Faculté des Sciences de Monastir
Département des Sciences de l'Informatique

Notes de cours sur

La technologie .NET

Par : Karim Kalti

SOMMAIRE

- Introduction à la technologie .NET
- Le Langage C#.NET
- Les IHM en .NET
- Accès aux données (ADO.NET)
- Architectures des applications Web dynamiques
- ASP.NET
- Les services Web
- Les serveurs de média (audio et vidéo streaming)
- Développement et déploiement d'applications Web multimédia.

V - 3.1

INTRODUCTION AU FRAMEWORK .NET

Qu'est ce que le FrameWork .NET :

D'après MicroSoft, le Framework .NET est une nouvelle plate-forme informatique qui simplifie le développement d'applications dans l'environnement fortement distribué d'Internet. (*Source MSDN*)

Principaux objectifs

Le Framework .NET est conçu pour remplir les objectifs suivants : (*Source MSDN*)

- Fournir un environnement cohérent de programmation orientée objet que le code objet soit stocké et exécuté localement, exécuté localement mais distribué sur Internet ou exécuté à distance.
- Fournir un environnement qui garantit l'exécution sécurisée de code.
- Fournir aux programmeurs un environnement cohérent qui permet de développer une grande variété de types d'applications comme les applications Windows, les services Windows, les applications embarquées et les applications Web.
- Générer toutes les communications à partir des normes d'industries pour s'assurer que le code basé sur le Framework .NET peut s'intégrer à n'importe quel autre code. (forte utilisation du XML)

L'architecture du FrameWork .NET

Le Framework .NET possède une architecture composée de trois principales couches :

- La CLS qui est une spécification visant à garantir l'interopérabilité des langages de programmation en .NET.
- La bibliothèque de classes .NET (API.NET): qui permet de développer des applications .NET
- La **C**ommon **L**anguage **R**untime (CLR) : qui permet d'exécuter les applications .NET

La Common Language Specification (CLS)

- La *Common Language Specification* est une normalisation qui doit être respecté et implémenté par tout langage de programmation qui se veut capable de créer des applications .NET.
- Cette spécification s'intéresse à la normalisation d'un certain nombre de fonctionnalités comme par exemple la manière d'implémenter les types de données, les classes, les délégués, la gestion des événements, etc ...
- La CLS a pour objectif de garantir l'interopérabilité entre les langages .NET. En effet, deux langages conformes à la CLS peuvent facilement échanger des données entre eux (ils implémentent au bas niveau de la même manière ces données). Il est possible par exemple de créer une classe dans un langage conforme à la CLS qui hérite d'une autre classe définie dans un autre langage conforme à la CLS.

© Karim Kalti

La bibliothèque de classes .NET

- La bibliothèque de classes .NET est une API complètement orientée objet qui offre la possibilité de développer des applications allant des traditionnelles applications à ligne de commande ou à interface graphique utilisateur (GUI, Graphical User Interface) jusqu'aux applications qui exploitent les dernières innovations fournies par ASP.NET, comme les services Web XML et les Web Forms.
- La version 1.0 du .NET FrameWork comporte une bibliothèque d'environ 9000 classes et 270000 méthodes.
- Ces classes et leurs méthodes couvrent presque tous les besoins du développement d'applications (Accès aux données, programmation réseau, développement d'interfaces graphiques, développement Web, fonctions mathématiques, manipulation des chaînes, des dates …).
- Ces classes sont regroupées d'une manière thématique et hiérarchique en espaces de noms.

La bibliothèque .NET : une API de programmation multi-langages

Langages classiques de programmation

- Les langages classiques : chaque langage possède sa syntaxe de base (liste de mots-clès plus les règles d'écriture)
- Un langage est toujours accompagné d'une bibliothèque de fonctions. Ces fonctions enrichissement les possibilités du langage.
- Exemples : fonctions mathématiques, fonctions de manipulation des chaînes de caractères, fonctions graphiques, fonctions systèmes, …

Langage pour programmer avec la bibliothèque .NET

- La bibliothèque .NET n'est pas spécifique à un langage donné (indépendante des langages).
- Elle est supportée par plusieurs langages (environ 27 langages d'après Microsoft : VB, C++, C#, Java, Jscript, Delphi, Eiffel, Perl, Python, COBOL, …).
- Les trois langages les plus utilisées pour développer en .NET sont VB.NET, C#.NET, et C++.NET.
- Chaque langage qui supporte la bibliothèque .NET (conforme à la CLS) peut instancier les classes de cette bibliothèque et utiliser ses méthodes.

Exemple :

```
////   VB.NET ////
Dim i as Integer
i=5
Console.WriteLine(i)

////  C++.NET ////
int i;
i=5;
Console::WriteLine(i);

////   C#.NET ////
int i;
i=5;
Console.WriteLine(i);
```

Cet exemple montre que chacun des trois langages utilise sa propre syntaxe pour la déclaration des variables, mais les trois font appel à la même méthode .NET pour faire l'affichage bien sûr chacun avec ses propres règles d'écriture.

© Karim Kalti

Organisation de la bibliothèque de classes .NET

La bibliothèque .NET est constituée de trois couches de classes offrant trois catégories de services :

Première couche : Base classes Library (BCL)

La BCL rassemble des classes permettant d'effectuer les opérations de base telles que la manipulation de chaînes de texte, la gestion des entrées/sorties, des communications réseaux, des threads, etc.

Deuxième couche : les classes de données et XML

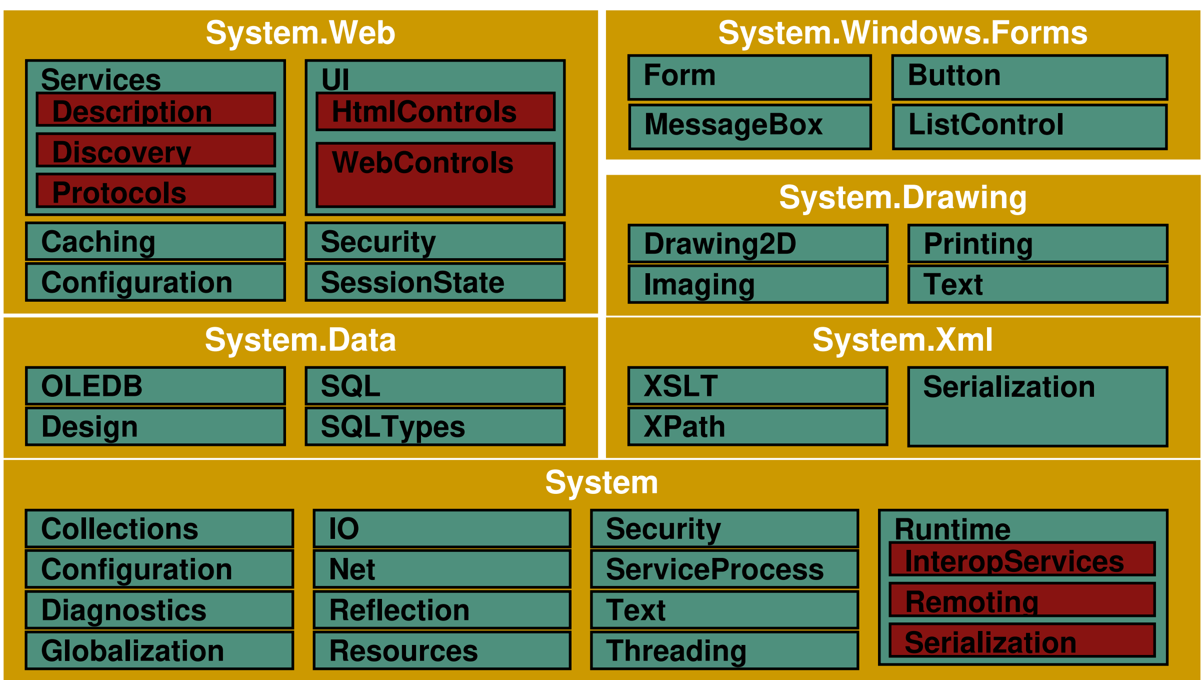
La deuxième couche est composée de deux bibliothèques de classes d'accès aux données :

- La bibliothèque ADO.NET, s'élevant sur les bases de l'ancien ADO (*ActiveX Data Objects*) et permettant l'accès sous format XML aux interfaces de bases de données SQL Server ODBC, OLEDB, ORACLE, et aux fichiers XML.
- Une bibliothèque de classes permettant de manipuler les données XML. On y trouve par exemple les classes XSLT permettant la transformation d'un document XML vers n'importe quel type d'autre document.

Troisième couche : les services Web, les applications Web et les applications Windows

La dernière couche, la plus élevée, est utilisée pour la création des applications Web et des applications Windows et notamment la partie interface :

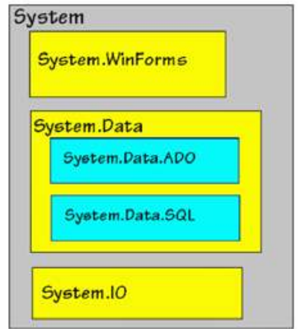
- Les applications Web peuvent se présenter sous formes de pages Web dynamiques et statiques ou sous forme de services Web. La technologie utilisée pour leur création est l'*ASP.NET*. Elle utilise pour la réalisation des interfaces de nouveaux composants appelés les *WebForms*.
- Les applications Windows et particulièrement leurs interfaces sont créées à l'aide des *WinForms*.



La bibliothèque de classes .NET

© *Karim Kalti*

- Les classes de la bibliothèque .NET sont organisées sous forme d'espaces de noms hiérarchisés. Chaque espace de noms peut comporter un ensemble de classes et/ou un ensemble de sous-espaces de noms.
- L'accès à une classe s'effectue à l'aide son nom complet. Ce nom complet se compose de la liste hiérarchique des espaces de noms plus le nom de la classe en question. Ces noms étant reliés entre eux par des points.

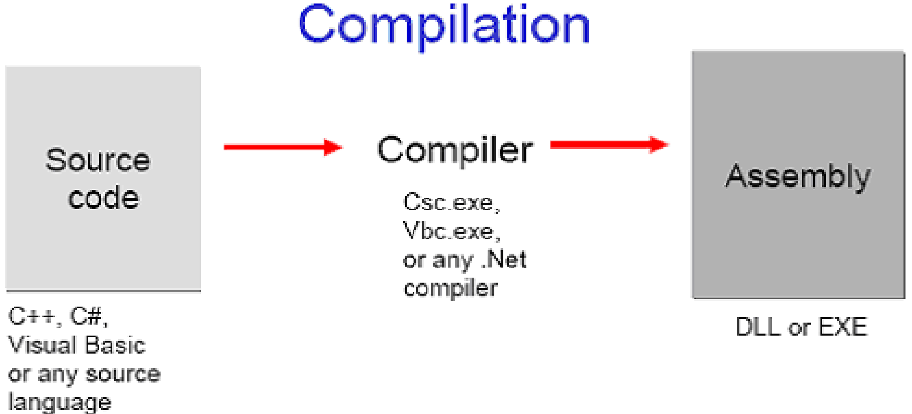


Exemple :

- La classe DataSet qui se trouve dans l'espace de noms "System.Data.ADO" se déclare comme "System.Data.ADO.Dataset".
- La classe Console qui se trouve dans l'espace de noms "System" se déclare comme "System.Console".

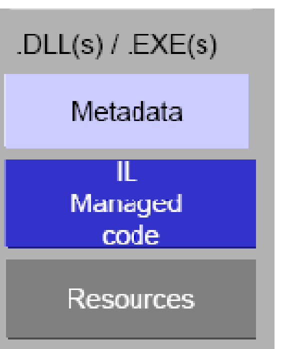
Le MSIL

- La compilation d'un programme écrit en .NET conduit vers la création d'un fichier exécutable (fichier .exe).
- Cet exécutable n'est pas écrit en code machine comme les exécutables classiques mais avec un langage intermédiaire appelé MSIL acronyme de **M**icro**S**oft **I**ntermediate **L**anguage.
- L'exécution des fichiers compilés en MSIL ne peut pas être directement assurée par les services du système d'exploitation.



L'assemblage

- L'assemblage est le fichier *exe* ou *dll* produit par la compilation d'un code .NET.
- Un assemblage contient trois types de données :
  - Le code MSIL qui résulte de la compilation.
  - Les méta données.
  - Les ressources.



MSIL dans un assembly

Un code .NET peut comporter plusieurs classes. Au moment de la compilation, toutes ces classes sont compilées en MSIL dans un même assemblage. Les classes d'un même assemblage doivent avoir un seul point d'entrée (Une seule classe parmi celles d'un assemblage doit avoir une méthode qui joue le rôle de fonction principale). Il est à rappeler que le point d'entrée est défini par la méthode *Main* pour les applications de type console, par la méthode *WinMain* pour les applications Windows et par *DllMain* pour les dll (Dynamic Linked Library).

© Karim Kalti

Les méta données

Metadata

Type Descriptions
Classes
Base classes
Implemented interfaces
Attributes
Methods

Assembly Manifest
Name
Version
Culture
Other assemblies
Security permissions
Exported types

Les données qui accompagnent le code MSIL dans un assemblage sont de deux catégories :

- Des données de description des types : les types sont les classes compilées dans l'assemblage. La description concerne alors les spécifications de ces classes (noms, attributs, méthodes, droits d'accès, interfaces implémentées, …). Ces informations sont généralement utilisées par les EDI pour la complétion de code.
- L'*assembly manifest* : Le manifeste contient des informations du genre :
    - *Nom* : le nom de l'assemblage. Il sert à la résolution de portée. Il est le même que celui du fichier de l'assemblage moins l'extension.
    - *Version* : la version de l'assemblage.
    - *Culture* : permet d'indiquer le type de ressource à utiliser en ce qui concerne la culture (langue, écriture de droite à gauche, type de calendrier …).
    - *Autres assemblages* : les assemblages nécessaires à l'exécution de l'assemblage courant. Ces assemblages sont généralement des dlls.

Les ressources

Les éventuelles ressources utilisées par l'assemblages : icônes, bitmap, …

La Common Language Runtime (CLR)

- La Common Language Runtime est un environnement qui assure l'exécution des programmes .NET. Elle joue le rôle de la machine virtuelle de Java mais pour les programmes écrits en .NET.
- Elle interprète les fichiers exécutables compilés en MSIL.
- Elle fournit des services tels que :
    - La gestion de la mémoire (à travers le Garbage Collector GC).
    - La gestion des exceptions.
    - La gestion des threads.
    - L'interopérabilité entre plusieurs langages.
    - Le chargement dynamique des modules à exécuter.
    - La compilation vers un code machine natif du MSIL et le contrôle de l'exécution des programmes.

- Un programme managé (Managed program) est un programme compilé en MSIL, son exécution est gérée par la CLR.
- Un programme non managé est un programme compilé en code natif. Son exécution est directement prise en charge par les services du système d'exploitation.
- VB.NET et C#.NET ne permettent de créer que des programmes managés. C++.NET permet de créer des programmes managés et non managés.

© Karim Kalti

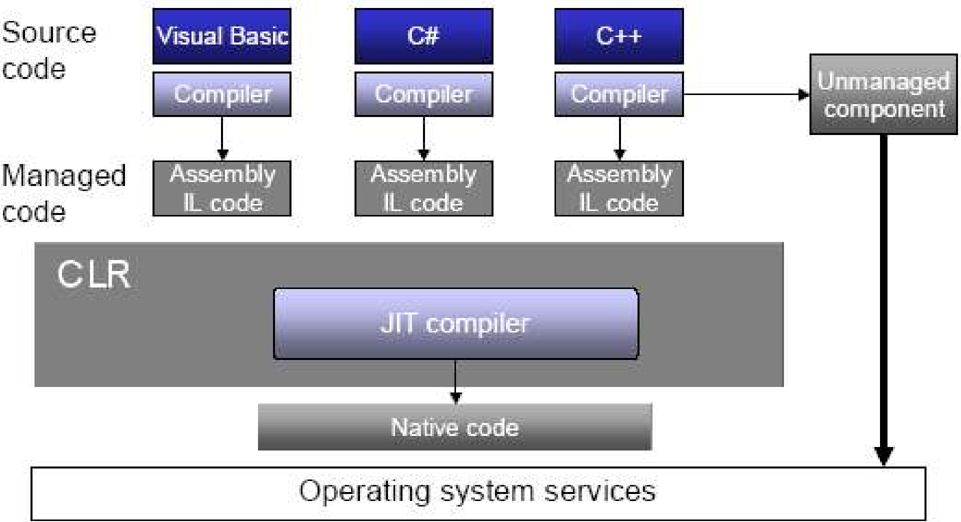
Avantages de la CLR

Sécurité de l'exécution des programmes.

Grâce à la gestion des exceptions et à la gestion automatique de la mémoire.

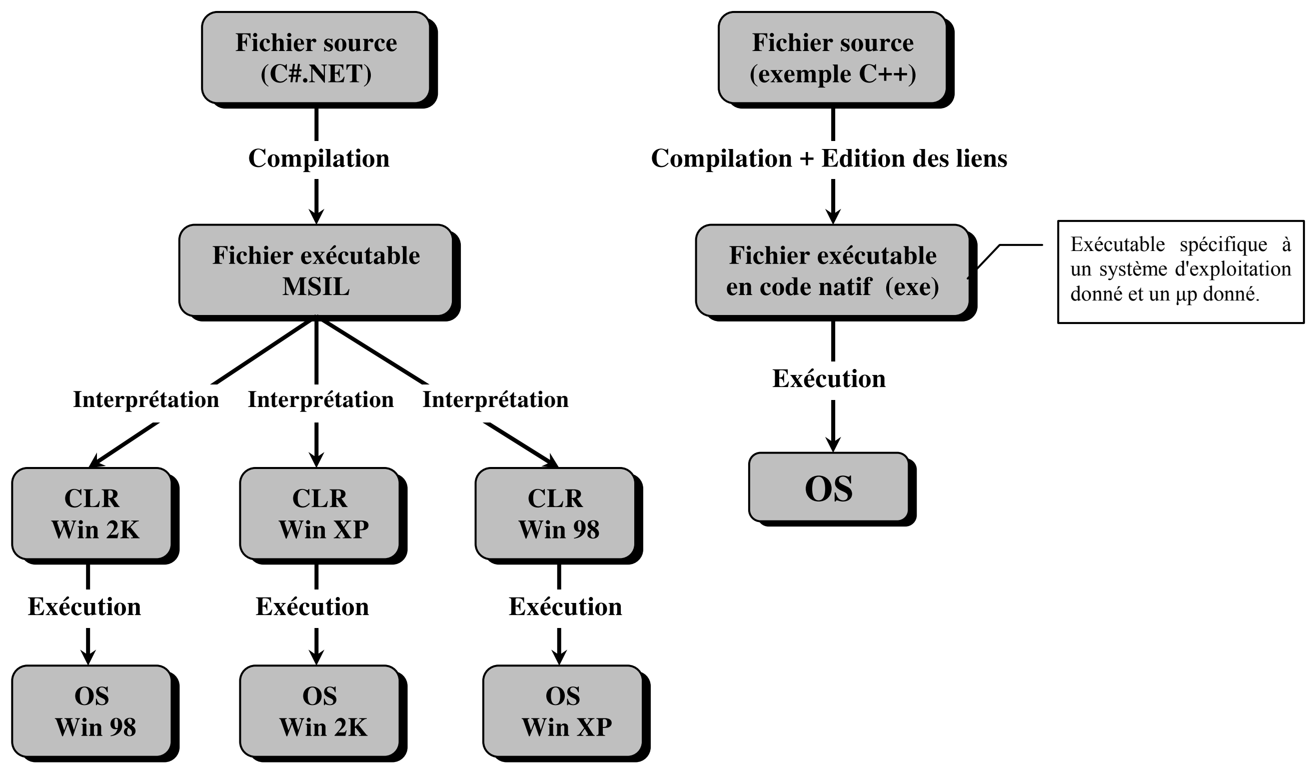
Interopérabilité de programmes écrits dans différents langages.

Tous les langages qui supportent le .NET compilent vers un même code intermédiaire → Possibilité de faire communiquer des programmes écrits dans des langages différents.



Portabilité des programmes

Etapes de génération d'un programme exécutable :



© Karim Kalti

- Au moment de l'exécution, la CLR assure entre autres la compilation Just In Time (JIT) du MSIL vers le code natif qui tient compte des services de la plateforme de destination.
- Il existe plusieurs versions de la CLR, chacune est spécifique à une plateforme donnée. Ce sont ces versions qui assurent la portabilité des exécutables .NET (assemblies)

Comparaison entre .net et Java :

.NET	JAVA
Code source	Code source
MS Intermediate Langage	Bytecode
CLR	Machine virtuelle (JVM)

La spécification .NET et l'implémentation .NET

- .NET FrameWork en tant que spécification est une spécification publique (proposée par MS).
- Il existe deux principales implémentations de cette spécification.
    - Implémentation proposée par MS : destinée aux plate-formes 2000,XP, 2003 server.
    - Projet Mono : implémentation destinée à tourner sous Linux (CLR pour Linux).

Outils nécessaires pour écrire un programme .NET

Au minimum il faut avoir :
- Pour écrire le code : un éditeur de texte (exemple : le bloc notes)
- Pour compiler : le SDK téléchargeable gratuitement (site Microsoft). Le SDK est un Kit comprenant un ensemble d'outils nécessaires pour le développement d'applications : compilateur, outils de déploiement etc. La compilation se fait dans ce cas en ligne de commande.
- Pour exécuter : la CLR : téléchargeable gratuitement (site Microsoft).

Environnement de Développement Intégré pour .NET

Qu'est ce qu'un environnement de Développement Intégré (EDI)

- Un environnement de développement intégré (EDI) est un outil qui facilite et accélère le développement d'applications.
- Il intègre les modules nécessaires pour le développement d'applications :
    - Editeur de code avec coloration syntaxique et complétion de code.
    - Lancement des processus de compilation et d'exécution à travers des menus sans passer en mode ligne de commande.
    - Outils de déboguage ...
    - Certains environnements de développement intègrent des fonctionnalités de type RAD (Rapid Application Developpment). Ces fonctionnalités permettent de créer d'une manière visuelle une grande partie de l'application (à l'aide d'assistants WYSIWYG) et génèrent automatiquement le code correspondant.

EDI payant

Visual Studio.NET (EDI + FrameWork).

EDI gratuit

SharpDevelop : pour developper des applications windows et mobiles. (EDI seulement)
WebMatrix : pour developper des applications Web. (EDI seulement)

© Karim Kalti

Ordre d'installation des composants :

- CLR (si elle n'est pas installée par défaut sur le système).
- IIS.
- MDAC.
- .NET FrameWork.
- EDI.

Pour VisualStudio.NET : l'installation de ces différents composants se fait d'une manière automatique.

© Karim Kalti

Les bases du langage C#

Caractéristiques et nouveautés

- C# : se présente comme la suite des langages C et C++, très proche de JAVA.
- Langage complètement orienté objet → pas de possibilité d'écrire des fonctions en dehors des classes, pas de variables globales.
- Permet l'écriture des programmes plus surs et plus stables grâce à la gestion automatique de la mémoire à l'aide du ramasse miette (garbage collector) et à la gestion des exceptions.
- Nouveautés par rapport au C++ :
  - Libération automatique des objets.
  - Disparition des pointeurs. (remplacés par des références).
  - Disparition du passage d'argument par adresse au profit du passage par référence.
  - Nouvelles manipulations des tableaux.
  - Nouvelle manière d'écrire les boucles. (foreach)
  - Disparition de l'héritage multiple mais possibilité d'implémenter plusieurs interfaces par une même classe.

Premier programme

```
using System ;
class Prog
{
  static void Main()
 {
   Console.WriteLine("Bonjour");
 }
}
```

- Un programme C# comporte obligatoirement une fonction Main (M majuscule).
- Main doit être obligatoirement membre d'une classe.
- Le point virgule à la fin de la définition d'une classe est optionnel (pas en C++).
- Pou afficher le message bonjour on utilise la méthode WriteLine de la classe Console. La classe Console fait partie de l'espace de noms (bibliothèque) System.
- La méthode WriteLine peut être appelée de deux manières :
  - Spécification du nom complet : System.Console.WriteLine
  - Spécification du nom relatif

```
using System ;
 … … … …
Console.WriteLine("Bonjour") ;
```

Les commentaires en C#

Trois types de commentaires peuvent être utilisés en C# :

//	Le reste de la ligne est considéré comme commentaire. (commentaire ligne)
/* … */	Tout ce qui se situe entre les deux délimiteurs est considéré comme commentaire. (commentaire multi-lignes)
///	Le reste de la ligne est considéré comme commentaire servant à la documentation automatique du programme. (commentaire de documentation)

© Karim Kalti

Les identificateurs en C#

- Premier caractère : lettre (y compris celles accentuées) ou le underscore _
- Distinction entre minuscule et majuscule.
- Les caractères accentués sont acceptés.
- Un mot réservé du C# peut être utilisé comme identificateur de variable à condition qu'il soit précédé de @.

Exemples d'identificateurs : *NbLignes, Nbécoles, @int*

Les instructions du C#

- Une instruction se termine obligatoirement par un point virgule.
- Une suite d'instructions délimitées par des accolades constitue un bloc. Les blocs définissent les zones de validité des variables (portée des variables).

Mots réservés du C#

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

Types de données

- Les types du C# sont de deux catégories : les types "*valeur*" et les types "*référence*".

Type "valeur"

- Une variable de type valeur est une variable *auto* au sens du C++. Elle est allouée sur la pile (*stack*). Sa durée de vie est gérée par le système (le runtime pour le C#).

Type "référence"

- Une variable de type référence est une variable dynamique au sens du C++.
- Une telle variable référence un objet alloué dans le tas managé. La variable elle même est allouée sur la pile.
- La durée de vie de l'objet référencé par une telle variable est gérée implicitement par le Garbage Collector.
- Les variables de type référence ne peuvent être accédées que d'une manière indirecte (à travers la référence). Elles sont de ce fait moins rapides que les variables de type "valeur".

© Karim Kalti

- La tentative d'accès à une référence invalide (non initialisée par exemple) engendre la levée d'une exception au moment de l'exécution.

Comparaison de valeurs et comparaison de références

- Une variable de type "référence" contient une référence à des données stockées dans un objet. Deux variables de types références peuvent donc référencer les mêmes données. Dans ce cas, la modification de ces données à travers l'une de ces deux références affecte automatiquement l'autre référence.
- Une variable de type "valeur" possède sa propre copie des données qu'elle stocke. Deux variables de type "valeur" peuvent avoir les mêmes données mais chacune possède sa propre copie distincte. Dans ce cas la modification de l'une des deux variables n'affecte pas l'autre.

Types C# et types .NET

- Il existe pour la majorité des types natifs du C# des types correspondants en .NET.
- La majorité des types natifs du C# sont de type "valeur". Les types .NET quant à eux, sont définis sous forme de classes et sont de ce fait de type "référence". (une classe définit toujurs un type référence en C#)
- Les types du C# ainsi que leurs types correspondants en .NET sont présentés dans les deux tableaux suivants :

Type "valeur" en C#	Type .NET Framework
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

Type référence en C#	Type .NET Framework
object	System.Object
string	System.String

- En C/C++ une valeur nulle est évaluée à *false* et une valeur ≠ 0 est évaluée à *true*, ce n'est plus le cas en C#, *true* est *true* et *false* est *false*.
- Le type *decimal* est utilisé pour les opérations financières. Il permet une très grande précision mais les opérations avec ce type sont deux fois plus lentes que les types *double* ou *float*.

Les variables

Déclaration

`Type NomVariable;`

- Toute variable doit être déclarée dans un bloc. Elle cesse d'exister en dehors de ce bloc, (pas de variable globale).
- Toute variable de type scalaire natif doit avoir obligatoirement une valeur (le compilateur signale une erreur sinon).
- En l'absence d'une initialisation explicite les champs d'un objet sont implicitement initialisés par le compilateur (numérique à 0 et chaîne à ""). Ce n'est pas le cas pour les variables de type scalaire.

Les constantes en C#

- Constante entière : `int i = 10 ;`
- Constante de type long (Ajout du suffixe L) : `long j = 10L ;`
- Constante réelle : `double k = 20.5;`
- Constante réelle simple (ajout du suffixe f) : `float v = 20.5f;`

© Karim Kalti

- Constante réelle grande précision (ajout du suffixe M ou m) : `decimal d=1.56M ou 1.56m;`
- Constante caractère (entre deux simples quotes) : `char c='A';`
- Constante chaîne de caractères (entre deux doubles quotes) : `string s="bonjour";`

Les constantes symboliques en C#

- La déclaration des constantes symboliques se fait avec le mot-clé *const*.
- Une constante symbolique doit être obligatoirement initialisée.

Exemple : `const int n=3 ;`

Les tableaux

Les tableaux en C# sont des types "référence".

Déclaration et allocation

`Type[] Tab ;` // Tab est une référence à un tableau.
`Tab = new Type[Taille] ;` // Allocation de l'espace mémoire du tableau.
L'espace mémoire du tableau est alloué sur le tas (heap).

Exemple :

```
int [] T ; // Déclaration d'une référence à un tableau d'entiers.
T=new int [3] ;   // Création effective du tableau
```

ou également : `int [] T = new int [3] ;`

Initialisation

```
float[] TF= {2.5f ,0.3f,5.9f} ; ou  float[] TF= new float[]{2.5f ,0.3f,5.9f} ;
string[] TS={"ALI","Salah", "Imed"};
```

Les cellules d'un tableau non explicitement initialisé sont automatiquement mises à 0.

Exemple :

`int[] TI = new int [3];` // Les cellules de TI sont automatiquement initialisées à 0

Libération d'un tableau

- La libération d'un tableau se fait automatiquement par le ramasse-miettes.
- Un tableau cesse d'exister :
  - Lorsqu'on quitte le bloc dans lequel il est déclaré.
  - Lorsqu'on assigne une nouvelle zone (nouvelle valeur y compris *null*) à la référence qui désigne le tableau.

Exemple :

`int[] T={10,5,23} ;` // déclaration et initialisation , allocation implicite sans new.
`...`
`T=new int[]{100,50,65,80}` ou également `T=null ;` // réallocation automatique de l'espace alloué au tableau.
La zone mémoire réservée aux 3 entiers précédents est libérée par le ramasse-miettes.

Accès aux cellules d'un tableau

Syntaxe : **`Tab[indice]`**

- Pour un tableau de n éléments, le premier élément a pour indice 0 et le dernier a pour indice n-1.
- Contrairement à C/C++, il n'est pas possible d'accéder à un tableau en dehors de ses bornes. Ceci est signalé comme une erreur. (levée d'exception)

© Karim Kalti

Tableau à plusieurs dimensions

Exemple : Tableau à deux dimensions

Déclaration : `Type[ , ] T = new Type[2,3] ;` // 2 lignes et 3 colonnes
Initialisation : `Type[ , ] T = {{Val1, Val2, Val3},{ Val4, Val5, Val6}};`
Accès : T[i,j]

Exemple : Tableau à 3 dimensions

```
Type[ , , ]  T = new Type[Dim1, Dim2, Dim3];
… … … …
T[i,j,k] = Valeur ;
```

Copie de tableaux

Exemple 1: Copie de références

```
int [] T1 = {1,2,3} ;
int [] T2 ; // T2 contient la valeur "null".
T2=T1 ;  // T1 et T2 font référence au même tableau.
```

Si on fait `T1[0]=100 ;` alors `Console.WriteLine(T2[0])` affichera 100 ;

Exemple 2: Copie de références

```
int [] T1 = {1,2,3} ;
int [] T2= new int[100] ;
T2=T1;
```

T2 fait référence à la zone mémoire contenant le tableau de trois éléments. La zone mémoire contenant les 100 cellules est signalée "à libérer". Le ramasse-miettes (Garbage Collector) la libérera lorsque un besoin en mémoire se manifestera.
Pour faire réellement une copie de tableaux il faut procéder comme suit :

Utilisation de la méthode CopyTo

```
int [] T1 = {1,2,3} ;
int [] T2= new int[100] ;
T2= new int[T1.Lenght] ; // l'espace de 100 éléments est signalé « à libérer ».
T1.CopyTo(T2,0) ; // fait la copie à partir de la cellule 0 ;
```

Utilisation de la méthode Clone

Exemple 1

```
int [] T1 = {1,2,3} ;
int [] T2;
T2 = (int[])T1.Clone() ;
T[1]=100 ;
```

Alors le contenu de T1 est : 100,2,3 et le contenu de T2 est : 1,2,3

Exemple 2

```
int [ , ] T1= {{1,2,3},{10,20,30}} ;
int [ , ] T2 ;
T2 = (int[ , ]) T1.Clone() ;
```

Tableaux avec des cellules de types différents

Il est possible de créer des tableaux ayant des cellules de types différents. Le type de base de ces tableaux doit être le type *object*. Une cellule de type *object* peut recevoir une valeur de n'importe quel type.

Exemple :

```
object[] T = new object[3] ;
T[0] = 12 ; T[1] = 1.2 ; T[2]= "Bonjour";
```

Tableaux déchiquetés

Un tableau déchiqueté est un tableau qui possède des composantes de dimensions irrégulières. Par exemple, pour les tableaux à deux dimensions on parle de tableau déchiqueté si chaque ligne possède un nombre différent de colonnes.

© Karim Kalti

Exemple :
```
int[][] T ;
T= new int[2][] ;
T[0] = new int[3] ;
T[0][0] =1 ; T[0][1]=2 ; T[0][2]=3 ;
T[1] = new int[]{10,20,30,40} ;
```

Les tableaux déchiquetés ne sont pas compatibles avec le .NET (ne sont pas CLS compliant). Ils sont plutôt spécifiques au C#.

Opérations de saisie et d'affichage en mode console

Affichage en mode console

- L'affichage en mode console se fait essentiellement à l'aide des méthodes `Write` et `WriteLine` de la classe `Console`.
- La méthode `Write` affiche une chaîne de caractères.
- La méthode `WriteLine` affiche une chaîne puis effectue un retour à la ligne.

Exemples
```
Console.WriteLine("bon");
Console.WriteLine("jour");
Console.Write("Bon");
Console.WriteLine("jour");
int k =10,i=5,j=6;
Console.WriteLine(k);
Console.WriteLine("k vaut :" + k);
Console.WriteLine("i="+i+"j="+j);
Console.WriteLine("i={0} et j = {1}",i,j);
Console.WriteLine(i+j);
Console.WriteLine("Somme ="+i+j);
Console.WriteLine("Somme =" +(i+j));
Console.WriteLine("Somme ={0}",i+j);
```

Saisie en mode console

- La saisie en mode console se fait essentiellement à l'aide des méthodes `Read` et `ReadLine` de la classe `Console`.
- La méthode `Read()` lit un caractère à partir du flux standard d'entrée.
- La méthode `ReadLine()` lit une chaîne à partir du flux standard d'entrée.
- La valeur retournée est "null" si aucune donnée n'a été saisie (l'utilisateur tape directement ENTREE).
- `ReadLine` ne peut retourner que des chaînes de caractères. Il faut par la suite faire les conversions explicites vers les types de destinations. Ces conversions se font à l'aide de la méthode `Parse` membre des classes représentant les types de base.

Exemple :
```
// Lecture d'un entier
string s = Console.ReadLine();
int i = Int32.Parse(s);
// Lecture d'un réel
string s = Console.ReadLine();
double d = Double.Parse(s);
```

Les opérateurs

- Opérateurs arithmétiques : `+ - / % *`
- Pré-Post Incrémentation : `++, --`
- Opérateurs logiques : `<, <=, >, >=, = =, !=, !,&&, ||.`
- Autres opérateurs : `+=, -=, *=`

Structures de contrôle

Similarités par rapport au C++

- `if.. else` : (identique qu'en C++)

© Karim Kalti

- `switch` : globalement, la même syntaxe qu'en C++. Il existe toutefois deux différences :
  - La valeur à tester peut être, en plus des types entiers, de type string.
  - Il y a une différence au niveau du passage d'un "case" à un autre en cas d'absence de "break".
- `for, while et do..while` : (identique qu'en C++)
- Les instructions `break, continue et goto` : (identique qu'en C++)

Différences par rapport au C++

Essentiellement au niveau de l'évaluation des expressions :

```
C++ : if(i) ⟶ C# : if(i!=0)
C++ : if(!i) ⟶ C# : if(i==0)
```

Nouvelle boucle : foreach

Permet de parcourir les tableaux et les collections.

