CalculerImpot(marié, nbEnfants, salaire));
```

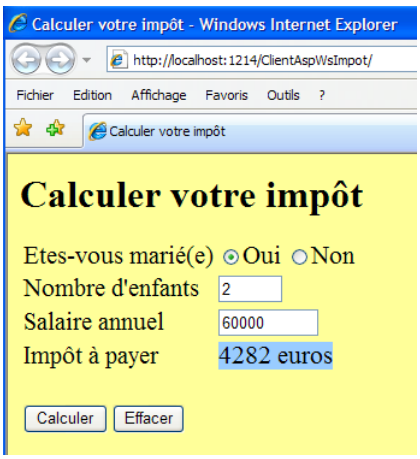
Nous sommes prêts pour un test. Il faut lancer le SGBD MySQL5, le service web distant et noter le port sur lequel il opère :



Ceci fait, il faut vérifier que dans le fichier [web.config] du client web, le port du service web distant est bon :

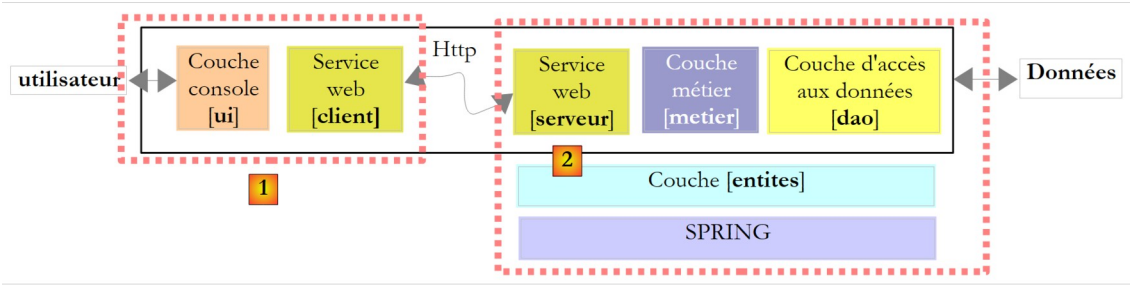
```
...
    <client>
      <endpoint address="http://localhost:2172/WsImpot/ServiceImpot.asmx"
                binding="basicHttpBinding" bindingConfiguration="ServiceImpotSoap"
                contract="WsImpot.ServiceImpotSoap" name="ServiceImpotSoap" />
    </client>
  </system.serviceModel>
```

Ceci fait, le client web du service web distant peut être lancé par Ctrl-F5 :



12.6 Un client console Java pour le service web de calcul d'impôt

Afin de montrer que les services web sont accessibles par des clients écrits dans un langage quelconque, nous écrivons un client Java console basique. L'architecture de l'application client / serveur sera la suivante :



- le client [1] sera écrit en Java

- le serveur [2] est celui écrit en C#

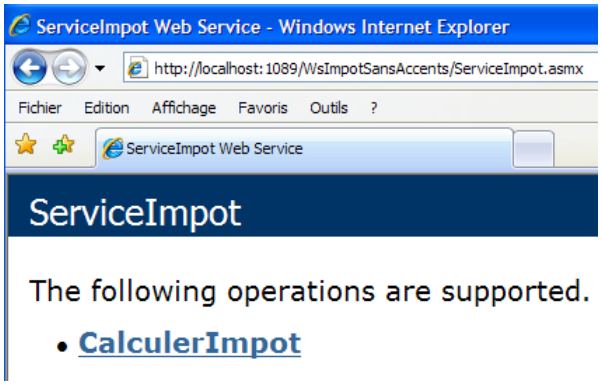
Tout d'abord, nous allons changer un détail dans notre service web de calcul d'impôt. Sa définition actuelle dans [ServiceImpot.cs] est la suivante :

```
1.  ...
2.  public class ServiceImpot : System.Web.Services.WebService
3.  {
4.
5.      [WebMethod]
6.      public int CalculerImpot(bool marié, int nbEnfants, int salaire)
7.      {
8.          return Global.Metier.CalculerImpot(marié, nbEnfants, salaire);
9.      }
10.
11. }
```

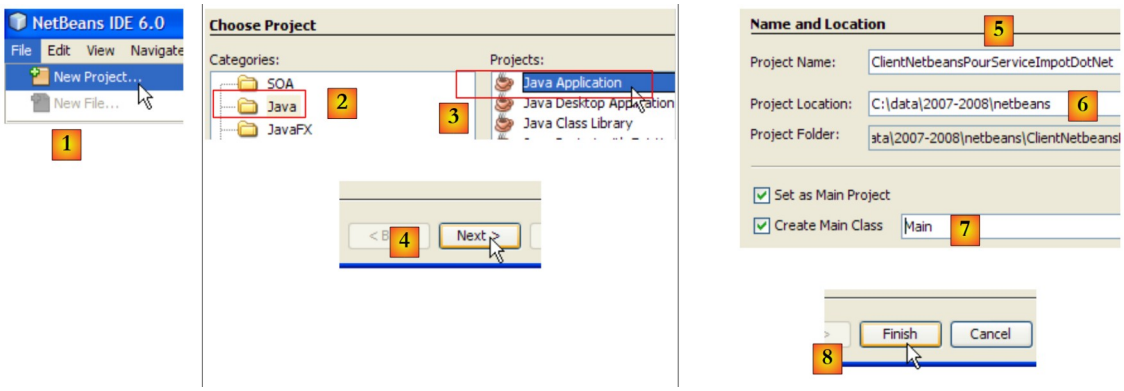
Les tests ont montré que l'accent du paramètre *marié* des lignes 6 et 8 pouvait être un problème dans l'interopérabilité Java / C#. Nous adopterons la nouvelle définition suivante :

```
1.  ...
2.  public class ServiceImpot : System.Web.Services.WebService
3.  {
4.
5.      [WebMethod]
6.      public int CalculerImpot(bool marie, int nbEnfants, int salaire)
7.      {
8.          return Global.Metier.CalculerImpot(marie, nbEnfants, salaire);
9.      }
10.
11. }
```

Ce service sera placé dans un nouveau projet Web Developer appelé *WsImpotsSansAccents*. Le service web aura alors l'Url [/WsImpotSansAccents/ServiceImpot.asmx].

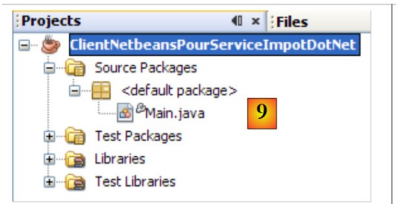


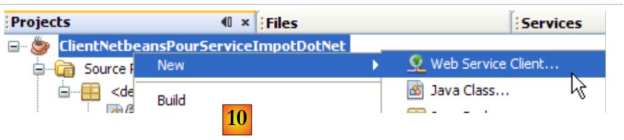
Pour écrire le client Java, nous utiliserons l'IDE Netbeans [http://www.netbeans.org/] :



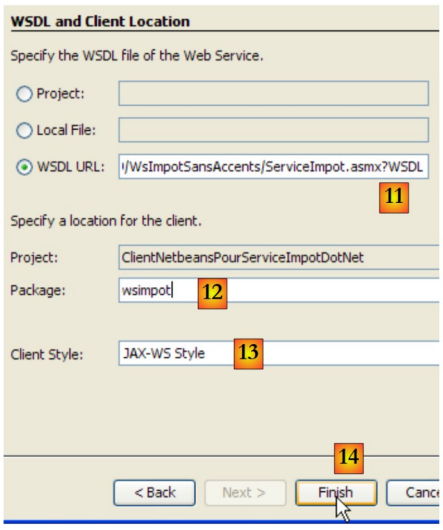
- en [1], créer un nouveau projet

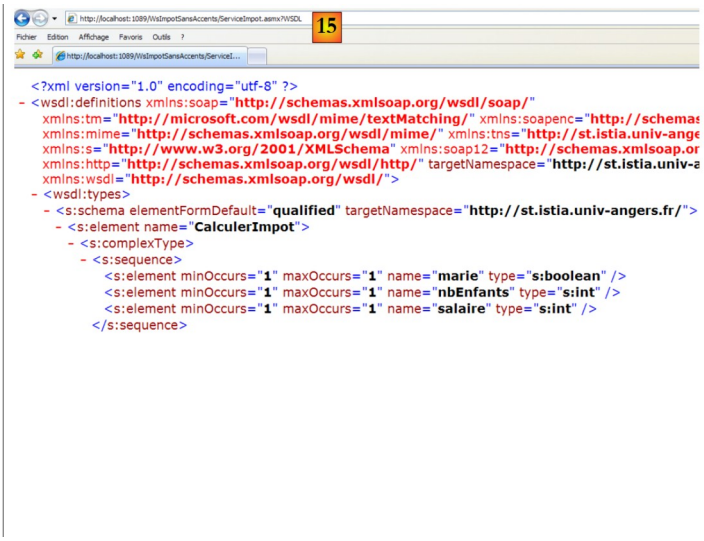
- en [2,3], choisir un projet *Java* de type *Java Application*.