Syntaxe :

```
Type[] Tab = new Type [Taille] ;
… … … …
foreach (Type Var in Tab)
        Bloc d'instructions
```

Remarque :

Cette boucle ne peut être utilisée que pour lire les valeurs des éléments d'un tableau. Elle ne permet pas d'effectuer des modifications du contenu d'un tableau.

Exemple :

```
int [] TI = new int[]{15,30,45,60} ;
foreach (int i in TI)
        Console.WriteLine(i);

string[] TS={ "ALI", "SALAH","KAMEL"};
foreach(string s in TS)
        Console.WriteLine(s);
```

Les fonctions

Déclaration :

- Syntaxe : Globalement comme en C++.
- Pas de fichiers d'entêtes contenant les déclarations des fonctions.

Passage de paramètres :

- Pas de passage par adresse.
- Passage par valeur : comme en C++.
- Passage par référence : il existe une différence syntaxique par rapport au C++.
- Les tableaux et les objets sont toujours passés par référence. Les paramètres possédant les autres types (int, float, string, …, struct) peuvent être passés par valeur ou par référence.

Passage par valeur

```
using System ;
class Prog
{
        // Fonction prenant comme paramètres deux entiers et retournant un entier
        static int Somme(int a, int b)
        {return a+b;}
        // Fonction prenant comme paramètre un tableau et ne retournant aucune valeur
        static void AfficherTab(int[] T)
        {
          foreach (int i in T)
            { Console.Write(i+" "); }
        }
        // Fonction retournant un tableau
        static int[] CréerTableau()
        {  int[] T= new int[]{1,2,3};
           return T;
```


© Karim Kalti

```
        }
        static void Main( )
        {int a = 3; int b = 7;
         Console.WriteLine("Somme de a et b:"+Somme(a,b));
         Console.WriteLine("Contenu du tableau");
         AfficherTab(CréerTableau());
        }
}
```

Passage par référence

- Il remplace le passage par adresse du C++.
- Si le paramètre à passer à la fonction est de type "valeur" alors le passage de sa référence se fait à l'aide du mot-clé *ref* comme suit :

  Déclaration : `void  f(ref Type Param) ;`
  Appel : `f(ref Param) ;`

Exemple :

```
using System;
class Prog
{ static void Permuter(ref int x, ref int y)
   { int temp;
     temp = x;
     x=y;
     y=temp;
   }
  static void AfficherValeurs(int a, int b)
   {  Console.WriteLine("a :"+a);
      Console.WriteLine("b :"+b);
   }
  static void Main()
   {  int a =5; int b =6;
      Permuter(ref a, ref b);
      AfficherValeurs(a,b);
   }
}
```

- Une variable ayant un type référence (Exemple : tableau, objet d'une classe, …) passe par défaut par référence lorsqu'elle est transmise comme paramètre à une fonction. Le mot-clé *ref* n'est pas utilisé dans ce cas.

Passage d'arguments non initialisés

- Un argument passé à l'aide du mot-clé *ref* doit être obligatoirement initialisé. En d'autres mots, il doit avoir nécessairement une valeur au moment de sa transmission à la fonction lors de l'appel de cette dernière.
- Toutefois, plusieurs situations en programmation nécessitent l'utilisation d'arguments sans valeurs initiales. C'est le cas par exemple des fonctions qui ont besoin de retourner plusieurs valeurs. Ces valeurs ne peuvent pas être renvoyées toutes par *return*. Elles peuvent plutôt être placées dans des arguments qui sont passées par référence à la fonction.
- Le passage d'arguments non initialisées ne peut pas se faire avec *ref*. Il est fait plutôt avec le mot-clé *out*.
- Un argument de type *out* est un paramètre qui passe par référence sans nécessiter toutefois d'être initialisé au moment de sa transmission.
- Un argument de type *out* est initialisé par la fonction appelée.

Syntaxe :

Déclaration : **`Type  Fct(out TypeParam NomParam)`**
Appel : **`Fct(out NomArgument)`**

Exemple 1:

```
using System;
class Prog
{
        static void CalculerTriple(int x, out int triple)
        { triple = 3*x;}

        static void Main()
        {
                int x = 5;
```


© Karim Kalti

```
        int t;
        CalculerTriple(x,out t);
        Console.WrilteLine("le triple est : " + t);
    }
}
```

Le paramètre triple n'a pas de valeur au moment de l'appel de la méthode *CalculerTriple*. Il prend sa valeur à l'intérieur de cette dernière. C'est pourquoi il est passé à l'aide de *out*.

Exemple 2: Saisie d'un tableau

```
using System;
class Prog
{
    static void Saisie(out int[] Tab)
    {
        Tab = new int[3];
        Tab[0]=21; Tab[1]=85; Tab[2]=5;
    }


    static void Main()
    {
        int[] Tab;
        Saisie(out Tab);
        ....
        ....
    }
}
```

Remarques :

- Une propriété de classe ne peut pas être passée comme un argument de type *ref* ou de type *out*.
- La surcharge de méthodes est possible si les signatures de deux méthodes diffèrent seulement par un *ref* ou seulement par un *out*.
- La surcharge n'est pas valide pour deux méthodes dont les signatures diffèrent par *ref* et par *out*.

Exemples:

// Exemple de surcharge correcte car les deux signatures diffèrent seulement par *out*

```
void f(int i){… … …}
void f(out int i){… … …}
```

// Exemple de surcharge incorrecte car les deux signatures ne peuvent pas se distinguer par *ref* et *out*.

```
void g(ref int i){… … …}
void g(out int i){… … …}
```

Les structures

- Les structures constituent un moyen de construction de types personnalisés. Une structure permet de regrouper plusieurs champs dans une même entité.
- Une structure est toujours de type valeur.

Déclaration

struct NomStructure
{ ModificateurAcces type1 champ1;
 ModificateurAcces type2 champ2;
 … … …
 ModificateurAcces type3 champ3;
}

- *ModificateurAcces* désigne le modificateur de droit d'accès (*public, private, internal*).
- L'utilisation d'un point virgule à la fin de la déclaration d'une structure est optionnelle.

Exemple :

```
struct Pers
{       public string Nom;
        public int Age;
}
```


© Karim Kalti

Déclaration d'une variable de type structure

Syntaxe : **`NomStructure NomVar;`**

NomVar est une variable de type valeur.

Exemple : `Pers P;`

Accès aux champs

Lorsqu'un champ d'une structure est publique alors il peut être accédé comme suit :
NomVar.Champ;

Exemple :

```
P.Nom = "Toto";
P.Age = 20;
```

Constructeur d'une structure

- Une structure peut avoir 0, un ou plusieurs constructeurs.
- Une structure ne peut pas avoir un constructeur par défaut explicitement défini. Le seul constructeur par défaut accepté est celui implicite. Ce dernier est automatiquement généré même si la structure comporte des constructeurs paramétrés explicitement définis. Le constructeur par défaut d'une structure initialise automatiquement les champs numérique à 0 et les champs de type chaînes à "".
- Un constructeur paramétré d'une structure doit obligatoirement initialiser tous les champs de la structure. (une initialisation partielle engendre une erreur).

Exemple 1 :

```
struct Pers
{       public string Nom;
        public int Age;
       Pers(){Nom = Toto;Age=20;} // Erreur
}
```

Exemple 2 :

```
struct Pers
{       public string Nom;
        public int Age;
       Pers(string N){Nom = N;} // Erreur
}
```

- Une structure ne peut pas hériter d'une classe ou d'une structure.
- Les variables structurées peuvent également être déclarées à l'aide de new comme suit :

  Syntaxe : **`NomStructure NomVar = new NomStructure();`**

  Exemple : `Pers P = new Pers();`

Mais même avec cette syntaxe, P reste une variable de type valeur. Cette syntaxe permet tout simplement d'appeler explicitement les constructeurs paramétrés chose qui n'est pas possible avec la première écriture.

Méthodes d'une structure

- Une structure peut avoir des méthodes. Ces dernières sont déclarées à l'intérieure de la structure.

Exemple :

```
using System;
struct Pers
{
        public string Nom;
        public int Age;
        public Pers(string N, int A){ Nom=N; Age=A;}

        public void Afficher()
        { Console.WriteLine("Nom :"+Nom+" Age : "+Age);}
}
class Prog
{
        static void Main()
        {
                Pers P = new Pers("Ali",20);
                P.Afficher();
        }
```


© Karim Kalti

```
}
```

Remarque :

En C#, tous les types de données scalaires tels que *int, float, double,* … sont implémentés sous forme de structures intégrées.

Exemple :

```
int i = 5;
int j = new int(5); // initialisation de j à 5.
int k = new int(); // initialisation automatique de k à zéro.
int L; // L n'est pas initialisé.
```

Les énumérations

- Une énumération est un outil permettant de construire un type ayant un ensemble fini de valeurs.
- Une énumération définit toujours un type valeur.

Déclaration : enum NomEnumeration {Valeur1, Valeur2, …, ValeurN};

- Les valeurs d'une énumération sont codées comme des entiers de type *int*. Dans ce cadre *Valeur1* vaut par défaut 0, *Valeur2* vaut 1 et ainsi de suite.
- Il est possible d'attribuer d'une manière explicite d'autres constantes entières aux valeurs d'une énumération comme suit :
  `enum NomEnumeration {Valeur1=20, Valeur2=65, …, ValeurN};`

Accès aux valeurs d'une énumération

Syntaxe : **`NomEnumération.Valeur`**

Déclaration d'une variable de type énumération

Syntaxe : **`NomEnumération Var;`**

Affectation de valeur

Syntaxe : **`Var = NomEnumération.Valeur;`**

Une variable énumération est toujours de type "valeur".

Exemple :

```
using System;
enum Couleur {Rouge, Orangé, Vert};
class Prog
{  static void Message(Couleur Feu)
   { switch (Feu)
        {
          case Couleur.Vert :
               Console.WriteLine("Vous pouvez passer");
                break;
          case Couleur.Orangé :
               Console.WriteLine("Préparez vous à vous arrêter");
                break;
           case Couleur.Rouge :
               Console.WriteLine("Arrêtez immédiatement");
                break;
        }
  }
 static void Main()
 {
  Couleur Feu;
  Feu = Couleur.Vert;
  Message(Feu);
 }
}
```


© Karim Kalti

La conversion de données

Conversion implicite

Certaines conversions sont effectuées implicitement par le compilateur car de par leur sens et les types qu'elles font intervenir elles ne peuvent jamais engendrer des erreurs ou des pertes de données.

Exemple:

```
int a=5;
long b=a;
```

Conversion explicite

Certaines conversions peuvent ne pas être sûres. Il incombe alors au programmeur de les faire explicitement s'il en a besoin. Ces conversions ne mènent pas toujours vers un résultat correct.

Syntaxe :

Type1 Var1;
Type2 Var2 = (Type2) Var1;

Exemple :

```
long a=5;
int b= (int)a;
```

Si *a* est dans la plage de valeur du type *int* alors la conversion est acceptée. En cas de dépassement de capacité, la conversion sera refusée et une exception de type *System.OverflowException* sera levée.

Conversion entre types "référence"

Pour les objets la conversion n'est définie qu'entre un objet parent et un objet enfant. Elle se fait :

- D'une manière implicite d'un objet enfant vers un objet parent.
- Elle doit être effectuée d'une manière explicite dans le sens contraire (parent-enfant). Aucune vérification n'est effectuée au moment de la compilation. Si une perte de données peut être causée alors la conversion est annulée et une exception de type *System.InvalidCastException* est levée.

Syntaxe :

TypeParent Var1;
TypeEnfant Var2 = (TypeEnfant) Var1;

Le boxing et le unboxing

Le boxing

- Le boxing consiste à convertir une variable de type "valeur" en une référence de type *System.Object*.
- Le boxing peut se faire d'une manière implicite ou explicite:

Syntaxe :

TypeValeur Var;
Object box = Var; // Boxing implicite
Object box = (Object) Var; // Boxing explicite

Exemple1 :

```
int i =123;
Object box =(object)i; // boxing explicite
```

Ou également d'une manière implicite

```
Object box = i;
```

Exemple 2 :

Un des prototypes de *WriteLine* est le suivant : `void WriteLine(Object);`
L'appel suivant de cette méthode est rendu possible grâce au boxing implicite :

```
int n=5;
Console.WriteLine(n); // boxing de int dans object
```


© Karim Kalti

Le unboxing

- Le unboxing est l'opération inverse du boxing. Il consiste à convertir une référence de type *System.Object* vers une variable de type "valeur".
- Le unboxing ne peut se faire que d'une manière explicite.

Syntaxe :

```
Object box;
…
TypeValeur Var = (TypeValeur) box;  // La conversion n'est pas toujours possible
```

Exemple :

```
i = (int) box;
```

Remarque :

Le compilateur C# assure des opérations de boxing et de unboxing implicites entre les types "valeur" intégrés (natifs) et les classes qui leur correspondent en .NET. Ainsi par exemple :

- Une méthode qui prend comme paramètre formel *Int32* peut accepter un paramètre effectif de type *int* (boxing implicite).
- Une méthode qui prend un paramètre formel de type *int* peut accepter un paramètre effectif de type *Int32* (unboxing implicite).

© Karim Kalti

Quelques classes usuelles du FrameWork.NET

La classe String

Cette classe permet de stocker et de gérer les chaînes de caractères.

Caractéristique d'une chaîne.

- Chaque caractère est codé sur deux octets.
- La gestion de la fin de la chaîne est une affaire interne (transparente % à l'utilisateur, pas de zéro de fin de chaîne).

Constante chaîne de caractères

Une constante chaîne de caractères se présente comme une suite de caractères placée entre deux double-quotes.
Exemple : "abcd".

Opérateurs et chaînes de caractères

La majorité des opérateurs (=, ==, !=) ont été redéfinis pour travailler avec les chaînes à la manière des types "valeur" malgré que string soit un type "référence".

Copie de chaînes

L'opérateur = effectue une copie du contenu des chaînes et non une copie de références.

Exemple :

```
string s1 ="ABC";
string s2;
s2=s1;
s1="XYZ";
// s2 contient ABC alors que S1 contient XYZ.
```

Comparaison de chaînes

L'opérateur == effectue une comparaison des contenus des chaînes et non une comparaison de références

Exemple :

```
string s1="Bonjour", s2="Hello";
if(s1==s2)...
if(s1==bonjour)
```

Extraction d'un caractère

L'opérateur [] permet d'extraire un caractère de la chaîne mais il n'agit qu'en lecture :

Exemple :

```
string s= "bonjour";
char c=S[0]; // C contient 'B'
s[0] = 'X'; // erreur de syntaxe
```

Concaténation de chaînes

Les opérateurs + et += permettent de faire des concaténations de chaînes

Exemple :

```
string  s, s1 = "Bon", s2 ="jour";
s = s1+s2;
s+=" Monsieur";  // Bonjour monsieur
```


© Karim Kalti

Exemples de méthodes de la classe String

Méthode	Signification
`int IndexOf(char c)`	Renvoie la position du caractère *c* dans la chaîne qui porte l'opération.(0 pour la première position et –1si *c* n'a pas été trouvé)
`int IndexOf(char c, int pos)`	Même chose que la version précédente sauf que la recherche commence à partir de la position *pos*.
`int IndexOf(string s)`	Recherche la position de la chaîne *s*.
`int IndexOf(string s,int Pos)`	Recherche la chaîne *s* à partir de la position *pos*.
`String Insert(int pos, string s)`	Insère la chaîne *s* dans la chaîne appelante et ce à partir de la position *pos*.
`String ToUpper()`	Convertit la chaîne en majuscule.
`String ToLower()`	Convertit la chaîne en minuscule.

Classe Array

- Cette classe représente les tableaux en .NET.
- Tout tableau créé avec la syntaxe du C# se comporte comme un objet de type *Array*.

Quelques Propriétés

- *Length* : Nombre total d'éléments, toutes dimensions confondues.
- *Rank* : Nombre de dimensions.

Exemple :

```
int[] t1 = {1,2,3};  // Length = 3 rank = 1
int [ , ] t2 = {{1,2,3},{10,20,30}};  // Length = 6 rank = 2
int [][] t3 = new int[2][];  // Length = 2 Rank = 1
t3[0]= new int[]{1,2,3};
t3[1] = new int[]{10,20,30,40};
// t3.Length vaut 2, t3[0].Length vaut 3, t3[1].Length vaut 4
```

Quelques méthodes

Méthode	Signification
`void copyTo(Array t, int pos)`	Copie tout le tableau sur lequel porte l'opération dans le tableau passé en argument à partir de la position *pos*. (Il existe d'autres surdéfinitions de cette méthode).

`int IndexOf(Array t, object o)`	Méthode statique qui renvoie l'indice de la première occurrence du second argument dans le tableau *t* ou –1 si *o* n'a pas été trouvé. (Il existe d'autres surdéfinitions de cette méthode).
	`string[] ts={"Ali", "salah", "salem", "mehdi"};`
	`n = Array.IndexOf("Salah"); // n=1`

`void reverse(array)`	Méthode statique qui inverse la séquence d'éléments dans le tableau
	`int [] ts ={10,20,30};`
	`Array.reverse(ts); // 30 , 20 , 10`

`void Sort(array)`	Méthode statique qui trie un tableau à une dimension
`int Binarysearch(Array, Objet)`	Méthode statique qui recherche un objet dans un tableau. Elle renvoie l'emplacement de l'objet dans le tableau ou une valeur négative si l'objet n'a pas été trouvé. Le tableau doit être trié car c'est la technique de recherche dichotomique qui est utilisée.

`int GetLength(int Dimension)`	Renvoie le nombre d'éléments suivant une dimension spécifique.
	`int[ , ] M = new int[3,6];`
	`for(int i = 0;i<M.GetLength(0);i++){`
	`for(int j = 0;j<M.GetLength(0);j++){`
	`M[i][j] = i+j;`
	`}}`

© Karim Kalti

Autres classes intéressantes

La classe Math

Elle renferme toutes les fonctions mathématiques.

<u>Exemple : fonction de calcul de la valeur absolue</u>

```
int i;
i = Int32.Parse(Console.ReadLine());
Console.WriteLine("Valeur absolue de i :"+ Math.Abs(i));
```

Il existe également d'autres classes qui peuvent s'avérer utiles comme :

- La classe *ArrayList* : pour la gestion des tableaux dynamiques.
- La classe *DateTime* : pour la gestion de la date et l'heure.
- Etc…

© Karim Kalti

Propriétés et Indexeurs

Les propriétés

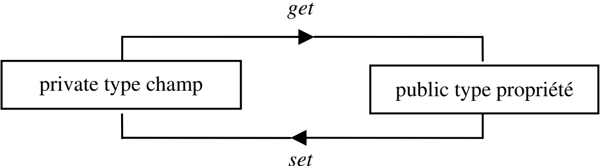
- Une propriété est un champ public particulier, qui est généralement associé à un champ privé de la classe et qui permet de définir les droits d'accès en lecture et en écriture à ce champ privé.
- La définition de ces droits d'accès est faite à travers deux méthodes spéciales associées à la propriété. Ces méthodes, appelées accesseurs, sont les méthodes *get* et *set*.

Accesseur get

- Cette méthode rend la propriété accessible en lecture.
- Elle retourne la valeur du champ auquel est associée la propriété.
- La méthode *get* est automatiquement exécutée lors de la lecture de la propriété.

Accesseur set

- Cette méthode rend la propriété accessible en écriture.
- Elle affecte une valeur au champ associé à la propriété.
- La méthode *set* est automatiquement exécutée lors de l'affectation d'une valeur à la propriété.



Syntaxe :

```
public type propriété{
  get{return champ ;}
  set{champ = value;}
}
```

value est un mot-clé du C# qui contient la valeur affectée à la propriété.

Exemple :

```
int public prop1{
  get{return champ ;}
  set{champ = value; }
}
```

Remarque :

- Si l'accesseur get est absent alors la propriété est en écriture seule.
- Si l'accesseur set est absent alors la propriété est en lecture seule.

Exemple :

```
.....
private int champ;
public int prop1{
get {return champ*10;}
set {champ = value + 5 ;}
}
.....
prop1 = 14 ; // Champ contient alors 19
int X = prop1; // X vaut 190
```

- Outre la protection des champs, les accesseurs servent également à automatiser les opérations usuelles appliquées aux champs. L'exemple suivant illustre ceci.

© Karim Kalti

Exemple :

```
using System ;
namespace ProjProp
{
  class clA {
    private double prixTTC ;
    private double Taxe = 1.12;
    public double prix
        {
         get {return Math.Round(PrixTTC);}
         set {PrixTTC = value*Taxe;}
        }
  }
  class Class {
    static void Main(){
        clA Obj = new clA();
        double val = 55;
        System.Console.WriteLine("Valeur entrée:"+val);
        Obj.prix = val ;
        System.Console.WriteLine ("Valeur stockée:"+PrixTTC);
        val = Obj.prix ;
        System.Console.WriteLine ("valeur arrondie:"+val) ;
        System.Console.ReadLine ( );
    }
  }
}
```

Propriétés de classes et propriétés d'objets

- Comme c'est le cas pour les champs classiques, il est possible de déclarer des propriétés de classe et des propriétés d'instance.
- Une propriété de classe est déclarée à l'aide du mot-clé *static*.
- Une propriété statique ne peut être associée qu'à un champ statique.

Exemple de propriété de classe:

```
using System ;
namespace ProjProp
{
  class Class {
    static private double PrixTTC ;
    static private double Taxe = 1.12 ;
    static public double prix {
      get {return Math.Round(PrixTTC);}
      set {PrixTTC = value * Taxe;}
      }
    static void Main(){
       double val = 55 ;
        System.Console.WriteLine("Valeur entrée:"+val );
        prix = val ;
        System.Console.WriteLine ("Valeur stockée:"+PrixTTC);
        val = prix ;
        System.Console.WriteLine ("valeur arrondie:"+val) ;
        System.Console.ReadLine();
        }
   }
}
```

Masquage de propriétés

Il est possible de masquer une propriété par une autre à la manière des méthodes et ce à l'aide du mot réservé new.

Exemple :

```
using System;
  class clA {
    private double PrixTTC ;
    private double Taxe = 1.12 ;
    public double prix { // propriété de la classe mère
      get { return Math.Round(PrixTTC);}
      set { PrixTTC = value * Taxe;}
    }
}
```


© Karim Kalti

```
class clB : clA {
     private double prixLocal ;
     public new double prix { // masquage de la propriété de la classe mère
         get {return Math.Round(prixLocal);}
         set { prixLocal = value * 1.05 ; }
   }
}
class Class {
    static void Main(){
    clA Obj = new clA();
    double val = 55 ;
    System.Console.WriteLine("Valeur entrée obj1:"+val);
    Obj.prix = val ;
    val = Obj.prix ;
    System .Console.WriteLine ("valeur arrondie obj1:"+val);
    System .Console.WriteLine ("--------------------");
    clB Obj2 = new clB ( );
    val = 55 ;
    System .Console.WriteLine ("Valeur entrée Obj2:"+val );
    Obj2.prix = val ;
    val = Obj2.prix ;
    System .Console.WriteLine ("valeur arrondie Obj2:"+val ) ;
  }
}
```

Remarque :

Les propriétés peuvent également être virtuelles et redéfinies dans les classes dérivées.

Les indexeurs

- Les indexeurs constituent un moyen permettant d'accéder aux objets à la manière des tableaux.
- Un indexeur est déclaré avec le mot-clé *this*. Il possède deux méthodes *get* et *set* comme les propriétés.

Exemple 1 : Définition d'un indexeur

```
class clA
{
   private int[] champ;
   public clA(){champ = new int [5]};
   public int this [int index]{
   get { return champ[ index ] ; }
   set { champ[ index ] = value ; }
}
```

Exemple 2 : Utilisation d'un indexeur

```
 clA Obj = new clA( );
 for (int i =0; i<5; i++ )
   Obj[i] = i*10 ;
 int x = Obj[ 2 ] ; // x = 20
 int y = Obj[ 3 ] ; // y = 30
 ...
```

Remarque :

- L'index utilisé par l'indexeur peut être de n'importe quel type.
- Plusieurs indexeurs peuvent être définies pour une même classe. Ils doivent se distinguer obligatoirement par leurs signatures.

Exemple :

```
using System;
class Père
{
        string[] Enfants;
        public void SesEnfants(string[] E)
        {
                Enfants = new string[E.Length];
                for(int i=0;i<Enfants.Length;i++)
                        Enfants[i]=E[i];
        }
```


© Karim Kalti

```
        public string this[int n]
        {
                get{ return Enfants[n];}
                set{ Enfants[n] = value;}
        }

        public int this[string Nom]
        {
                get{
                        for(int i=0;i<Enfants.Length;i++)
                                if(Enfants[i]==Nom)
                                  return i;

                        return -1;
                    }
        }

 }
class Prog
{
        public static void Main()
        {
         Père Ammar = new Père();
         String[] Noms = {"Salah","Ali","Mehdi"}
         Ammar.SesEnfants(Noms);
         Console.WriteLine(Ammar[1]);
         Console.WriteLine(Ammar["Salah"]);
         Ammar[0]="Salem";
         Console.WriteLine(Ammar[0]);
        }
}
```


© Karim Kalti

Délégués et événements

Rappel C/C++

En C/C++, le nom d'une fonction, utilisé seul, désigne l'adresse en mémoire où sont stockées les instructions de cette fonction. De ce fait, il est possible de déclarer des pointeurs sur les fonctions et d'initialiser ces pointeurs.
La déclaration d'un pointeur sur une fonction se fait comme suit : **`Type (*pf)(arg1, …, argn)`**

Exemple :

`int (*pf)(int i double j)`

pf est un pointeur qui peut pointer sur toute fonction qui prend comme paramètres un entier et un double et qui retourne un entier.
Soit la fonction : `int f (int K, double L) ;`
Si on initialise *pf* avec *f* : *pf* = *f*
Alors l'appel de *f* peut être fait à travers *pf* :
`int i = f(5,7.8)` est équivalent à `int i = (*pf)(5,7.8)`

La notion de délégué

- En C#, les pointeurs sur les fonctions sont implémentés à l'aide des délégués.
- Un délégué est un objet particulier qui peut référencer (pointer vers) une ou plusieurs méthodes.
- Comme les pointeurs sur les fonctions, un délégué doit être défini pour un prototype donné de méthodes. Il peut par la suite référencer n'importe quelle méthode ayant ce prototype.
- La méthode référencée peut changer en cours de l'exécution du programme.
- L'appel de la méthode peut se faire dans ce cas à travers le délégué.

Déclaration d'un type délégué

Pour pouvoir déclarer un délégué, il faut définir auparavant le modèle (prototype) des méthodes qui peuvent être référencées par ce délégué. Ce modèle de méthodes définit ce que l'on appelle un type délégué.
La définition d'un type délégué se fait à l'aide du mot-clé *delegate* comme suit :

```
delegate TypeRetour TypeDéléqué(type1 arg1,..., typeN argN)
```

Instanciation d'un objet de type délégué

Une fois le type délégué défini, il devient possible de créer des délégués de ce type. Cette création se fait comme suit :

```
Typedélégué Deleg = new Typedélégué(NomMéthode)
```

- *Deleg* est un délégué qui peut référencer des méthodes ayant le prototype donné dans la déclaration de *TypeDélégué*.
- *NomMéthode* désigne ici le nom de la méthode qui sera référencée par *Deleg*.
- L'appel de la méthode *NomMéthode* peut se faire à travers *Deleg*.

Exemple:

```
delegate string Délégué(int x);
class ClasseA {
static string Fonc1(int x)
{
  return (x*10).ToString();
}
```


© Karim Kalti

```
static string Fonc2(int x)
{return (x*100).ToString();}
static void Main(){
string s = Fonc1(50); // appel de fonction classique
System.Console.WriteLine("Fonc1(50) = " + s );
s = Fonc2(50); // appel de fonction classique
System.Console.WriteLine("Fonc2(50) = " + s );
System.Console.WriteLine ("délégué référence Fonc1 :");
Délégué FoncDeleg = new Délégué (Fonc1) ;
s = FoncDeleg(50); // appel au délégué qui appelle la fonction
System.Console.WriteLine ("FoncDeleg(50) = " + s );
System.Console.WriteLine ("délégué référence maintenant Fonc2 :");
FoncDeleg = new Délégué ( Fonc2 ) ; // on change d'objet référencé (de fonction)
s = FoncDeleg ( 50 ); // appel au délégué qui appelle la fonction
System.Console.WriteLine ("FoncDeleg(50) = "+s);
System.Console.ReadLine ( );
}
}
```

Remarque :

Il est à noter qu'il est tout à fait possible de créer un délégué faisant référence à une méthode d'instance. Cette création se fait comme suit :

1- Déclarer une classe contenant une méthode publique :

```
class clA {
  public int meth1(char x) { .... }
}
```

2- Déclarer un type délégation :

```
delegate int Deleg( char x ) ;
```

3- Instancier un objet de la classe clA :

```
clA ObjA = new clA ( ) ;
```

4- Instancier à partir de la classe Deleg un délégué :

```
Deleg FoncDeleg = new Deleg(ObjA.meth1);
```

Délégué multicast

- En C/C++, la fonction pointée par un pointeur peut être variable et peut donc changer durant l'exécution du programme. Toutefois, à un instant donné, le pointeur ne peut pointer que sur une seule fonction.
- Par rapport à ceci, en C#, un délégué peut faire référence à un instant donné à plusieurs fonctions en même temps. Ce type de délégué est appelé délégué multicast.
- Les méthodes référencées par un délégué multicast doivent avoir le même prototype. Elles peuvent être des méthodes de classes ou des méthodes d'instances.
- L'ajout d'une référence à un délégué multicast se fait à l'aide de l'opérateur d'adition += comme suit :

  `NomDélégué += new TypeDélégation(NomMéthode);`

Exemple :

```
delegate int Délégué(char x);
class ClasseA {
public int champ;
public int meth100 ( char x ) {
System.Console.WriteLine ("Exécution de meth100('"+x+"')");
return x+100 ;
}
public int meth101 ( char x ) {
System.Console.WriteLine ("Exécution de meth101('"+x+"')");
return x+101 ;
}
public int meth102 ( char x ) {
System.Console.WriteLine ("Exécution de meth102('"+x+"')");
return x+102 ;
}
public static int meth103 ( char x ) {
System.Console.WriteLine ("Exécution de meth103('"+x+"')");
return x+103 ;
}}
class Prog {
static void Main ( ) {
ClasseA ObjA = new ClasseA( );
//-- instanciation du délégué avec ajout de 4 méthodes :
Délégué FoncDeleg = new Délégué( ObjA.meth100 ) ;
```


© Karim Kalti

```
FoncDeleg += new Délégué ( ObjA.meth101 ) ;
FoncDeleg += new Délégué ( ObjA.meth102 ) ;
FoncDeleg += new Délégué ( ClasseA.meth103 ) ;
//--la méthode meth103 est en tête de liste :
//--Appel du délégué sur le paramètre effectif 'a' :
ObjA.champ = FoncDeleg('a') ; //code ascii 'a' = 97
System.Console.WriteLine ( " valeur du champ : "+ObjA.champ) ;
}}}
```

Résultat de l'exécution du programme :

```
Exécution de meth100('a')
Exécution de meth101('a')
Exécution de meth102('a')
Exécution de meth103('a')
valeur du champ : 200
```

Suppression d'une référence à une méthode dans un délégué multicast :

Il est possible de supprimer une référence à une méthode à l'aide de l'opérateur -= comme suit :

```
NomDélégué -= new TypeDélégation(NomMéthode);
```

Exemple :

```
FoncDelg -= new délégué (ClasseA.meth103) ;
```

Les événements

- Les événements permettent de capturer une action dans un programme. Ils jouent un rôle très important dans les programmes tournant sous windows (utilisant les contrôles graphiques de windows : bouttons, formulaires, menus).
- Les événements définissent un style de programmation à part entière appelé programmation événementielle.
- Lorsqu'un événement a lieu, il doit être intercepté (par la machine virtuelle CLR dans les programmes managés et par Windows dans les programmes Win32). Le programme doit réagir également à cet événement et entreprendre les actions nécessaires.

Implémentation d'un mécanisme de détection et de gestion d'événements.

Pour réaliser une gestion des événements en .Net, il faut procéder comme suit :

- Définir le type des événements à détecter.
- Définir un module de détection (MD) de ce type d'événements appelé également source d'événements.
- Définir un module de traitement des événements (MT) appelé également gestionnaire d'événements.
- Lorsqu'un événement est détecté par le MD ; ce dernier doit connaître le MT capable de traiter cet événement. Pour cela le MT doit être connu par le MD pour que ce dernier puisse l'informer. On dit que le MT doit être dans la liste de notification du MD.
- La liste de notification d'un MD peut comporter un ou plusieurs MT.
- Un MT peut se retirer de la liste de notification d'un MD s'il ne veut plus traiter les événements détectés par ce dernier.
- Le traitement d'un événement par le MT se fait en réalité à travers une fonction FT de ce dernier ayant le même prototype que l'événement.
- Le MD doit être capable d'appeler les FT de n'importe quel MT faisant partie de sa liste de notification. Pour cela il doit disposer d'un délégué qui peut faire référence à ces FT.

Déclaration du type de l'événement (délégué)

1- Définition du type de délégation

```
public delegate TypeRetour TypeEvenement(Liste d'arguments);
```

2- Déclaration de l'événement à l'aide du mot-clé event.

```
public event TypeEvenement NomEvenement;
```

Normalisation des événements en .NET

pour déclarer l'événement il est possible d'utiliser un délégué ayant n'importe quel prototype, suivant les besoins du programme bien sûr. Toutefois le .Net recommande d'utiliser l'un des deux prototypes normalisés suivants :

```
delegate void TypeDelegation(Object sender, EventArgs e)
delegate void TypeDelegation(Object sender, MonEventArgs e)
```


© Karim Kalti

où :

- *sender* désigne l'objet source de l'événement.
- *e* désigne un objet contenant les paramètres de l'événement.
- *e* peut ne pas contenir d'informations. Dans ce cas il est de type *EventArgs*. *EventArgs* est la classe de base des classes qui peuvent contenir les informations associées aux événements. Elle comporte quelques fonctions utilitaires telles que le test d'égalité entre objets, la conversion de données vers le type string, etc.
- *e* peut comporter des informations personnalisées. Dans ce cas *e* doit être un objet dérivé de la classe *EventArgs* (exemple *MonEventArgs*). Les champs ajoutés lors de la dérivation contiennent normalement les données personnalisées de l'événement.

Autres recommandations :

- Le nom du type de délégation doit se terminer par *EventHandler*.
- Le nom de la méthode qui déclenche l'événement doit être de la forme ***On****NomEvenement.* Cette méthode est généralement déclaré comme protégée.
- Le lancement de l'événement se fait à travers une méthode publique qui appelle la méthode protégée de déclenchement de l'événement.

Etapes pour la mise en place d'un événement avec informations personnalisées

1- Une classe d'informations personnalisées sur l'événement.
2- Une déclaration du type délégation normalisée (nom qui se termine par *EventHandler*).
3- Une déclaration d'une référence *NomEvenement* du type délégation normalisée spécifiée *event***.**
4- Une méthode protégée *OnNomEvenement* qui déclenche l'événement *NomEvenement*.
5- Une méthode publique qui lance l'événement par appel de la méthode *OnNomEvenement*.
6- Un ou plusieurs gestionnaires de l'événement *NomEvenement*.
7- Abonner ces gestionnaires au délégué *NomEvenement.*
8- Consommer l'événement *NomEvenement*.

Déroulement des étapes à travers un exemple :

```
using System;
//1°) une classe d'informations sur l'événement
public class ExamenEventArgs
{
        public string DateExamen;
        public ExamenEventArgs( string S){ DateExamen = S;}
}

// 2°) une déclaration du type délégation
public delegate void EvenementExamen(Object sender, ExamenEventArgs e);

public class MD
{
        // 3°) une déclaration d'une référence NomEvenement du type de délégation
        public event EvenementExamen Examen;
        // 4°) une méthode protégée qui déclenche l'événement NomEvenement
        protected void OnExamen (Object sender, ExamenEventArgs e)
        {
                if (Examen!=null)
                        Examen(sender,e);
        }
        // 5°) une méthode publique qui lance l'événement par appel de la
        // méthode NomEvenement
        public void LancerEvenement()
        {
                ExamenEventArgs e = new ExamenEventArgs("15-01-05");
                OnExamen(this, e);
        }
}
class MT
{
        // 6°) un ou plusieurs gestionnaires de l'événement NomEvenement
        static void Message(Object sender, ExamenEventArgs e)
        {
                Console.WriteLine("Attention l'examen s'approche");
                Console.WriteLine("la date de l'examen est: " + e.DateExamen);
        }
```


© Karim Kalti

```
        // 7°) abonner ces gestionnaires au délégué NomEvenement
        public MT(MD md)
        {
                md. Examen+= new EvenementExamen(Message);
        }
}

class Prog
{
        static void Main()
        {
                MD md = new MD();
                MT mt = new MT(md);
                // 8°) consommer l'événement
                md.LancerEvenement();
        }
}
```

Complément sur les événements

- Un événement est un message envoyé par un objet pour signaler l'occurrence d'une action. Cette action peut être causée par une interaction avec l'utilisateur (généralement à travers les interfaces graphiques comme par exemple le clic de la souris). Elle peut être également déclenchée par la logique d'exécution du programme.
- L'objet qui déclenche l'événement est appelé "source de l'événement" ou également émetteur de l'événement (*event sender*).
- L'objet qui capture l'événement et qui le traite est appelé "gestionnaire de l'événement" (*event handler*) ou également récepteur de l'événement (*event receiver*).
- La classe qui émet un événement ne connaît pas l'objet ou la méthode qui va traiter cet événement. Un intermédiaire entre l'émetteur et le récepteur est alors nécessaire. Cet intermédiaire n'est autre qu'un délégué qui référence le récepteur. L'appel de ce délégué par l'émetteur va engendrer par conséquent l'exécution du récepteur.

Différence entre un délégué et un événement

Les événements sont implémentés en .NET sous forme de délégués. Toutefois, par rapport à un délégué classique, un événement possède quelques restrictions supplémentaires notamment en ce qui concerne l'accès et l'invocation.

Différence au niveau de l'invocation :

- Même s'il est déclaré en tant que membre public, un événement ne peut être invoqué que de l'intérieur de sa classe. Ce n'est pas le cas des délégués qui se comportent avec les modificateurs d'accès à la manière des champs classiques.
- La seule invocation possible d'un événement de l'extérieur d'une classe est une invocation indirecte qui passe par l'utilisation d'une méthode publique de cette classe qui invoque en interne l'événement.

Exemple :

```
delegate void TypeEvent(int i);
        class X
        {
                // Déclaration de l'événement
                public event TypeEvent MyEvent;

// Méthode publique qui invoque l'événement
                public void InvokeEvent()
                {
                        if(MyEvent!=null)
                                MyEvent();
                }
        }

class MainClass
        {
                // Fonction de traitement de l'événement
                static void FT()
                {
                        Console.WriteLine(" Traitement de l'événement);
                }
```


© Karim Kalti

```
        public static void Main()
        {
                X x = new X();
                // Inscription de la fonction de traitement
                x.MyEvent+= new TypeEvent(FT);
                // Invocation indirecte de l'événement
                x.InvokeEvent();
        }
}
```

- Pour pouvoir invoquer un événement à partir d'une classe dérivée, il suffit de déclarer une méthode protégée dans la classe de base qui invoque en interne l'événement. (il s'agit toujours d'une invocation indirecte)
- Même si la méthode protégée est définie comme virtuelle et qu'elle est redéfinie dans la classe dérivée, alors la redéfinition ne pourra pas invoquer directement l'événement. Tout ce qu'elle peut faire c'est l'appel de sa version protégée de la classe de base.

Exemple :

```
class Y:X
{
        protected override void InvokeEvent()
        {
                // MyEvent(); ERROR
                base.InvokeEvent();
        }


}
```

Le mot-clé *event* permet donc de créer un délégué dont les droits d'invocation sont plus restreints que ceux d'un délégué classique. En effet, par défaut, le droit d'invocation d'un événement est restreint à l'intérieur de la classe dans laquelle est déclarée cet événement. Ensuite, c'est le propriétaire de cette classe qui peut étendre ou non ce droit d'invocation aux utilisateurs de la classe en définissant ou non une méthode (publique ou protégée) d'invocation indirecte.
A la lumière de ce qui vient d'être susmentionné la question suivante se pose : pourquoi déclare-t-on un événement public s'il ne peut pas être invoqué de l'extérieur ?
La réponse à cette question réside dans le fait qu'un événement a besoin de référencer une (ou plusieurs) méthode(s) pour qu'il puisse être traité. Cette opération se fait à travers l'inscription de la (des) fonction(s) de traitement auprès de l'événement. L'inscription se fait de l'extérieur de la classe qui contient l'événement (par un objet externe) ce qui oblige par conséquent l'événement à être accessible de l'extérieur donc public.

<u>Différence au niveau de l'accès</u>

Accès à un délégué

Un délégué classique est accessible de plusieurs manières :

- `Deleg = new TypeDelegation(NomMethode)` pour un délégué unicast ou pour inscrire une première fonction auprès d'un délégué multicast.
- `Deleg+= new TypeDelegation(NomMethode)` pour ajouter une inscription à un délégué multicast qui référence déjà au moins une fonction.
- L'instruction `Deleg = new TypeDelegation(NomMethode)` appliquée à un délégué qui référence déjà une (ou plusieurs) méthode(s) engendre la suppression de toutes ces méthodes de la liste de notification du délégué. Le même effet est obtenu avec l'instruction : `Deleg = null;`
- La suppression d'un gestionnaire de la liste de notification d'un délégué multicast se fait à l'aide de -=.

Accès à un événement

A la différence d'un délégué, un événement n'accepte que les accès à l'aide des opérateurs += et -=. Cette restriction permet d'introduire un niveau de sécurité supplémentaire. En effet, de par sa définition un événement doit être capable de déclencher une ou plusieurs actions (*Exemple :* un accident peut déclencher un appel au SAMU et un autre appel à la police). Techniquement cela se traduit par le fait qu'un événement doit être capable d'accepter les inscriptions indépendantes de plusieurs gestionnaires d'événements (fonctions de traitement). Toutefois, si par erreur une inscription se fait à l'aide de l'opérateur = alors que d'autres gestionnaires sont déjà inscrits alors ces derniers seront tout simplement éliminés de la liste de notification. Pour éviter ce risque et pour donner plus de sécurité et d'indépendance au processus d'inscription, le .NET n'autorise un gestionnaire à s'inscrire auprès d'un événement qu'à l'aide de l'opérateur += (ceci est valable même pour l'inscription du premier gestionnaire). De même la désinscription ne peut être réalisée qu'avec l'opérateur -= et elle touche un seul gestionnaire à la fois. Ainsi, si un événement est sollicité par plusieurs gestionnaires, alors chaque gestionnaire

© Karim Kalti

ne pourra agir que sur sa propre fonction de traitement sans risquer de toucher les fonctions des autres gestionnaires.

Remarque :

Une telle restriction permet par exemple de garantir aux composants logiciels que leurs utilisateurs (généralement les programmes qui intègrent ces composants) ne peuvent pas engendrer (par erreur ou par malveillance) en ajoutant un traitement personnalisé à un événement la suppression d'un éventuel traitement prédéfini, intégré par défaut, et indispensable pour le bon fonctionnement de ces composants.

© Karim Kalti

Les classes

Déclaration

- Une classe définit un nouveau type de données.
- Une classe peut comporter :
  - des attributs, des propriétés et des indexeurs.
  - des méthodes, des délégués et des événements.
- Une classe est définie avec le mot réservé *class*.
- Le point virgule après la déclaration d'une classe est optionnel.

Exemple :

```
class Pers
{ string Nom ;
  string Prenom;
  int Age;
}
```

Instanciation des classes (Les objets)

Création

- C# ne permet que la déclaration dynamique d'objets : pas de déclaration d'instance de type auto (au sens du C++).
- Tout objet est une référence.
- Syntaxe :

```
NomClasse RefObjet; // Déclaration d'une référence à un objet
RefObjet = new ConstructeurNomClasse(<arguments>); // instanciation
```

Les deux instructions précédentes peuvent être regroupées en une seule comme suit :

```
NomClasse RefObjet = new ConstructeurNomClasse(<arguments>);
```

Exemple :

```
Pers p ;  // p est une référence à un objet de type Pers
p = new Pers( ) ;  // Instanciation
// Les parenthèses sont requises même si le constructeur est absent ou n'admet pas
// d'arguments.
```

Libération

- Elle se fait d'une manière automatique par le GC.
- La libération se fait si :
  - On quitte le bloc où l'objet a été déclaré.
  - On affecte à l'objet la référence d'une nouvelle instance de la classe ou si on lui affecte la référence "null".

Droits d'accès aux membres d'une classe

- Les droits d'accès à un membre d'une classe en C# sont définies à l'aide des mots-clés suivant :
  - *public* : toutes les méthodes (du même assemblage ou non) peuvent accéder à ce champ ou appeler cette méthode.
  - *private* : seules les méthodes de la classe peuvent accéder à ce champ ou appeler cette méthode.
  - *protected* : seules les méthodes de la classe et des classes dérivées (du même assemblage ou non) peuvent accéder à ce champ ou appeler cette méthode.
  - *internal* : seules les méthodes du même assemblage peuvent accéder à ce champ ou appeler cette méthode.

© Karim Kalti

- Les droits d'accès en C# doivent être définis individuellement pour chaque membre de la classe.
- Si aucun droit d'accès n'est explicitement spécifié devant un membre, alors ce dernier sera considéré par défaut comme étant privé.

Valeur initiale d'un champ

L'initialisation d'un champ d'une classe se fait au moment de la définition de cette dernière. (*cf le champ TotalPoints de la classe Jeu*).

Exemple

```
class Jeu
{  int TotalPoints = 100;
   int PointsPerdus;
   int PointsGagnés;
}
```

- Le champ *TotalPoints* est explicitement initialisé à 100. Pour toute nouvelle instance de la classe *Jeu*, ce champ vaudra 100.
- Un champ non explicitement initialisé prend la valeur 0 s'il est numérique, false s'il est booléen, espace s'il est caractère et chaîne vide s'il est de type chaîne.
- La valeur du champ peut être modifiée à tout moment durant l'exécution.

Les champs constants

- Il est possible de définir des champs constants à l'aide des mots-clés *const* et *readonly*.
- Un champ défini par *const* représente une constante de classe. Il prend la même valeur constante pour toutes les instances de sa classe. Un champ *const* doit être initialisé au moment de la définition de la classe.
- Un champ défini par *readonly* représente une constante d'instance. Il prend une valeur constante spécifique pour chaque instance de la classe. Un champ *readonly* est initialisé par le constructeur.
- Une fois initialisé, un champ constant (défini par *const* ou *readonly*) ne peut plus changer de valeur.

Les méthodes

- Le corps d'une méthode doit se trouver dans la définition de la classe. (pas de possibilité de définition des méthodes à l'extérieur de la classe comme en C++).
- Il n'y a pas de fichiers d'entêtes (fichiers .h).
- L'emplacement de la définition d'une classe ou d'une méthode n'a pas d'importance par rapport au lieu de son appel.

Exemple :

```
using System;
class Pers
{ public string Nom;
  public int age;
  public void Affiche()
  {Console.WriteLine(Nom+"("+Age+")");}
}
class Prog
{  static void Main()
   { Pers p = new Pers();
     P.Nom ="Ali";
     P.Age=20;
     P.Affiche();
}}
```

La surcharge de méthodes

- Plusieurs méthodes d'une classe peuvent porter le même nom. Elles doivent cependant se distinguer par leurs signatures (nombre, ordre et types des paramètres). Ces méthodes sont dites surchargées.
- Deux méthodes surchargées ne peuvent pas se distinguer seulement par le type de leurs valeurs de retour.

© Karim Kalti

Exemple :

```
class Pers
{       string Nom="Moi";
        int Age = 20;
        public void Modifier(string N){Nom=N;}
        public void Modifier(int A){Age=A;}
        public void Modifier(string N, int A){Nom=N;Age=A;}
}
class Prog
{       static void Main()
        {
                Pers P = new Pers();
                P.Modifier("Ali");
                P.Modifier("Salah",25);
        }
}
```

Accès aux membres d'une classe

Accès de l'intérieur de la classe

- Les champs et les méthodes d'une classe sont directement accessibles de l'intérieur d'une classe (indépendamment des droits d'accès).
- La manipulation des champs et l'appel des méthodes dans ce cas peuvent se faire directement à travers leurs noms sans nécessiter aucune qualification supplémentaire.

Accès de l'extérieur de la classe

- L'accès aux champs et aux méthodes d'une classe de l'extérieur n'est possible que pour les membres publics.
- Cet accès se fait généralement à partir d'une instance de la classe à l'aide de l'opérateur " . ".
  NomObjet.Attribut
  NomObjet.Methode(<arguments>)

Le mot réservé this

Le mot réservé *this* sert à référencer l'objet en cours d'utilisation.

Exemple d'utilisation : (Arguments de méthodes ayant les mêmes noms que les champs)

```
public void Modifier(string Nom, int Age)
        {this.Nom=Nom;this.Age=Age;}
```

Les constructeurs

Les constructeurs possèdent globalement les mêmes caractéristiques qu'en C++.

- Une classe peut comporter un ou plusieurs constructeurs. Dans le cas d'existence de plusieurs constructeurs ces derniers doivent respecter les règles de surcharge de méthodes.
- Ils doivent porter le même nom que celui de la classe.
- Ils peuvent admettre 0, un ou plusieurs paramètres.
- Ils ne renvoient rien.
- Leur déclaration explicite n'est pas obligatoire. En cas d'absence de déclaration explicite d'un constructeur, un constructeur par défaut est automatiquement généré pour la classe.
- En cas de présence d'au moins un constructeur explicitement défini, alors aucun constructeur par défaut n'est généré.

Exemple 1 : Utilisation du constructeur par défaut

```
Class NomClasse
{
  … … … …
}
objet = new NomClasse(); // Appel du constructeur par défaut
```


© Karim Kalti

Exemple 2 : Classe avec constructeurs paramétrés

```
class Pers
{
        string Nom;
        int Age;
        public Pers(string N)
        {Nom=N;Age=20;}
        public Pers(string N, int A)
        {Nom=N;Age=A;}
        public void Afficher()
        {Console.WriteLine(Nom+"("+Age+")");}
}

class Prog
{       static void Main()
        {       Pers P1 = new Pers("Ali");
                Pers P2 = new Pers("Mehdi",23);
                P1.Afficher();
                P2.Afficher();
}}
```

Liste d'initialiseurs

- Une liste d'initialiseurs permet l'appel d'un constructeur d'une classe par un autre constructeur de la même classe.
- Une liste d'initialiseurs commence par le symbole *:* suivi de *this* et d'une liste d'arguments définie par la signature du constructeur appelé.

Exemple :

```
class Text
{ int a;
  int b;
  string str;
  public Text(): this(0, 0, null) {}
  public Text(int x, int y): this(x, y, null) {}
  public Text(int x, int y, string s)
  { a=x; b=y; str=s; }
}
```

Destructeur

- La notion de destructeur existe en C# bien qu'elle ne soit pas aussi utile qu'en C++.
- Un destructeur possède le même nom que celui de la classe, précédé par ~.
- Un destructeur n'accepte aucun paramètre.
- Un destructeur n'accepte aucun modificateur d'accès (*public private*, …).
- Un destructeur en C# ne peut pas être explicitement appelé (pas d'instruction *delete*).
- Les destructeurs en C# ne sont pas déterministes comme en C++ :
  - En C++ le destructeur est automatiquement appelé quand l'objet quitte sa zone de validité.
  - En C# le destructeur devient "appelable" pour un objet lorsque il n'existe plus de référence à cet objet dans le programme. Toutefois on ne peut pas connaître exactement le moment de l'appel effectif du destructeur. Cet appel est en effet géré par le Garbage Collector. Il est effectué lorsqu'un besoin en mémoire se manifeste.

Exemple :

```
class Pers
{
   string Nom;
   int Age;
   public Pers(string N)
   {// code du constructeur}

   ~Pers()
    {// code du destructeur}
    ...
}
```


© Karim Kalti

La méthode Dispose

- .NET met à la disposition des programmeurs une méthode pouvant être appelée automatiquement lorsque la fin de la zone de validité d'un objet est atteinte. Cette méthode est appelée *Dispose*.
- Il est possible de mettre dans *Dispose* le code destiné à être placé dans le destructeur (libération des ressources).
- Pour qu'une classe garantisse au runtime que la méthode *Dispose* qu'elle implémente est celle devant être appelée à la fin de la zone de validité de l'objet (méthode du .NET), cette classe doit implémenter l'interface *IDisposable* du .NET.

Le prototype de *Dispose* est le suivant :
```
void Dispose() ;
```

Une classe doit implémenter IDisposable comme ceci :
```
class Pers : IDisposable
{
 ... ... ...
 public void Dispose()
 { // mettre ici le code de libération des ressources
 }
 ... ... ...
}
```
- Il est à noter que la zone de validité dont il est question ici n'est pas automatiquement déterminée par le système (en se basant sur les limites du bloc de déclaration comme c'est le cas en C++). Elle doit plutôt être définie explicitement par le programmeur à l'aide de *using* comme suit :

```
Pers p = new Pers() ;
using (p) // Definition de la zone de validité
{
  ... ...
} // fin de la zone de validité pour p
```

- Il est également possible de créer des zones de validité pour plusieurs objets :
```
Pers p = new Pers() ;
Animal a = new animal();
using (p) using (a)
{ …
}
```
Les méthodes *a.Dispose* et *p.Dispose* seront automatiquement invoquées (dans le sens inverse des *using* : *a* avant *p*) lorsque l'exécution atteint l'accolade fermante.

- *Dispose* peut être :
    - explicitement appelée : `NomObjet.Dispose( );`
    - implicitement appelée : en utilisant les zones de validité.

La méthode Finalize

- La méthode *Finalize* joue en .NET le même rôle joué par les destructeurs en C#.
- Cette méthode est automatiquement exécutée lorsqu'un objet est pris en charge par le GC.
- *Finalize* est une méthode de la classe *Object*, classe de base du .NET. Tout objet possède donc par défaut une telle méthode.
- Comme pour les destructeurs, on met dans *Finalize* le code de libération des ressources.
- *Finalize* ne peut pas être explicitement appelée.
- Le moment d'appel de *Finalize* n'est pas déterministe. Cet appel étant décidé et fait par le GC lorsqu'un besoin en mémoire se manifeste.
- En .NET *Finalize* assure un mécanisme de destruction même pour les langages qui ne supportent pas en natif un tel mécanisme (absence de la notion de destructeur au sens du C++ ou du C#).

© Karim Kalti

Les tableaux d'objets

- Les tableaux d'objets présentent des différences par rapport au C++.
- En C#, le constructeur de la classe doit être explicitement appelé pour chaque élément (objet) du tableau. En effet les tableaux d'objets tels qu'ils sont déclarés en C# sont des tableaux de références. Chaque référence doit donc être initialisée par une instance de la classe.
- Il est possible d'appeler n'importe quel constructeur (paramétré ou non) pour initialiser les tableaux d'objets.

Syntaxe :

```
NomClasse[] Tab; // Déclaration du tableau
Tab = new NomClasse[Taille]; // Allocation du tableau
for(int i = 0; i <Taille; i++) // instanciation des éléments du tableau.
   Tab[i] = new NomClasse(<arguments>);
```

Exemple :

```
Pers[] TP ;
TP = new Pers[10] ;
for(int i = 0; i <TP.Length ; i++)
   TP[i] = new Pers();
```

Membres statiques

- Les champs et les méthodes d'une classe peuvent être qualifiés de statiques.
- Les champs et les méthodes statiques existent sans nécessité d'instanciation de la classe.
- Le mot-clé *static* est utilisé pour qualifier un champ ou une méthode de statique.

Exemple :

```
class X
{
   public int  a ; // champ non statique
   public static int b ; // champ statique
   public void f(){…} // méthode non statique
   public static void g(){b++;} // méthode statique
}
```

Les champs statiques

Un champ statique (appelé également champ de classe) :

- est accessible indépendamment de tout objet de la classe. Il existe même si aucun objet de la classe n'a encore été créé.
- est partagé par tous les objets de la classe.
- est accessible à partir du nom de la classe : **`NomClasse.Champ;`** (*Exemple : X.b*)
- n'est pas accessible à partir d'un objet : `Obj.ChampStatique // erreur`
- peut être initialisé et manipulé par une méthode de la classe comme n'importe quel champ.

Les méthodes statiques

Une méthode statique (appelée également méthode de classe) :

- Peut être appelée en spécifiant directement le nom de la classe suivi du nom de la méthode. **`NomClasse.Methode(arguments)`** (`Exemple : X.g() ;`).
- Contrairement au C++, elle ne peut pas être appelée à partir d'un objet : `Obj.MethodeStatique // erreur`
- Ne peut accéder qu'aux champs statiques de sa classe.
- Ne peut appeler que les méthodes statiques (de sa classe ou d'une autre classe).
- Une méthode non statique (appelée également méthode d'instance) peut accéder à la fois aux champs statiques et non statiques. Elle peut appeler également les méthodes statiques et non statiques.

Constructeur statique

- Un constructeur statique est un constructeur qui ne peut initialiser que les membres statiques d'une classe.
- Un constructeur statique ne possède jamais d'arguments.
- Un constructeur statique n'accepte pas les modificateurs d'accès (public, private, …).

© Karim Kalti

Syntaxe :

```
class X
{ static X()
  { … }
  … … …
}
```

- Un constructeur statique ne peut pas être appelé explicitement.
- Un constructeur statique est appelé implicitement suite à l'instanciation d'un objet de la classe (il précède l'appel du constructeur d'instance) ou lorsqu'un membre statique de la classe est appelé.

Exemple :

```
using System;
class A
{ public static int i;
  public int j;

  static A()
  { Console.WriteLine("Constructeur statique");
    i=5;
  }

  public A(int v)
  { Console.WriteLine("Constructeur d'instance");
    j=v;
  }

  public A(int v1, int v2)
  { Console.WriteLine("Constructeur d'instance");
    i=v1; j=v2;
  }

  public void Afficher()
  { Console.WriteLine("i vaut :"+i);
    Console.WriteLine("j vaut :"+j);
  }
}
class Prog
{  static void Main()
   { A a1 = new A(3);
     a1.Afficher();  // i = 5 et j=3
     A a2 = new A(4,9);
     a2.Afficher();  // i = 4 et j=9
   }
}
```

Héritage de composition et imbrication de classes

- Une classe peut contenir comme membre un objet d'une autre classe. On parle dans ce cas d'héritage de composition.
- Une classe peut être déclarée dans une autre classe. Elle ne peut alors être utilisée qu'à l'intérieur de cette classe. On parle dans ce cas de classes imbriquées.

Exemple :

```
class A
{
  public class B
  { int chpr;
    public int chpu;
    public B(int arg){chpre =arg; chpu = 10*arg;}
    public int getchpr(){return chpr;}
  }
  B prB;
  public B puB;
  public A(int arg)
  { prB = new B(2*arg);
    puB = new B(10*arg);
  }
  public int getchprB()
    {return prb.getchpr();}
}
```


© Karim Kalti

- Un objet de la classe *B* ne peut pas être créé en dehors de la classe *A*. *B* est un objet local à *A*.
- La partie publique de *B* peut être accédée par le membre public de *A*.

```
A a = new A(5) ;
int n = a.getchprB(); // Ok car getchprB est publique.
n= a.puB.chpu; // Ok car puB et chpu sont publiques
```

- Les objets privés de B restent privés pour A :

`a.pub.chpr` // n'est pas accessible.
`a.prb.chpu` // erreur car prb est privé

Héritage

- Globalement similaire au C++.
- Pas d'héritage multiple.
- Syntaxe : **class ClasseDérivée : ClasseDeBase**

```
{… … … …
 … … … …
}
```

***Exemple :** Héritage*

```
class Pers
{  protected string Nom ;
   protected int Age;
   public Pers(string N, int A){ Nom = N; Age = A;}
   public void Afficher( ) { Console.WriteLine(Nom+ "(" + Age + ")");}
}
class Tunisien : Pers
{       string ville;
        public Tunisien(String N, int A, string V) : base (N,A)
        { Ville = V; }
        public new void Afficher()
        { Console.WriteLine(Nom+" Agé de " + Age + " ans (" + Ville+")"); }
}
```

Appel du constructeur de la classe de base

- L'appel du constructeur de la classe de base se fait à l'aide du mot-clé *base*. Lorsqu'il est utilisé dans une classe dérivée, ce mot-clé permet de référencer la classe de base (classe mère).
- L'appel du constructeur de la classe de base peut être omis. Dans ce cas c'est le constructeur par défaut qui est appelé. Si la classe de base possède un constructeur avec paramètres et si le constructeur par défaut n'est pas explicitement défini alors une erreur sera signalée par le compilateur.

Redéfinition de méthode

- Il est possible de redéfinir une méthode d'une classe de base dans une classe dérivée. Dans ce cas, la redéfinition est généralement précédée du mot-clé *new*.
- le *new* est optionnel, il sert généralement à améliorer la lisibilité du code en indiquant qu'il s'agit bien de la redéfinition d'une méthode.
- Tout appel d'une méthode redéfinie à partir d'une instance de la classe dérivée va engendrer l'utilisation de la version définie dans la classe dérivée.

Exemple 1 :

```
Tunisien T = new Tunisien ("Ali",20,"Monastir");
T.Afficher();
```

T est un objet de la classe *Tunisien*. L'instruction *T.Afficher( )* va engendrer l'appel de la méthode *Afficher* de la classe *Tunisien*. Cette méthode cache la méthode *Afficher* de la classe *Pers*.

© Karim Kalti

Exemple 2 :

Considérons le code suivant :

```
class Prog
{       static void Main()
        {
                Pers p1, p3;
                Tunisien p2;
                p1=new Pers("Ali",20);
                p1.Afficher();
                p2 = new Tunisien ("Salah",22,"Monastir");
                p2.Afficher();
                p3 = new Tunisien ("Salem",25,"Sousse");
                p3.Afficher();
        }
}
```

- Pour *P1* : c'est la méthode *Afficher* de la classe *Pers* qui est appelée.
- Pour *P2* : c'est la méthode *Afficher* de la classe *Tunisien* qui est appelée. *Afficher* de *Tunisien* cache dans ce cas *Afficher* de *Pers*. Si *Afficher* de *Tunisien* n'était pas définie, ça serait *Afficher* de *Pers* qui aurait été appelée.
- *P3 = new Tunisien( )* est possible car tout objet *Tunisien* est de type *Pers*.
- *P3.Afficher( )* exécute la méthode *Afficher* de *Pers* car *P3* a été déclarée en tant que tel. Le compilateur se base dans ce cas sur la définition stricte.

Les fonctions virtuelles

Pour tenir compte du véritable type de *P3* au moment de l'exécution de *P3.Afficher( )*, il faut qualifier *Afficher* de *virtual* dans la classe de base et de *override* dans la classe dérivée.

```
class Pers
{       protected string Nom ;
        protected int Age;
        public Pers(string N, int A){ Nom = N; Age = A;}
        public virtual void Afficher( ) { Console.WriteLine(Nom+ "(" + Age + ")");
}

class Tunisien : Pers
{       string Ville;
        public Tunisien(String N, int A, string V) : base (N,A)
        {Ville = V;}
        public override void Afficher()
        {Console.WriteLine(Nom+" Agé de " + Age + " ans (" + Ville+")");}
}

Pers p3;
p3 = new Tunisien ("Salem",25,"Sousse");
p3.Afficher(); // Salem agé de 25 ans (Sousse)
```

Appel des méthodes cachées par la redéfinition

Il est possible d'appeler une méthode d'une classe mère dans une classe fille même si cette méthode a été redéfinie dans la classe fille. Pour cela il suffit de faire précéder le nom de cette méthode par une référence à la classe mère. Cette référence est obtenue à l'aide du mot-clé *base*.

Exemple :

La méthode *Afficher* de *Tunisien* cache celle de *Pers*. Mais rien n'empêche d'appeler cette dernière dans la classe *Tunisien*. Il suffit de faire précéder le nom de cette méthode par *base*.

```
Exemple : base.Afficher();
```

Redéfinition de champs

Il est possible de définir dans une classe dérivée un champ qui porte le même nom qu'un champ de la classe de base. Pour distinguer entre les deux, il est possible d'écrire :

- *This.NomChamp* : pour désigner le nom du champ de la classe courante (classe dérivée).
- *base.NomChamp* : pour désigner le nom du champ de la classe de base.

Remarque :

Il n'est pas possible de chaîner l'utilisation de "base" pour accéder aux membres d'une classe grand-mère.

```
Exemple : base.base.Affiche() // erreur
```


© Karim Kalti

Identification du véritable objet instancié

Il est possible de connaître le véritable type d'un objet instancié et ce à l'aide du mot-clé "*is*".

Exemple :

```
class Pers
{   ..........
   public void Mariage(Pers Conjoint)
   { if(Conjoint is Tunisien)
        Console.WriteLine("Mariage avec compatriote");
     else
        Console.WriteLine("Mariage avec étranger");
   }
}

class Tunisien : Pers
{       string ville;
        public Tunisien(String N, int A, string V) : base (N,A)
        {Ville = V;}
        public override void Afficher()
        {Console.WriteLine(Nom+" Agé de " + Age + " ans (" + Ville+")");}
}

class Prog
{  static void Main()
    {   Tunisien T = new Tunisien("Ammar",50,"Nabeul");
        Pers P1 = new Tunisien ("Salma",45,"Kairouan");
        Pers P2 = new Pers("Carla",45);
        T.Mariage(P1); // Mariage avec compatriote
        T.Mariage(P2); // Mariage avec étranger
    }
}
```

Opérations applicables aux objets

La copie d'objets

Une affectation directe entre deux objets effectue en réalité une copie de références et non une copie d'instances.

Exemple 1 :

```
Pers P1,P2;
P1 = new Pers("Ali",20);
P2= P1;
```

Dans l'exemple 1, c'est une copie de référence qui a été effectuée et non une copie d'objet. *P1* et *P2* font référence donc au même objet.

Pour effectuer une véritable copie d'objets il faut que la classe en question implémente la méthode *Clone* de l'interface *IClonable*. La méthode *Clone* possède le prototype suivant : `public object Clone()`

Exemple 2 :

```
   class Pers : IClonable
    { ...
      public object Clone()
         {return new Pers(Nom, Age);}
     }}
    ...
    p2=(Pers)p1.Clone();
```

Pers implémente l'interface *IClonable*. Elle propose en effet une définition de la méthode *Clone* dans laquelle est mis le code effectuant la copie explicite.

La comparaison d'objets

- Une comparaison globale d'objets effectue en réalité une comparaison de références et non une comparaison d'instances.
- Pour effectuer une véritable comparaison, il faut comparer les deux objets champ par champ.

Exemple :

```
Pers P1,P2;
...
```

`P2==P1;` est une comparaison de références d'objets.

© Karim Kalti

Les protections

Protection des classes

Le seul qualificatif qui peut être utilisé avec les classes est *public*. Ainsi, si la classe est précédée de :

- *public* : alors il est possible de créer partout un objet de la classe.
- rien du tout : alors il est possible de créer un objet de la classe mais seulement dans le même assemblage.

Protection des membres

Les membres d'une classe peuvent utiliser les qualificatifs suivants :

- *public* : signifie que toutes les méthodes (du même assemblage ou non) peuvent accéder à ce champ ou appeler cette méthode.
- *private* : signifie que seules les méthodes de la classe ont accès à un tel champ ou à une telle méthode. Le fait d'omettre le qualificatif revient à spécifier *private*.
- *protected* : signifie que seules les méthodes de la classe et des classes dérivées (du même assemblage ou non) ont accès à un tel champ ou à une telle méthode.
- *internal* : signifie que seules les méthodes du même assemblage ont accès à un tel champ ou peuvent appeler une telle méthode.

Interdiction de dérivation

Il est possible d'interdire toute dérivation à partir d'une classe et ce à l'aide du qualificatif *sealed*.

Exemple :

```
sealed class Tunisien : Pers
{
...
}
```

Avec l'utilisation de *sealed* il devient impossible de dériver à partir de la classe *Tunisien*.

Les classes abstraites

- Une classe abstraite est une classe qui possède une ou plusieurs méthodes qui ne sont pas implémentées.
- Une classe abstraite est définie avec le mot-clé *abstract* comme suit :

Syntaxe :

```
abstarct class NomClasse
{ … }
```

- Une classe abstraite ne peut pas être instanciée.
- Une classe abstraite est utilisée généralement pour faire des dérivations. Les classes dérivées qui sont complètement implémentées peuvent être instanciées.

Exemple :

```
abstract class Forme
{  int Surface;
   void CalculerSurface(); // pas d'implémentation
   … … …
}
class Rectangle : Forme
{ int x1, int y1; int x2; int y2;
  void CalculerSurface(){… …}
  … … …
}
class Circle : Forme
{ int cx; int cy; int Rayon
  void CalculerSurface(){… …}
  … … …
}
```


© Karim Kalti

La classe *Forme* possède une méthode de calcul de la surface, mais cette méthode ne peut être implémentée que dans les classes dérivées (car il faut connaître la nature de la forme pour savoir comment calculer la surface).

Les interfaces

Définition

- Une interface est une classe qui ne comporte que des méthodes sans implémentation. On trouve seulement les prototypes de ces fonctions (il faut mettre les noms des paramètres formels).
- Les membres d'une interface ne peuvent être que des méthodes, des indexeurs, des propriétés ou des événements. Une interface ne peut pas comporter des champs.
- Une interface ne comporte pas des qualificatifs de protection (public, private,….).
- Le nom d'une interface commence par I.
- Une interface est définie avec le mot-clé interface comme suit :

```
interface NomInterface
 {  Methode1(<Liste d'arguments>);
      .........
    Methode2(<Liste d'arguments>);
 }
```

Utilité des interfaces

- Une interface définie une sorte de cahier de charge concernant un certain comportement (ensemble d'opérations).
- Chaque classe qui implémente une interface garantit à ses utilisateurs qu'elle est capable d'assurer les opérations définies par l'interface.

Interface et dérivation

- Il est possible de dériver des classes à partir d'une interface.
- Si la classe dérivée propose des définitions aux méthodes de l'interface alors le processus de dérivation s'appelle implémentation.
- L'implémentation d'une interface par une classe dérivée peut être complète ou partielle.
- Une classe dérivée d'une interface n'est instanciable que si elle propose une implémentation complète de l'interface.
- Les méthodes implémentées dans les classes dérivées acceptent les qualificatifs de protection.
- Une interface peut dériver d'une autre interface (dérivation sans aucune implémentation).