- en [4], passer à l'étape suivante
- en [5], donner un nom au projet
- en [6], indiquer le dossier où un sous-dossier portant le nom du projet sera créé pour lui
- en [7], donner un nom à la classe qui va contenir la méthode *main* exécutée au démarrage de l'application
- en [8], terminer l'assistant





- en [9] : le projet Java généré
- en [10] : clic droit sur le projet pour générer le client du service web de calcul d'impôt

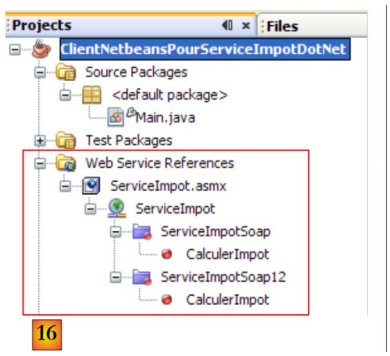


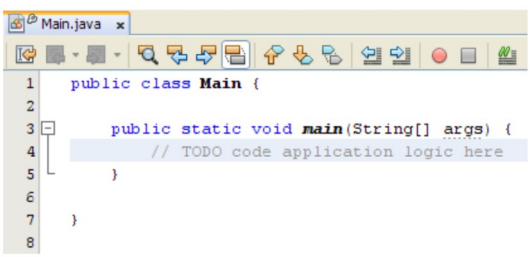


- en [11], l'Url du fichier décrivant le service web de calcul d'impôt :
  http://localhost:1089/WsImpotSansAccents/ServiceImpot.asmx?WSDL

  Cette Url est celle du service [ServiceImpot.asmx] à laquelle on ajoute le paramètre ?WSDL. Le document situé à cette Url décrit en langage Xml ce que sait faire le service [15]. C'est un élément standard d'un service web.

- en [12], le package (équivalent de l'espace de noms du C#) dans lequel mettre les classes qui vont être générées
- en [13], laisser la valeur par défaut
- en [14], terminer l'assistant

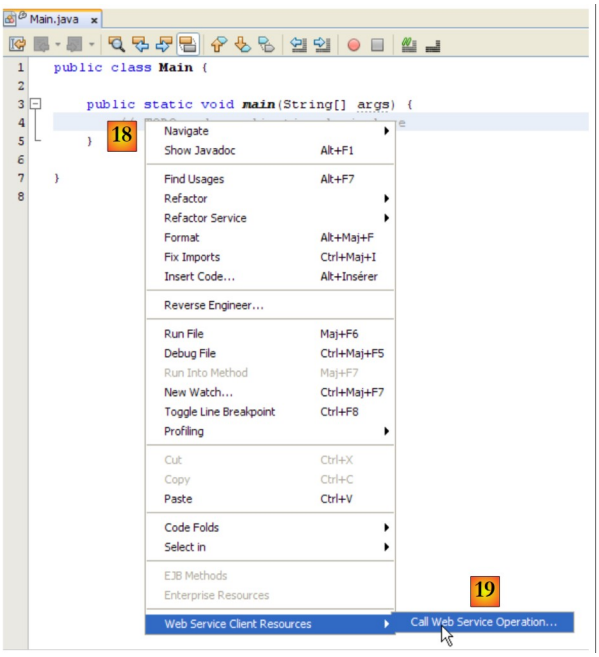


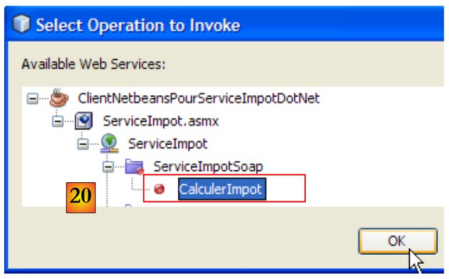


17

- en [16], le service web importé a été intégré au projet Java. Il supporte deux protocoles de communication *Soap* et *Soap12*.

- en [17], la classe [Main] dans laquelle nous allons utiliser le client généré





- en [18], nous allons insérer du code dans la méthode [main]. Mettre le curseur à l'endroit où le code doit être inséré, cliquer droit et prendre l'option [19]
- en [20], indiquer que vous voulez générer le code d'appel de la fonction *CalculerImpot* du service distant de calcul d'impôt puis faire Ok.

Le code généré dans [Main] est le suivant :

```
1.  public class Main {
2.
3.      public static void main(String[] args) {
4.          // TODO code application logic here
5.        try { // Call Web Service Operation
6.          wsimpot.ServiceImpot service = new wsimpot.ServiceImpot();
7.          wsimpot.ServiceImpotSoap port = service.getServiceImpotSoap();
8.          // TODO initialize WS operation arguments here
9.          boolean marie = false;
10.         int nbEnfants = 0;
11.         int salaire = 0;
12.         // TODO process result here
13.         int result = port.calculerImpot(marie, nbEnfants, salaire);
14.         System.out.println("Result = "+result);
15.       } catch (Exception ex) {
16.         // TODO handle custom exceptions here
17.       }
18.     }
19. }
```

Le code généré montre comment appeler à la fonction *CalculerImpot* du service distant de calcul d'impôt. Si on fait le parallèle avec ce qui a été vu en C#, la variable *port* de la ligne 7 est l'équivalent du client utilisé en C#. Nous ne commenterons pas davantage ce code. Nous le réaménageons de la façon suivante :

```
1.  import wsimpot.ServiceImpot;
2.  public class Main {
3.      public static void main(String[] args) {
4.        try {
5.          // on appelle la fonction CalculerImpot du service web
6.          System.out.println(String.format("Montant à payer : %d euros", new
    ServiceImpot().getServiceImpotSoap().calculerImpot(true, 2, 60000)));
7.        } catch (Exception ex) {
8.          System.out.println(String.format("L'erreur suivante s'est produite %s",ex.getMessage()));
9.        }
10.     }
11. }
```

- ligne 1 : nous importons la classe *ServiceImpot* qui représente le client généré par l'assistant.
- ligne 6 : nous appelons la méthode distante *CalculerImpot* en suivant la procédure indiquée dans le code généré dans main.

Les résultats obtenus dans la console à l'exécution (F6) sont les suivants :

```
1.  init:
2.  deps-jar:
3.  wsimport-init:
4.  wsimport-client-check-ServiceImpot.asmx:
5.  wsimport-client-ServiceImpot.asmx:
6.  wsimport-client-generate:
7.  wsimport-client-compile:
8.  Compiling 1 source file to C:\data\2007-
    2008\netbeans\ClientNetbeansPourServiceImpotDotNet\build\classes
9.  compile:
10. run:
11. Montant à payer : 4282 euros
12. BUILD SUCCESSFUL (total time: 7 seconds)
```

13 A suivre…

Il resterait des thèmes importants à couvrir. En voici trois :

1. LINQ (Language INtegrated Query) qui permet de requêter des collections d'objets, des flux XML et la base de données SQL Server avec un langage unifié similaire à SQL.
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.
4. .NET Remoting qui permet de faire du client / serveur avec un protocole propriétaire mais efficace
5. ...

Le point 3 est développé dans les documents suivants écrits en 2004 mais sans doute encore d'actualité en 2008 :

- Programmation ASP.NET / VB.NET, volume 1, [http://tahe.developpez.com/dotnet/aspnet/vol1/]
- Programmation ASP.NET / VB.NET , volume 2, [http://tahe.developpez.com/dotnet/aspnet/vol2/]

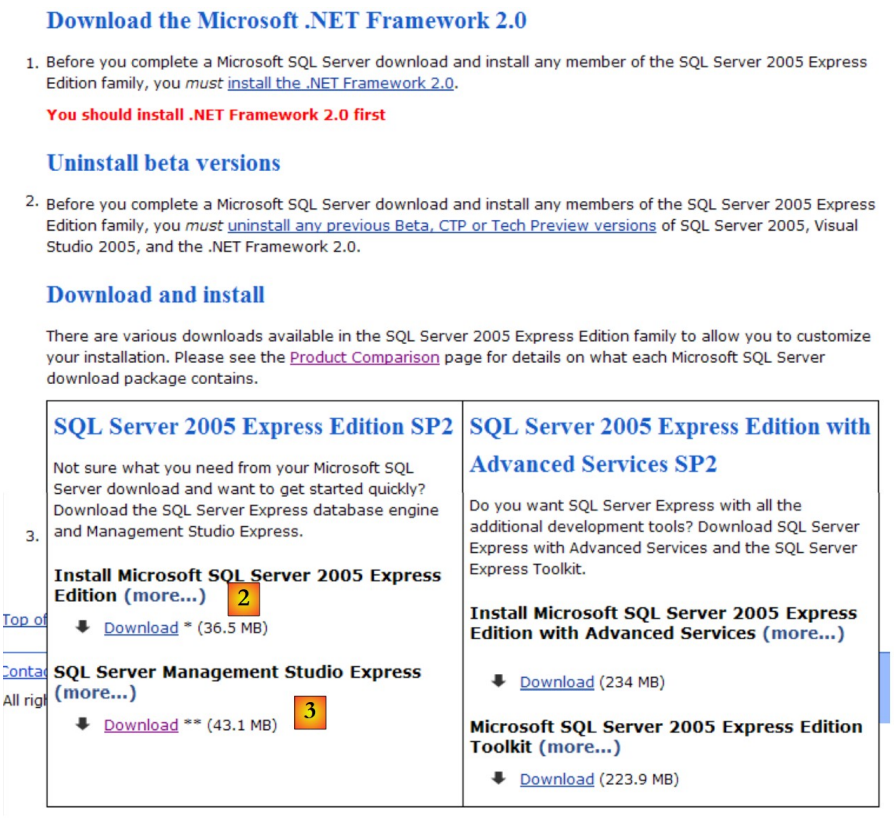
Les exemples de ces documents sont en VB.NET.