Exemple : implémentation d'une interface par une classe

```
interface IB
{ void f1(int a);
  void f2();
}
class X : IB
{ ...
  void f1(int a){...}
  void f2(){...}
}
```

Implémentation de plusieurs interfaces

- Une classe ne peut dériver que d'une seule autre classe. Elle peut par contre implémenter plusieurs interfaces en même temps.

Exemple :

Soient IB et IC deux interfaces

```
class X : IB, IC
{  ... // champs et méthodes de X
   void f1(int a){...} // corps de méthodes de IB
   void f2(){...} // corps de méthodes de IB
   void g1(){...} // corps de méthodes de IC.
}
```


© Karim Kalti

- Une classe dérivée peut avoir comme classes de base une classe et des interfaces. Dans ce cas, dans la définition de la classe dérivée, la classe de base doit être listée en premier lieu.

Exemple :

```
class ClassA: BaseClass, Iface1, Iface2
{
   // les membres de la classe
}
```

Ambiguïté d'implémentation de plusieurs interfaces

Deux interfaces peuvent comporter des méthodes portant le même nom. Une classe qui implémente ces deux interfaces en même temps doit alors résoudre le problème d'ambiguïté d'appel de telles méthodes.

Exemple :

```
interface IB
{
  void f();
  void g();
}
interface IC
{
  double f(int a);
  void h();
}
```

Soit *A* une classe qui implémente *IB* et *IC*. L'implémentation de la méthode *f* doit alors être associée à chaque fois à une référence de l'interface à laquelle elle se rapporte.

```
class A : IB, IC
{
  void IB.f(){...}
  double IC.f(int a){...}
}
```

Pour appeler ces fonctions il faut écrire comme suit :

```
A a = new A();
IB rib = a;
rib.f(); // Exécution de f de IB appliquée à a
IC ric= a;
double d=ric.f(5); //Exécution de f de IC appliquée à a.
```


© Karim Kalti

Les exceptions

Les exceptions sont des erreurs qui ont lieu au moment de l'exécution des programmes (pas au moment de la compilation) et qui ont pour cause une situation inattendue (exemple : tentative de lecture d'un fichier qui n'existe plus ou qui est déplacé).

Ancienne technique de gestion des erreurs d'exécution

Avec les langages qui ne supportent pas la gestion des exceptions (exemple C), le traitement des erreurs se fait généralement en faisant des tests sur les valeurs de retour des fonctions. Ces dernières retournent le plus souvent des codes indiquant le type de l'erreur. Ces codes ne comportent pas des informations supplémentaires sur les causes et les paramètres des erreurs. En plus cette technique devient fastidieuse dans le cas d'appels imbriqués de fonctions (le gestion des erreurs complique le code surtout au niveau des fonctions internes).

Avantage de la gestion d'exceptions

Les mécanismes de gestion des exceptions permettent d'éviter ces problèmes :
- Il devient ainsi possible de gérer à un seul niveau les exceptions, même celles engendrées par des fonctions internes (Possibilité de centralisation de la gestion en des points précis).
- Il est également possible d'avoir plus d'informations sur les causes des erreurs. Les erreurs ne sont plus décrites par des codes mais par des objets qui dérivent tous d'une classe de base appelée *Exception* et qui sont spécifiques chacun à un type d'erreur bien précis. Chaque objet comporte des informations qui décrivent l'erreur.

Familles d'exceptions

Il existe deux grandes familles d'exceptions :
- Les exceptions générées par le système (runtime exception) comme les divisions par zéro, l'accès à un tableau en dehors de ses bornes, l'accès à un fichier absent …
- Les exceptions générées par les applications. Ces exceptions sont généralement de type sémantique et dépendent du contexte de l'application. Comme exemple de ces exceptions, il est possible de citer la saisie d'une valeur trop élevée pour l'age (250). C'est au programmeur de définir dans ce cas ce type d'exceptions.

Gestion des exceptions

La programmation orientée objet offre une solution structurée de gestion d'erreurs sous la forme de bloc *try* et *catch*. L'idée est de séparer physiquement les instructions essentielles du programme qui gèrent son déroulement normal des instructions de gestion d'erreurs.
- Le bloc *try* contient les instructions qui sont susceptibles d'engendrer une erreur.
- Le bloc *catch* contient le code de gestion d'erreurs.

Syntaxe :

```
try
{
   // Bloc d'instructions du programme
}
catch(ClasseException e)
{
   // Bloc de gestion des erreurs
}
```

- Les accolades dans les blocs de *try* et de *catch* sont obligatoires même si ces derniers comportent une seule instruction.
- Le type de *ClasseException* doit être la classe *System.Exception* ou une classe qui dérive de cette dernière.
- La portée de l'identificateur *e* est limitée au bloc *catch*. Il est généralement utilisé pour récupérer les informations sur l'erreur à partir des attributs de *ClasseException.* S'il n'est pas fait usage de e dans le bloc catch alors cet identificateur peut être omis.

```
catch(ClasseException)
{
   // Bloc de gestion des erreurs
}
```


© Karim Kalti

Déroulement de l'exécution

Le programme entre dans le bloc de *try* comme si cette dernière instruction n'existait pas. Il commence alors l'exécution séquentielle des instructions de ce bloc. Dès qu'une erreur est détectée par le système ou par une méthode (qui lève une exception) un objet de la classe *ClasseException* est créé et le programme quitte le bloc *try* et rentre tout aussi automatiquement dans le bloc *catch*.

Exemple :

```
using System;
class Prog
{
 static void Main()
 { int a = 5,b = 0,c;
   try
   { Console.WriteLine("Avant division");
     c=a/b;
     Console.WriteLine("Après division c vaut "+c);
   }
   catch(Exception)
   {Console.WriteLine("Erreur sur opération arithmétique");}
 }
}
```

Blocs catch multiples

- Le bloc de *try* peut comporter plusieurs instructions. Chaque instruction peut lever une ou plusieurs exceptions. Il est possible d'associer à un tel bloc plusieurs blocs *catch* qui interceptent chacun un type d'exception particulier.
- En cas d'erreur le contrôle passe au premier bloc *catch*. Si cette erreur correspond à l'argument de ce premier bloc *catch* alors l'erreur est traitée par ce bloc. Dans le cas contraire le *catch* suivant est inspecté et ainsi de suite.
- Dès qu'une erreur est traitée par un *catch* les *catchs* suivants ne sont plus envisagés.

Exemple :

```
using System;
class Prog
{
  static void Main()
  { int a = 5,b = 0,c;
    int[] T = new int[3];
    try
    { Console.WriteLine("Donner un nombre");
       c= Int32.Parse(Console.ReadLine());
       Console.WriteLine("Avant division et accès au tableau");
       T[c]=a/b;
       Console.WriteLine("Après division T["+c+"] vaut "+T[c]);
    }
    catch(ArithmeticException)
    {Console.WriteLine("Erreur sur opération arithmétique");}

    catch(IndexOutOfRangeException)
    {Console.WriteLine("Accès au tableau en dehors de ses bornes");}

    catch(Exception)
    {Console.WriteLine("Erreur");}

    Console.WriteLine("Après le traitement des erreurs");
  }
}
```

Commentaires

- Trois blocs *catch* sont associés au *try*. Le premier bloc gère les erreurs arithmétiques. La classe *ArithmeticException* est la classe de base de ce type d'erreurs. Voici la liste des classes exceptions qui en dérivent :

```
System.ArithmeticException
 System.DivideByZeroException (Division par zero)
 System.NotFiniteNumberException (Valeur réelle infinie ou valeur non numérique pour un réel)
 System.OverflowException (Dépassement de capacité)
```

- Le deuxième bloc gère les erreurs d'accès à un tableau en dehors de ses bornes.

© Karim Kalti

- Le troisième bloc gère les autres exceptions pouvant éventuellement avoir lieu. En effet *Exception* est la classe de base de toutes les exceptions. Donc toute exception ayant échappé aux deux premiers blocs sera obligatoirement interceptée par ce troisième bloc.

Remarque :

L'ordre des catchs est important surtout si leurs arguments sont des objets qui dérivent les uns des autres.
Par exemple si les blocs catchs de l'exemple précédents étaient organisés comme ceci :

```
catch(Exception)
  {Console.WriteLine("Erreur");}
catch(ArithmeticException)  // Bloc jamais exécuté
  {Console.WriteLine("Erreur sur opération arithmétique");}
catch(IndexOutOfRangeException) // Bloc jamais exécuté
  {Console.WriteLine("Accès au tableau en dehors de ses bornes");}
```

Alors les deux derniers blocs ne pourraient jamais être exécutés car toute erreur est de type *Exception* (toutes les exceptions de type système ou application dérivent de *System.Exception*). Toute erreur serait de ce fait toujours traitée par le premier bloc.

Le bloc finally

- En plus des bloc *try* et *catch*, il est possible d'utiliser un troisième bloc optionnel défini par *finally*. Les instructions de ce bloc sont toujours exécutées quel que soit le déroulement de l'exécution dans le bloc *try* (erreur ou pas d'erreur).
- Le bloc *finally* est généralement utilisé pour assurer la libération des ressources (fermeture de fichiers, …) et ce en cas de levée d'une exception.

Remarque :

- Un seul bloc *finally* peut être associé à un bloc *try*.
- Les accolades sont obligatoires même si le bloc comporte une seule instruction.

Mécanisme complet de gestion d'exceptions

La syntaxe complète d'un mécanisme de gestion d'exception est :

```
try
{
   // Bloc d'instructions du programme
}
catch(ClasseException e)
{
   // Bloc de gestion de l'erreur
}
finally
{
   // Bloc toujours exécuté
}
```

Exemple :

```
using System;
class Prog
{ static void Main()
  {
    try
    { Console.WriteLine("Début du try du Main");
      f();
      Console.WriteLine("Fin du try du Main");
    }
    catch(Exception)
    {
      Console.WriteLine("Erreur");
    }
  }
```


© Karim Kalti

```
  static void f()
  {
    int a = 5,b,c;
    int[] T = new int[3];
    ... // code de saisie de b et c
    try
    {Console.WriteLine("Début du try de f");
     T[b] = a/c;
     Console.WriteLine("Fin du try de f");
    }
    finally
    {Console.WriteLine("finally de f);}
  }
}
```

Exemples de résultats d'exécution :

Si b= 1 et c=1	Si b =1 et c=0	Si b =5 et c=1
Début du try du Main Début du try de f Fin du try de f Finally de f Fin du try de Main	Début du try du Main Début du try de f Finally de f Erreur	Début du try du Main Début du try de f Finally de f Erreur

Remarque : Propagation du mécanisme d'interception

L'exemple précédent montre qu'il est possible de ne pas intercepter une erreur à la source (absence de *catch* dans le *try* de *f*). En effet suite à la génération d'une exception dans un bloc *try*, le runtime commence à chercher un bloc *catch* capable de la traiter (en se basant sur le type de l'exception). S'il ne trouve pas un tel bloc dans la fonction appelée, la recherche se propage vers la fonction appelante. Ce processus de propagation continu :

- Jusqu'à ce que un *catch* approprié soit trouvé, auquel cas le runtime considère que l'exception à été interceptée et il reprend l'exécution du programme à partir du corps du bloc catch.
- Jusqu'à ce que la fonction *Main* soit atteinte sans trouver un *catch* approprié. Dans ce cas le runtime met fin à l'exécution du programme.

Définition d'exceptions personnalisées

- En plus des exceptions "système" qui sont prédéfinies, il est possible au programmeur de définir ses propres exceptions. Ces dernières sont généralement d'ordre sémantique et dépendent de ce fait du contexte de l'application développée. C'est pourquoi elles sont dites exceptions de type "application".

- La définition d'une exception personnalisée passe par la définition d'une classe qui dérive obligatoirement de la classe *Exception.* Il suffit généralement de faire une simple dérivation sans ajouter de membres supplémentaires. Il est également possible d'inclure des attributs supplémentaires qui donneraient des informations utiles quant à l'erreur.

Levée d'exceptions

Contrairement aux exceptions prédéfinies où c'est le runtime qui lève automatiquement l'exception, pour les exceptions personnalisées cette tâche est à la charge du programmeur. Elle est assurée par l'instruction *throw* qui prend en argument un objet de la classe représentant l'exception.

Exemple 1 :

```
if(minute<1 || minute >=60)
{
   string Erreur = minute + "n'est pas une minute valide";
   throw new InvalidTimeException(Erreur);
   // Partie jamais atteinte
}
```

Exemple 2 :

```
using System;
class MyException : Exception
{
  public MyException(string msg) : base(msg)
  {}
}
```


© Karim Kalti

```
class Prog
{
  static void f(int i)
  {
    if(i>10)
      throw new MyException(i+" est une valeur trop grande");

    Console.WriteLine("Fin de la méthode f");
  }
  static void Main()
  {
    int i=20;
    try
    { f(i); }

    catch(MyException e)
    { Console.WriteLine(e.Message); }
  }
}
```

Principaux membres de la classe *System.Exceptions*

Constructeurs		
`Exception()`		Constructeur par défaut
`Exception(string msg)`		Constructeur qui initialise l'attribut Message de la classe Exception.
Propriétés		
Nom	**Type**	**Signification**
`HelpLink`	string	Nom de la page Web d'aide associée à l'exception.
`Message`	string	Message d'explication
`Source`	string	Nom de l'objet ou de l'application ayant généré l'erreur.

Classes d'exceptions usuelles en C# (ref MSDN)

`System.ArithmeticException`	Classe de base pour les exceptions qui se produisent pendant des opérations arithmétiques, telles que `System.DivideByZeroException` et `System.OverflowException`.
`System.ArrayTypeMismatchException`	Levée lorsqu'un stockage dans un tableau échoue car le type réel de l'élément stocké est incompatible avec le type réel du tableau.
`System.DivideByZeroException`	Levée à l'occasion d'une tentative de division d'une valeur intégrale par zéro.
`System.IndexOutOfRangeException`	Levée lors d'une tentative d'indexation d'un tableau via un index qui est inférieur à zéro ou en dehors des limites du tableau.
`System.InvalidCastException`	Levée lorsqu'une conversion explicite d'une interface ou d'un type de base vers des types dérivés échoue au moment de l'exécution.
`System.NullReferenceException`	Levée lorsqu'une référence `null` est utilisée d'une manière qui rend obligatoire l'objet référencé.
`System.OutOfMemoryException`	Levée par l'échec d'une tentative d'allocation de mémoire (via `new`).
`System.StackOverflowException`	Levée lorsque la pile d'exécution est épuisée par un trop grand nombre d'appels de méthode en attente ; c'est généralement le signe d'une récurrence très profonde ou non liée.
`System.TypeInitializationException`	Levée lorsqu'un constructeur statique lève une exception, alors qu'il n'existe aucune clause catch pouvant l'intercepter.

© Karim Kalti

Les IHM en C#

Introduction

Les Interfaces Homme-Machine (IHM) permettent de développer des programmes informatiques dans lesquelles l'interaction entre l'utilisateur et le système se fait dans un cadre convivial basé essentiellement sur des composants visuels. Le système d'exploitation Windows donne une bonne illustration de ces interfaces et de leurs composants (*Exemples* : Fenêtres, Boites de dialogue, menus, boutons, boutons radio, cases à cocher, etc.). En utilisant le .NetFrameWork, il est possible de développer des applications disposant de telles interfaces.

Classes des composants visuels en .NET

- Avec le .NetFramework, chaque composant visuel, appelé également contrôle, est représenté par une classe. Ainsi, et à titre d'exemple, la classe *Button* représente les boutons, la classe *Form* représente les fenêtres, etc.
- Toutes les classes représentant les composants visuels font partie de l'espace de noms *System.Windows.Forms*. Elles dérivent toutes de la classe *Control.*
- Chaque composant visuel possède :
  - Un ensemble de caractéristiques (Exemple : emplacement, taille, couleur, …). Ces caractéristiques sont codées comme des propriétés de la classe qui représente le composant. Certaines de ces propriétés sont en lecture seule, d'autres sont en lecture et en écriture.
  - Un ensemble d'actions qui peuvent lui être appliquées. Ces actions sont codées sous forme de méthodes de la classe qui représente le contrôle. (Exemple d'actions : la méthode *Close* de la classe *Form* permet de fermer une fenêtre).
  - Un ensemble d'événements qu'il peut traiter. Ces événements sont signalés à l'aide de messages envoyés par Windows. Les fonctions de traitement de ces événements sont codées sous forme de méthodes de la classe qui représente le composant. Ces méthodes doivent être ajoutées aux délégués responsables de la gestion des événements.

La classe Form

Cette classe représente une fenêtre.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ContainerControl
            System.Windows.Forms.Form

Constructeur : `public Form();`

© Karim Kalti

Quelques propriétés :

Propriété	Type	Description
AllowDrop	bool	Si vrai, la fenêtre pourra jouer le rôle de destination dans une opération Drag&Drop.
AutoScroll	bool	Si la valeur est true alors des barres de défilement (scrollbar) sont automatiquement affichées lorsque la fenêtre devient trop réduite pour que tous les composants soient visibles. Si la valeur est false alors les barres de défilement ne sont jamais affichées.
BackColor	enum Color	Couleur d'arrière plan de la fenêtre.
BackGroundImage		Image de fond de la fenêtre. Il est possible de spécifier une image au format bmp, gif, jpg ou ico. Cette image sera greffée dans l'exécutable du programme. L'image ainsi spécifiée est répétée autant de fois que nécessaire pour remplir toute la fenêtre.
Cursor	enum Cursors	Forme que prend le curseur de la souris lorsqu'il survole la fenêtre.
ForeColor	enum Color	Couleur d'affichage du texte.
Location	Objet Point	Position du coin supérieur gauche de la fenêtre.
Menu	Objet MainMenu	Nom du menu associé à la fenêtre
Size	Objet Size	Taille de la fenêtre. Size représente deux champs: Width et Height. Ces champs sont exprimés en nombre de pixels.
Text	String	Représente le texte qui sera affiché dans la barre de titre de la fenêtre.
Name	String	Nom interne de la fenêtre.

Quelques méthodes :

Méthode	Description
`void BringToFront()`	Amène la fenêtre à l'avant-plan.
`Void Close()`	Ferme la fenêtre
`Void Show( )`	Fait apparaître la fenêtre.

Quelques événements :

Evénement	Description
BackColorChanged (hérité de **Control**)	Se produit lorsque la valeur de la propriété BackColor change.
Click (hérité de **Control**)	Se produit suite à un clic sur le contrôle (la fenêtre dans ce cas).
Closed	Se produit lorsque le formulaire est fermé.
Closing	Se produit pendant la fermeture du formulaire.
GotFocus (hérité de **Control**)	Se produit lorsque le contrôle reçoit le focus.
LostFocus (hérité de **Control**)	Se produit lorsque le contrôle perd le focus.
KeyDown (hérité de **Control**)	Se produit lorsqu'une touche est enfoncée alors que le contrôle a le focus.
KeyPress (hérité de **Control**)	Se produit lorsqu'une touche est enfoncée alors que le contrôle a le focus.
KeyUp (hérité de **Control**)	Se produit lorsqu'une touche est relâchée alors que le contrôle a le focus.
Load	Se produit avant le premier affichage d'un formulaire.

Programme 1

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
public class Exemple
    {
        static void Main( ) {
        Form fiche = new Form( );
        fiche.Text="Exemple de Form";
        fiche.BackColor=System.Drawing.Color.RosyBrown;
        fiche.Show( );
        }
    }
}
```

Résultat : la fenêtre disparaît aussi vite qu'elle est apparue.

© Karim Kalti

Explication des étapes de l'exécution :

Ligne d'instruction du programme	Que fait le CLR
{	Initialisation de l'exécution de la méthode main
`Form MaFenêtre = new Form( );`	Instanciation d'une fenêtre nommée MaFenêtre
`MaFenêtre.Show();`	Affichage de MaFenêtre.
}	Terminaison du processus.

La méthode Show de la classe Form affiche la fenêtre. Cet affichage va persister durant l'exécution du Main. Dès que l'accolade fermante du Main est atteinte, le programme se termine et la fenêtre disparaît en même temps. La fugacité de l'affichage de l'exemple précédent est due en fait à l'absence d'instructions après la méthode Show. Pour remédier à cela, il est possible d'ajouter une instruction qui bloque l'exécution (*Exemple :* `Console.Read();`)

```
static void Main()
{ char a;
  Form MaFenêtre = new Form();
  MaFenêtre.Show();
  a=Console.Read();
}
```

Le blocage de l'exécution avec la méthode Read, même s'il résout le problème de la fugacité de l'affichage reste inapproprié pour le développement d'application avec IHM. En effet, dans ce cas, et tout juste après l'instruction `MaFenêtre.Show()`, le programme passe directement à l'instruction `a=Console.Read()` et attend la saisie d'un caractère. Toute autre action peut bloquer son exécution et il devient dans ce cas difficile d'interagir avec la Fenêtre. Pour éviter ce problème, il vaut mieux utiliser une boucle à l'intérieur du Main dont le rôle sera d'intercepter les événements engendrés par le système ou par l'interaction de l'utilisateur avec l'application. Cette boucle permettra entre autres d'éviter d'atteindre la fin de l'application tant qu'un message d'arrêt n'a pas été intercepté. Elle assurera par la même occasion une persistance de l'affichage des interfaces graphiques. Le .NET framework implémente une telle boucle et ce dans la classe Application. Le lancement de cette boucle se fait à l'aide de la méthode Run.

Exemple :

```
static void Main()
{       Form MaFenêtre = new Form();
        MaFenêtre.Show();
        Application.Run();
}
```

La méthode Run exécute une boucle de messages d'application sur le thread en cours (processus associé à l'application en cours d'exécution). La structure de la boucle peut être vue comme suit :

tantque non ArrêtSysteme **faire**
  si événement **alors**
    Construire **Message ;**
    si Message ≠ ArrêtSysteme **alors**
      Reconnaître le composant auquel est destiné ce **Message**;
      Distribuer ce **Message**
    fsi
  fsi
ftant

Autres surcharges de Run

La méthode *Run* possède deux autres surcharges. La surcharge la plus utilisée est celle qui prend en argument un objet de la classe *Form* et qui l'affiche. Cet objet représente généralement la fenêtre principale de l'application.

Prototype : `public static void Run(Form);`

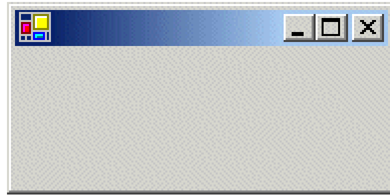
© Karim Kalti

Deuxième programme :

Programme 2

```
using System;
using System.Windows.Forms;
namespace Application
{
        class PremierWinProg
        {
                static void Main()
                {
                        Form MaFenêtre = new Form();
                        Application.Run(MaFenêtre);
                }
        }
}
```

Résultat de l'exécution du programme 2 :



Remarques :

- Une seule boucle de messages est généralement associée à une application. Elle est lancée avec la méthode Run.
- La surcharge la plus utilisée de Run est celle qui prend en argument la fenêtre principale de l'application. Si l'application contient d'autres fenêtres, alors leur affichage sera fait à travers leurs propres méthodes Show et non à travers `Application.Run` comme c'est le cas pour la fenêtre principale.

Utilisation d'un RAD pour la génération du code :

Les environnements de développement en C#.NET permettent de développer de grandes parties des applications à l'aide des assistants visuels. Ils génèrent également d'une manière automatique le code correspondant aux parties ainsi développées. Le programme suivant représente le code généré pour un projet minimal affichant seulement un formulaire vide.

Programme 3

```
using System;
using System.Windows.Forms;

namespace DefaultNamespace
{
 public class MainForm : System.Windows.Forms.Form
  {
    public MainForm()
     {
      InitializeComponent();
     }

 public static void Main()
  {Application.Run(new MainForm());}

 private void InitializeComponent()
  {// Form1
   this.ClientSize= new System.Drawing.Size(292, 266);
   this.Text = "MainForm";
   this.Name = "MainForm";
  }
 }
}
```


© Karim Kalti

Commentaires :

- La méthode InitialiazeComponent contient tout le code nécessaire à l'initialisation du formulaire et éventuellement ses composants. Cette méthode est générée par le RAD. Elle est par la suite appelée dans le constructeur de la fenêtre de base de l'application.

- Il est possible d'ajouter d'autres instructions d'initialisation dans la méthode InitailizeComponent ou directement dans le constructeur de la fenêtre.

La classe Button

Cette classe représente les boutons de commande.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ButtonBase
          System.Windows.Forms.Button

Constructeur : `public Button();`

Quelques propriétés :

Propriété	Type	Description
BackColor	enum Color	Couleur d'arrière plan du bouton.
Cursor	enum Cursors	Forme que prend le curseur de la souris lorsqu'il survole le bouton.
Enabled (hérité de Control)	bool	Obtient ou définit une valeur indiquant si un contrôle peut répondre à une interaction utilisateur.
Font (hérité de Control)	Font	Obtient ou définit la police du texte affiché par le contrôle.
FlatStyle	enum FlatStyle	Obtient ou définit le style à deux dimensions (flat) du contrôle bouton.
ForeColor	enum Color	Couleur d'affichage du texte.
Image	Image	Image d'avant plan.
Location	Point	Position du coin supérieur gauche du bouton.
Name	String	Nom interne du bouton.
TabIndex (hérité de Control)	int	Obtient ou définit l'ordre de tabulation du contrôle dans son conteneur.
TabStop (hérité de Control)	bool	Obtient ou définit une valeur indiquant si l'utilisateur peut octroyer le focus à ce contrôle à l'aide de la touche TAB.
Size	Size	Taille du bouton. Size représente deux champs: Width et Height. Ces champs sont exprimés en nombre de pixels.
Text	String	Représente le texte qui sera affiché sur le bouton.
Visible (hérité de Control)	bool	Obtient ou définit une valeur indiquant si le contrôle est affiché.

Quelques méthodes :

Méthode	Description
`void Hide()`	Masque le bouton à l'utilisateur.
`void scale(float,float)`	Dimensionne le contrôle et ses contrôles enfants.
`Void select()`	Surchargée. Active un contrôle.
`Void Show( )`	Affiche le bouton à l'utilisateur.

© Karim Kalti

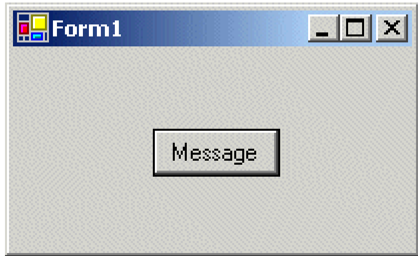
Quelques événements :

Evénement	Description
BackColorChanged (hérité de **Control**)	Se produit lorsque la valeur de la propriété BackColor change.
Click (hérité de **Control**)	Se produit suite à un clic sur le contrôle.
GotFocus (hérité de **Control**)	Se produit lorsque le contrôle reçoit le focus.
LostFocus (hérité de **Control**)	Se produit lorsque le contrôle perd le focus.
KeyDown (hérité de **Control**)	Se produit lorsqu'une touche est enfoncée alors que le contrôle a le focus.
KeyPress (hérité de **Control**)	Se produit lorsqu'une touche est enfoncée alors que le contrôle a le focus.
KeyUp (hérité de **Control**)	Se produit lorsqu'une touche est relâchée alors que le contrôle a le focus.

Programme 4

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace WindowsApplication3
{
public class Form1 : System.Windows.Forms.Form
{private System.Windows.Forms.Button button1;
 public Form1()
 {InitializeComponent();}
 #region Windows Form Designer generated code
 private void InitializeComponent() {
 this.button1 = new System.Windows.Forms.Button();
 this.SuspendLayout();
 // button1
 this.button1.Location = new System.Drawing.Point(70,40);
 this.button1.Name = "button1";
 this.button1.Size = new System.Drawing.Size(64, 24);
 this.button1.TabIndex = 0;
 this.button1.Text = "Message";
 // Form1
 this.ClientSize = new System.Drawing.Size(200, 100);
 this.Controls.Add(this.button1);
 this.Name = "Form1";
 this.Text = "Form1";
 this.ResumeLayout(false);
}
#endregion
static void Main()
{Application.Run(new Form1());}
}
}
```

<u>Résultat de l'exécution du programme 4 :</u>



© Karim Kalti

La classe Label

Cette classe représente une zone d'affichage de texte. Les contrôles **Label** sont généralement utilisés pour fournir un texte descriptif à un contrôle. Par exemple, il est possible d'utiliser **Label** pour ajouter un texte descriptif à un contrôle **TextBox** afin d'informer l'utilisateur du type des données attendues dans le contrôle. Le contrôle **Label** peut également être utilisé pour ajouter un texte descriptif à un **Form** afin de fournir des informations utiles à l'utilisateur. Par exemple, il est possible d'ajouter un **Label** au début d'un **Form** qui fournit des instructions à l'utilisateur sur la façon de faire entrer des donnés dans les contrôles du formulaire. Le contrôle **Label** peut également être utilisé pour afficher des informations sur l'état d'une application au moment de l'exécution.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.Label

Constructeur : `public Label();`

Quelques propriétés :

Propriété	Type	Description
BackColor	enum Color	Couleur d'arrière plan du Label.
BorderStyle	enum BorderStyle	Obtient ou définit le style de bordure du Label.
Cursor	enum Cursors	Forme que prend le curseur de la souris lorsqu'il survole le Label.
Enabled	bool	Obtient ou définit une valeur indiquant si le Label peut répondre à une interaction avec l'utilisateur.
Font	Font	Obtient ou définit la police du texte affiché par le Label.
FlatStyle	enum FlatStyle	Obtient ou définit le style à deux dimensions (flat) du contrôle Label.
ForeColor	enum Color	Couleur d'affichage du texte.
Image	Image	Image d'avant plan.
Location	Point	Position du coin supérieur gauche du contrôle Label
Name	string	Nom interne du Label.
TabIndex	int	Obtient ou définit l'ordre de tabulation du contrôle dans son conteneur.
TabStop	bool	Obtient ou définit une valeur indiquant si l'utilisateur peut octroyer le focus à ce contrôle à l'aide de la touche TAB.
Size	Size	Taille du contrôle Label. Size représente deux champs: Width et Height. Ces champs sont exprimés en nombre de pixels.
Text	string	Représente le texte qui sera affiché par le Label.
Visible	bool	Obtient ou définit une valeur indiquant si le contrôle Label est affiché.

Quelques méthodes :

Méthode	Description
`void Hide()`	Masque le label à l'utilisateur.
`void scale(float,float)`	Surchargée. Dimensionne le contrôle Label et ses contrôles enfants.
`Void select()`	Surchargée. Active un contrôle Label.
`Void Show( )`	Affiche le contrôle Label à l'utilisateur.

Quelques événements :

Un label peut traiter la majorité des événements associés à la classe contrôle. Toutefois, sur le plan pratique il est très rare de gérer ces événements.

© Karim Kalti

La classe TextBox

Le contrôle TextBox représente les zones d'édition. Il permet d'entrer du texte dans une application. Il a été enrichi de fonctionnalités absentes du contrôle zone de texte Windows standard, dont la modification multiligne et le masquage des caractères d'un mot de passe.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.TextBoxBase
          System.Windows.Forms.TextBox

Constructeur : `public TextBox();`

Quelques propriétés :

Propriété	Type	Description
AcceptsReturn	bool	Obtient ou définit une valeur indiquant si le fait d'appuyer sur la touche ENTRÉE dans un contrôle TextBox multiligne entraîne la création d'une nouvelle ligne de texte dans le contrôle ou bien l'activation du bouton par défaut sur le formulaire.
BackColor	enum Color	Couleur d'arrière plan du contrôle TextBox.
BorderStyle	enum BorderStyle	Obtient ou définit le type des bordures du contrôle zone de texte.
Cursor	enum Cursors	Forme que prend le curseur de la souris lorsqu'il survole le contrôle TextBox.
Enabled	bool	Obtient ou définit une valeur indiquant si un contrôle peut répondre à une interaction utilisateur.
Font	Objet Font	Obtient ou définit la police du texte affiché par le contrôle TextBox.
ForeColor	enum Color	Couleur d'affichage du texte.
Location	Point	Position du coin supérieur gauche du contrôle TextBox.
Name	string	Nom interne du contrôle TextBox.
MaxLength	int	Obtient ou définit le nombre maximal de caractères que l'utilisateur peut taper dans le contrôle zone de texte.
Multiline	bool	Obtient ou définit une valeur indiquant si ce contrôle est un contrôle zone de texte multiligne.
PasswordChar	char	Obtient ou définit le caractère servant à masquer les caractères d'un mot de passe dans un contrôle TextBox monoligne.
ReadOnly	bool	Obtient ou définit une valeur indiquant si le texte de la zone de texte est en lecture seule.
RightToLeft	bool	Obtient ou définit une valeur indiquant si les éléments du contrôle sont alignés pour prendre en charge les paramètres régionaux utilisant des polices de droite à gauche (comme l'arabe).
TabIndex	int	Obtient ou définit l'ordre de tabulation du contrôle dans son conteneur.
TabStop	bool	Obtient ou définit une valeur indiquant si l'utilisateur peut octroyer le focus à ce contrôle à l'aide de la touche TAB.
Size	Objet Size	Taille du contrôle TextBox. Size représente deux champs: Width et Height. Ces champs sont exprimés en nombre de pixels.
Text	string	Représente le texte qui sera affiché par le contrôle TextBox.
Visible	bool	Obtient ou définit une valeur indiquant si le contrôle est affiché.

© Karim Kalti

Quelques méthodes :

Méthode	Description
`void Clear();`	Efface tout le texte du contrôle zone de texte.
`void Copy();`	Copie la sélection active dans la zone de texte vers le Presse-papiers.
`void Cut();`	Déplace la sélection active entre la zone de texte et le Presse-papiers.
`void Paste();`	Remplace la sélection active de la zone de texte par le contenu du Presse-papiers.
`void Undo()`	Annule la dernière modification apportée dans la zone de texte.
`void Select(int start,int length)`	Sélectionne le texte qui commence depuis le caractère d'indice *start* et qui s'étend sur *length* caractère.
`void SelectAll()`	Sélectionne tout le texte du contrôle.
`void Hide()`	Masque la zone de texte à l'utilisateur.
`void Show( )`	Affiche le contrôle TextBox à l'utilisateur.