14 Annexes

1.1 Le SGBD SQL Server Express 2005

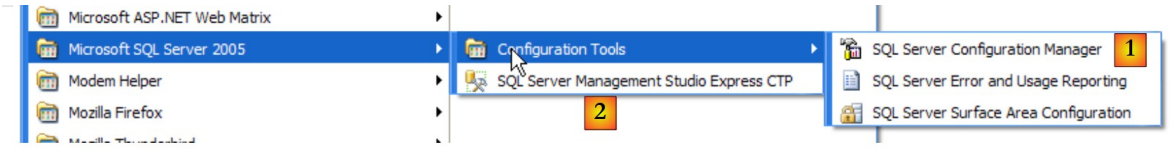
1.1.1 Installation

Le SGBD SQL Server Express 2005 est disponible à l'url [http://msdn.microsoft.com/vstudio/express/sql/download/] :



- en [1] : d'abord télécharger et installer la plate-forme .NET 2.0
- en [2] : puis installer et télécharger SQL Server Express 2005
- en [3] : puis installer et télécharger SQL Server Management Studio Express qui permet d'administrer SQL Server

L'installation de SQL Server Express donne naissance à un dossier dans [Démarrer / Programmes] :



- en [1] : l'application de configuration de SQL Server. Permet également de lancer / arrêter le serveur
- en [2] : l'application d'administration du serveur

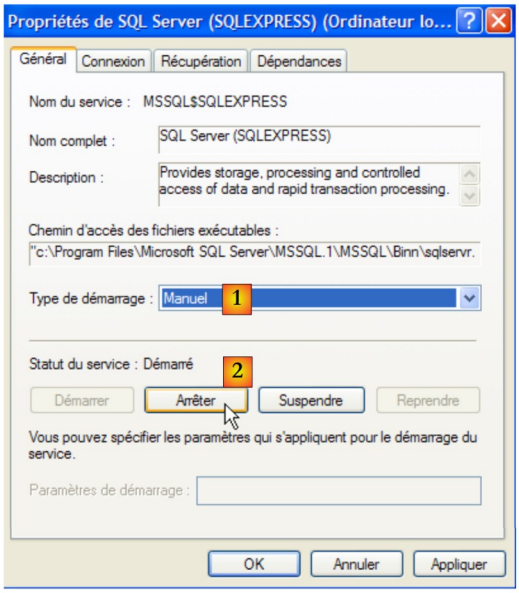
1.1.2 Lancer / Arrêter SQL Server

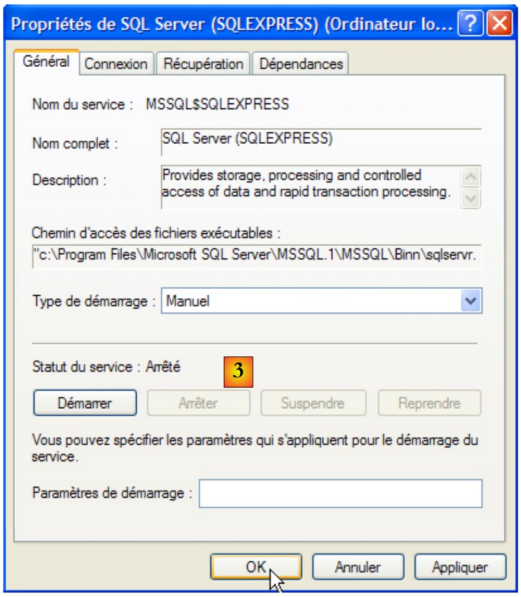
Comme pour les SGBD précédents, SQL server Express a été installé comme un service windows à démarrage automatique. Nous changeons cette configuration :

[Démarrer / Panneau de configuration / Performances et maintenance / Outils d'administration / Services] :

- en [1] : nous double-cliquons sur [Services]
- en [2] : on voit qu'un service appelé [SQL Server] est présent, qu'il est démarré [3] et que son démarrage est automatique [4].
- en [5] : un autre service lié à SQL Server, appelé "SQL Server Browser" est également actif et à démarrage automatique.

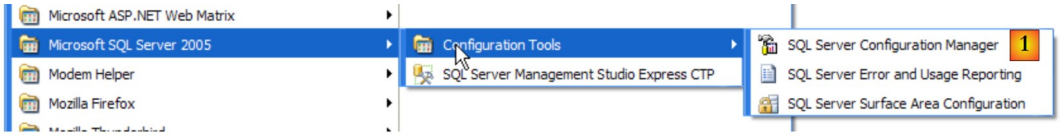
Pour modifier ce fonctionnement, nous double-cliquons sur le service [SQL Server] :

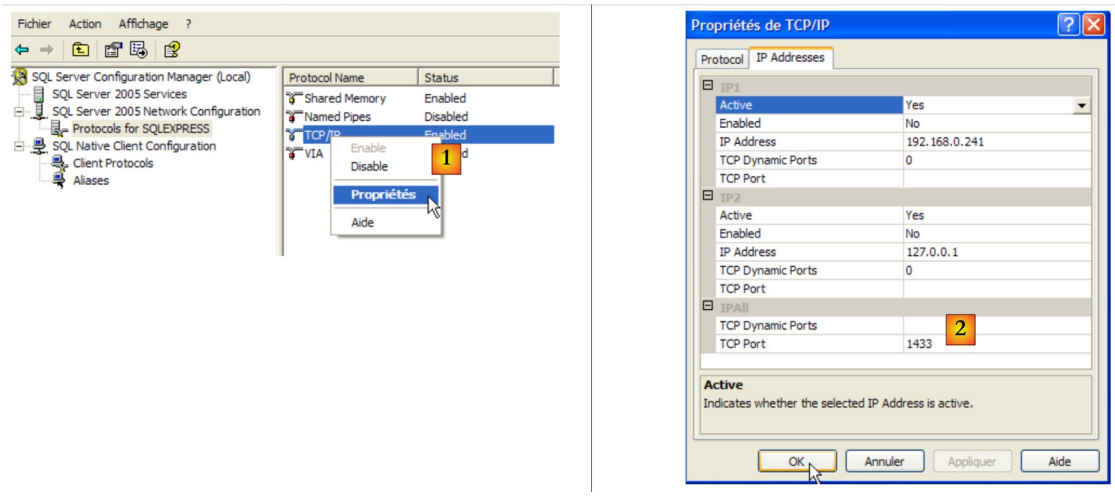




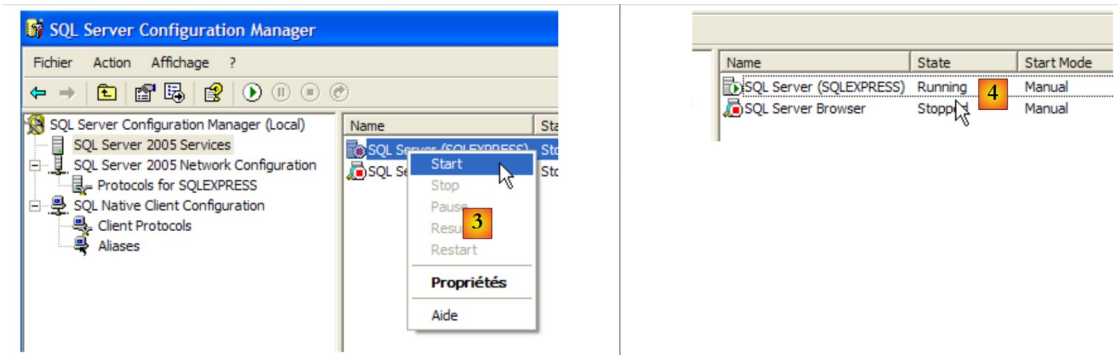
- en [1] : on met le service en démarrage manuel
- en [2] : on l'arrête
- en [3] : on valide la nouvelle configuration du service

On procèdera de même avec le service [SQL Server Browser] (cf [5] plus haut). Pour lancer et arrêter manuellement le service SQL Server 2005, on pourra utiliser l'application [1] du dossier [SQL server] :





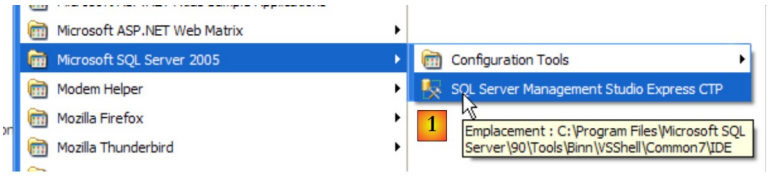
- en [1] : s'assurer que le protocole TCP/IP est actif (enabled) puis passer aux propriétés du protocole.
- en [2] : dans l'onglet [IP Addresses], option [IPAll] :
  - le champ [TCP Dynamic ports] est laissé vide
  - le port d'écoute du serveur est fixé à 1433 dans [TCP Port]

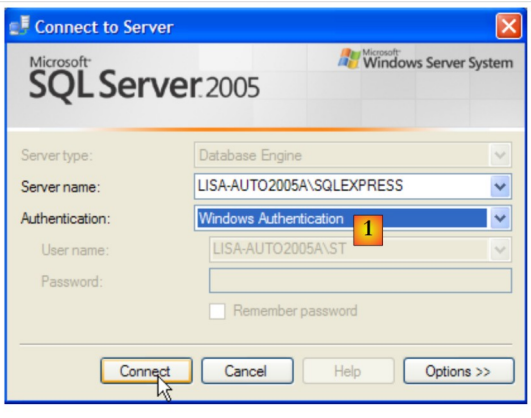


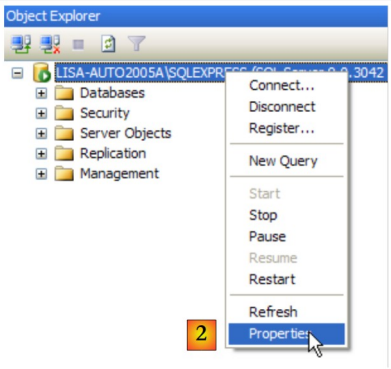
- en [3] : un clic droit sur le service [SQL Server] donne accès aux options de démarrage / arrêt du serveur. Ici, on le lance.