Exemple d'écriture du contenu d'un Label

```
Label Lb = new Label();
Lb.text = "Premier message";
```

Exemple de lecture du contenu d'une zone d'édition

```
TextBox TB = new TextBox();
............
String s = TB.text ;
```

Exemple montrant comment vider le contenu d'une zone d'édition

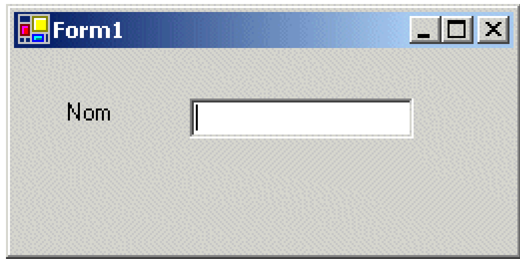
```
TB.text = ""; ou TB.Clear() ;
```

Programme 5 : Exemple d'utilisation des contrôles TextBox et Label

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

public class Form1 : System.Windows.Forms.Form
 {private System.Windows.Forms.Label label1;
  private System.Windows.Forms.TextBox textBox1;
  public Form1()
   {InitializeComponent();}
  private void InitializeComponent()
  {this.label1 = new System.Windows.Forms.Label();
   this.textBox1=new System.Windows.Forms.TextBox();
   this.SuspendLayout();
   // label1
   this.label1.Location = new System.Drawing.Point(24, 24);
   this.label1.Name = "label1";
   this.label1.Size = new System.Drawing.Size(72, 24);
   this.label1.TabIndex = 0;
   this.label1.Text = "Nom";
   // textBox1
   this.textBox1.Location =new System.Drawing.Point(88, 24);
   this.textBox1.Name = "textBox1";
   this.textBox1.Size = new System.Drawing.Size(112, 20);
   this.textBox1.TabIndex = 1;
   this.textBox1.Text = "";
   // Form1
   this.ClientSize = new System.Drawing.Size(250, 100);
   this.Controls.Add(this.textBox1);
   this.Controls.Add(this.label1);
   this.Name = "Form1";
   this.Text = "Form1";
   this.ResumeLayout(false);
}

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


© Karim Kalti

Résultat de l'exécution du programme 5 :



La classe CheckBox

- Cette classe représente un contrôle case à cocher. Ce contrôle permet de présenter à l'utilisateur la possibilité de sélectionner ou non un choix.
- Une case à cocher est généralement accompagnée d'un texte expliquant le choix à faire.
- Un contrôle case à cocher peut avoir trois états : non coché, coché, estompé.
- Un contrôle case à cocher peut être utilisé seul ou en groupe.
- Il est possible de regrouper thématiquement des cases à cocher dans un GroupBox. Ceci permet de faciliter leur manipulation (déplacement et positionnement) et permet à l'utilisateur de faire des choix (non exclusifs) concernant le thème du groupe.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ButtonBase
          System.Windows.Forms.CheckBox

Constructeur : `public CheckBox();`

Quelques propriétés :

Propriété	Type	Description
Appearance	enum Appearance	Indique l'apparence de la case à cocher. Les valeurs possibles de l'énumération *Appearance* sont : *Button* (la case prend la forme d'un petit bouton) et *Normal* (la case prend sa forme normale).
AutoCheck	bool	Indique si la case change automatiquement d'état suite à un clic sur la case. Si *AutoCheck* vaut *false*, c'est alors au programmeur de gérer l'état de la case en traitant l'événement *Click*.
Checked	bool	Indique l'état (coché ou non) d'une case à deux états. Pour les contrôles à trois états la propriété *Checked* retourne *true* pour les deux états *Checked* et *Intermediate*.
CheckState	enum CheckState	Etat d'une case à trois états. *CheckState* prend l'une des trois valeurs suivantes de l'énumération *CheckState* (*Checked*, *Unchecked*, *Indeterminate*). *Indeterminate* correspond à une case estompée (grisée).
ThreeState	bool	Indique si la case est à deux ou à trois états : *false* : correspond à une case à deux états. Elle peut être cochée (*Checked*) ou non cochée (*Unchecked*). Cet état est contrôlé par la propriété *Checked.* *true* : correspond à une case à trois états. (coché, non coché, estompé). Cet état est contrôlé par la propriété *CheckState*.
Text	string	Libellé de la case. Le caractère & sert à spécifier un accélérateur. ALT+Accélérateur permet de changer l'état de la case.
Enabled	bool	Obtient ou définit une valeur indiquant si le contrôle peut répondre à une interaction utilisateur.
Location	Point	Position du coin supérieur gauche du contrôle case à cocher.
Name	string	Nom interne du contrôle case à cocher.
TabIndex	int	Obtient ou définit l'ordre de tabulation du contrôle dans son conteneur.
TabStop	bool	Obtient ou définit une valeur indiquant si l'utilisateur peut octroyer le focus à ce contrôle à l'aide de la touche TAB.
Visible	bool	Obtient ou définit une valeur indiquant si le contrôle est affiché.

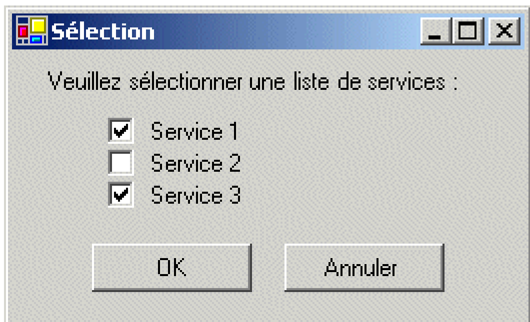
© Karim Kalti

Quelques méthodes :

Méthode	Description
`void Hide()`	Masque le contrôle à l'utilisateur.
`Void Show( )`	Affiche le contrôle à l'utilisateur.

Quelques événements :

Evénement	Description
Click (hérité de **Control**)	Se produit suite à un clic sur le contrôle. Il est généralement géré lorsque la propriété *AutoChek* est égale à *false*.
CheckedChanged	Se produit suite au changement d'état d'une case à deux états.
CheckedStateChanged	Se produit suite au changement d'état d'une case à trois états.



Exemple d'interface comportant des cases à cocher

<u>Exemple 1 :</u> Code de création du contrôle CheckBox "Service 1"

```
// Dans la liste des attributs du formulaire il faut ajouter
private System.Windows.Forms.CheckBox checkBox1;

// Dans le code d'initialisation du formulaire il faut mettre
this.checkBox1 = new System.Windows.Forms.CheckBox();
this.checkBox1.Location = new System.Drawing.Point(48, 32);
this.checkBox1.Name = "checkBox1";
this.checkBox1.Size = new System.Drawing.Size(176, 16);
this.checkBox1.TabIndex = 1;
this.checkBox1.Text = " Service 1";
```

<u>Exemple 2 :</u> Code de traitement d'un clic sur le contrôle CheckBox "Service 1"

Pour un contrôle à trois états :

```
private void checkBox1_Click(object sender, System.EventArgs e)
{
   switch(checkBox1.CheckState)
   {
      case CheckState.Checked:
         // Mettre ici le code pour l'état checked.
         break;
      case CheckState.Unchecked:
         // Mettre ici le code pour l'état unchecked.
         break;
      case CheckState.Indeterminate:
         // Mettre ici le code pour l'état intermediate.
         break;
   }
}
```


© Karim Kalti

La classe RadioButton

- La classe RadioButton représente un contrôle bouton radio.
- Les contrôles boutons radio représentent un ensemble de choix mutuellement exclusifs. Ils s'utilisent essentiellement en groupe et permettent alors à l'utilisateur de sélectionner une option parmi plusieurs.
- Le contrôle bouton possède à quelques exceptions prés les mêmes caractéristiques que le contrôle case à cocher, toutefois :
  - Dans un groupe de boutons radio, une et une seule option peut être sélectionnée alors que dans un groupe de cases à cocher, il est possible de sélectionner autant d'options que voulues.
  - Un contrôle bouton radio ne possède que deux états possibles (Checked et Unchecked). Il n'a pas comme pour les cases à cocher un état estompé.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ButtonBase
          System.Windows.Forms.RadioButton

Constructeur : `public RadioButton();`

Quelques propriétés :

Propriété	Type	Description
Appearance	Enum Appearance	Indique l'apparence de la case à cocher. Les valeurs possibles de l'énumération *Appearance* sont : *Button* (la case prend la forme d'un petit bouton) et *Normal* (la case prend sa forme normale).
AutoCheck	bool	Indique si le bouton radio change automatiquement d'état suite à un clic. Si *AutoCheck* vaut *false*, c'est alors au programmeur de gérer l'état de la case en traitant l'événement *Click*.
Checked	bool	Indique l'état (coché ou non) du bouton.
Text	string	Libellé du bouton radio. Le caractère & sert à spécifier un accélérateur. ALT+Accélérateur permet de changer l'état du bouton.
Enabled	bool	Obtient ou définit une valeur indiquant si le contrôle peut répondre à une interaction utilisateur.
Location	Point	Position du coin supérieur gauche du contrôle bouton radio.
Name	string	Nom interne du contrôle bouton radio.
TabIndex	int	Obtient ou définit l'ordre de tabulation du contrôle dans son conteneur.
TabStop	bool	Obtient ou définit une valeur indiquant si l'utilisateur peut octroyer le focus à ce contrôle à l'aide de la touche TAB.
Visible	bool	Obtient ou définit une valeur indiquant si le contrôle est affiché.

Quelques méthodes :

Méthode	Description
`void Hide()`	Masque le contrôle à l'utilisateur.
`Void Show( )`	Affiche le contrôle à l'utilisateur.

Quelques événements :

Avec le contrôle bouton radio, seul l'événement *CheckedChanged* présente de l'intérêt. Il est signalé lorsqu'un bouton radio change d'état.

Evénement	Description
Click (hérité de **Control**)	Se produit suite à un clic sur le contrôle. Il est généralement géré lorsque la propriété *AutoCheck* vaut *false*.
CheckedChanged	Se produit suite au changement d'état du bouton radio.

© Karim Kalti

La classe GroupBox

- Cette classe représente un contrôle groupe. Ce contrôle est un conteneur qui permet de regrouper logiquement un ensemble de contrôles de n'importe quel type.
- Les groupes sont généralement utilisés pour regrouper des contrôles de type "bouton radio".
- Les boutons radio d'un groupe sont mutuellement exclusifs.
- Des boutons radio appartenant à un même formulaire mais faisant partie de deux groupes différents fonctionnent de manière complètement indépendante.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.GroupBox

Constructeur : `public GroupBox()`

Quelques propriétés :

Outre les propriétés héritées des classes de base et présentées dans les paragraphes précédents : *Texte*, *Enabled*, *Location*, *Name*, *TabIndex*, *TabStop*, *Visible*, etc, le contrôle GroupBox possède une autre propriété intéressante qui est *Controls*.

Propriété	Type	Description
Controls	ControlCollection	C'est une collection qui comporte les contrôles contenus dans le groupe.

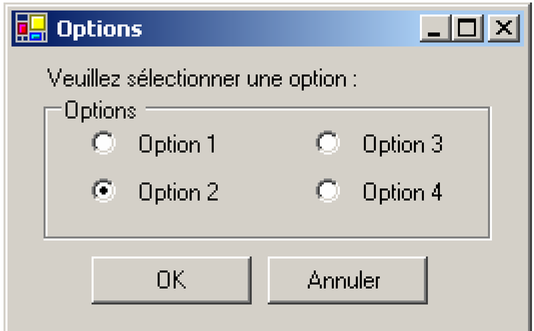
Remarque :
Comme toute collection, *ControlCollection* possède les méthodes usuelles *Add*, *AddRange, Remove, RemoveAt, Cleat, Contains, IndexOf.*

Quelques méthodes :

Méthode	Description
`void Hide()`	Masque le contrôle à l'utilisateur.
`Void Show( )`	Affiche le contrôle à l'utilisateur.

Quelques événements :

Le GroupBox gère les événements gérés par les classes de base (click, etc), toutefois la gestion des événements au niveau de ce contrôle reste rare sur le plan pratique.



Exemple d'interface comportant des boutons radio

© Karim Kalti

Exemple 1 : ***Code de création du groupe "Options" et du bouton radio "Option 1"***

```
// Dans la liste des attributs du formulaire il faut ajouter
// un groupebox et le contrôle radioButton 1.
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.RadioButton radioButton1;

// Dans le code d'initialisation du formulaire il faut mettre
// Code de création du groupe box
this.groupBox1 = new System.Windows.Forms.GroupBox();
this.groupBox1.Location = new System.Drawing.Point(16, 24);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(224, 72);
this.groupBox1.TabIndex = 6;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Options";
// Code de création du bouton radio
this.radioButton1 = new System.Windows.Forms.RadioButton();
this.radioButton1.Location = new System.Drawing.Point(24, 16);
this.radioButton1.Name = "radioButton1";
this.radioButton1.Size = new System.Drawing.Size(72, 16);
this.radioButton1.TabIndex = 0;
this.radioButton1.Text = "Option 1";
// Ajout du bouton radio à la liste des contrôles du groupe
this.groupBox1.Controls.Add(this.radioButton1);
```

Exemple 2 : ***Détermination du bouton radio sélectionné dans un groupe***

```
RadioButton rb = null;
for(int i=0,i<groupBox1.Controls.Count;i++)
{
   rb=(RadioButton) groupBox1.Controls[i];
   // On sort dès qu'on trouve le bouton sélectionné
   if(rb.Checked)
      break;
}
```

La classe ListBox

Cette classe représente une boîte de liste d'articles. Ce contrôle sert à opérer une sélection d'un ou de plusieurs articles à partir de la liste.

Hiérarchie :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ListControl
          System.Windows.Forms.ListBox

Constructeur : `public ListBox();`

Quelques propriétés :

Propriété	Type	Description
BorderStyle	enum BorderStyle	Type de contour. Les valeurs possibles sont *None* (aucun contour), *FixedSingle* (Contour formé d'une simple ligne), *Fixed3D* (contour avec effet de relief).
DrawMode	enum DrawMode	Indique la manière d'affichage de la liste. Les valeurs possibles de cette propriété sont : *Normal* (Affichage automatique, tous les articles ayant la même hauteur), *OwnerDrawFixed* (boîte de liste dont tous les articles ont la même taille), *OwnerDrawVariable* (boîte de liste dont tous les articles non pas la même taille à cause de l'utilisation de police de caractère différente par exemple).
HorizontalScrollBar	bool	Indique si une barre de défilement horizontale doit être automatiquement ajoutée si nécessaire.
ItemHeight	int	Hauteur d'un article. Par défaut, la hauteur d'un article est de 13 pixels. Cette valeur est utilisée dans le cas ou *DrawMode* prend la valeur *OwnerDrawFixed.*

© Karim Kalti

Items	Collection de Items	Collection de libellés d'articles.
ScrollAlwaysVisible	bool	La valeur *true* force la barre de défilement verticale à être affichée même si tous les articles sont visibles.
SelectionMode	enum SelectionMode	Mode de sélection des articles. Les valeurs possibles sont : *One* (un seul article peut être sélectionné), *None* (aucun article ne peut être sélectionné), *MultiSimple* (Chaque clic sélectionne un article supplémentaire), *MultiExtended* (sélection multiple par les combinaisons avec MAJ et CTRL).
Sorted	bool	Indique si les articles de la boîte de liste doivent être triés par ordre alphabétique.

Quelques propriétés runtime :

Propriété	Type	Description
SelectedIndex	int	Index de l'article sélectionné. (0 pour le premier). La valeur -1 indique qu'aucun article n'est sélectionné.
SelectedIndices	SelectedIndex Collection	Collection d'indices des articles sélectionnés.
SelectedItem	Object	Article sélectionné. Cet article peut être de n'importe quel type, il est toutefois dans la majorité des cas de type string.
SelectedItems	SelectedObject Collection	Collection d'articles sélectionnés.

Quelques méthodes :

Méthode	Description
`void ClearSelected()`	Désélectionne tous les articles sélectionnés.
`int FindString(string)`	Renvoie l'index du premier article qui commence par la chaîne passée en argument.
`bool GetSelected(int Index)`	Indique si l'article dont l'index est passé en argument est sélectionné ou non.
`void SetSelected(int Index, bool Value)`	Sélectionne (*Value=true*) ou désélectionne (*Value=false*) l'article dont l'index est passé en argument.

Quelques événements :

Les événements les plus souvent traités par les boîtes de liste sont :

- *Click* et *DoubleClick* : pour détecter un clic ou un double-clic sur un article.
- *KeyPress* : généralement pour détecter une frappe de ENTREE sur un article sélectionné.
- *SelectedIndexChanged* : pour détecter le passage d'un article à l'autre (par clic sur un article ou par les touches de direction du clavier).

Exemple d'interface comportant une boîte de liste

© Karim Kalti

<u>Exemple 1 :</u> Code de création de la liste

```
// Dans la liste des attributs du formulaire il faut ajouter
private System.Windows.Forms.ListBox listBox1;

// Dans le code d'initialisation du formulaire il faut mettre
this.listBox1 = new System.Windows.Forms.ListBox();
this.listBox1.Items.AddRange(new object[] {"Sousse","Monastir","Mahdia","Kairouan"});
this.listBox1.Location = new System.Drawing.Point(40, 32);
this.listBox1.Name = "listBox1";
this.listBox1.Size = new System.Drawing.Size(120, 43);
this.listBox1.TabIndex = 7;
```

<u>Exemple 2 :</u>

L'exemple suivant suppose l'existence de deux listes *listBox1* et *listBox2*. Suite à la sélection d'un article dans la première liste, le programme le récupère et cherche sa présence dans la deuxième liste. Si cet article est présent alors il sera également sélectionné dans la deuxième liste, sinon un message indiquant sont absence sera affiché.

```
private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e)
{
   string ArticleCourant = listBox1.SelectedItem.ToString();
   int index = listBox2.FindString(ArticleCourant);
   if(index == -1)
      MessageBox.Show("l'article est absent de la liste 2");
   else
      listBox2.SetSelected(index,true);
}
```


© Karim Kalti

IHM et gestion des événements

Les événements jouent un rôle très important dans les applications utilisant les IHM. Ils influent en effet sur l'organisation du code dans de telles applications. Ainsi, et à la différence des applications de type console dans lesquelles la fonction principale (main) détermine le déroulement de l'exécution du programme vu qu'elle contient les appels explicites aux fonctions constituant l'application, dans les applications utilisant les IHM l'écriture du code suit une autre organisation qui est plutôt orientée événement (Programmation événementielle). Dans une telle organisation, le rôle de la fonction Main se réduit à l'invocation de la boucle de détection des événements et de dispatching des messages résultant, vers les contrôles composant l'IHM. Il n'y a plus donc d'appels explicites des fonctions constituant l'application dans le Main. Ces dernières sont plutôt codées sous forme de fonctions de traitement des événements interceptés. À titre d'exemple, si on veut effectuer une action A suite au clic sur un bouton B, il suffit de mettre le code correspondant à A dans une fonction F et de signaler F comme étant une fonction de traitement de l'événement Click associé à B.
Pour pouvoir faire de la programmation événementielle en .NET, il faut connaître :

- La liste des événements pouvant être interceptés.
- Le type délégué associé à chaque type d'événements et ce afin de pouvoir traiter ces derniers.

Les événements liés à la souris

Les événements liés à la souris sont nombreux. Parmi les événements les plus importants de cette catégorie il est possible de citer :

- *MouseDown* : signale un enfoncement du bouton de la souris.
- *MouseUp* : relachement du bouton de la souris.
- *MouseMove* : Déplacement de la souris.
- *MouseEnter* : Entrée de survol.
- *MouseLeave* : Sortie de survol
- *MouseHover* : la souris marque un court temps d'arrêt.

Il existe également deux autres événements très fréquemment utilisés qui sont *Click* et *DoubleClick*. La différence entre ces deux événements d'une part et *MouseDown* ainsi que *MouseUp* d'autre part réside essentiellement dans les informations qui accompagnent l'occurrence de ces événements.

L'événement Click

Cet événement a lieu lorsqu'un clic survient sur un contrôle. Le contrôle peut être un objet de n'importe quel classe dérivée de la classe *Control* du .NetFrameWork (Exemple : *Form*, *Button*, *TextBox*, *Label*,…).

Evénement : `public event EventHandler Click;`
Type délégué : `public delegate void EventHandler(Object sender, EventArgs e);`

Exemple de traitement de l'événement Click associé à un bouton

Programme 1

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace WindowsApplication3
{
public class Form1 : System.Windows.Forms.Form
{private System.Windows.Forms.Button button1;
public Form1()
{InitializeComponent();}
#region Windows Form Designer generated code
private void InitializeComponent() {
 this.button1 = new System.Windows.Forms.Button();
```


© Karim Kalti

```
 this.SuspendLayout();
 // button1
 this.button1.Location = new System.Drawing.Point(96,40);
 this.button1.Name = "button1";
 this.button1.Size = new System.Drawing.Size(64, 24);
 this.button1.TabIndex = 0;
 this.button1.Text = "Message";
 this.button1.Click+=new EventHandler(this.FTButton);
 // Form1
 this.ClientSize = new System.Drawing.Size(292, 266);
 this.Controls.Add(this.button1);
 this.Name = "Form1";
 this.Text = "Form1";
 this.ResumeLayout(false);
}
#endregion
void FTButton(Object sender, EventArgs e)
{if(this.button1.Text=="Message1")
   this.button1.Text="Message2";
 else
   this.button1.Text="Message1";
}
static void Main()
{Application.Run(new Form1());}
}}
```

Programme 2 : une autre écriture du programme 1

```
namespace WindowsApplication3
{
public class MyBt : Button
{
 public MyBt()
 {// button1
 this.Location = new System.Drawing.Point(96, 40);
 this.Name = "button1";
 this.Size = new System.Drawing.Size(64, 24);
 this.TabIndex = 0;
 this.Text = "button1";
 this.Click+= new EventHandler(this.FTButton);
}

void FTButton(Object sender, EventArgs e)
{  if(this.Text=="Message1")
        this.Text="Message2";
  else
        this.Text="Message1";
}}

public class Form1 : System.Windows.Forms.Form
{private MyBt button1;
 public Form1()
 {InitializeComponent();}
  #region Windows Form Designer generated code
  private void InitializeComponent() {
  this.button1 = new MyBt();
  this.SuspendLayout();
  // Form1
 this.ClientSize = new System.Drawing.Size(292, 266);
 this.Controls.Add(this.button1);
 this.Name = "Form1";
 this.Text = "Form1";
 this.ResumeLayout(false);
}
#endregion
static void Main()
{Application.Run(new Form1());}
}}
```

Programme 3 : Traitement de deux événements par une seule fonction

```
public class Form2 : System.Windows.Forms.Form
{private System.Windows.Forms.Button button1;
 private System.Windows.Forms.Button button2;
 public Form2()
  {InitializeComponent();}
 private void InitializeComponent()
```


© Karim Kalti

```
 {this.button1 = new System.Windows.Forms.Button();
  this.button2 = new System.Windows.Forms.Button();
  this.SuspendLayout();
  // button1
 this.button1.Location = new System.Drawing.Point(48, 48);
 this.button1.Name = "button1";
 this.button1.Size = new System.Drawing.Size(80, 24);
 this.button1.TabIndex = 0;
 this.button1.Text = "button1";
 this.button1.Click+=new EventHandler(this.FT);
  // button2
 this.button2.Location=new System.Drawing.Point(144, 48);
 this.button2.Name = "button2";
 this.button2.Size = new System.Drawing.Size(80, 24);
 this.button2.TabIndex = 1;
 this.button2.Text = "button2";
 this.button2.Click+=new EventHandler(this.FT);
 // Form2
this.ClientSize = new System.Drawing.Size(272, 117);
this.Controls.Add(this.button2);
this.Controls.Add(this.button1);
this.Name = "Form2";
this.Text = "Form2";
this.ResumeLayout(false);
}
void FT(Object sender, EventArgs e)
{ if(sender == this.button1)
   this.Text="Bouton1 est cliqué";
  if(sender == this.button2)
   this.Text="Bouton2 est cliqué";
}}
```

L'événement DoubleClick

Cet événement a lieu lorsqu'un double clic survient sur un contrôle. La décision relative à la distinction entre un double clic et deux clics successifs est faite par le système d'exploitation. Cette décision se base sur le temps qui sépare les deux clics. Ce paramètre peut être réglé à travers le panneau de configuration de Windows.

Evénement : `public event EventHandler DoubleClick;`
Type délégué : `public delegate void EventHandler(Object sender,  EventArgs e);`

Remarque : Si un contrôle est capable de gérer à la fois les événements *Click* et *DoubleClick* (exemple *Form*) alors un double clic sur ce contrôle va déclencher automatiquement et dans l'ordre l'événement *Click* suivi de l'événement *DoubleClick*.

Les événements MouseEnter, MouseLeave et MouseHover

Evénement : `public event EventHandler MouseEnter;`
Evénement : `public event EventHandler MouseLeave;`
Evénement : `public event EventHandler MouseHover;`

Type délégué : Ces trois événements utilisent tous le même type de délégation

```
public delegate void EventHandler(Object sender,  EventArgs e);
```

Les événements MouseDown, MouseUp et MouseMove

Evénement : `public event MouseEventHandler MouseDown;`
Evénement : `public event MouseEventHandler MouseUp;`
Evénement : `public event MouseEventHandler MouseMove;`

Type délégué : Ces trois événements utilisent tous le même type de délégation

```
public delegate void MouseEventHandler(Object sender, MouseEventArgs e);
```


© Karim Kalti

La classe MouseEventArgs :

Hiérarchie:

System.Object
  System.EventArgs
    System.Windows.Forms.MouseEventArgs

Spécification:

Propriété	Type	Description
Button	Enum MouseButtons	Indique quel bouton de la souris est enfoncé. Les valeurs possibles de l'énéumération MouseButtons sont {Left, Midle, None, Right,…}
Clicks	int	Indique le nombre de fois le bouton de la souris a été appuyé puis relâché.
Delta	int	Nombre de détente sur la molette (si la souris en possède une)
X	int	Coordonné X par rapprot à l'aire client. L'axe des x est orienté de gauche vers la droite à partir du coin supérieur gauche.
Y	int	Coordonné Y. L'axe des y est orienté du haut vers le bas à partir du coin supérieur gauche.

Exemple :

Affichage des coordonnées de la souris suite à l'appui sur l'un de ses boutons (la souris survole un formulaire).

```
 // this référence le formulaire
… … …
this.MouseDown += new System.Windows.Forms.MouseEventHandler(this.FT);
… … …
// FT est une méthode de la classe qui représente le formulaire
void FT(object sender, System.Windows.Forms.MouseEventArgs e)
{
       MessageBox.Show("X: "+e.X+", Y :"+e.Y);
}
```

Les événements liés au clavier

Le focus :

- Dans une interface graphique, le contrôle qui est sélectionné est dit le contrôle qui a le focus.
- Le focus peut être passé d'un contrôle graphique à un autre à l'aide de la touche TAB ou des touches MAJ+TAB.
- Si la propriété *TabStop* d'un contrôle vaut *true*, alors il y a arrêt sur le composant lors du passage du focus.
- L'ordre de réception du focus par les contrôles est défini par la propriété *TabIndex* de chaque contrôle.
- Un événement engendré par le clavier est automatiquement envoyé par Windows au contrôle ayant le focus.
- Trois types d'événements sont liés au clavier :
    - *KeyDown* : signalé lors de l'enfoncement d'une touche du clavier. Cet événement peut être déclenché par n'importe quelle touche.
    - *KeyUp* : signalé lors du relâchement d'une touche. Il peut être déclenché par n'importe quelle touche.
    - *KeyPress* : signalé lors d'une frappe d'une touche correspondant à un caractère (alphanumérique).

- Lors de la détection de l'un de ces événements par Windows, ce dernier exécute automatiquement le délégué qui lui correspond. A titre d'exemple, lors de la détection de l événement *KeyDown*, Windows lance automatiquement le délégué *KeyDown* du contrôle ayant le focus.

Les événements KeyDown et KeyUp

Evénement : `public event KeyEventHandler KeyDown;`
Evénement : `public event KeyEventHandler KeyUp;`

Type délégué :

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e);
```


© Karim Kalti

La classe KeyEventArgs :

Hiérarchie:

System.Object
  System.EventArgs
    System.Windows.Forms.KeyEventArgs

Spécification :

Propriété	Valeur	Description
Alt	true, false	Indique si la touche Alt est enfoncée
Control	true, false	Indique si la touche Ctrl est enfoncée
Handled	true, false	Indique que l'événement a été traité
Shift	true, false	Indique si la touche Maj est enfoncée
KeyCode	Enum Keys	Une des valeurs de l'énumération Keys

Pour avoir la spécification complète de la classe *KeyEventArgs*, il faut consulter la documentation du .NetFramework

Exemples de valeurs de l'énumération Keys

- Les touches de F1 à F24 correspondent à Keys.Fi (exemple : la valeur Keys.F5 correspond à la touche F5).
- Les valeurs Keys.D0 jusqu'à Keys.D9 correspondent aux touches numériques.
- Les valeurs Keys.A jusqu'à Keys.Z correspondent aux touches alphabétiques.
- Keys possède d'autres valeurs qui correspondent aux touches spéciales du clavier. Comme exemple de ces valeurs, il est possible de citer : Keys.Back, Keys.Cancel, Keys.Home, Keys.End, Keys.Del, etc.

La liste complète des valeurs de l'énumération Keys peut être consultée dans la documentation du .NetFrameWork.

Exemple : Manipulation du paramètre e dans une fonction de traitement d'un événement KeyUp ou KeyDown

Si on veut vérifier si l'utilisateur a appuyé en même temps sur la touche MAJ et sur le bouton de déplacement vers la gauche, alors la condition de test doit être comme suit :

```
void FT(object sender, System.Windows.Forms.KeyEventArgs e)
{
  if(e.Shift== true && e.KeyCode==Keys.Left)
  {    // Mettre le code de traitement ici
        … … … … … …
  }
}
```

Exemple d'inscription d'une fonction de traitement auprès du gestionnaire d'un événement KeyUp

```
This.KeyUp += new System.Windows.Forms.KeyEventHandler(FT)
```

Remarque :

- Aucune distinction n'est faite entre les minuscules et les majuscules par les événements *KeyUp* et *KeyDown* : `e.KeyCode==Keys.A` est vraie pour 'A' et 'a'.
- Pour intercepter les événements au niveau de la fenêtre il faut que la propriété *KeyPreview* de cette fenêtre soit mise à *true*.

L'événement KeyPress

Evénement : `public event KeyPressEventHandler KeyPress;`

Type délégué :

```
public delegate void KeyPressEventHandler(Object sender, KeyPressEventArgs e);
```


© Karim Kalti

La classe KeyPressEventArgs :

Hiérarchie:

System.Object
  System.EventArgs
    System.Windows.Forms.KeyPressEventArgs

Spécification:

Propriété	Valeur	Description
Handled	true, false	Indique que l'événement a été traité
KeyChar	char	Caractère tapé au clavier

Exemple :

Si on veut tester si le caractère tapé est é alors la condition de test doit être la suivante :

```
if(e.KeyChar == 'é')
```

Remarques :

- l'événement *KeyPress* fait la distinction entre minuscule et majuscule.
- Généralement les événements *KeyUp* et *KeyDown* sont utilisés pour détecter les touches de direction, les touches de fonctions et les touches spéciales du clavier comme par exemple Del, Back,…
- *KeyPress* est utilisé pour traiter les événements liés aux touches alphanumériques.

© Karim Kalti

Les menus

Exemple de menu

La classe MainMenu

Un menu en .NET est représenté par le contrôle MainMenu. Ce contrôle représente en réalité la structure qui contient le menu. En mode programmation, ce contrôle est créé à travers la classe qui porte le même nom "MainMenu".

Hiérarchie de la classe :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Menu
        System.Windows.Forms.MainMenu

Quelques propriétés :

Propriété	Description
Container (hérité de Component)	Obtient une référence au conteneur du composant.
Handle	Obtient une valeur qui représente le handle de la fenêtre contenant le menu.
IsParent	Obtient une valeur indiquant si ce menu contient des éléments de menu. Le MainMenu joue dans ce cas le rôle de parent pour ces éléments de menu. Cette propriété est en lecture seule.
MdiListItem	Obtient une valeur qui indique le MenuItem utilisé pour afficher une liste de formulaires enfants MDI.
MenuItems	Obtient une valeur qui indique la collection d'objets MenuItem associée au menu.

La classe MenuItem

Un Menu est généralement composé d'un ensemble de contrôles appelés éléments de menu. Chaque élément de menu est représenté par la classe MenuItem.

Hiérarchie de la classe :

System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Menu
        System.Windows.Forms.MenuItem

© Karim Kalti

Quelques propriétés :

Propriété	Valeur	Desciption
BarBreak		Obtient ou définit une valeur qui indique si MenuItem est placé sur une nouvelle ligne (pour un élément de menu ajouté à un objet MainMenu) ou dans une nouvelle colonne (pour un élément de sous-menu ou un élément de menu affiché dans ContextMenu).
Break		Obtient ou définit une valeur qui indique si l'élément est placé sur une nouvelle ligne (pour un élément de menu ajouté à un objet MainMenu) ou dans une nouvelle colonne (pour un élément de menu ou un élément de sous-menu affiché dans un ContextMenu).
Checked	T/F	Obtient ou définit une valeur qui indique si une coche apparaît en regard du texte de l'élément de menu.
Container (hérité de Component)		Obtient le IContainer qui contient Component.
DefaultItem		Obtient ou définit une valeur indiquant si l'élément est l'élément de menu par défaut.
Enabled	T/F	Obtient ou définit une valeur qui indique si l'élément de menu est activé.
Handle (hérité de Menu)		Obtient une valeur qui représente le handle de fenêtre pour le menu.
Index	int	Obtient ou définit une valeur qui indique la position de l'élément de menu dans son menu parent.
IsParent		Substitué. Obtient une valeur qui indique si l'élément de menu contient des éléments de menu enfants. (sous menu)
MdiList		Obtient ou définit une valeur qui indique si l'élément de menu va être rempli avec une liste des fenêtres enfants MDI (interface multidocument) affichées dans le formulaire associé.
MdiListItem (hérité de Menu)		Obtient une valeur qui indique le MenuItem utilisé pour afficher une liste de formulaires enfants MDI.
MenuItems (hérité de Menu)		Obtient une valeur qui indique la collection d'objets MenuItem associée au menu.
MergeOrder		Obtient ou définit une valeur qui indique la position relative de l'élément de menu lorsqu'il est fusionné avec un autre.
MergeType		Obtient ou définit une valeur qui indique le comportement de l'élément de menu lorsque son menu est fusionné avec un autre.
Mnemonic		Obtient une valeur qui indique le caractère mnémonique associé à cet élément de menu.
OwnerDraw		Obtient ou définit une valeur qui indique si l'élément de menu est dessiné par le code que vous fournissez ou par Windows.
Parent		Obtient une valeur indiquant le menu qui contient cet élément de menu.
RadioCheck	T/F	Obtient ou définit une valeur qui indique si le MenuItem affiche une case d'option au lieu d'une coche lorsqu'il est activé.
Shortcut	Enum Shortcut	Obtient ou définit une valeur qui indique la touche de raccourci associée à l'élément de menu.
ShowShortcut	T/F	Obtient ou définit une valeur qui indique si la touche de raccourci associée à l'élément de menu est affichée en regard de la légende de l'élément de menu.
Site (hérité de Component)		Obtient ou définit le ISite de Component.
Text	string	Obtient ou définit une valeur qui indique la légende de l'élément de menu.
Visible		Obtient ou définit une valeur qui indique si l'élément de menu est visible.

© Karim Kalti

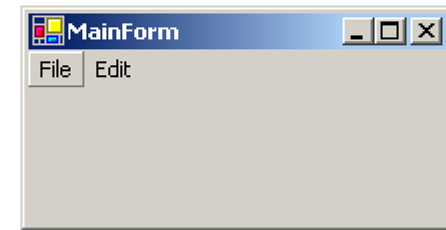
- Pour qu'un menu principal (*MainMenu*) soit affiché sur un formulaire (*Form*), il faut qu'il soit affecté à la propriété *Menu* du formulaire.

- Pour qu'un élément de menu (*MenuItem*) soit affiché dans un menu principal (*MainMenu*), il faut qu'il soit ajouté à la liste des *MenuItems* du menu principal. Le menu principal joue dans ce cas le rôle de parent ou de conteneur pour l'élément de menu.

- Un élément de menu (*MenuItem*) peut jouer également le rôle de parent pour un sous-menu composé lui aussi de plusieurs éléments de menu. Pour que ce menu soit affiché, il faut que ses éléments soient ajoutés à la liste des *MenuItems* de l'élément parent.

Exemple 1:

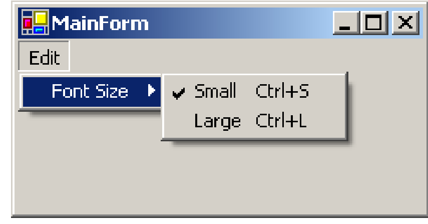
```
// Méthode membre de la classe qui représente le formulaire
public void CreateMyMainMenu()
{
   // Création d'un MainMenu vide.
   MainMenu mainMenu1 = new MainMenu();

   MenuItem menuItem1 = new MenuItem();
   MenuItem menuItem2 = new MenuItem();

   menuItem1.Text = "File";
   menuItem2.Text = "Edit";

   // Ajout de deux éléments de menu au menu principal.
   mainMenu1.MenuItems.Add(menuItem1);
   mainMenu1.MenuItems.Add(menuItem2);

   // Attachement du menu principal au formulaire.
   // this référence le formulaire
   this.Menu = mainMenu1;
}

private void InitializeComponent()
{ … … … …
  //L'appel de CreateMyMainMenu
  CreateMyMainMenu();
  … … … …
}
```



Résultat de l'exemple 1

Exemple 2:

```
// Méthode membre de la classe qui représente le formulaire
public void CreateMyMenu()
{
    MainMenu mainMenu1 = new MainMenu();
    MenuItem menuItem1 = new MenuItem();
    MenuItem menuItem2 = new MenuItem();
    MenuItem menuItem3 = new MenuItem();
    MenuItem menuItem4 = new MenuItem();
    menuItem1.Text = "&Edit";
    menuItem2.Text = "Font Size";
    menuItem3.Text = "Small";
    menuItem3.Checked = true;
    menuItem3.Shortcut = Shortcut.CtrlS;
    menuItem4.Text = "Large";
    menuItem4.Shortcut = Shortcut.CtrlL;
    menuItem4.Index = 1;
    menuItem2.MenuItems.Add(menuItem3);
    menuItem2.MenuItems.Add(menuItem4);
    menuItem1.MenuItems.Add(menuItem2);
    mainMenu1.MenuItems.Add(menuItem1);
    this.Menu = mainMenu1;
}
```



Résultat de l'exemple 2

© Karim Kalti

Quelques événements interceptés par les éléments de menu

L'événement Click :

Evénement : `public event EventHandler Click;`
Type délégué : `public delegate void EventHandler(object sender,  EventArgs e);`

Exemple de gestion de l'événement Click

```
public void CreateMyMenu()
{

    MainMenu mainMenu1 = new MainMenu();

    MenuItem topMenuItem = new MenuItem();
    MenuItem menuItem1 = new MenuItem();

    topMenuItem.Text = "&File";
    menuItem1.Text = "&Open";

    topMenuItem.MenuItems.Add(menuItem1);
    mainMenu1.MenuItems.Add(topMenuItem);

    menuItem1.Click += new System.EventHandler(this.menuItem1_Click);

    this.Menu=mainMenu1;
}

private void menuItem1_Click(object sender, System.EventArgs e)
{
   // Instanciation d'une boîte d'ouverture de fichier.
   OpenFileDialog fd = new OpenFileDialog();
   fd.DefaultExt = "*.*";
   fd.ShowDialog();
}
```

Traitement de plusieurs événements de Click par une seule fonction

```
… … … …
MenuItem m = (MenuItem)sender;
if(m.text=="Ouvrir)
  {// Code du menu ouvrir
   … … … }
if(m.text=="Enregistrer)
  {// Code du menu Enregistrer
   … … …}
… … … …
… … … …
```

L'événement PopUp

- Cet événement se produit avant l'affichage de la liste des éléments de menu d'un élément de menu.
- Cet événement est généralement utilisé pour faire des modifications des paramètres du menu (activation, désactivation,…) suivant les données du programme.

Evénement: public event EventHandler Popup;
Type délégué: `public delegate void EventHandler(object sender,  EventArgs e);`

Changement du menu principal suivant l'état du programme

Il suffit de créer plusieurs menus principaux (*MainMenu*) et d'affecter l'un d'eux à la propriété *Menu* du formulaire et ce suivant l'état de l'exécution du programme.

© Karim Kalti

Accès aux données en .NET (ADO.NET)

Introduction

L'accès aux données est un élément important dans la majorité des applications développées aujourd'hui. Microsoft a proposé une panoplie de technologies d'accès aux données (ODBC, DAO, RDO, ADO). ADO.NET (anciennement appelé ADO+) constitue la dernière technologie proposée dans ce cadre.

Rappel

Terminologie

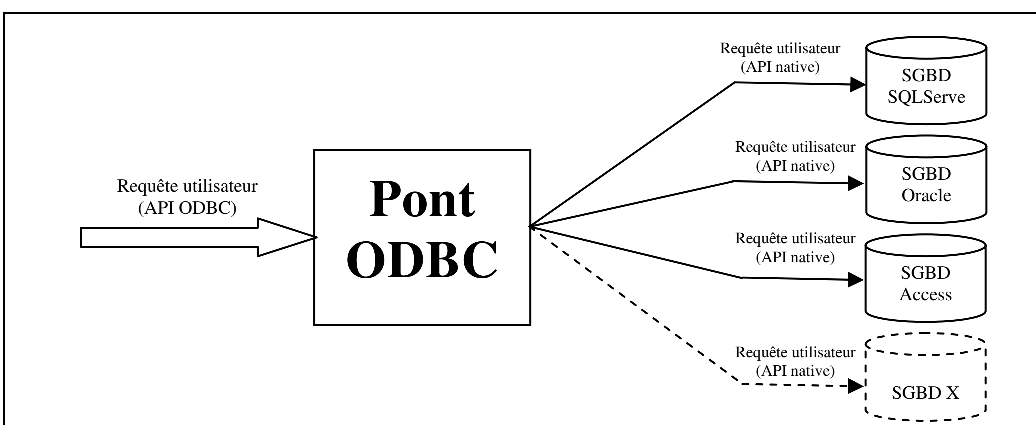
- **API (Application Programming Interfaces) :** une API est une bibliothèque de modèles de fonctions (interfaces) généralement de bas-niveau réalisant une tâche bien précise. Dans un contexte orienté objet, une API se présente comme une bibliothèque de classes interfaces disposant de méthodes permettant de réaliser des tâches bas-niveau.

- **API d'accès aux données :** Une API d'accès aux données est une bibliothèque de fonctions ou d'objets permettant de faire les opérations d'accès aux bases de données, d'interrogation de ces bases (envoi des requêtes) et de récupération de données à partir de ces bases.

- **API native d'accès aux données :** Chaque SGBD possède une API propriétaire tenant compte de ses spécificités et qui permet de faire les opérations d'accès à cet SGBD. Une telle API est appelée une API native.

ODBC (Open Data Base Connectivity)

ODBC (Open Data Base Connectivity) est une API de fonctions (non orientée objet) unifiée d'accès aux données qui a pour objectif de permettre l'accès à n'importe quel SGBD sans nécessiter une connaissance des API natives.



Architecture d'un pont ODBC :

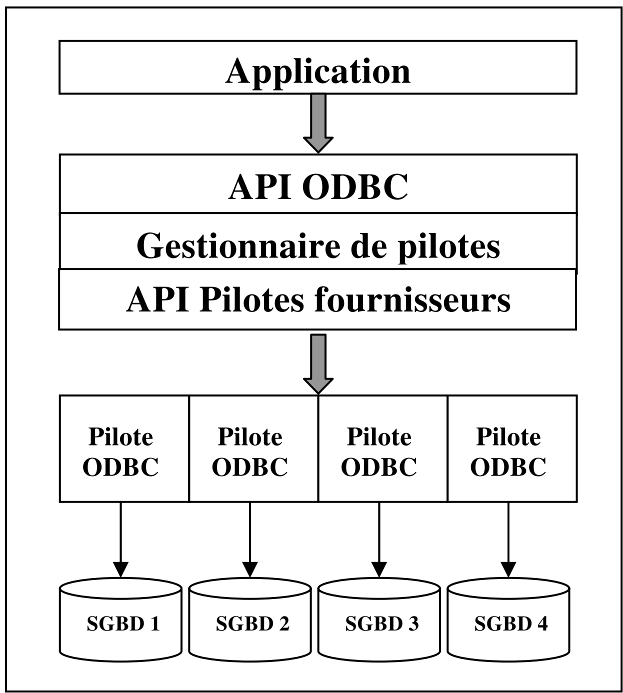
Un pont ODBC est composé de trois couches :

- La première couche est une couche de réception des requêtes adressées par les applications à un SGBD. Ces requêtes sont spécifiées par les applications à l'aide de l'API ODBC.
- La deuxième couche est constituée d'un composant logiciel appelé gestionnaire de pilotes. Ce gestionnaire joue le rôle d'intermédiaire qui va décider du choix du pilote ODBC approprié qui va prendre en charge la requête adressée par l'application. Ce choix se base sur le type de la base de données interrogée. Le

© Karim Kalti

gestionnaire assure le chargement en mémoire et la libération du pilote sélectionné.

- La troisième couche assure la communication entre le gestionnaire des pilotes et les pilotes ODBC. Cette communication est faite à travers des appels effectués par le gestionnaire, aux fonctions des pilotes ODBC. Ces fonctions font partie de l'API unifiée définie par le standard ODBC et utilisée par les développeurs des pilotes ODBC.



Architecture d'un pont ODBC

Pilote ODBC

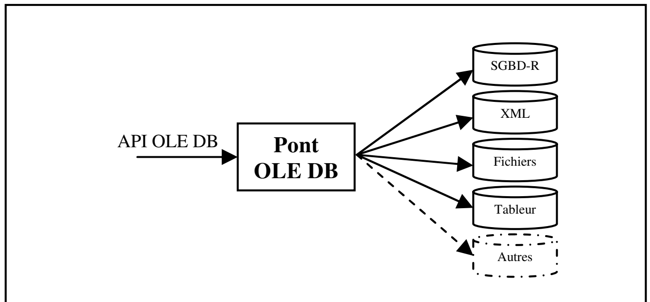
Un pilote ODBC est un composant logiciel qui a pour objectif de traduire la requête écrite avec l'API ODBC unifiée vers un format (écrit à l'aide de l'API native) propre au SGBD interrogé. Tout SGBD qui veut être accessible à travers ODBC doit fournir un pilote ODBC qui lui est spécifique. Ainsi par exemple, pour accéder à une base de données SQLServer ou Oracle à travers ODBC, il faut installer sur la machine sur laquelle tourne l'application le pilote ODBC pour SQL Server, respectivement pour Oracle. Pour voir sous Windows XP la liste des pilotes ODBC installés sur une machine il faut aller à :
Panneau de configuration → Outils d'administration → Sources de données ODBC

OLE DB :

OLE DB (Open Linking and Embeding for DataBases) est une API unifiée d'accès aux données qui offre plus de possibilités que l'API ODBC. En effet ODBC permet l'accès aux bases de données relationnelles seulement. Avec OLE DB il est possible d'accéder à n'importe quelle source de données, relationnelle ou non (Système de fichiers, XML, Tableur, Messagerie,…).

OLE DB utilise une collection d'objets qui définit un modèle différent de celui de ODBC. Ce modèle s'articule autour des objets de base suivants : *Data Source, Session, Command, Rowset*.
A ces objets de base s'ajoutent d'autres objets qui sont : *Enumerator, Transaction, Error*.

© Karim Kalti



Exemples de sources de données accessibles à travers un pont OLE DB

Avantages et inconvénients de l'utilisation des APIs unifiées.

Sans l'utilisation d'une API unifiée, les programmeurs d'applications sont obligés d'appendre différentes API natives pour faire des accès à différents SGBDs. Ceci alourdit la charge de travail des développeurs surtout dans les milieux professionnels où l'on est amené souvent à travailler sur des SGBD différents d'un projet à un autre. L'intérêt des APIs unifiées dans ce cas est grand puisqu'elles vont alléger la charge de travail des développeurs. Les APIs unifiées sont toutefois moins rapides que les API natives à cause notamment des étapes de conversion de requêtes qui sont nécessaires dans ce cas.

Présentation de ADO.NET

ADO.NET (ActiveX Database Object) est une nouvelle API d'accès aux données proposée par Microsoft. Cette API permet aux applications développées en .NET de travailler avec différents types de sources de données.

Caractéristiques :

- ADO.NET permet de travailler en mode connecté et en mode déconnecté.
- ADO.NET permet de travailler d'une manière complètement indépendante du type de la base de données.
- ADO.NET permet l'interopérabilité avec d'autres sources de données (relationnelle ou non telles que les systèmes de fichiers, Active Directory, etc.). Cette interopérabilité est obtenue grâce à la possibilité d'importer et d'exporter les données au format XML.

Configuration requise pour ADO.NET

- ADO.NET est fourni avec le FrameWork.NET.
- Il est nécessaire par ailleurs d'installer MDAC (Microsoft Data Access Components version 2.6 ou plus) pour pouvoir utiliser les fournisseurs de données SQL Server ou OLE DB.
- ADO.NET peut être utilisé sous Windows CE/95/98/ME/NT4 SP6a/2K/XP/2003 Server.

Les modes de travail avec ADO.NET

ADO.NET offre deux modes de travail :

Le mode déconnecté

Ce mode permet de travailler d'une manière déconnectée de la source de données. Les étapes de travail avec ce mode se déroulent comme suit :

- Le client entre en communication avec le serveur de BD pour effectuer une requête (généralement un SELECT).
- Le serveur envoie en une seule fois le résultat de la requête au client et la communication est ensuite coupée. Le client dispose donc d'une copie locale d'une partie de la BD serveur (qui correspond à sa requête).
- Le client peut travailler sur la copie locale sans solliciter ni le serveur ni le réseau. Il peut modifier cette partie puis l'envoyer à nouveau au serveur à travers une nouvelle connexion à la BD.

© Karim Kalti

Le mode connecté

Ce mode permet de travailler d'une manière connectée par rapport à la source de données. Les étapes de travail avec ce mode se déroulent comme suit :

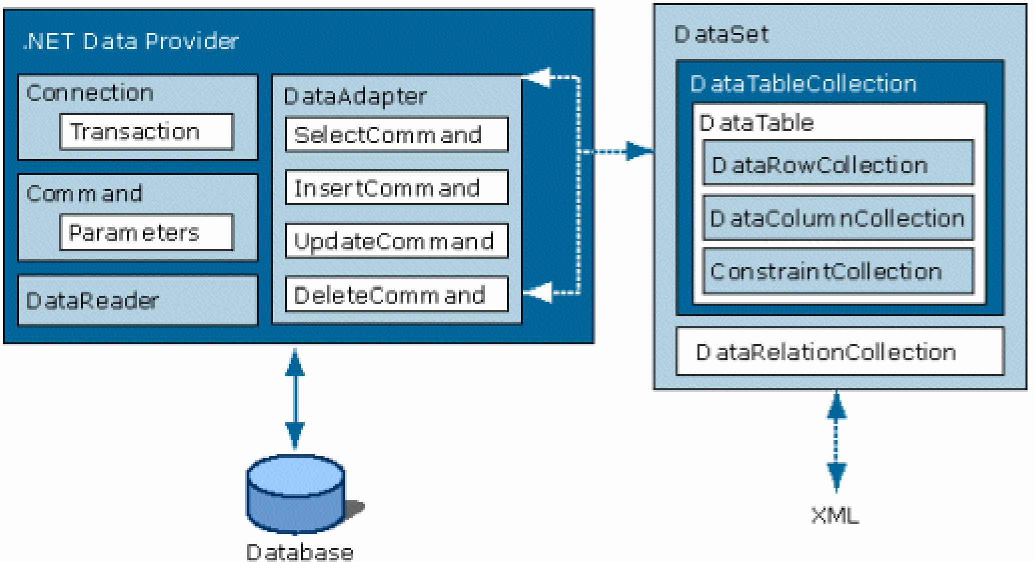
- Le client effectue une requête auprès du serveur (Exemple un SELECT).
- Le serveur calcule le résultat de la requête, il maintient une communication logique avec le client et lui envoie le résultat de son SELECT ligne par ligne suite aux demandes de ce client (déplacement entre les enregistrements de son RecordSet).
- Le serveur doit savoir où il en est dans sa communication avec chaque client qui lui est connecté.

Comparaison des deux modes

- Le mode déconnecté est plus adapté à la manière de travailler sur Internet. En effet sur Internet on ne peut pas connaître quand est ce que le client (léger dans ce cas) décide de se déconnecter de la page d'accès aux données (Il peut la laisser ouverte et passer à d'autres pages). Maintenir plusieurs connexions ouvertes en continue avec la source de données constitue un frein à la capacité de montée en charge d'une application. (ce problème peut toutefois être évité en utilisant la propriété *TimeOut* liée à la variable de Session).
- Le mode déconnecté engendre une plus grande surcharge du réseau mais seulement au moment de la transmission du résultat de la requête ce qui risque de provoquer des courts blocages.
- Le mode connecté est préféré lorsque le jeu d'enregistrement à traiter est trop grand pour être facilement transféré et stocké localement en mémoire.

Architecture générale de ADO.NET

ADO.NET est une API composée d'une collection de classes interfaces répartie suivant un modèle en deux couches comme le montre la figure ci-dessous :



Les deux couches qui composent ce modèle sont :

- Une couche connectée qui comporte des classes managées qui font des accès physiques à la base de données. Ces classes constituent ce que ADO.NET appelle un fournisseur de données ou Data Provider (*Connection, Transaction, Command, Parameters, DataReader, DataAdapter,…*).

- Une couche déconnectée qui comporte des classes qui ne sont pas en liaison directe avec les bases de données. Ces classes constituent une sorte de cache pour le stockage en local d'une partie des données de la base.

© Karim Kalti

Le modèle de fournisseur de données

Un fournisseur de données est une API qui permet de faire des accès physiques à la source de données. Cette API est composée de la collection de classes interfaces de la couche connectée de ADO.NET. Les quatre objets de cette couche sont présentés dans le tableau ci-dessous :

Objet	Description
`Connection`	Etablit une connexion à une source de données spécifique.
`Command`	Exécute une commande adressée à une source de données.
`DataReader`	Lit un flux à partir d'une source de données. Seule la lecture est possible. Elle est faite dans un sens unique (vers l'avant).
`DataAdapter`	Remplit un DataSet à partir d'une source de données et met à jour cette dernière à partir d'un DataSet.

A ces classes de base viennent s'ajouter d'autres qui les complètent :

Objet	Description
`Transaction`	Permet de lister les commandes dans une transaction.
`CommandBuilder`	Un assistant qui génère automatiquement les propriétés des commandes d'un DataAdapter.
`Parameter`	Définit les entrées, les sorties et retourne les valeurs des paramètres pour les commandes et les procédures stockées.
`Exception`	Exception retournée lorsqu'une erreur est rencontrée au niveau de la source de données. Pour une erreur rencontrée au niveau du client le fournisseur de données intercepte une exception du FrameWork.NET.
`Error`	Expose les informations accompagnant un avertissement ou une erreur retourné(e) par une source de données.

Les classes susmentionnées ne sont pas directement instanciables. Elles sont en effet définies dans le FrameWork .NET sous forme d'interfaces (car elles définissent un modèle). Par exemple l'objet *Connection* est représenté par l'interface *IDbConnection*, l'objet *DataAdpater* est représenté par l'interface *IDbDataAdapter*, l'objet *Command* est représenté par l'interface *IDbCommand* et l'objet *DataReader* est représenté par l'interface *IDbDataReader*.

Fournisseurs de données de ADO.NET

Un fournisseur de données est une implémentation de l'API du modèle des fournisseurs (collection d'interfaces présentée dans le paragraphe précédent). ADO.NET dans sa version 1.1 propose quatre fournisseurs différents (quatre implémentations différentes) :

- **Le fournisseur managé (OLEDB managed Provider) :** Ce fournisseur est une implémentation du modèle des fournisseurs à l'aide de l'API de la technologie OLEDB proposée par Microsoft. Les classes qui représentent cette implémentation sont définies dans l'espace de noms ***System.Data.OleDb***. Ce fournisseur permet aux applications écrites en .NET d'accéder à la majorité des SGBDs et ce à travers le pont OLEDB.
- **Le fournisseur managé ODBC (ODBC managed Provider) :** Ce fournisseur est une implémentation du modèle des fournisseurs à l'aide de l'API ODBC. Il permet de ce fait d'accéder à n'importe quel SGBD possédant un pilote ODBC. Les classes composant ce fournisseur sont définie dans l'espace de noms ***System.Data.ODBC***.
- **Le fournisseur managé SQL Server (SQL Server managed Provider) :** Ce fournisseur est une implémentation du modèle des fournisseurs à l'aide de l'API native du SGBD SQL Server. Il permet d'accéder seulement au SGBD SQL Server (version 7 ou plus) pour lequel il offre de meilleurs performances que celles obtenues avec le fournisseur OLEDB (grâce à l'accès natif bien sûr). Les classes composant ce fournisseur sont définies dans l'espace de noms ***System.Data.SqlClient***.
- **Le fournisseur managé ORACLE (ORACLE managed Provider) :** Ce fournisseur est une implémentation du modèle des fournisseurs à l'aide de l'API native du SGBD ORACLE. Il ne peut être utilisé donc qu'avec ORACLE. Les classes composant ce fournisseur sont définies dans l'espace de noms ***System.Data.OracleClient***.

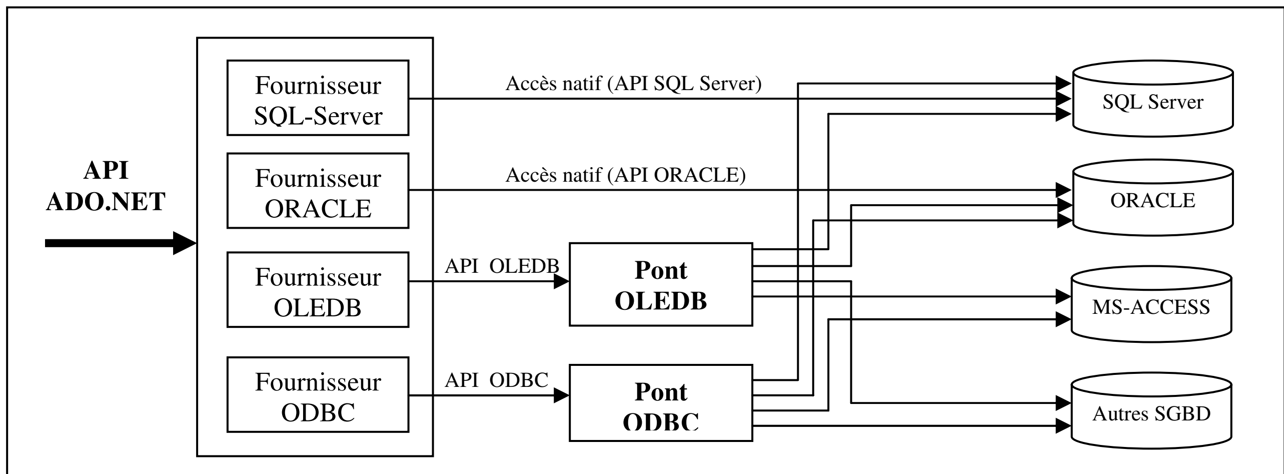
Les deux premiers fournisseurs utilisent des ponts (OLEDB et ODBC) et offrent par conséquent un accès à un grand nombre de SGBD. Les deux derniers fournisseurs font respectivement des accès natifs aux SGBD SQL Server et Oracle. Ils offrent de meilleures performances pour ces deux SGBDs par rapport à celles obtenues à travers les ponts OLEDB et ODBC.

© Karim Kalti

Remarque 1:
Tout SGBD qui se veut accessible d'une manière native à travers une application .NET doit proposer sa propre implémentation du modèle des fournisseurs de données (son propre pilote ADO.NET).

Remarque 2 :
Chaque classe qui implémente une interface faisant partie d'un fournisseur porte le nom de cette interface préfixé du nom du fournisseur auquel elle appartient. Le tableau suivant donne quelques exemples de noms de classes implémentées par les différents fournisseurs :

Interface	OLE DB Provider	ODBC Provider	SQL Provider	Oracle Provider
`Connection`	`OleDbConnection`	`ODBCConnection`	`SqlConnection`	`OracleConnection`
`Command`	`OleDbCommand`	`ODBCCommand`	`SqlCommand`	`OracleCommand`
`DataReader`	`OleDbDataReader`	`ODBCDataReader`	`SQLDataReader`	`OracleDataReader`
`DataAdapter`	`OleDbDataAdapter`	`ODBCDataAdapter`	`SqlDataAdapter`	`OracleDataAdapter`



Les différentes APIs implémentant les fournisseurs de données de ADO.NET

Le reste de ce chapitre présente les différentes étapes de l'accès à une base de données à l'aide du fournisseur OLEDB Provider. L'accès à l'aide des autres fournisseurs se fait de la même manière puisque tous les fournisseurs implémentent le même modèle.

Connexion à une base de données

La connexion à une base de données en ADO.NET se fait à l'aide de l'interface *Connection*. Le fournisseur OLEDB implémente cette interface à l'aide de la classe *OleDbConnection*.

Espace de nom de OleDbConnection:

System.Data.OleDB ;

Constructeurs :

Constructeur	Description
`OLeDbConnection()`	Crée un objet de connexion non initialisé.
`OleDbConnection(string connectionString)`	Crée un objet de connexion tout en spécifiant une chaîne de connexion. Ce constructeur décortique automatiquement la chaîne pour remplir les propriétés de l'objet de connexion.

© Karim Kalti

Principales propriétés de OleDbConnection :

Propriété	Type	Description
`ConnectionString`	string	Il s'agit d'une chaîne de caractères au format bien particulier qui est passée au constructeur de l'objet de connexion. Ce constructeur la place dans l'attribut courant et la décortique pour initialiser les autres attributs de la classe. Il est à noter que certains attributs sont en lecture seule et ne peuvent être initialisée en dehors du constructeur de la classe et donc à travers la chaîne de connexion.
`ConnectionTimeOut`	int	Durée en secondes durant laquelle l'établissement d'une connexion au serveur de BD est tenté. Si pendant ce laps temps la connexion échoue alors une erreur est signalée. Une valeur nulle indique une attente infinie (à éviter). Par défaut la valeur est de 15 secondes.
`DataBase`	string	Nom de la base de données (à utiliser avec SQL Server).
`DataSource`	string	Nom de la source de données (à utiliser dans le cas de MSACCESS ou autres).
`Password`	string	Mot de passe pour accéder à la base
`Provider`	string	Identificateur du type de la BD

Principales méthodes de OleDbConnection :

Méthode	Description
`void Open()`	Ouverture explicite de la base. Une exception est générée en cas d'échec de l'ouverture.
`void Close()`	Fermeture explicite de la base de données.

Spécification de la valeur de la propriété DataSource :

La valeur de la propriété *DataSource* désigne le chemin de la source de données. Ce chemin peut être :

- Un chemin absolu. (Exemple : data source = C:\repertoire\MaBase.mdb)
- Un chemin relatif si la source se trouve dans le même dossier que l'exécutable de l'application ou si elle se trouve dans un sous dossier de celui-ci. (Exemple pour un même dossier : data source = MaBase.mdb)
- Un chemin réseau si la source se trouve sur une autre machine du réseau local. (Exemple: data source = \\machine\Lecteur\...\ MaBase.mdb)

Spécification de la valeur de la propriété Provider :

Cette propriété désigne le type de la source de données à laquelle est fait l'accès. Le tableau ci-dessous donne quelques exemples de valeurs utilisées pour un certain nombre de SGBD.

SGBD	Valeur
Access	Provider = Microsoft.Jet.OLEDB.4.0
SGL Server	Provider =SQLOLEDB
AS400	Provider = IBMA400 ;
DB2	Provider = DB2OLEDB ;
Fox Pro	Provider = vfpOLEDB ;
Oracle	Provider = MSDAORA

Exemple d'utilisation de l'objet de connexion :

```
using System;
using System.Data;
using System.Data.OleDb;
...
// Création de la chaîne de connexion
string StrCnn = @"Provider=Microsoft.Jet.OleDb.4.0 ; Data Source=C:\Repertoire\MaBase.mdb";

// Création de l'objet de connexion
OleDbConnection cnn = new OleDbConnection(StrCnn);

// Ouverture de la connexion
cnn.Open();
...
// Mettre ici le code d'interrogation de la source de données
...
// Fermeture de la connexion
cnn.Close();
```


© Karim Kalti

Commentaire :

Suite à la création de l'objet de connexion, l'établissement effectif de la liaison avec la base est amorcé par l'appel à la méthode *Open*. Une fois la connexion ouverte, il devient possible d'adresser à la base des requêtes de différents types (insertion, sélection, …). Enfin, lorsque l'interrogation se termine, il faut fermer la connexion avec la méthode *Close*.

Remarque : utilisation de @ devant la chaîne de connexion

Par défaut le caractère '\' introduit dans les chaînes un caractère de formatage. Pour ne pas le considérer ainsi, il faut faire précéder la chaîne de @. Si non lorsqu'il s'agit de spécifier un chemin, il faut utiliser \\ au lieu de \. Dans ce cas la chaîne de connexion devient :

```
"Provider=Microsoft.Jet.OleDb.4.0 ; Data Source = C:\\Repertoire\\MaBase.mdb";
```

Interrogation de la base de données en mode connecté

L'interrogation de la base de données peut se faire dans un objectif d'insertion, de suppression, de mise à jour ou de sélection de données. En mode connecté, la réalisation de toutes ces opérations passe par l'interface *Command*. Le fournisseur OLEDB implémente cette interface par la classe *OleDbCommand*.

Espace de nom de OleDbCommand :

System.Data.OleDB ;

Constructeurs de OleDbCommand :

Constructeur	Description
`OLeDbCommand()`	Crée un objet de commande non initialisé
`OLeDbCommand (string sqlstr)`	Crée une commande et lui passe une requête SQL.
`OLeDbCommand (string sqlstr, OleDbConnection cnn)`	Crée une commande et lui passe une requête SQL plus un objet de connexion.

Principales propriétés de OleDbCommand :

Propriété	Type	Description
`CommandType`	enum CommandType	Indique le type de la commande. Trois valeurs sont possibles dans ce cadre : • *Text* : Signifie que la commande est une requête SQL. • *StoredProcedure* : signifie que la commande représente un appel à une procédure stockée. • *TableDirect* : signifie que la commande est une requête de sélection du contenu entier d'une table.
`CommandText`	string	Contient le texte de la commande. La manière d'interprétation de ce texte dépend de la valeur de la propriété CommandType. • Si *CommandType = Text* alors *CommandText* représente le texte de la requête SQL. • Si *CommandType=StoredProcedure* alors *CommandText* contient le nom de la procédure stockée qui sera appelée. • Si *CommandType=TableDirect* alors *CommandText* contient le nom de la table dont le contenu sera renvoyé. Cette commande est équivalente à une requête SLECT * appliquée à la table en question.
`CommandTimeOut`	int	Nombre de secondes d'attente avant qu'une commande ne soit déclarée en erreur. La valeur par défaut est de 30 sec.
`Connection`	OleDbConnection	L'objet connexion associé à la commande. Il sert à spécifier la source de données à laquelle est destinée la commande.
`Parameters`	OleDbParameterCollection	Liste des paramètres utilisés par la commande. Par défaut cette collection est vide.

Principales méthodes de OleDbCommand :

Les méthodes les plus utilisées de la classe *OleDbCommand* sont celles qui permettent d'exécuter les commandes. Trois méthodes sont proposées à cet effet :

Méthode	Description
`int ExecuteNonQuery()`	Elle exécute la commande spécifiée dans *CommandText*. Dans ce cas la commande doit être une commande d'insertion, de mise à jour ou de suppression (INSERT, UPDATE, DELETE). La méthode retourne le nombre de lignes sur lesquelles a porté la commande (nombre de lignes insérées, supprimées, mises à jour). La méthode *ExecuteNonQuery* ne peut pas exécuter une commande de sélection (SELECT). Elle retourne -1 dans ce cas.

© Karim Kalti

`object ExecuteScalar()`	Cette méthode est essentiellement utilisée pour exécuter une commande qui renvoie une valeur scalaire (Exemple : SELECT COUNT* …). D'une manière générale, *ExecuteScalar* renvoie sous la forme d'une valeur de type object la première colonne de la première ligne de la collection d'enregistrements représentant le résultat de la requête. Elle retourne une référence nulle si ce résultat est vide.
`OleDbDataReader ExecuteReader()`	Cette méthode est utilisée lorsque la commande renvoie une collection d'enregistrements (Généralement le résultat d'une requête de sélection). La collection d'enregistrements renvoyée est placée dans un objet de type *OleDbDataReader* (implémentation de l'interface *DataReader*). Cet objet permet de faire le parcours des enregistrements un par un, en lecture seule et dans un seul sens (vers l'avant).

L'objet OleDbParameter

Un objet *Parameter* représente un paramètre d'une commande. Il est à noter qu'un objet *Command* peut comporter une collection de paramètres stockée dans sa propriété *Parameters*.
Pour le fournisseur *OleDb* les paramètres sont créés à l'aide de la classe *OleDbParameter*.

Propriété	Type	Description
`ParameterName`	string	Nom du paramètre.
`Direction`	enum ParameterDirection	- Input : le paramètre est un paramètre d'entrée. - Output : le paramètre est un paramètre de sortie. - InputOutput : le paramètre est un paramètre d'entrée et de sortie. - ReturnValue : le paramètre représente la valeur de retour (généralement d'une procédure stockée).
`OleDbType`	OleDbType	Le type du paramètre.
`IsNullable`	Bool	Indique si le paramètre accepte la valeur null ou non (par défaut false).
`Precision`	Byte	Obtient ou définit le nombre maximal de chiffres utilisés pour représenter la propriété Value. (la valeur par défaut est 0, elle signifie que c'est le fournisseur qui détermine la précision)
`Scale`	Object	Obtient ou définit le nombre de décimales (chiffres après la virgule) appliqué à la résolution de Value. La propriété **Scale** est utilisée uniquement pour les paramètres d'entrée décimaux et numériques. (la valeur par défaut est 0, elle signifie que c'est le fournisseur qui détermine la valeur de cette propriété).
`Size`		Obtient ou définit la taille maximale, en octets, des données figurant dans la colonne. Elle est utilisée essentiellement pour les chaînes de caractères. Pour les types de taille fixe cette valeur est tout simplement ignorée.
`SourceColumn`	string	Obtient ou définit le nom de la colonne source mappée à DataSet et utilisée pour charger et retourner Value.
`SourceVersion`	DataRowVersion	Obtient ou définit la version de la ligne à utiliser lors du chargement de Value.
`Value`	object	Obtient ou définit la valeur du paramètre.

Les constructeurs les plus utilisés sont :

`public OleDbParameter();`	Construit un objet paramètre non initialisé.
`public OleDbParameter(string parameterName, OleDbType oledbType, int size, string sourceColumn);`	Construit un objet paramètre avec initialisation de quelques propriétés. Ce constructeur est souvent utilisé pour créer des paramètres devant passer des nouvelles valeurs à la requête stockée dans un objet Command.
`public OleDbParameter(string parameterName, OleDbType oleDbType, int size, ParameterDirection direction, bool isNullable, byte precision, byte scale, string srcColumn, DataRowVersion srcVersion, object value);`	Construit un objet paramètre avec initialisation de l'ensemble des propriétés. Ce constructeur est souvent utilisé pour créer des paramètres devant passer des anciennes valeurs d'une ligne à la requête stockée dans un objet Command.

Utilisation des espaces réservées dans les requêtes

- Il est possible de faire passer des paramètres au texte SQL d'une requête et ce en utilisant des espaces réservés introduits par le caractère "?". Les espaces réservés ne sont pas acceptés par tous les SGBD (Certains SGBDs demandent des paramètres nommés comme SQL Server par exemple).

- Les espaces réservés d'une requête prennent leurs valeurs à partir de la collection des paramètres de l'objet *Command* qui représente cette requête. La correspondance entre les espaces et les paramètres se fait sur la base de l'ordre d'apparition des "?" dans le texte de la requête et de l'ordre d'ajout des paramètres à la collection *Parameters* de l'objet *Command*. Ainsi, le premier paramètre ajouté à la collection correspond au premier espace réservé, le deuxième paramètre au deuxième espace réservé, etc.

- Au moment de l'exécution de la commande, les espaces réservés de la requête seront remplacés par le contenu des propriétés Value des objets paramètres qui leurs correspondent.