- en [4] : SQL Server est lancé

1.1.3 Création d'un utilisateur jpa et d'une base de données jpa

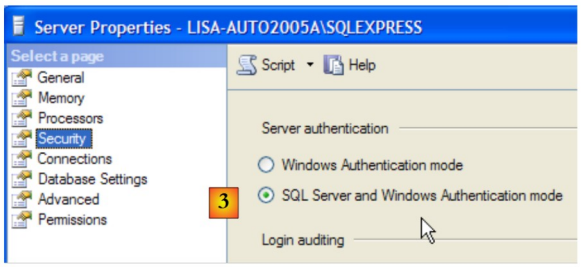
Lançons le SGBD comme indiqué ci-dessus, puis l'application d'administration [1] via le menu ci-dessous :

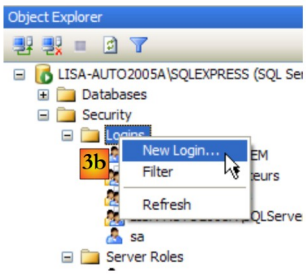




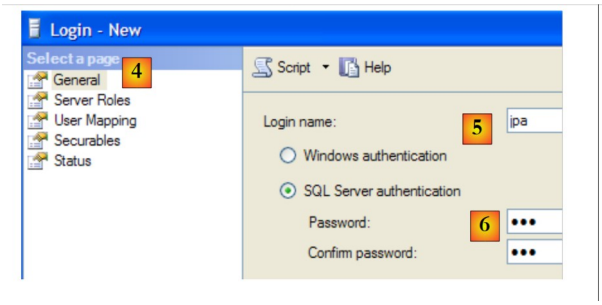


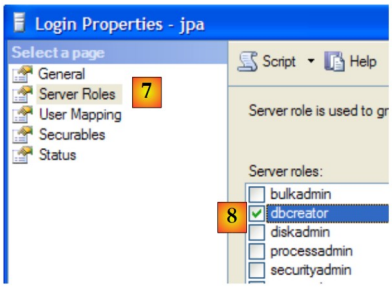
- en [1] : on se connecte à SQL Server en tant qu'**administrateur** Windows
- en [2] : on configure les propriétés de la connexion





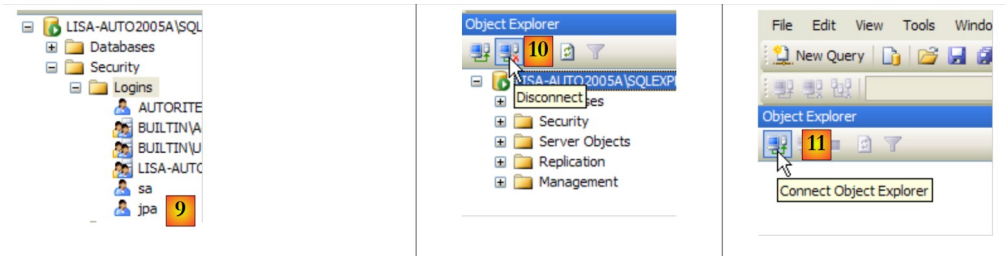
- en [3] : on autorise un mode mixte de connexion au serveur : soit avec un login windows (un utilisateur windows), soit avec un login SQL Server (compte défini au sein de SQL Server, indépendant de tout compte windows).
- en [3b] : on crée un utilisateur SQL Server



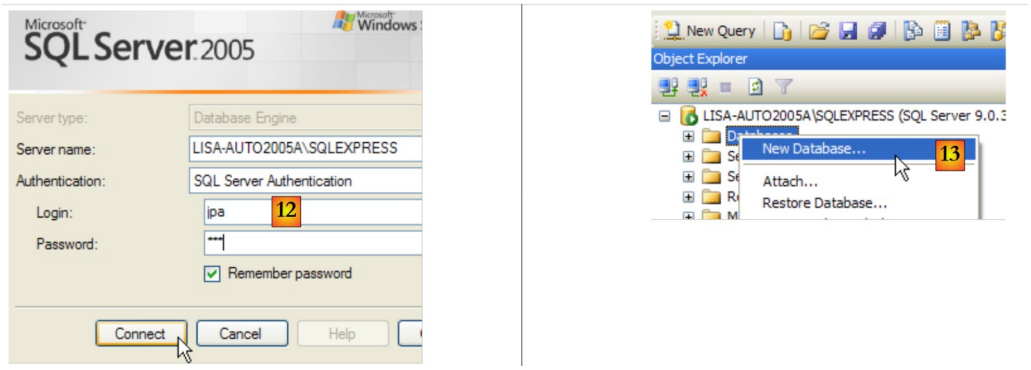


- en [4] : option [General]
- en [5] : le login
- en [6] : le mot de passe (**jpa** ici)
- en [7] : option [Server Roles]
- en [8] : l'utilisateur **jpa** aura le droit de créer des bases de données

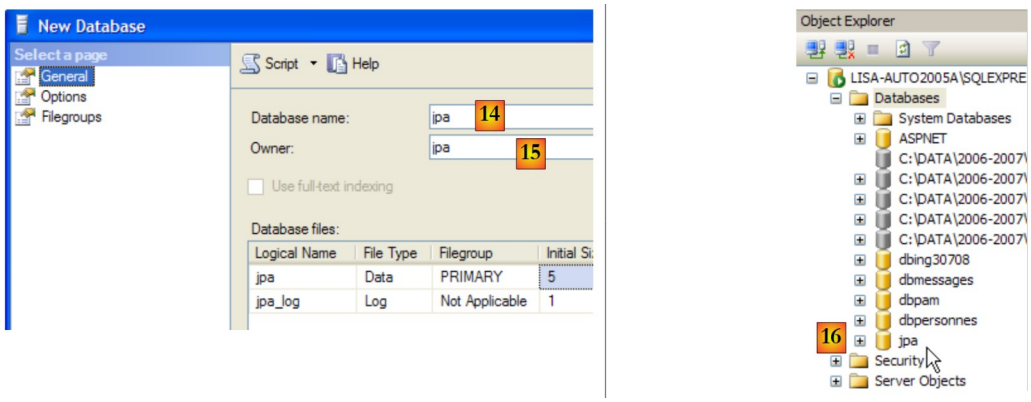
On valide cette configuration :



- en [9] : l'utilisateur **jpa** a été créé
- en [10] : on se déconnecte
- en [11] : on se reconnecte



- en [12] : on se connecte en tant qu'utilisateur jpa/jpa
- en [13] : une fois connecté, l'utilisateur **jpa** crée une base de données



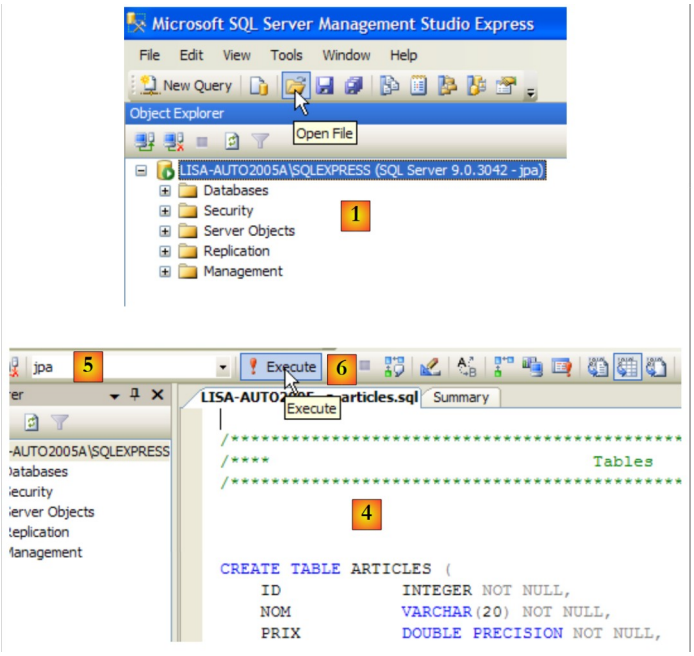
- en [14] : la base s'appellera **jpa**
- en [15] : et appartiendra à l'utilisateur **jpa**
- en [16] : la base **jpa** a été créée

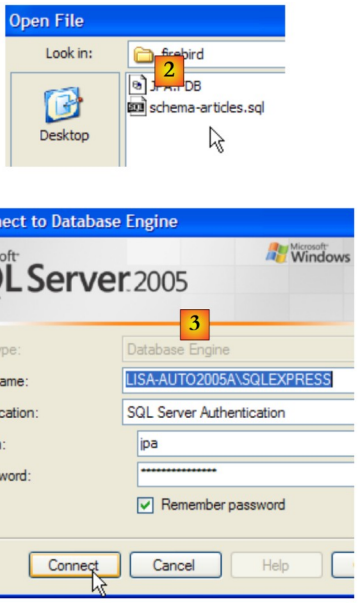
1.1.4 Création de la table [ARTICLES] de la base de données jpa

Nous créons une table [ARTICLES] à partir du script SQL suivant :

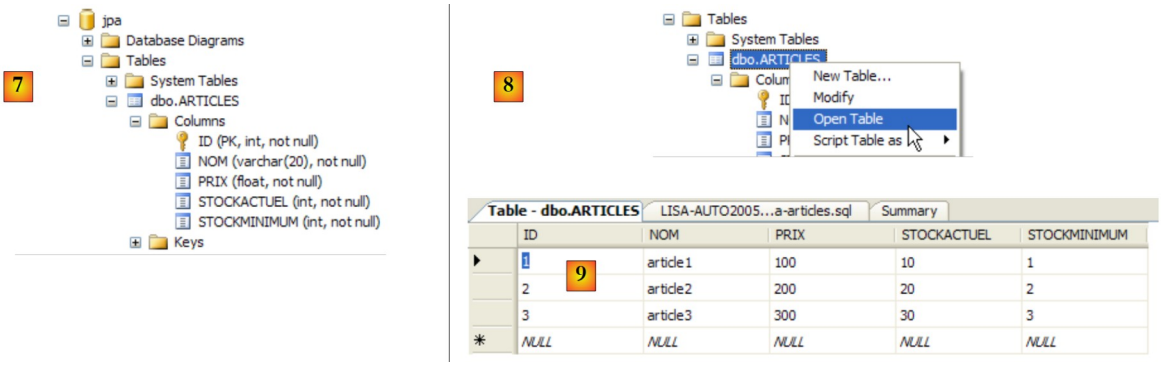
```
1.  /* création table */
2.