© Karim Kalti

Exemple :

Si on considère le texte de la requête suivante : "INSERT INTO NomTable Values (?, ?, ?)".
L'objet command basé sur cette requête peut faire passer à cette dernière des arguments qui prendront la place des espaces réservés. Ces arguments doivent provenir de la collection *Parameters* de cet objet *Command*.

Récupération des enregistrements en mode connecté

Les requêtes de sélection retournent généralement une collection d'enregistrements. En mode connecté, la gestion de cette collection se fait à l'aide de l'objet *OleDbDataReader*. Cet objet :

- n'est pas créé à l'aide d'un constructeur mais par appel à la méthode *ExecuteReader* de l'objet *Command*.
- Fait un balayage du résultat de la requête de sélection enregistrement par enregistrement, d'une manière séquentielle, du début vers la fin et uniquement dans ce sens.
- Référence à un instant donné un seul enregistrement de la collection d'enregistrement renvoyée par la requête de sélection.
- Ne permet de faire que des accès en lecture aux données récupérées.
- Assure le transfert de chaque enregistrement du serveur de données vers le client juste au moment de la demande de lecture de cet enregistrement.

Espace de noms de OleDbDataReader :

System.Data.OleDB ;

Principales propriétés de OleDbDataReader :

Propriété	Type	Description
`FieldCount`	`int`	Nombre de champs dans l'enregistrement renvoyé (nombre de colonnes dans la ligne).
`Item`	`[]`	Indexeur d'accès aux champs de l'enregistrement courant de l'objet *OleDbDataReader*. Il peut être indexé sur le numéro du champ ou sur le nom du champ.

Principales méthodes de OleDbDataReader :

Méthode	Description
`void Close()`	Ferme l'objet *OleDbDataReader*.
`string  GetName(int n)`	Renvoie le nom du n^{ième} champ.
`int GetOrdinal(string)`	Renvoie la position d'un champ dont le nom est passé comme argument. Une exception est générée si le nom du champ n'est pas connu.
`bool IsDBNull(int N)`	Renvoie *true* si le champ en n^{ième} position contient une valeur nulle.
`bool Read()`	Lit la ligne suivante dans la table représentant la collection d'enregistrements. *Read* renvoie *true* si une ligne a effectivement pu être lue. Elle renvoie *false* quand la fin du résultat du SELECT est atteinte.
`bool NextResult()`	Permet de passer directement à l'enregistrement suivant (ligne suivante de la table). Renvoie *true* si cette ligne existe.
`int GetValues(Object[])`	Remplit le tableau en argument avec le contenu des différentes colonnes d'une ligne (le contenu de l'enregistrement référencé par le *DataReader*). Elle renvoie le nombre de cellules remplies dans ce tableau. Si le tableau contient un nombre n de cellules inférieur au nombre de champs alors seules les n cellules sont remplies.

Exemples de code d'interrogation de base de données

La base de données interrogé dans ces exemples s'appelle *Biblio* et contient une table *Ouvrage*. Cette base est créée avec MS-ACCESS.

Ouvrage	**Champ**	<u>Inventaire</u>	Titre	Auteur	Année d'édition
	Type	Entier	Chaîne	Chaîne	Entier

© Karim Kalti

Exemple 1 : requête renvoyant une valeur (Calcul du nombre d'ouvrages)

Le calcul du nombre d'ouvrages se fait à l'aide d'une requête qui retourne une valeur de type scalaire (entier). C'est la méthode *ExecuteScalar* de l'objet *Command* qui doit alors être utilisée :

```
using System;
using System.Data;
using System.Data.OleDb;
class DataAccess
{
 static int CalculNbOuvrages()
 {
   // Création de la chaîne de connexion
  string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source = C:\ADO\ Biblio.mdb";

   // Création de l'objet de connexion
   OleDbConnection cnn = new OleDbConnection(StrCnn);

   // Ouverture de la connexion
   cnn.Open();

   // Création du texte de la commande
   string StrSql = "SELECT Count(*) FROM Ouvrage";

   // Création de l'objet Command
   OleDbCommand cmd=new OleDbCommand(StrSql,cnn);

   // Exécution de la commande
      int n =(int)cmd.ExecuteScalar();

   // Fermeture de la connexion
   cnn.Close();

   return n;
  }
  static void Main()
   {
       Console.WriteLine("le résultat est "+CalculNbOuvrages());
       Console.ReadLine();
   }
}
```

Exemple 2 : Requête d'insertion dans une table (Ajout d'un ouvrage)

L'insertion d'un enregistrement dans une table se fait généralement à l'aide de la méthode *ExecuteNonQuery* de l'objet *Command*.

```
using System;
using System.Data;
using System.Data.OleDb;

class DataAccess
{
   static int CalculNbOuvrages(){...}
   static void AjoutOuvrage()
   {
     // Création de la chaîne de connexion
     string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source = C:\ADO\Biblio.mdb";

     // Création de l'objet de connexion
     OleDbConnection cnn = new OleDbConnection(StrCnn);

     // Ouverture de la connexion
     cnn.Open();

     // Texte de la commande
     string StrSql = "INSERT INTO Ouvrage VALUES (12,'Programmation C','Didier Le Blanc',
     #10/12/1999#)";

     // Création de l'objet Command
     OleDbCommand cmd=new OleDbCommand(StrSql,cnn);

     // Exécution de la commande
     cmd.ExecuteNonQuery();
```


© Karim Kalti

```
      // Fermeture de la connexion
      cnn.Close();
     }
   static void Main()
   {
      Console.WriteLine("le résultat est "+CalculNbOuvrages());
      AjoutOuvrage();
      Console.WriteLine("le résultat est "+CalculNbOuvrages());
      Console.ReadLine();
   }
}
```

Exemple 3 : Requête de suppression et de mise à jour

Les requêtes de suppression et de mise à jour s'exécutent de la même manière que la requête d'insertion (à l'aide de la méthode *ExecuteNonQuery*). Il suffit tout simplement de changer le texte de la requête.

Exemple de texte d'une commande de mise à jour :

```
string StrSql = "UPDATE Ouvrage SET Titre = 'Programmation Java' WHERE Inventaire = 12";
```

Exemple de texte d'une commande de suppression :

```
string StrSql = "DELETE FROM Ouvrage WHERE Inventaire = 12";
```

Exemple 4 : Requête de sélection

L'exécution d'une requête de sélection se fait à l'aide de la méthode *ExecuteReader* de l'objet *Command*. La collection d'enregistrements renvoyée est placée dans un objet *DataReader*. C'est à travers cet objet que se fait la consultation de ces enregistrements.

```
using System;
using System.Data;
using System.Data.OleDb;
class DataAccess
{
  static void ConsultationOuvrage()
  {
     // Création de la chaîne de connexion
     string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source = C:\ADO\Biblio.mdb";

     // Création de l'objet de connexion
     OleDbConnection cnn = new OleDbConnection(StrCnn);

     // Ouverture de la connexion
     cnn.Open();

     // Texte de la commande
     string StrSql = "SELECT * FROM Ouvrage";

     // Création de l'objet Command
     OleDbCommand cmd=new OleDbCommand(StrSql,cnn);

     // Execution de la commande et récupération des données dans un DataReader
     OleDbDataReader rd = cmd.ExecuteReader();

     // Parcours des enregistrements renvoyés
     string s;
     int i =0;
     if(rd!=null)
     {  while (rd.Read())
         { s = (string)rd["Titre"]; // (string)rd[1]
           i+=1;
          Console.WriteLine("Ouvrage"+i+":"+s);
         }
     }

     // Fermeture de la connexion
     cnn.Close();
}
 static void Main()
 {
  ConsultationOuvrage();
  Console.ReadLine();
 }
}
```


© Karim Kalti

Requête paramétrée

On considère dans ce qui suit la requête de sélection de livres présentée au paragraphe précédent. Supposons que l'on veuille sélectionner seulement les livres dont l'année d'édition est ultérieure à une année saisie par l'utilisateur. Le texte de la requête doit dans ce cas être construit d'une manière dynamique. Deux solutions se présentent à cet égard.

Solution 1 : Construction de la requête par concaténation.

Dans cette solution, les données saisies par l'utilisateur sont converties en chaînes de caractères et introduites dans le texte de la requête par concaténation comme le montre l'exemple suivant :

Exemple :

```
… … … … …
int Année;
string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source =C:\Data\Bilbio.mdb";
OleDbConnection cnn = new OleDbConnection(StrCnn);
cnn.Open();

// Saisie de l'année de référence
Console.WriteLine("Donner l'année de référence");
Année=Int32.Parse(Console.ReadLine());

// Création du texte de la requête par concaténation
string StrSql = "SELECT * FROM Ouvrage WHERE [Année d'édition]>"+Année.ToString();

// Création d'une commande basée sur la requête de sélection
OleDbCommand cmd=new OleDbCommand(StrSql,cnn);
OleDbDataReader rd = cmd.ExecuteReader();
… … … … …
```

Solution 2 : Utilisation des objets *Parameters*

L'ADO.NET propose une alternative permettant d'automatiser le passage de paramètres aux requêtes. Cette alternative se base sur l'utilisation des espaces réservés et des objets *Parameters*. L'exemple suivant illustre l'utilisation de cette technique :

Exemple :

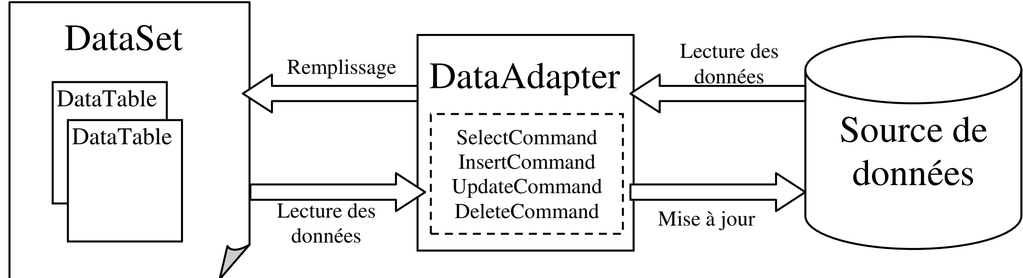
```
… … … … …
int Année;
string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source =C:\Data\Bilbio.mdb";
OleDbConnection cnn = new OleDbConnection(StrCnn);
cnn.Open();

// Saisie de l'année de référence
Console.WriteLine("Donner l'année de référence");
Année=Int32.Parse(Console.ReadLine());

// Texte de la requête avec utilisation d'un espace réservé permettant
// de faire passer une année d'édition comme argument
string StrSql = "SELECT * FROM Ouvrage WHERE [Année d'édition]>?";

// Création d'un objet command basé sur la requête de sélection
OleDbCommand cmd=new OleDbCommand(StrSql,cnn);

// Création d'un objet Parameter qui va servir pour faire passer l'année.
// Il  suffit de passer l'année à la propriété Value de l'objet Parameter
OleDbParameter Par = new OleDbParameter("PDate",Année);

// Ajout du paramètre à la liste des paramètres de la commande
cmd.Parameters.Add(Par);

// Exécution de la commande
OleDbDataReader rd = cmd.ExecuteReader();
… … … … …
```


© Karim Kalti

Présentation du mode déconnecté

Une application déconnectée ne possède aucune connexion permanente à sa source de données. Le modèle de telles applications se base sur deux composants qui sont le *DataAdapter* et le *DataSet*.

Architecture du modèle déconnecté



- Le composant *DataSet* représente une sorte de cache de données. Il stocke en local une partie de la base de données qui correspond au résultat d'une requête de sélection adressée par l'application à la source de données. Ce résultat est stocké dans une table (*DataTable*) du *DataSet*. Il est à noter que le *DataSet* peut comporter plusieurs tables créées chacune par une requête de sélection.
- Le composant *DataAdapter* joue le rôle d'intermédiaire entre la source de données et le *DataSet*. Dans ce cadre ce composant permet de :
  - Récupérer le résultat correspondant à une requête de sélection à partir de la source de données et remplit avec, le *DataSet*.
  - Assurer la mise à jour de la source de données dans le cas où les données locales du *DataSet* ont été modifiées par l'utilisateur.

Remarque :

La copie locale des données stockée par un *DataSet* se trouve sur la machine cliente du serveur de base de données. Dans les applications Windows, il s'agit généralement de la machine de l'utilisateur (cas des clients lourds). Pour les applications Web ce client est généralement le serveur qui stocke les pages d'accès aux données (Serveurs Web). Les internautes à travers leurs navigateurs (clients légers) ne peuvent recevoir en effet que du HTML ou du javaScript.

Les objets d'adaptation de données (DataAdapter)

- Le *DataAdapter* joue le rôle d'un pont entre une source de données et une table du *DataSet*.

- Un *DataAdapter* fait des accès physiques à la source de données chose qui le rend dépendant des fournisseurs de données. Il est spécifié en ADO.NET par l'interface *IDataAdapter*. Cette interface possède plusieurs implémentations proposées par les différents fournisseurs de données. Ces implémentations sont : *OleDbDataAdapter*, *ODBCDataAdapter*, *SqlDataAdapter*, *OracleDataAdapter*.

- Un objet *DataAdapter* possède deux méthodes importantes qui lui permettent d'assurer l'échange de données entre la source et le *DataSet*.
  - La méthode *Fill* : qui permet de remplir une table du *DataSet* à l'aide du résultat d'une requête SELECT adressée à la source.
  - La méthode *Update* : qui permet de mettre à jour la source de données dans le cas où des modifications ont été apportées sur le *DataSet*.

Un *DataAdapter* possède quatre propriétés de type *Command* qui représentent les actions que peut assurer l'adaptateur entre une source de données et un *DataSet*. Ces commandes sont :

© Karim Kalti

Objet Command	Signification
`SelectCommand`	Contient une instruction SQL SELECT permettant de sélectionner des données à partir de la base et de les placer dans un DataSet.
`InsertCommand`	Contient une instruction SQL INSERT permettant d'insérer physiquement dans la base de données les lignes nouvellement ajoutées au DataSet.
`UpdateCommand`	Contient une instruction SQL UPDATE permettant d'appliquer à la base de données les modifications apportées sur les lignes du DataSet.
`DeleteCommand`	Contient une instruction SQL DELETE permettant de supprimer physiquement de la base de données les lignes supprimées dans le DataSet.

- L'appel de la méthode *Fill* du *DataAdapter* engendre l'exécution automatique de l'objet *SelectCommand*. Le *DataSet* est alors rempli avec le résultat de cette commande.
- L'appel de la méthode *Update* engendre l'exécution des objets *InsertCommand*, *UpdateCommand* et *DeleteCommand*.

L'adaptateur OleDbDataAdapter

DataAdapter est un modèle. Les classes réellement instanciables par les programmeurs sont celles qui implémentent ce modèle. Ce paragraphe présente l'implémentation *OleDbDataAdapter* proposée par le fournisseur *OleDb*.

Espace de noms :

System.Data.OleDb

Constructeurs :

Constructeurs	
`OleDbDataAdapter()`	Crée un objet DataAdapter non initialisé.
`OleDbDataAdapter(OleDbCommand)`	Crée un objet DataAdapter à partir d'un objet OleDbCommand (il s'agit d'une commande de sélection. Cette commande est affectée à la propriété SelectCommand de l'adaptateur).

`OleDbDataAdapter(`
`string ClauseSelect,`
`string ChaînedeConnexion`
`)`
	Crée un objet DataAdapter à partir d'une requête de sélection qui sera utilisée par Fill, et d'une chaîne de connexion indiquant la source de données. Dans ce cas, il n'est pas nécessaire d'établir la connexion à l'aide d'un objet OleDbConnection. Ce dernier est automatiquement créé.

`OleDbDataAdapter(`
`string ClauseSelect,`
`OleDbConnection Cnn`
`)`
	Même effet que le constructeur précédent. Mais ici l'objet de connexion doit être explicitement créé et passé au constructeur.

Principales propriétés :

Les principaux attributs sont : *OleDbSelectCommand, OleDbInsertCommand, OleDbUpdateCommand, OleDbDeleteCommand*.dont les interfaces viennent d'être présentées au paragraphe précédent.

Principales méthodes :

Méthode	Signification
`int Fill(DataSet)`	Remplit un DataSet passé comme argument avec le résultat de la requête de sélection passée au constructeur. Fill renvoie le nombre de lignes reçues du serveur. Une table est ainsi ajoutée au DataSet. La table porte par défaut le nom "Table". Aucune table n'est cependant ajoutée au DataSet si ce nombre de lignes est égal à zéro.

`int Fill(`
`DataSet DS,`
`string NomTable`
`)`
	Même effet que la version précédente mais en donnant la possibilité de spécifier explicitement un nom à la table ajoutée au DataSet. Si une table qui porte le même nom existe déjà dans le DataSet alors la méthode Fill remplit cette table et aucune nouvelle table n'est créée.

`int Fill(`
`DataSet DS,`
`int StartRecord,`
`int NbRecords;`
`string NomTable`
`)`
	Comme la fonction précédente mais en remplissant le DataSet avec NbRecords lignes à partir de la ligne numéro StartRecord. (Les lignes provenant du SELECT).

© Karim Kalti

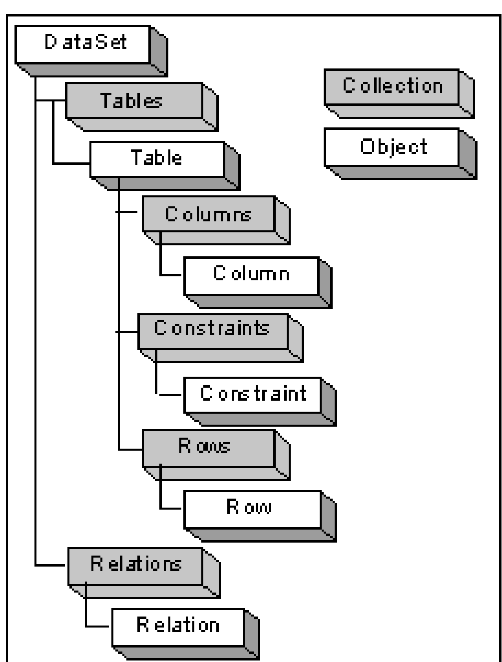
`int Fill(DataTable)`	Insère directement le résultat du SELECT dans un objet table en mémoire.
`int Update(DataSet DS)`	Engendre l'exécution des commandes INSERT, UPDATE et DELETE du DataAdapter afin de répercuter dans la source de données les mises à jour effectuées dans le DataSet. Update retourne le nombre de lignes mises à jour. Dans le cas où la table de la source n'est pas trouvée une exception de type SystemException est levée. Dans le cas de l'echec d'une commande (insert,…) une exception de type DbConcurrencyException est levée.
`int Update(DataTable DT)`	Engendre l'exécution des commandes INSERT, UPDATE et DELETE du DataAdapter afin de répercuter dans la source de données les mises à jour effectuées dans le DataTable. Update retourne le nombre de lignes mises à jour. Dans le cas où la table de la source n'est pas trouvée une exception de type SystemException est levée. Dans le cas de l'echec d'une commande (insert,…) une exception de type DbConcurrencyException est levée.

L'objet DataSet

- Un DataSet représente une copie locale d'une partie de la base de données. Les données de cette copie locale peuvent être consultées et modifiées sans avoir besoin à accéder au serveur de données.

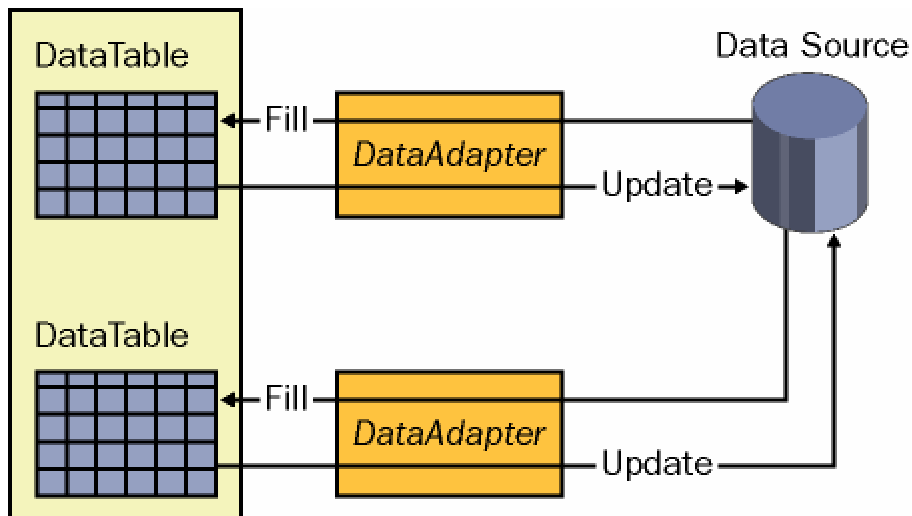
Modèle objet d'un DataSet

- Le modèle objet d'un DataSet est donné par la figure suivante :



- Un objet DataSet comporte généralement une collection de tables (objets *DataTable*) et une collection de relations (objets *DataRelation*).
- Chaque table comporte une collection de colonnes (objets *DataColumn*), une collection de lignes (objets *DataRow*) et une collection de contraintes (objets *Constraint*).
- Les tables peuvent être créées et ajoutées au *DataSet* d'une manière explicite ou par appel aux méthodes *Fill* de plusieurs adaptateurs de données. Chaque adaptateur de données s'occupe généralement de la gestion (remplissage et mise à jour) d'une table du *DataSet*.

© Karim Kalti



- Un objet *DataSet* est complètement déconnecté de la source de données. Il est indépendant des différents fournisseurs de données (implémentation unique pour les différents fournisseurs OleDb, SQL Server, Oracle,…).
- En .NET, les données d'un *DataSet* sont écrites en XML et le schéma est écrit en XSD (XML Schema Definition Language). Un *DataSet* peut ainsi importer et exporter d'une manière native les données au format XML. Il peut également être utilisé pour échanger des données entre différentes sources de données.

Espace de noms :

System.Data

Constructeurs :

Constructeur	Signification
`DataSet()`	Crée un objet DataSet.
`DataSet(string)`	Crée un objet DataSet en lui donnant un nom.

Principales propriétés publiques :

Nom	Type	Signification
`DataSetName`	`string`	Nom donné au DataSet non initialisé.
`EnforceConstraints`	`bool`	Indique si les contraintes doivent être vérifiées ou non lors des opérations de mise à jour effectuées dans le DataSet. (true signifie qu'il y a vérification, c'est la valeur par défaut pour cette propriété).
`Relations`	`DataRelationCollection`	Collection de relations (objets de type DataRelation)
`Tables`	`DataTableCollection`	Collection de tables (objets de type DataTable)

Principales méthodes :

Méthode	Signification
`void Clear()`	Efface le contenu du DataSet en supprimant toutes les lignes des tables.
`DataSet Clone()`	Copie la structure du DataSet (tables, relations, contraintes) sans copier les données.
`DataSet Copy()`	Copie à la fois la structure et les données du DataSet.
`string GetXml()`	Retourne la représentation XML des données stockées dans le DataSet.
`string GetXmlSchema()`	Retourne le schéma XSD de la représentation des données du DataSet.
`void Merge(DataSet)`	Fusionne le DataSet passé en argument et son schéma avec le DataSet courant.
`void Merge(DataTable)`	Fusionne la table passée en argument et son schéma dans le DataSet courant.

Exemple 1 :

On considère la base de données Biblio comportant la table Ouvrage donnée comme suit :

Ouvrage	**Champ**	Inventaire	Titre	Année d'édition
	Type	Entier	Chaîne	Entier

© Karim Kalti

L'exemple suivant montre la création d'un *DataSet* qui récupère en local la liste des ouvrages de la base Biblio. Cette liste est placée dans un objet *DataTable* nommé *Ouvrage*.

```
// Création de la chaîne de connexion
string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source = C:\Data\Biblio.mdb";
// Création de l'objet de connexion
OleDbConnection Cnn = new OleDbConnection(StrCnn);
// Ouverture de la connexion
Cnn.Open();
// Texte de la requête qui sera stocké dans la propriété SelectCommand de l'adaptateur
string StrSelect = "SELECT * FROM Ouvrage";
// Création de l'adaptateur de données
OleDbDataAdapter DA = new OleDbDataAdapter(StrSelect,Cnn);
// Création du DataSet
DataSet DS = new DataSet();
// Remplissage du DataSet
DA.Fill(DS, "Ouvrage");
… … …
… … …
```

L'objet DataTable

- L'objet *DataTable* est une représentation en mémoire locale (côté client) d'une table de base de données. Une table dans un *DataSet* comporte un schéma et des données.
- Le schéma est défini par les colonnes de la table (collection d'objets de type *DataColumn*) et par les contraintes appliquées aux colonnes. Ces contraintes concernent la spécification de la clé primaire (objet *UniqueConstraint*) et de la clé étrangère (objet *ForeignKeyConstraint*).
- Les données d'une table sont stockées dans des lignes (collection d'objets de type *DataRow*).

Espace de noms :

System.Data

Constructeurs :

`DataTable()`	Crée une table sans lui donner de nom.
`DataTable(string)`	Crée une table en lui donnant un nom.

Principales propriétés publiques :

Nom	Type	Signification
`TableName`	`string`	Nom donné à la table.
`DataSet`	`DataSet`	En lecture seule, cette propriété permet de récupérer le DataSet auquel appartient la table.
`ChildRelations`	`DataRelationCollection`	En lecture seule. Cette propriété donne la collection de relations (objets DataRelation) de type enfant de la table.
`ParentRelations`	`DataRelationCollection`	En lecture seule. Cette propriété donne la collection de relations (objets DataRelation) de type parent de la table.
`Columns`	`DataColumnsCollection`	En lecture seule. Cette propriété donne la collection de colonnes (objets de type DataColumn) de la table.
`Rows`	`DataRowCollection`	En lecture seule. Cette propriété donne la collection des lignes (objets de type DataRow) de la table.
`Constraints`	`Constraintcollection`	En lecture seule. Cette propriété donne la collection des contraintes associées à la table.
`PrimaryKey`	`DataColumn[]`	En lecture et écriture. Cette propriété contient la liste des colonnes qui composent la clé primaire de la table.

Principales méthodes :

Méthode	Signification
`void Clear()`	Efface le contenu d'une table (les lignes).
`DataTable Clone()`	Clone la structure d'une table (schéma et contraintes).
`DataTable Copy()`	Copie à la fois la structure et les données d'une table.
`void ImportRow(DataRow dr)`	Ajoute la ligne dr à la table.
`DataRow NewRow()`	Créer une nouvelle ligne ayant le même schéma que la table.

© Karim Kalti

L'objet DataTableCollection

Cet objet représente une collection de tables (*DataTable*).

Principales propriétés :

Nom	Type	Signification
`Count`	`int`	Retourne le nombre de tables dans la collection.
`Item`	`DataTable`	C'est un indexeur qui permet d'accéder à une table particulière de la collection. La table peut être indexée par un numéro (ordre d'ajout de la table dans la collection) ou par son nom.

Principales méthodes :

Méthode	Signification
`DataTable Add(string)`	Ajoute une nouvelle table à la collection. La table possède comme nom la chaîne passée en argument. La méthode retourne une référence sur la table ajoutée.
`void Add(DataTable)`	Ajoute la table passée en argument à la collection.
`void AddRange(DataTable[])`	Ajoute un tableau de tables à la collection. (à la fin de la collection)
`void Clear()`	Supprime toutes les tables de la collection.
`bool Contains(string name)`	Vérifie si la collection contient la table dont le nom est passé en argument.
`int IndexOf(string name)` `int IndexOf(DataTable)`	Renvoi la position de l'indexe (position ordinale dans la collection) de la table qui est passée ou dont le nom est passé en argument.
`void Remove(string name)` `void Remove(DataTable)`	Supprime de la collection la table qui est passée ou dont le nom est passé en argument.
`void RemoveAt(int index)`	Supprime de la collection la table dont l'indexe est passé en argument.

<u>Exemple 1 :</u>

Cet exemple montre les différentes possibilités d'accès à la table *Ouvrage* du *DataSet* DS créé dans l'exemple précédent.

```
DS.Tables[0];
```

Accès à l'aide de l'indice de la table dans le *DataSet*. (Les indices commencent à partir de 0. Ils sont affectés aux tables suivant l'ordre d'ajout de ces dernières *au DataSet*.

```
DS.Tables["Ouvrage"];
```

Accès à l'aide d'un indexeur de la collection *Tables* qui prend un indice de type chaîne. Cet indice représente le nom de la table à accéder.

<u>Exemple 2:</u>

Il est possible de compter le nombre de tables du *DataSet* et ce à l'aide de la propriété *Count* de la collection *Tables* du *DataSet*.

```
int n = DS.Tables.Count;
```

Remarque :

Les objets *DataColumnCollection* et *DataRowCollection* possèdent les mêmes caractéristiques que *DataTableCollection*. (Ce sont toutes des collections). On trouve ainsi presque les mêmes propriétés et méthodes mais c'est seulement le type de l'objet manipulé par la collection qui change (respectivement *DataColumn* et *DataRow* au lieu de *DataTable*).

L'objet DataColumn

L'objet *DataColumn* représente une colonne d'une table en mémoire.

Espace de noms :

System.Data

Constructeurs :

`DataColumn()`	Crée une colonne sans lui donner de nom.
`DataColumn(string N)`	Crée une colonne tout en lui attribuant un nom.
`DataColumn(string N, Type T)`	Crée une colonne tout en lui attribuant un nom et un type. Type est une classe du .NET de l'espace de noms System qui permet de stocker des types de données.

© Karim Kalti

Principales propriétés publiques :

Nom	Type	Signification
AllowDbNull	bool	Indique si des valeurs nulles peuvent être introduites dans la colonne.
AutoIncrement	bool	Indique si le contenu de cette colonne doit être automatiquement incrémenté lors de chaque ajout.
AutoIncrementSeed	int	Valeur de début d'un champ autoincrémenté.
AutoIncrementStep	int	Valeur d'incrémentation (par défaut 1).
Caption	string	Libellé de la colonne. Si ce champ n'est pas explicitement spécifié alors il prend par défaut la valeur de ColumnName.
ColumnName	string	Nom du champ correspondant à la colonne.
DataType	Type	Type du champ.
DefaultValue	Object	Valeur par défaut lors d'un ajout
Ordinal	int	Position de la colonne dans la collection des colonnes de la table.
ReadOnly	bool	Indique si le contenu de la colonne est en lecture seule.
Unique	bool	Indique si la valeur de chaque ligne de la colonne doit être unique.

Les quelques méthodes publiques de *DataColumn* sont rarement utilisées.

<u>Exemple 1 :</u>

L'accès à une colonne d'une table peut se faire de différentes manières comme le montrent les quatre instructions équivalentes suivantes :

```
DS.Tables["Ouvrage"].Columns[0];
DS.Tables["Ouvrage"].Columns["Inventaire"];
DS.Tables[0].Columns[0];
DS.Tables[0].Columns["Inventaire"];
```

<u>Exemple 2 : Détermination dynamique de la structure d'un DataSet</u>

```
// Détermination dynamique du nombre des tables et de leurs noms
Console.WriteLine("Nombre de tables du DataSet : "+DS.Tables.Count);
// Liste des noms des tables
for(int i=0;i<DS.Tables.Count;i++)
   Console.WriteLine(DS.Tables[i].TableName+" ("+i+")");
// Détermination dynamique des structures des tables du DataSet
Console.WriteLine("\n\n ////// Structures des tables du DataSet //////");
foreach(DataTable dt in DS.Tables)
   {
     Console.WriteLine("**** Table "+dt.TableName+" ****");
     foreach(DataColumn dc in dt.Columns)
        Console.WriteLine(dc.ColumnName+" ("+dc.DataType.Name+")");
   }
```

L'objet DataRow

L'objet *DataRow* représente une ligne d'une table.

Espace de noms :

System.Data

Constructeurs :

Pas de constructeurs explicites (constructeur par défaut seulement). Généralement c'est la méthode *NewRow* de la classe *DataTable* qui est utilisée pour construire un *DataRow*.

Principales propriétés :

Nom	Type	Signification
Item	Indexeur de la forme Object this[ ]	Indexeur donnant accès au contenu de chaque colonne de la ligne. Dans les crochets de l'indexeur il est possible de spécifier le numéro de colonne, le nom de la colonne ou encore un objet DataColumn.
ItemArray	object[]	Tableau des contenus des différentes colonnes de la ligne (contenu de la ligne).
RowState	enum DataRowState	Etat de la ligne. RowState peut prendre l'une des valeurs suivantes de l'énumération DataRowState : (Detached, Unchanged, Added, Deleted, Modified).
Table	DataTable	Table à laquelle appartient la ligne.

© Karim Kalti

L'état d'une ligne

L'état d'une ligne est déterminé par la propriété *RowState* de la classe *DataRow*. Les valeurs possibles de cette propriété sont celles de l'énumération *DataRowState* :

Valeur de DataRowState	Signification
`Unchanged`	Aucune modification n'a été effectuée sur la ligne depuis l'appel à AcceptChanges ou depuis la création de la ligne à l'aide de la méthode Fill du DataAdapter.
`Added`	La ligne a été ajoutée à la table mais la méthode AcceptChanges n'a pas été appelée.
`Modified`	Certains éléments de la ligne ont été modifiés.
`Deleted`	La ligne a été supprimée de la table et AcceptChanges n'a pas été encore appelée.
`Detached`	Cette valeur signifie que la ligne a été créée mais qu'elle ne fait pas partie d'un DataRowCollection. Le RowState d'une ligne créée prend la valeur Detached. Une fois la ligne ajoutée à un DataRowCollection à l'aide de la méthode Add, la propriété RowState prend la valeur Added. La valeur Detached est également définie pour une ligne qui a été supprimée d'un DataRowCollection à l'aide de la méthode Remove ou par la méthode Delete suivie de l'appel à la méthode AcceptChanges.

Le *DataAdapter* se base sur la propriété *RowState* pour décider de la commande à exécuter pour chaque ligne lors de l'appel de la méthode *Update* : (*InsertCommand* pour *Added*, *UpdateCommand* pour *Modified* et *DeleteCommand* pour *Deleted*).

La version d'une ligne

La version d'une ligne peut prendre l'une des valeurs de l'énumération *DataRowVersion*.

DataRowVersion	Signification
`Current`	C'est la version qui contient les valeurs actuelles de la ligne. Cette version de ligne n'existe pas pour les lignes dont le RowState est Deleted.
`Default`	C'est la version par défaut d'une ligne particulière. La version par défaut d'une ligne Added, Modified ou Unchanged est Current. La version par défaut d'une ligne Deleted est Original. La version par défaut d'une ligne Detached est Proposed.
`Original`	C'est la version qui contient les valeurs d'origine de la ligne. Cette version de ligne n'existe pas pour les lignes ayant un RowState Added.
`Proposed`	C'est la version qui contient les valeurs proposées d'une ligne. Cette version existe pendant une opération de modification sur une ligne ou pour une ligne qui ne fait pas partie d'un DataRowCollection.

Il est possible de savoir si une version particulière d'une ligne existe et ce en utilisant la méthode *HasVersion*. Les versions des lignes sont essentiellement utilisées durant les opérations de mise à jour de données à partir du DataSet vers la base de données.

Principales méthodes :

Méthode	Signification
`void AcceptChanges()`	La méthode *AcceptChanges* permet de valider tous les changements effectués sur une ligne depuis le dernier appel à *AcceptChanges.* Si le RowState d'une ligne est Added ou Modified alors il devient Unchanged. Si le RowState d'une ligne est Deleted alors la ligne sera supprimée de la table et son état devient Detached.
`bool HasVersion(DataRowVersion)`	Retourne une valeur indiquant si la version passée en argument existe pour la ligne courante.