3.  CREATE TABLE ARTICLES (
4.      ID              INTEGER NOT NULL,
5.      NOM             VARCHAR(20) NOT NULL,
```

```
6.      PRIX           DOUBLE PRECISION NOT NULL,
7.      STOCKACTUEL    INTEGER NOT NULL,
8.      STOCKMINIMUM   INTEGER NOT NULL
9.  );
10.
11. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (1, 'article1', 100, 10,
    1);
12. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (2, 'article2', 200, 20,
    2);
13. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (3, 'article3', 300, 30,
    3);
14.
15. /* contraintes d'intégrité */
16.
17. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID check (ID>0);
18. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX check (PRIX>0);
19. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL check (STOCKACTUEL>0);
20. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM check (STOCKMINIMUM>0);
21. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM check (NOM<>'');
22. ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
23.
24.
25. /* clé primaire */
26.
27. ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);
```



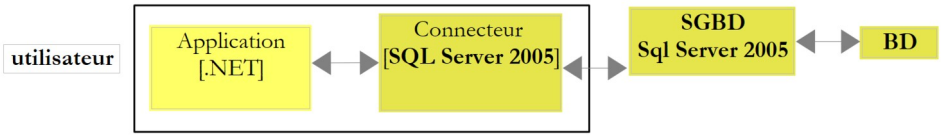


- en [1] : on ouvre un script SQL
- en [2] : on désigne le script SQL
- en [3] : on doit s'identifier de nouveau (jpa/jpa)
- en [4] : le script qui va être exécuté
- en [5] : sélectionner la base dans laquelle le script va être exécuté
- en [6] : l'exécuter



- en [7] : le résultat de l'exécution : la table [ARTICLES] a été créée.
- en [8] : on demande à voir son contenu
- en [9] : le contenu de la table.

1.1.5 Le connecteur ADO.NET de SQL Server Express

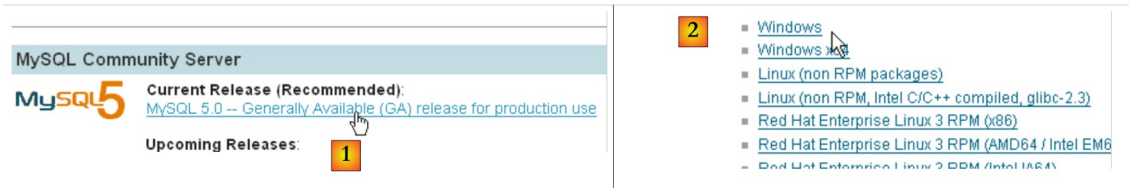


Le connecteur ADO.NET est l'ensemble des classes qui permettent à une application .NET d'utiliser le SGBD SQL Server Express 2005. Les classes du connecteur sont dans l'espace de noms [System.Data], nativement disponible sur toute plate-forme .NET.

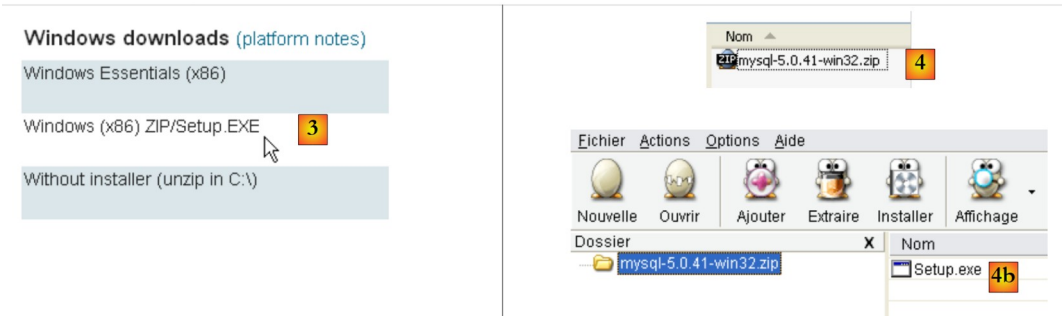
1.2 Le SGBD MySQL5

1.2.1 Installation

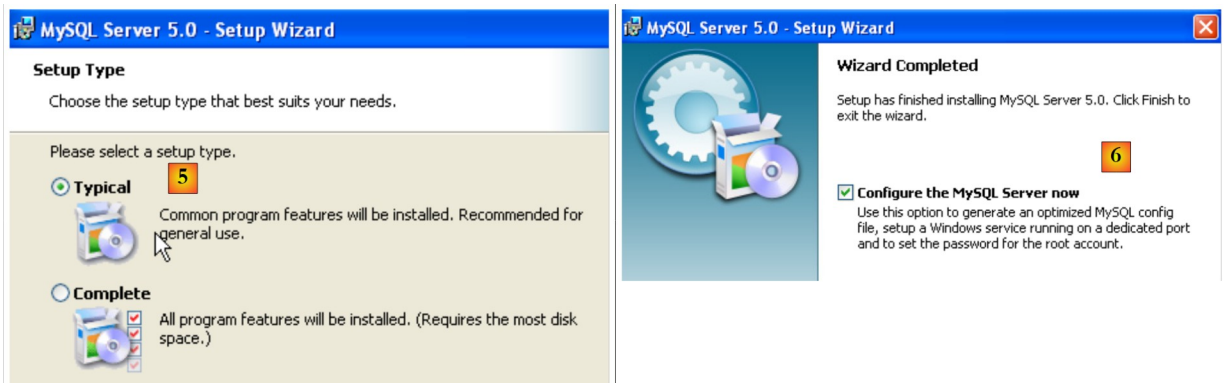
Le SGBD MySQL5 est disponible à l'url [http://dev.mysql.com/downloads/] :



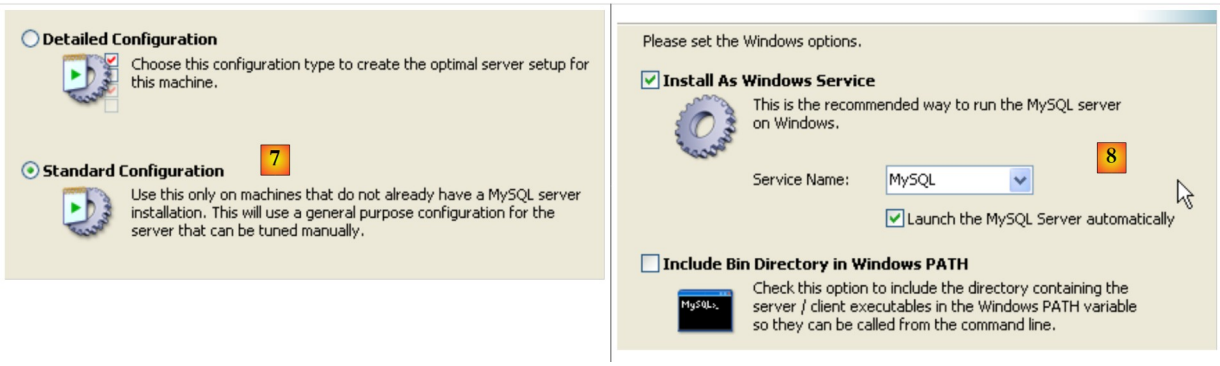
- en [1] : choisir la version désirée
- en [2] : choisir une version Windows



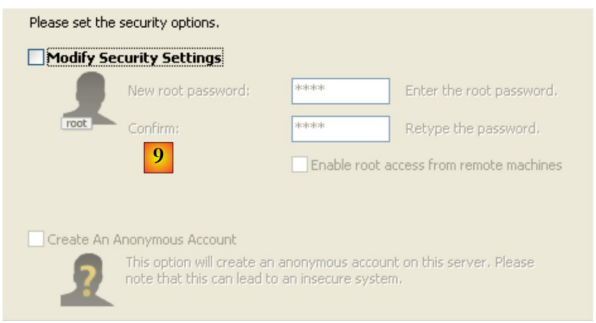
- en [3] : choisir la version windows désirée
- en [4] : le zip téléchargé contient un exécutable [Setup.exe] [4b] qu'il faut extraire et exécuter pour installer MySQL5

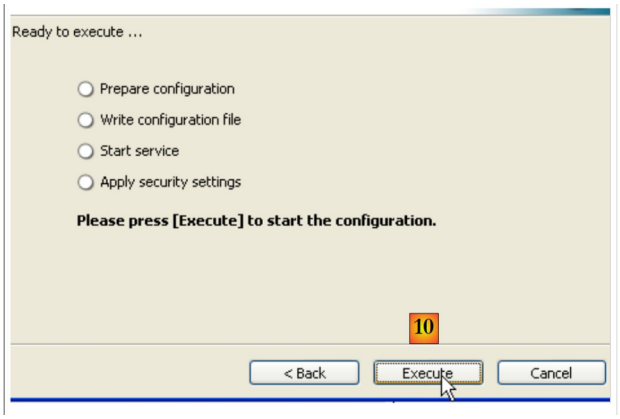


- en [5] : choisir une installation typique
- en [6] : une fois l'installation terminée, on peut configurer le serveur MySQL5



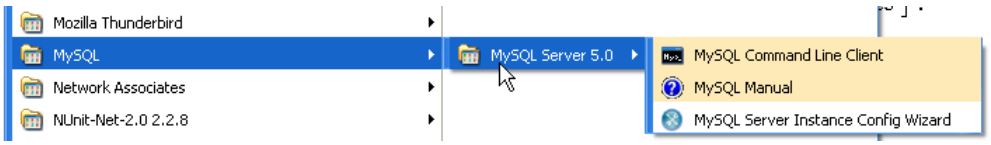
- en [7] : choisir une configuration standard, celle qui pose le moins de questions
- en [8] : le serveur MySQL5 sera un service windows



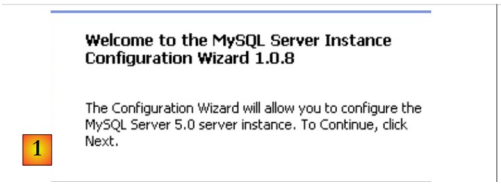


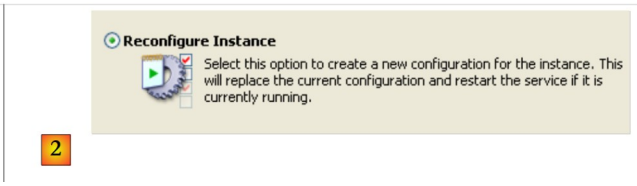
- en [9] : par défaut l'administrateur du serveur est **root** sans mot de passe. On peut garder cette configuration ou donner un nouveau mot de passe à **root**. Si l'installation de MySQL5 vient derrière une désinstallation d'une version précédente, cette opération peut échouer. Il y a moins moyen d'y revenir.
- en [10] : on demande la configuration du serveur

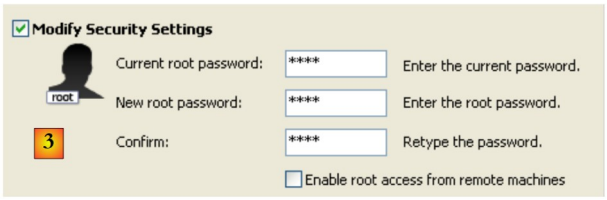
L'installation de MySQL5 donne naissance à un dossier dans [Démarrer / Programmes] :

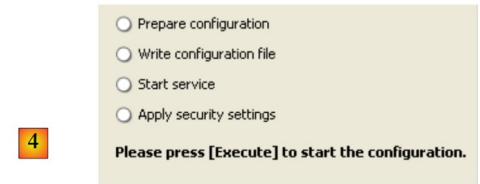


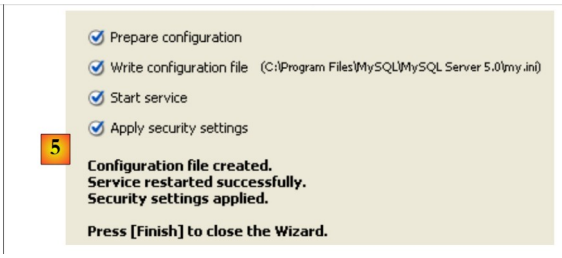
On peut utiliser [MySQL Server Instance Config Wizard] pour reconfigurer le serveur :









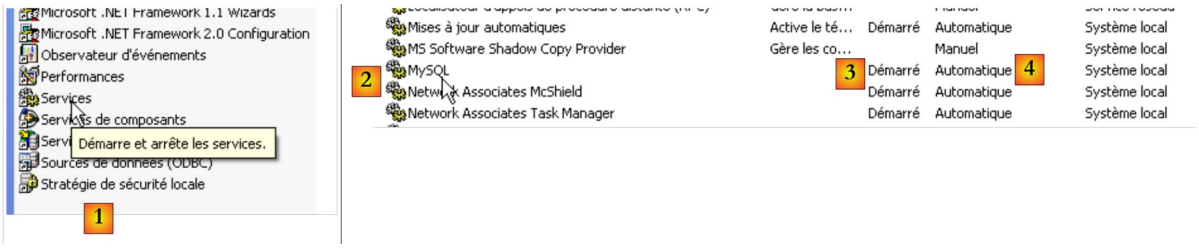


- en [3] : nous changeons le mot de passe de root (ici root/root)

1.2.2 Lancer / Arrêter MySQL5

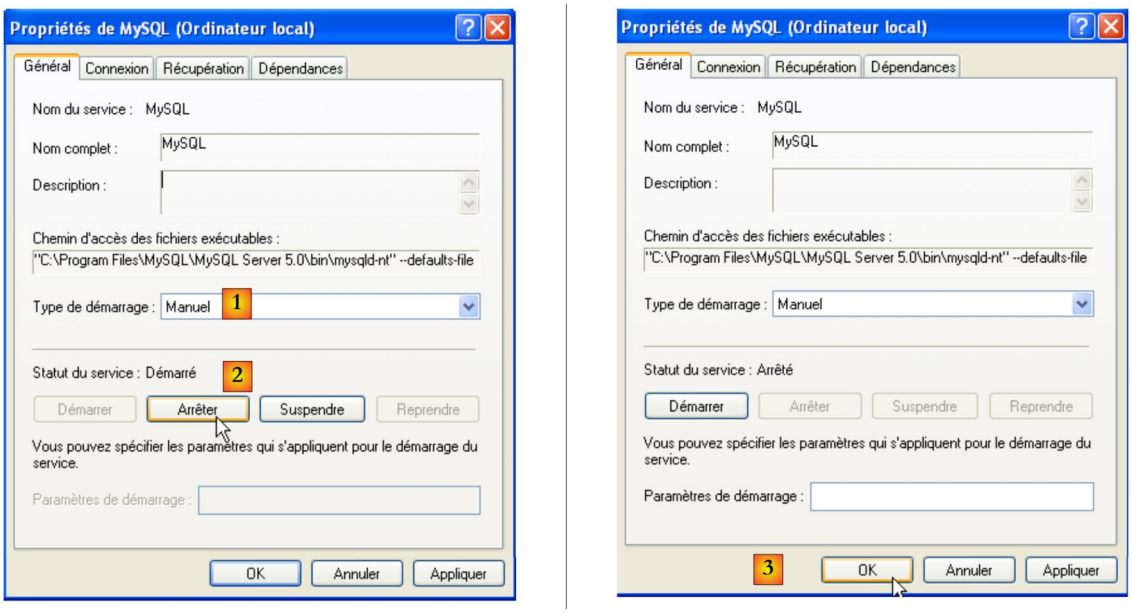
Le serveur MySQL5 a été installé comme un service windows à démarrage automatique, c.a.d lancé dès le démarrage de windows. Ce mode de fonctionnement est peu pratique. Nous allons le changer :

[Démarrer / Panneau de configuration / Performances et maintenance / Outils d'administration / Services] :



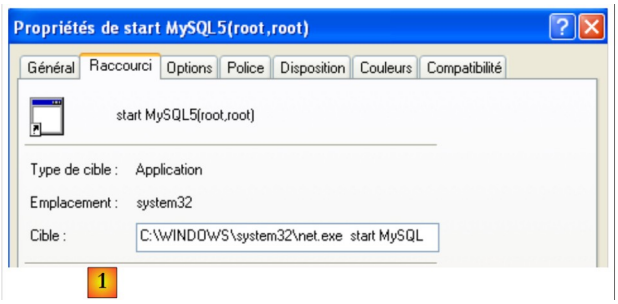
- en [1] : nous double-cliquons sur [Services]
- en [2] : on voit qu'un service appelé [MySQL] est présent, qu'il est démarré [3] et que son démarrage est automatique [4].

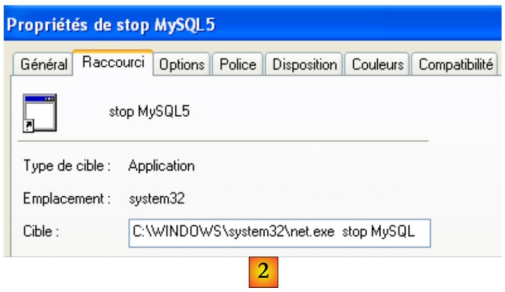
Pour modifier ce fonctionnement, nous double-cliquons sur le service [MySQL] :



- en [1] : on met le service en démarrage manuel
- en [2] : on l'arrête
- en [3] : on valide la nouvelle configuration du service

Pour lancer et arrêter manuellement le service MySQL, on pourra créer deux raccourcis :

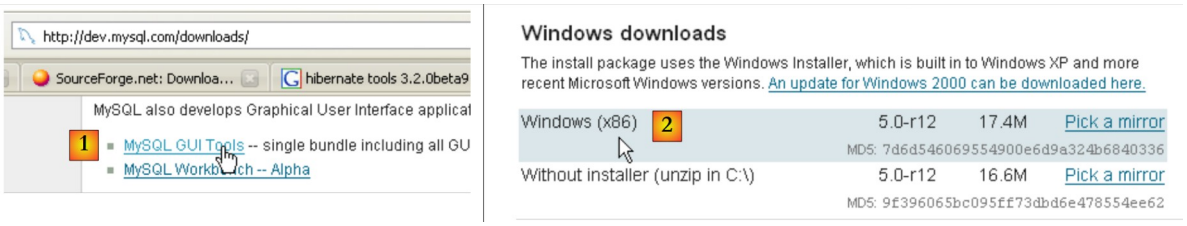




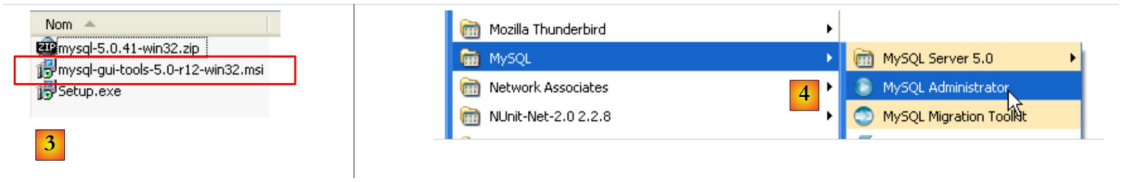
- en [1] : le raccourci pour lancer MySQL5
- en [2] : le raccourci pour l'arrêter

1.2.3 Clients d'administration MySQL

Sur le site de MySQL, on peut trouver des clients d'administration du SGBD :

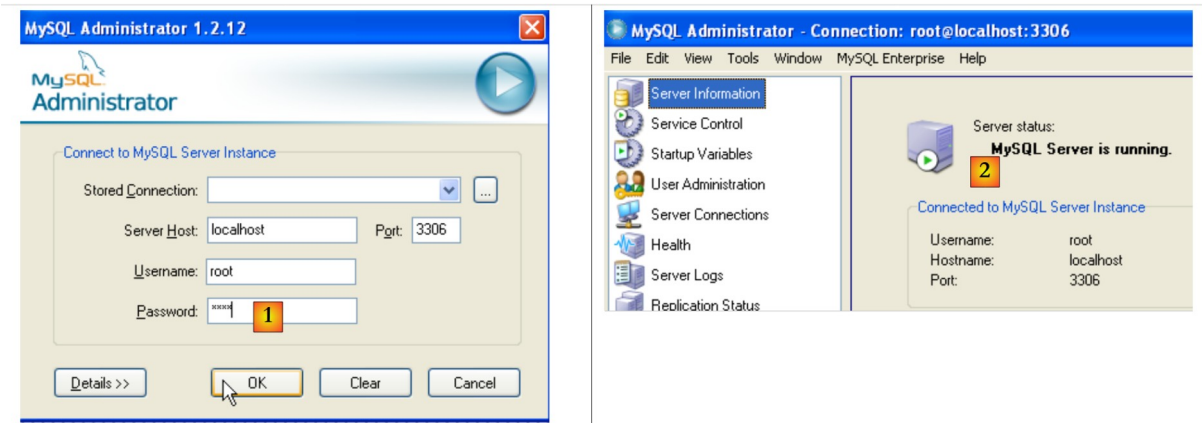


- en [1] : choisir [MySQL GUI Tools] qui rassemble divers clients graphiques permettant soit d'administrer le SGBD, soit de l'exploiter
- en [2] : prendre la version Windows qui convient



- en [3] : on récupère un fichier .msi à exécuter
- en [4] : une fois l'installation faite, de nouveaux raccourcis apparaissent dans le dossier [Menu Démarrer / Programmes / mySQL].

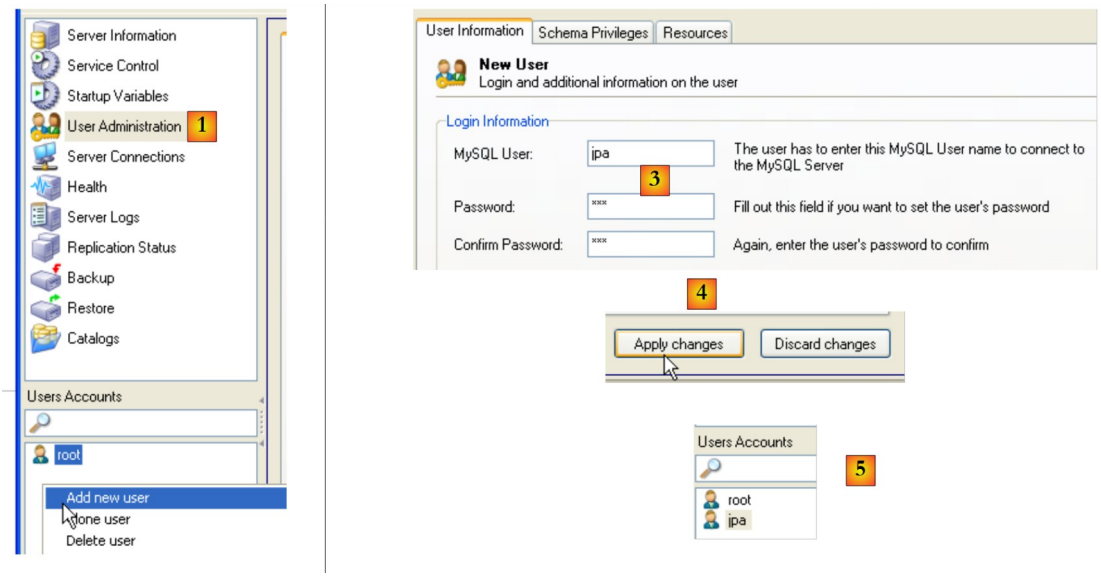
Lançons MySQL (via les raccourcis que vous avez créés), puis lançons [MySQL Administrator] via le menu ci-dessus :



- en [1] : mettre le mot de passe de l'utilisateur root (root ici)
- en [2] : on est connecté et on voit que MySQL est actif

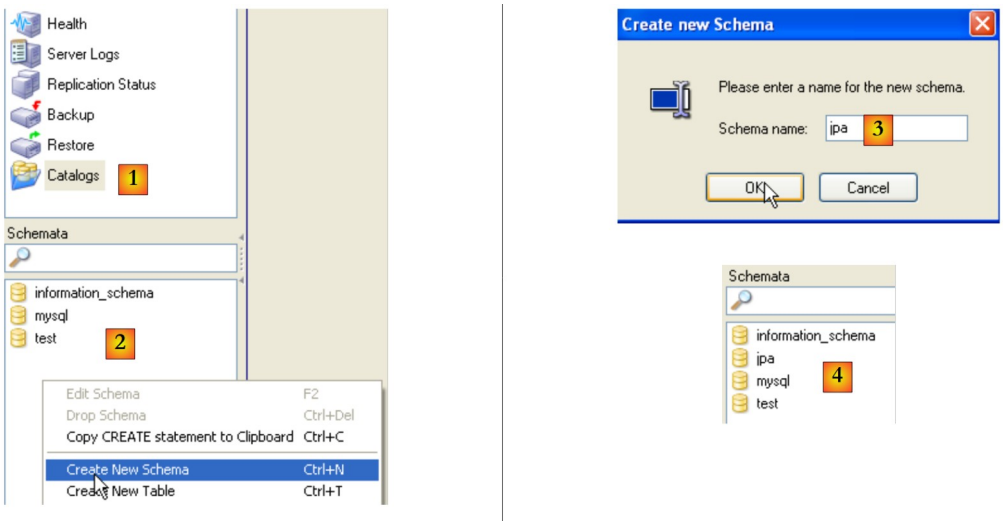
1.2.4 Création d'un utilisateur jpa et d'une base de données jpa

Nous créons maintenant une base de données appelée **jpa** et un utilisateur de même nom. D'abord l'utilisateur :

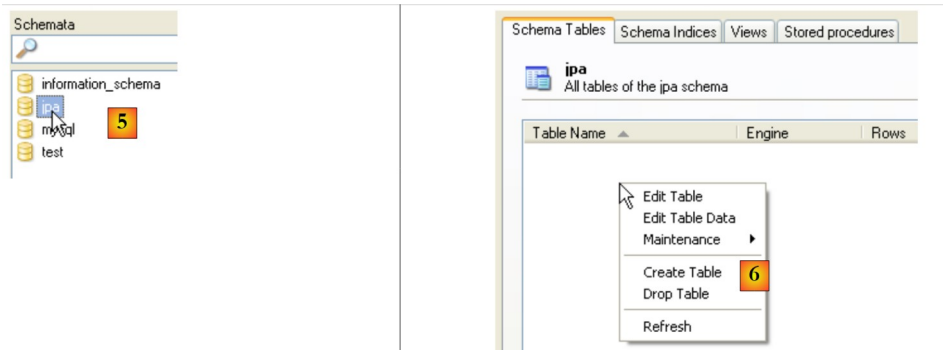


- en [1] : on sélectionne [User Administration]
- en [2] : on clique droit dans la partie [User accounts] pour créer un nouvel utilisateur
- en [3] : l'utilisateur s'appelle **jpa** et son mot de passe est **jpa**
- en [4] : on valide la création
- en [5] : l'utilisateur [jpa] apparaît dans la fenêtre [User Accounts]

La base de données maintenant :

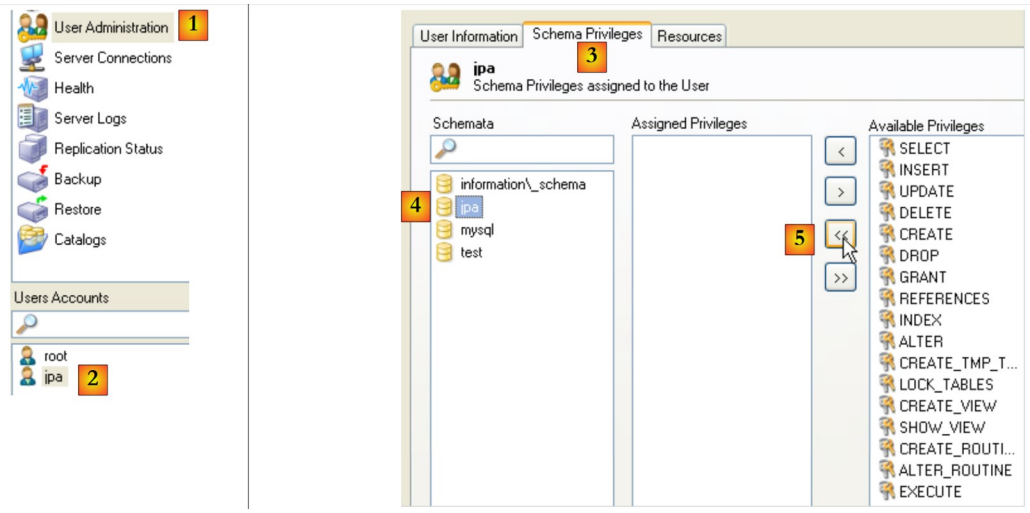


- en [1] : choix de l'option [Catalogs]
- en [2] : clic droit sur la fenêtre [Schemata] pour créer un nouveau schéma (désigne une base de données)
- en [3] : on nomme le nouveau schéma
- en [4] : il apparaît dans la fenêtre [Schemata]

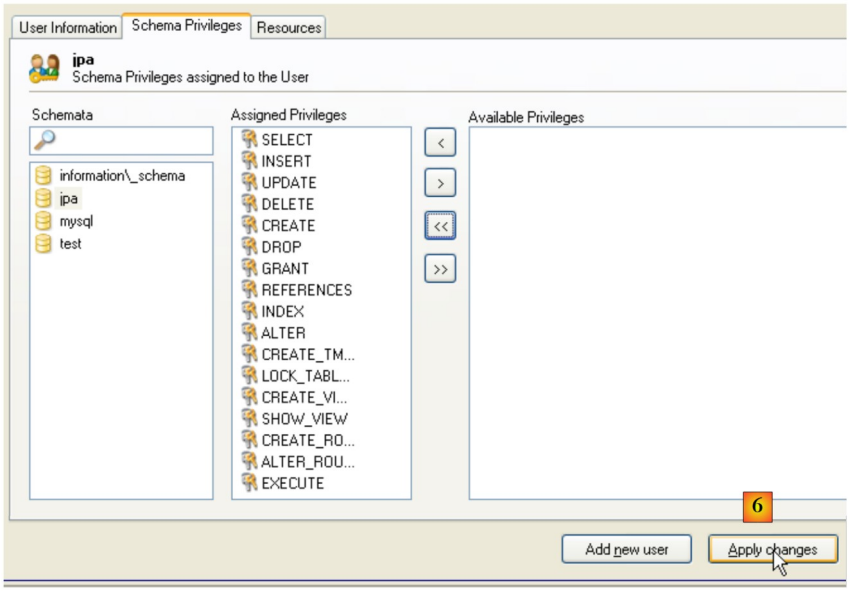


- en [5] : on sélectionne le schéma [jpa]
- en [6] : les objets du schéma [jpa] apparaissent, notamment les tables. Il n'y en a pas encore. Un clic droit permettrait d'en créer. Nous laissons le lecteur le faire.

Revenons à l'utilisateur [jpa] afin de lui donner tous les droits sur le schéma [jpa] :

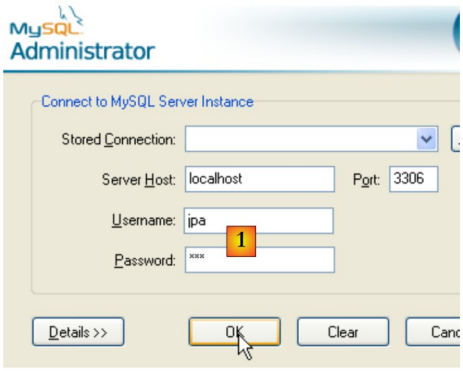


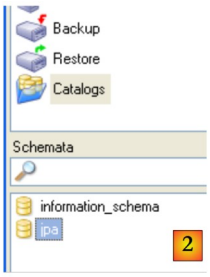
- en [1], puis [2] : on sélectionne l'utilisateur [jpa]
- en [3] : on sélectionne l'onglet [Schema Privileges]
- en [4] : on sélectionne le schéma [jpa]
- en [5] : on va donner à l'utilisateur [jpa] tous les privilèges sur le schéma [jpa]



- en [6] : on valide les changements faits

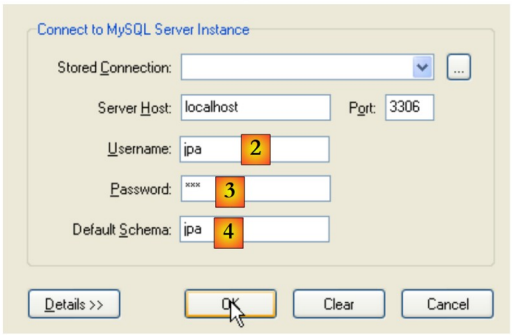
Pour vérifier que l'utilisateur [jpa] peut travailler avec le schéma [jpa], on ferme l'administrateur MySQL. On le relance et on se connecte cette fois sous le nom [jpa/jpa] :



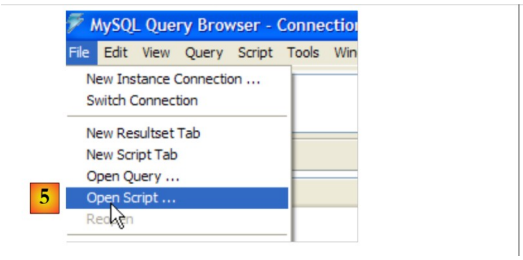


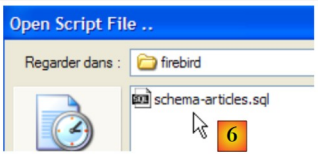
- en [1] : on s'identifie (jpa/jpa)
- en [2] : la connexion a réussi et dans [Schemata], on voit les schémas sur lesquels on a des droits. On voit le schéma [jpa].

Nous allons maintenant créer une table [ARTICLES] à l'aide d'un script SQL.



- en [1] : utiliser l'application [MySQL Query Browser]
- en [2], [3], [4] : s'identifier (jpa / jpa / jpa)

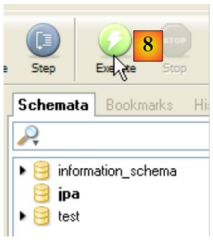


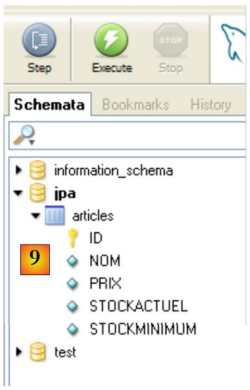


- en [5] : ouvrir un script SQL afin de l'exécuter
- en [6] : désigner le script [schema-articles.sql] suivant :

```
1.  /*******************************************************************************/
2.  /****                                Tables                                 ****/
3.  /*******************************************************************************/
4.
5.
6.
7.  CREATE TABLE ARTICLES (
8.      ID              INTEGER NOT NULL,
9.      NOM             VARCHAR(20) NOT NULL,
10.     PRIX            DOUBLE PRECISION NOT NULL,
11.     STOCKACTUEL     INTEGER NOT NULL,
12.     STOCKMINIMUM    INTEGER NOT NULL
13. );
14.
```

```
15. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (1,'article1', 100, 10, 1);
16. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (2,'article2', 200, 20, 2);
17. INSERT INTO ARTICLES (ID, NOM, PRIX, STOCKACTUEL, STOCKMINIMUM) VALUES (3,'article3', 300, 30, 3);
18.
19. COMMIT WORK;
20.
21.
22.
23. /* Check constraints definition */
24.
25. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID check (ID>0);
26. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX check (PRIX>0);
27. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL check (STOCKACTUEL>0);
28. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM check (STOCKMINIMUM>0);
29. ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM check (NOM<>'');
30.
31.
32. /*****************************************************************************/
33. /****                          Unique Constraints                         ****/
34. /*****************************************************************************/
35.
36. ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
37.
38.
39. /*****************************************************************************/
40. /****                            Primary Keys                             ****/
41. /*****************************************************************************/
42.
43. ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);
```





- en [7] : le script chargé
- en [8] : on l'exécute
- en [9] : la table [ARTICLES] a été créée

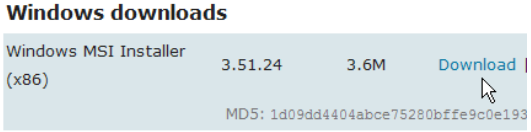
1.2.5 Installation du connecteur ADO.NET de MySQL5

Le connecteur ADO.NET de MySQL5 est disponible (avril 2008) à l'adresse [http://dev.mysql.com/downloads/connector/net/5.1.html] :

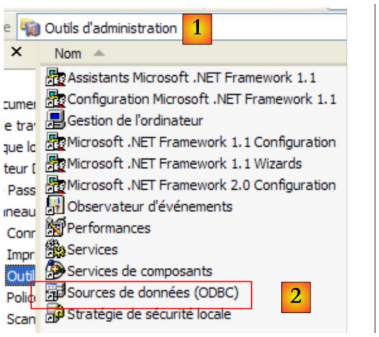
L'installation de ce connecteur ajoute un espace de noms à la plate-forme .NET :

1.2.6 Installation du pilote ODBC de MySQL5

Le connecteur ODBC (Open DataBase Connectivity) de MySQL5 est disponible (avril 2008) à l'adresse [http://dev.mysql.com/downloads/connector/odbc/3.51.html] :



Après installation, la présence du connecteur ODBC peut être vérifiée de la façon suivante :



- en [1], sélectionner [Outils d'administration] (sur XP Pro : Menu Démarrer / Panneau de configuration / Performances et maintenance / Outils d'administration)
- en [2], double-cliquer sur [Sources de données (ODBC)]
- en [3], sélectionner l'onglet [Pilotes ODBC]
- en [4], le pilote ODBC de MySQL