- Sur le plan pratique l'appel explicite de *AcceptChanges* est rare. Cet appel est souvent implicite. Il est effectué surtout par la méthode *Update* du *DataAdapter*. Ceci permet de ne faire que la mise à jour des nouvelles modifications si un autre appel à *Update* est effectué. L'appel implicite de *AcceptChanges* après l'appel de la méthode *Fill* du *DataAdapter* est déterminé quant à lui par la propriété *AcceptChangesDuringFill* de la classe *DataAdapter* (la valeur par défaut est *true*. Elle signifie qu'il y a appel).

- La méthode *AcceptChanges* est membre non seulement de la classe *DataRow* mais également des classes *DataSet* et *DataTable*.
  - Pour *DataRow* son effet touche la ligne.
  - Pour *DataTable* son effet touche toutes les lignes d'une table.
  - Pour *DataSet* son effet touche toutes les lignes de toutes les tables du *DataSet*.

© Karim Kalti

***Exemple 1 :* Accès à une ligne**

L'accès à la n^{ième} ligne d'une table se fait comme suit :

```
DS.Tables["Ouvrage"].Rows[n-1];
```

Les indices commencent toujours à partir de 0. Si la n ^{ième} ligne n'existe pas alors une exception sera levée.

L'accès au m^{ième} champ de la n^{ième} ligne se fait comme suit :

```
// Accès à l'aide des indices
DS.Tables["Ouvrage"].Rows[n-1][m-1];
// Accès à travers le nom du champ.
DS.Tables["Ouvrage"].Rows[n-1]["Inventaire"];
```

***Exemple 2*: Affichage du contenu d'une table**

```
foreach(DataRow rw in DS.Tables["Ouvrage"].Rows)
  {
     Console.WriteLine("************************************");
     Console.WriteLine("Inventaire      : "+rw["Inventaire"]);
     Console.WriteLine("Titre           : "+rw["Titre"]);
     Console.WriteLine("Année d'édition : "+rw["Année d'édition"]);
  }
```

***Exemple 3:* une autre version de l'exemple précédent.**

```
for(int i=0;i<DS.Tables["Ouvrage"].Rows.Count;i++)
{
  Console.WriteLine("************************************");
  Console.WriteLine("Inventaire      : "+DS.Tables["Ouvrage"].Rows[i]["Inventaire"]);
  Console.WriteLine("Titre           : "+DS.Tables["Ouvrage"].Rows[i]["Titre"]);
  Console.WriteLine("Année d'édition : "+DS.Tables["Ouvrage"].Rows[i]["Année d'édition"]);
}
```

***Exemple 4 :* Code complet**

```
class DisconnectedMode
{
       private string StrCnn; // La chaîne de connexion
       private OleDbConnection Cnn; // Objet de Connexion
       private string StrSelect; // Texte de la commande Select du DataAdapter
       private OleDbDataAdapter DA; // Adaptateur de données
       private DataSet DS;

       public DisconnectedMode(string strcnn)
       {
              StrCnn =strcnn;
              Cnn = new OleDbConnection(StrCnn);
              Cnn.Open();
              StrSelect = "SELECT * FROM Ouvrage";
              DA = new OleDbDataAdapter(StrSelect,Cnn);
              DS = new DataSet();
              // Remplissage du DataSet
              DA.Fill(DS, "Ouvrage");
       }
       public void Disconnect()
       {
              //Fermeture de la connexion
              Cnn.Close();
       }
       public void ShowStructure()
       {
              // Détermination dynamique du nombre des tables et de leurs noms
              Console.WriteLine("Nombre de tables du DataSet : "+DS.Tables.Count);
              for(int i=0;i<DS.Tables.Count;i++)
                     Console.WriteLine(DS.Tables[i].TableName+" ("+i+")");
              // Détermination dynamique des structures des tables
              Console.WriteLine("\n/////// Structures des tables du DataSet ///////\n");


              foreach(DataTable dt in DS.Tables)
              {
                     Console.WriteLine("**** Table "+dt.TableName+" ****");
```


© Karim Kalti

```
                        foreach(DataColumn dc in dt.Columns)
                                Console.WriteLine(dc.ColumnName+" ("+dc.DataType.Name+")");
                }
        }
        public void ShowContent()
        {
                foreach(DataRow rw in DS.Tables["Ouvrage"].Rows)
                {
                        Console.WriteLine("************************************");
                        Console.WriteLine("Inventaire      : "+rw["Inventaire"]);
                        Console.WriteLine("Titre           : "+rw["Titre"]);
                        Console.WriteLine("Année d'édition : "+rw["Année d'édition"]);
                }
        }
}
class Prog
{
        public static void Main()
        {
                DisconnectedMode ins = new DisconnectedMode("C:\\Data\\Biblio.mdb");
                ins.ShowStructure();
                ins.ShowContent();
                ins.Disconnect();
        }
}
```

L'objet CommandBuilder

- L'objet *CommandBuilder* est utilisé pour générer d'une manière automatique les objets *DeleteCommand*, *InsertCommand* et *UpdateCommand* du *DataAdapter* nécessaires pour effectuer la mise à jour des données à partir du *DataSet*. L'objet *CommandBuilder* se base sur le texte de l'objet *SelectCommand* du *DataAdapter* pour déduire les informations nécessaires à cette génération comme le nom de la table, la liste des champs récupérés, les contraintes, etc.

- L'objet *CommandBuilder* est essentiellement utilisé pour générer des commandes relatives à des tables simples. (Généralement pour les tables du *DataSet* issues d'une seule table de la source de données. Le *CommandBuilder* ne convient pas aux tables simples créées par un SELECT qui ne retourne pas parmi la liste des champs le champ clé. Il ne convient pas également aux tables qui résultent d'une opération de jointure).

- L'objet *CommandBuilder* est dépendant du fournisseur de données. L'implémentation proposée par le fournisseur *OleDb* est *OleDbCommandBuilder*.

Espace de noms :

System.Data.OleDb

Quelques constructeurs :

`public OleDbCommandBuilder();`	Crée un *OleDbCommandBuilder* non initialisé.
`public OleDbCommandBuilder(OleDbDataAdapter);`	Crée un *OleDbCommandBuilder* et l'associe à un adaptateur de données.

Principales propriétés :

Nom	Type	Signification
`DataAdapter`	`OleDbDataAdapter`	Obtient ou définit un objet OleDbDataAdapter pour lequel les instructions SQL sont automatiquement générées.
`QuotePrefix`	`string`	Obtient ou définit le ou les caractères de début à utiliser lors de la spécification d'objets de base de données (par exemple, des tables ou colonnes) dont les noms contiennent des caractères tels que des espaces ou des jetons réservés.
`QuoteSuffix`	`string`	Obtient ou définit le ou les caractères de fin à utiliser lors de la spécification d'objets de base de données (par exemple, des tables ou colonnes) dont les noms contiennent des caractères tels que des espaces ou des jetons réservés.

© Karim Kalti

Principales méthodes :

Méthode	Signification
`public void RefreshSchema();`	Actualise les informations de schéma de la base de données utilisées pour générer des instructions INSERT, UPDATE ou DELETE. Cette méthode doit être appelée si l'adaptateur de données change sa commande SELECT après la construction du CommandBuilder. Cet appel va engendrer alors l'actualisation des informations de schéma du CommandBuilder et la construction de commandes valides.
`OleDbCommand GetDeleteCommand();`	Obtient l'objet OleDbCommand généré automatiquement et requis pour effectuer des suppressions au niveau de la source de données.
`OleDbCommand GetInsertCommand();`	Obtient l'objet OleDbCommand généré automatiquement et requis pour effectuer des insertions au niveau de la source de données.
`OleDbCommand GetUpdateCommand();`	Obtient l'objet OleDbCommand généré automatiquement et requis pour effectuer des mises à jour au niveau de la source de données.

La mise à jour des données à partir d'un DataSet

- La mise à jour des données d'un *DataSet* passe par l'utilisation des objets *DeleteCommand*, *InsertCommand* et *UpdateCommand* du *DataAdapter*. Ces commandes sont automatiquement exécutées suite à l'appel de la méthode *Update* du *DataAdapter*.
- Il existe globalement trois choix pour la génération des commandes de mise à jour du *DataAdapter* :
  - La génération à l'aide des assistants.
  - La génération par le développeur.
  - La génération à l'aide de l'objet *CommandBuilder*.

La génération à l'aide des assistants

Voir annexe.

La génération par le développeur

La génération par le développeur d'une commande de mise à jour doit suivre les 3 étapes suivantes :

- La création de la chaîne qui contient le texte de la requête. Si la requête est paramétrée alors le texte doit comporter des espaces réservés.
- La création d'un objet *Command* basé sur cette requête.
- La création d'une collection d'objets de type *Parameter*. Le nombre de ces objets doit correspondre au nombre des espaces réservés de la requête. Une fois initialisés, ces objets doivent être ajoutés à la collection *Parameters* de l'objet *Command* créé dans l'étape précédente.

Ces trois étapes doivent être effectuées pour chacune des commandes *DeleteCommand*, *InsertCommand* et *UpdateCommand* du *DataAdapter* comme le montre l'exemple suivant :

***Exemple* : Mise à jour des données à l'aide d'un DataAdapter**

Cet exemple montre la mise à jour des modifications apportées aux données de la table *Ouvrage* située dans le *DataSet* vers la table qui lui correspond dans la source de données.

Le tableau suivant donne l'état de la table *Ouvrage* avant la mise à jour des données :

Ouvrage		
Inventaire	**Titre**	**Année d'édition**
1	Programmation C	1996
2	Les réseaux	2004
3	Internet	2000
4	Les bases de données	1998

© Karim Kalti

```
using System;
using System.Data;
using System.Data.OleDb;

class DisconnectedMode
{
  private string StrCnn; // La chaîne de connexion
  private OleDbConnection Cnn; // Objet de Connexion
  private string StrSelect; // Texte de la commande Select du DataAdapter
  private OleDbDataAdapter DA; // Adaptateur de données
  private DataSet DS;
  private OleDbCommand DelCmd;
  private OleDbCommand InsCmd;
  private OleDbCommand UpdateCmd;

  public DisconnectedMode()
  {StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source = C:\Data\Biblio.mdb";
   Cnn = new OleDbConnection(StrCnn);
   Cnn.Open();
   StrSelect = "SELECT * FROM Ouvrage";
   DA = new OleDbDataAdapter(StrSelect,Cnn);
   DS = new DataSet();
   // Remplissage du DataSet
   DA.Fill(DS, "Ouvrage");
   }
   public void Disconnecte()
   { Cnn.Close();}

   public void ShowContent()
   {
     foreach(DataRow rw in DS.Tables["Ouvrage"].Rows)
     {
         Console.WriteLine("***********************************");
         Console.WriteLine("Inventaire      : "+rw["Inventaire"]);
         Console.WriteLine("Titre           : "+rw["Titre"]);
         Console.WriteLine("Année d'édition : "+rw["Année d'édition"]);
     }
   }

   public void UpdateDataSet()
   { // Code d'ajout d'une ligne par programmation
     DataRow dr = DS.Tables["Ouvrage"].NewRow();
     dr["Inventaire"]=5;
     dr["Titre"]= "UML";
     dr["Année d'édition"]=2004;
     DS.Tables["Ouvrage"].Rows.Add(dr);

     // Code de Modification d'une ligne
     DS.Tables["Ouvrage"].Rows[0]["Titre"]="Programmation C++";
     DS.Tables["Ouvrage"].Rows[0]["Année d'édition"]=2003;

     // Code de suppression d'une ligne
     DS.Tables["Ouvrage"].Rows[2].Delete();
   }
   public void UpdateDataSource()
   {
     // Création de la commande de suppression
     string StrDelete = "DELETE FROM Ouvrage WHERE Inventaire = ?";
     DelCmd = new OleDbCommand(StrDelete, Cnn);
     OleDbParameter Param = new OleDbParameter();
     Param.ParameterName="OriginInventaire";
     Param.OleDbType = OleDbType.Integer;
     Param.Direction = ParameterDirection.Input;
     Param.IsNullable = false;
     Param.SourceColumn="Inventaire";
     Param.SourceVersion=DataRowVersion.Original;
     DelCmd.Parameters.Add(Param);

     // Création de la commande d'insertion
     string StrInsert = "INSERT INTO Ouvrage VALUES (?,?,?)";
     InsCmd = new OleDbCommand(StrInsert, Cnn);

     OleDbParameter ParamInventaire = new  OleDbParameter("ParInv",OleDbType.Integer,0,
     ParameterDirection.Input,false,(byte)0,(byte)0,"Inventaire",DataRowVersion.Current,null);

     OleDbParameter ParamTitre = new OleDbParameter("ParTitre",OleDbType.VarWChar,50,
     ParameterDirection.Input,false,(byte)0,(byte)0,"Titre",DataRowVersion.Current,null);
```


© Karim Kalti

```
OleDbParameter ParamEdition = new OleDbParameter("ParEdition",OleDbType.Integer,0,
ParameterDirection.Input,false,(byte)0,(byte)0,"Année d'édition",DataRowVersion.Current,
null);

InsCmd.Parameters.Add(ParamInventaire);
InsCmd.Parameters.Add(ParamTitre);
InsCmd.Parameters.Add(ParamEdition);

// Création de la commande de modification
string StrUpdate = "Update Ouvrage SET Inventaire=?, Titre=?, [Année d'édition]=? WHERE
Inventaire=? AND (Titre = ?) AND ([Année d'édition]= ? OR ? IS NULL AND [Année d'édition]
IS NULL)";

UpdateCmd = new OleDbCommand(StrUpdate, Cnn);
UpdateCmd.Parameters.Add(new OleDbParameter("NouvInventaire",OleDbType.Integer,0,
"Inventaire"));

UpdateCmd.Parameters.Add(new OleDbParameter("NouvTitre", OleDbType.VarWChar,50,"Titre"));

UpdateCmd.Parameters.Add(new OleDbParameter("NouvEdition", OleDbType.Integer,0,"Année
d'édition"));

UpdateCmd.Parameters.Add(new OleDbParameter("OriginalInv", OleDbType.Integer,0,
ParameterDirection.Input,false,(byte)0,(byte)0,"Inventaire",DataRowVersion.Original,
null));

UpdateCmd.Parameters.Add(new OleDbParameter("OriginalTitre", OleDbType.VarWChar,50,
ParameterDirection.Input, false,(byte)0,(byte)0,"Titre",DataRowVersion.Original,null));

UpdateCmd.Parameters.Add(new OleDbParameter("OriginalEdition1", OleDbType.Integer,0,
ParameterDirection.Input,false,(byte)0,(byte)0,"Année d'édition",DataRowVersion.Original,
null));

UpdateCmd.Parameters.Add(new OleDbParameter("OriginalEdition2", OleDbType.Integer,0,
ParameterDirection.Input,false,(byte)0,(byte)0,"Année d'édition",DataRowVersion.Original,
null));

// Ajout des commandes à l'adaptateur
DA.DeleteCommand=DelCmd;
DA.InsertCommand=InsCmd;
DA.UpdateCommand=UpdateCmd;
DA.Update(DS.Tables["Ouvrage"]);

}

}

class Prog
{
    public static void Main()
    {
        DisconnectedMode ins = new DisconnectedMode();
        ins.UpdateDataSet();
        ins.UpdateDataSource();
        ins.ShowContent();
    }
}
```

Quelques précisions :

- Il n'est pas nécessaire de définir la précision pour les types numériques qui ne sont pas du type *OleDbType.Decimal*.
- Si la version d'une ligne n'est pas explicitement définie alors elle prend par défaut la valeur *Current*.
- Dans le mode déconnecté, la propriété Value d'un paramètre d'un objet commande d'un *DataAdapter* est déterminée d'une manière automatique en se basant sur les propriétés *SourceVersion* et *SourceColumn*.

Commentaire sur la commande de suppression :
A chaque fois que le *DataAdapter* trouve une ligne dont l'état est *Deleted*, il exécute la commande *DeleteCommand* du *DataAdapter*. Cette commande récupère la valeur originale de la ligne pour valoriser le paramètre de la requête qu'elle encapsule puis supprime physiquement cette ligne à partir de la source de données. (Il est à noter qu'une ligne supprimée dans le *DataSet* ne possède pas de version courrante).

© Karim Kalti

Commentaire sur la commande d'insertion :
A chaque fois que le *DataAdapter* trouve une ligne dont l'état est *Added*, il exécute la commande *InsertCommand* du *DataAdapter*. Cette commande récupère la version courante (la version par défaut) de la ligne dans le *DataSet* et ce pour valoriser les paramètres de la requête puis insère physiquement cette ligne dans la source de données.

Commentaire sur la commande de modification :

- A chaque fois que le *DataAdapter* trouve une ligne dont l'état est *Modified*, il exécute la commande *UpdateCommand* du *DataAdapter*. Cette commande récupère la valeur courante (version par défaut) de la ligne pour valoriser la première partie des paramètres de la requête (les paramètres avant le WHERE). Les conditions introduites par le WHERE permettent de vérifier si la ligne n'a pas été modifiée dans la source de données (par un autre utilisateur) depuis son chargement en local dans le *DataSet* (car dans le mode déconnecté de tels changements ne peuvent pas être immédiatement visibles dans le *DataSet*). Les paramètres utilisés par ces conditions prennent leurs valeurs de la version originale de la ligne (avant modification) et non de la version courante comme c'est le cas des paramètres de la première partie de la requête.

- L'expression Champ=NULL est toujours fausse (en BD, NULL n'est pas une valeur mais signifie que le champ n'a pas de valeur). Ainsi à chaque fois qu'un champ peut être NULL, le critère doit être de la forme Champ = ? OR ? IS NULL AND Champ IS NULL ce qui veut dire que le champ est égal au paramètre ou le champ et le paramètre sont tous les deux NULL. Cette condition est obligatoire pour les champs qui peuvent accepter la valeur NULL (*IsNullable = true*).

La génération à l'aide de l'objet CommandBuilder

Le code de mise à jour des données en utilisant un objet CommandBuilder est assez simple. Il se présente comme suit :

```
string StrCnn = @"Provider = Microsoft.Jet.OLEDB.4.0; Data Source = C:\Data\Biblio.mdb";
OleDbConnection Cnn = new OleDbConnection(StrCnn);
Cnn.Open();
string StrSelect = "SELECT * FROM Ouvrage";
OleDbDataAdapter DA = new OleDbDataAdapter(StrSelect,Cnn);
OleDbCommandBuilder CB = new OleDbCommandBuilder(DA);
CB.QuotePrefix = "[";
CB.QuoteSuffix = "]";
DataSet DS = new DataSet();
DA.Fill(DS, "Ouvrage");
// Mettre ici le code de modification des données du DataSet.
… … …
… … …
// Mise à jour des données vers la source
DA.Update(DS,"Ouvrage");
Cnn.Close();
```

L'objet DataRelation

Cette classe représente une relation Parent/Enfant entre deux tables du *DataSet*.

Espace de noms :

System.Data.DataRelation

Quelques constructeurs :

`DataRelation(string N, DataColumn P, DataColumn C)`	Crée une nouvelle relation qui possède comme nom interne N, comme champ parent P et comme champ enfant C.
`DataRelation(string N, DataColumn[] P, DataColumn[] C)`	Crée une nouvelle relation qui possède comme nom interne N, comme champs parents P et comme champs enfants C.

© Karim Kalti

Principales propriétés :

Nom	Type	Signification
`RelationName`	`String`	Nom de la relation.
`ChildColumns`	`DataColumn[]`	En lecture seule. Stocke la liste des colonnes "enfants" de la relation.
`ParentColumns`	`DataColumn[]`	En lecture seule. Stocke la liste des colonnes "parents" de la relation.
`ChildTable`	`DataTable`	En lecture seule. Indique la table contenant le (les) champ(s) "enfant(s)" de la relation.
`ParentTable`	`DataTable`	En lecture seule. Indique la table contenant le (les) champ(s) "parent(s)" de la relation.
`DataSet`	`DataSet`	En lecture seule. Indique le DataSet auquel appartient la relation.

<u>Exemple :</u> Ajout d'une relation entre Ouvrage et Prêt

On considère les trois tables suivantes :

Ouvrage	**Champ**	<u>Inventaire</u>	Titre	Année d'édition
	Type	Entier	Chaîne	Entier

Lecteur	**Champ**	<u>Id</u>	Nom
	Type	Entier	Chaîne

Prêt	**Champ**	<u>Numéro</u>	Inventaire	IdLecteur	Date
	Type	Entier	Chaîne	Entier	Date

L'exemple suivant montre l'ajout d'une relation entre les deux tables *Ouvrage* et *Prêt*.

```
DataColumn ParentCol;
DataColumn ChildCol;
ParentCol = DS.Tables["Ouvrage"].Columns["Inventaire"];
ChildCol = DS.Tables["Prêt"].Columns["Inventaire"];
DataRelation RelOuvragePret = new DataRelation("InventaireRelation", ParentCol, ChildCol);
DS.Relations.Add(RelOuvragePret);
```

Ou d'une manière équivalente :

```
DS.Relations.Add("InventaireRelation",DS.Tables["Ouvrage"].Columns["Inventaire"],
DS.Tables["Prêt"].Columns["Inventaire"]);
```

Les objets contraintes

Une contrainte est une règle utilisée pour maintenir l'intégrité des données dans un objet *DataTable*.
Les objets *DataTable* supportent deux types de contraintes représentés par les classes *UniqueConstraint* et *ForeignKeyConstraint*.

La classe UniqueConstraint

UniqueConstraint permet de spécifier que les valeurs d'une colonnes ou de plusieurs colonnes doivent être uniques. Si la propriété *EnforceConstraints* du *DataSet* est *true*, alors la violation de cette contrainte va engendrer la levée d'une exception.

Quelques constructeurs :

Constructeur	Description
`public UniqueConstraint(string N, DataColumn Cl);`	Crée une contrainte nommée N et associée à la colonne Cl.
`public UniqueConstraint(string N, DataColumn[] Cl);`	Crée une contrainte nommée N et associée à plusieurs colonnes.
`public UniqueConstraint(string N, DataColumn Cl, bool b);`	Crée une contrainte nommée N, associée à une colonne Cl et indique si la contrainte est une clé primaire.
`public UniqueConstraint(string N, DataColumn[] Cl, bool b);`	Crée une contrainte nommée N, associée à plusieurs colonnes et indique si la contrainte est une clé primaire.

© Karim Kalti

Principales propriétés :

Nom	Type	Signification
`ConstraintName`	`string`	Nom de la contrainte (lecture seule)
`Table`	`DataTable`	Nom de la table à laquelle est associée à la contrainte.
`Columns`	`Collection`	Liste des colonnes concernées par la contrainte.
`IsPrimaryKey`	`Bool`	Indique si la contrainte est une clé primaire. (true si c'est une contrainte de clé, c'est la valeur par défaut).

***Exemple* : Ajout d'une UniqueConstraint à une table (Source : MSDN)**

```
private void MakeTableWithUniqueConstraint(){
    DataTable myTable = new DataTable("myTable");
    DataColumn idCol = new DataColumn("id",System.Type.GetType("System.Int32"));
    DataColumn NameCol = new DataColumn("Name", System.Type.GetType("System.String"));
    myTable.Columns.Add(idCol);
    myTable.Columns.Add(NameCol);
    AddUniqueConstraint(myTable);
    // Add one row to the table.
    DataRow myRow;
    myRow = myTable.NewRow();
    myRow["id"] = 1;
    myRow["Name"] = "John";
    myTable.Rows.Add(myRow);

    // Display the table in a DataGrid control.
    dataGrid1.DataSource=myTable;


    // Try to add an identical row.
    try{
        myRow = myTable.NewRow();
        myRow["id"] = 1;
        myRow["Name"] = "John";
        myTable.Rows.Add(myRow);
        }
    catch(Exception e)
       {
         System.Diagnostics.EventLog log = new System.Diagnostics.EventLog();
         log.Source = "My Application";
         log.WriteEntry(e.ToString());
         Console.WriteLine("Exception of type {0} occurred.", e.GetType());
       }
 }
 private void AddUniqueConstraint (DataTable myTable){
    // Create the DataColumn array.
    DataColumn[] myColumns = new DataColumn[2];
    myColumns[0] = myTable.Columns["id"];
    myColumns[1] = myTable.Columns["Name"];
    UniqueConstraint myUniqueConstraint;
    myUniqueConstraint = new UniqueConstraint("idNameConstraint", myColumns);
    myTable.Constraints.Add(myUniqueConstraint);
 }
```

La classe ForeignKeyConstraint

Cette contrainte permet de spécifier comment agir lorsqu'une donnée dépendante de deux ou plusieurs tables différentes est supprimée. Faut-il la supprimer dans les autres tables, la mettre à *null* ou mettre une valeur par défaut.

Quelques constructeurs :

`public ForeignKeyConstraint(string N, DataColumn Parent, DataColumn Child);`	Crée une contrainte nommée N entre deux colonnes Parent et Enfant.
`public ForeignKeyConstraint(string N, DataColumn[] Parent, DataColumn[] Child);`	Crée une contrainte nommée N entre un ensemble de colonnes Parents et un ensemble de colonnes Enfants.

© Karim Kalti

Principales propriétés :

Nom	Type	Signification
`ConstraintName`	`string`	Nom de la contrainte (lecture seule)
`Table`	`DataTable`	Nom de la table "enfant" associée à la contrainte.
`Columns`	`Collection`	Liste des colonnes enfants de la contrainte.
`RelatedTable`	`DataTable`	Nom de la table "Parent" de la contrainte.
`RelatedColumns`	`Collection`	Liste des colonnes "Parent" de la contrainte.
`DeleteRule`	`Rule`	Permet de spécifier comment agir sur les données dépendantes lorsqu'une ligne est supprimée. Les actions possibles sont données par l'énumération Rule
`UpdateRule`	`Rule`	Permet de spécifier comment agir sur les données dépendantes lorsqu'une ligne est mise à jour. Les actions possibles sont données par l'énumération Rule

L'énumération Rule

Valeur	Signification
`Cascade`	Supprime ou met à jour les lignes reliées à la ligne supprimée ou modifiée. C'est la valeur par défaut pour les propriété DeleteRule et UpdateRule.
`None`	Aucune action n'est effectuée sur les lignes reliées.
`SetDefault`	Met les valeurs des colonnes des lignes reliées à la valeur spécifiée dans la propriété DefaultValue de l'objet DataColumn.
`SetNull`	Met les valeurs dans les lignes reliées à la valeur DBNull.

<u>Exemple</u> : Ajout d'une contrainte de clé étrangère

```
DataColumn pCol;
DataColumn cCol;
ForeignKeyConstraint myFKC;
pCol = DS.Tables["Ouvrage"].Columns["Inventaire"];
cCol = DS.Tables["Prêt"].Columns["Inventaire"];
myFKC = new ForeignKeyConstraint("PrêtFKConstraint", pCol, cCol);
myFKC.DeleteRule = Rule.SetNull;
myFKC.UpdateRule = Rule.Cascade;
myFKC.AcceptRejectRule = AcceptRejectRule.Cascade;
DS.Tables["Prêt"].Constraints.Add(MyFKC);
DS.EnforceConstraints = true;
```


© Karim Kalti

Annexe A

Code généré par VS.NET 2003 suite à la création à l'aide de l'assistant d'un Adaptateur de données et d'un DataSet

```
// Dans la partie attritbuts
private System.Data.OleDb.OleDbDataAdapter oleDbDataAdapter1;
private System.Data.OleDb.OleDbCommand oleDbSelectCommand1;
private System.Data.OleDb.OleDbCommand oleDbInsertCommand1;
private System.Data.OleDb.OleDbCommand oleDbUpdateCommand1;
private System.Data.OleDb.OleDbCommand oleDbDeleteCommand1;
private System.Data.OleDb.OleDbConnection oleDbConnection1;
private System.Data.DataSet dataSet1;
```

```
// Dans la méthode InitializeComponents
this.oleDbDataAdapter1 = new System.Data.OleDb.OleDbDataAdapter();
this.oleDbDeleteCommand1 = new System.Data.OleDb.OleDbCommand();
this.oleDbConnection1 = new System.Data.OleDb.OleDbConnection();
this.oleDbInsertCommand1 = new System.Data.OleDb.OleDbCommand();
this.oleDbSelectCommand1 = new System.Data.OleDb.OleDbCommand();
this.oleDbUpdateCommand1 = new System.Data.OleDb.OleDbCommand();
this.dataSet1 = new System.Data.DataSet();
((System.ComponentModel.ISupportInitialize)(this.dataSet1)).BeginInit();

// oleDbDataAdapter1
this.oleDbDataAdapter1.DeleteCommand = this.oleDbDeleteCommand1;
this.oleDbDataAdapter1.InsertCommand = this.oleDbInsertCommand1;
this.oleDbDataAdapter1.SelectCommand = this.oleDbSelectCommand1;

// Le Mapping est utilisé pour faire la correspondance entre les noms utilisés par le DataSet
et ceux utilisés par // la source de données. Il peut être exploité pour modifier les noms au
niveau du DataSet.

this.oleDbDataAdapter1.TableMappings.AddRange(new System.Data.Common.DataTableMapping[] {
new                                  System.Data.Common.DataTableMapping("Table","Ouvrage",new
System.Data.Common.DataColumnMapping[]{
new System.Data.Common.DataColumnMapping("Année d\'édition", "Année d\'édition"),
new System.Data.Common.DataColumnMapping("Inventaire", "Inventaire"),
new System.Data.Common.DataColumnMapping("Titre", "Titre")})});

this.oleDbDataAdapter1.UpdateCommand = this.oleDbUpdateCommand1;

// oleDbDeleteCommand1

this.oleDbDeleteCommand1.CommandText = "DELETE FROM Ouvrage WHERE (Inventaire = ?) AND ([Année
d\'édition] = ? OR ? IS NUL" +"L AND [Année d\'édition] IS NULL) AND (Titre = ?)";

this.oleDbDeleteCommand1.Connection = this.oleDbConnection1;

this.oleDbDeleteCommand1.Parameters.Add(new
System.Data.OleDb.OleDbParameter("Original_Inventaire",    System.Data.OleDb.OleDbType.Integer,
0,   System.Data.ParameterDirection.Input,   false,   ((System.Byte)(0)),   ((System.Byte)(0)),
"Inventaire", System.Data.DataRowVersion.Original, null));

this.oleDbDeleteCommand1.Parameters.Add(new
System.Data.OleDb.OleDbParameter("Original_Année_d\'édition",
System.Data.OleDb.OleDbType.Integer,     0,     System.Data.ParameterDirection.Input,     false,
((System.Byte)(0)),             ((System.Byte)(0)),             "Année             d\'édition",
System.Data.DataRowVersion.Original, null));

this.oleDbDeleteCommand1.Parameters.Add(new
System.Data.OleDb.OleDbParameter("Original_Année_d\'édition1",
System.Data.OleDb.OleDbType.Integer,     0,     System.Data.ParameterDirection.Input,     false,
```


© Karim Kalti

```
((System.Byte)(0)), ((System.Byte)(0)), "Année d\'édition", System.Data.DataRowVersion.Original, null));

this.oleDbDeleteCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Original_Titre", System.Data.OleDb.OleDbType.VarWChar, 50, System.Data.ParameterDirection.Input, false, ((System.Byte)(0)), ((System.Byte)(0)), "Titre", System.Data.DataRowVersion.Original, null));

// oleDbConnection1
this.oleDbConnection1.ConnectionString = @"Jet OLEDB:Global Partial Bulk Ops=2;Jet OLEDB:Registry Path=;Jet OLEDB:Database Locking Mode=1;Data Source=""C:\Data\Biblio.mdb""; Jet OLEDB:Engine Type=5;Provider=""Microsoft.Jet.OLEDB.4.0"";Jet OLEDB:System database=;Jet OLEDB:SFP=False;persist security info=False;Extended Properties=;Mode=Share Deny None;Jet OLEDB:Encrypt Database=False;Jet OLEDB:Create System Database=False;Jet OLEDB:Don't Copy Locale on Compact=False;Jet OLEDB:Compact Without Replica Repair=False;User ID=Admin;Jet OLEDB:Global Bulk Transactions=1";

// oleDbInsertCommand1
this.oleDbInsertCommand1.CommandText = "INSERT INTO Ouvrage([Année d\'édition], Inventaire, Titre) VALUES (?, ?, ?)";
this.oleDbInsertCommand1.Connection = this.oleDbConnection1;
this.oleDbInsertCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Année_d\'édition", System.Data.OleDb.OleDbType.Integer, 0, "Année d\'édition"));
this.oleDbInsertCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Inventaire", System.Data.OleDb.OleDbType.Integer, 0, "Inventaire"));
this.oleDbInsertCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Titre", System.Data.OleDb.OleDbType.VarWChar, 50, "Titre"));


// oleDbSelectCommand1
this.oleDbSelectCommand1.CommandText = "SELECT [Année d\'édition], Inventaire, Titre FROM Ouvrage";
this.oleDbSelectCommand1.Connection = this.oleDbConnection1;

// oleDbUpdateCommand1
this.oleDbUpdateCommand1.CommandText = "UPDATE Ouvrage SET [Année d\'édition] = ?, Inventaire = ?, Titre = ? WHERE (Inventaire = ?) AND ([Année d\'édition] = ? OR ? IS NULL AND [Année d\'édition] IS NULL) AND (Titre = ?)";

this.oleDbUpdateCommand1.Connection = this.oleDbConnection1;
this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Année_d\'édition", System.Data.OleDb.OleDbType.Integer, 0, "Année d\'édition"));

this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Inventaire", System.Data.OleDb.OleDbType.Integer, 0, "Inventaire"));

this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Titre", System.Data.OleDb.OleDbType.VarWChar, 50, "Titre"));

this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Original_Inventaire", System.Data.OleDb.OleDbType.Integer, 0, System.Data.ParameterDirection.Input, false, ((System.Byte)(0)), ((System.Byte)(0)), "Inventaire", System.Data.DataRowVersion.Original, null));

this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Original_Année_d\'édition", System.Data.OleDb.OleDbType.Integer, 0, System.Data.ParameterDirection.Input, false, ((System.Byte)(0)), ((System.Byte)(0)), "Année d\'édition", System.Data.DataRowVersion.Original, null));

this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Original_Année_d\'édition1", System.Data.OleDb.OleDbType.Integer, 0, System.Data.ParameterDirection.Input, false, ((System.Byte)(0)), ((System.Byte)(0)), "Année d\'édition", System.Data.DataRowVersion.Original, null));

this.oleDbUpdateCommand1.Parameters.Add(new System.Data.OleDb.OleDbParameter("Original_Titre", System.Data.OleDb.OleDbType.VarWChar, 50, System.Data.ParameterDirection.Input, false, ((System.Byte)(0)), ((System.Byte)(0)), "Titre", System.Data.DataRowVersion.Original, null));

// dataSet1
this.dataSet1.DataSetName = "NewDataSet";
this.dataSet1.Locale = new System.Globalization.CultureInfo("fr-FR");
((System.ComponentModel.ISupportInitialize)(this.dataSet1)).EndInit();
```


© Karim Kalti

Développement d'applications Web dynamiques

Rappel

Site Web :

Un site Web est un ensemble de pages Web, généralement liées entre elles par des liens hypertextes et stockées sur un ordinateur connecté en permanence à Internet.

Adresse IP :

C'est un identifiant numérique composé de 4 nombres codés chacun sur un octet (0-255) permettant d'identifier d'une manière unique un ordinateur faisant partie d'un réseau qui utilise le protocole TCP/IP.
Exemple : 193.94.15.3

Nom de domaine :

Le protocole TCP/IP permet d'associer des noms en langage courant aux adresses IP numériques des ordinateurs. Ces noms sont plus significatifs et permettent de désigner plus facilement ces ordinateurs.
Pour assurer la correspondance entre le nom du domaine et l'adresse IP, le TCP/IP utilise un service appelé *Domain Name Service*. Les machines qui hébergent ce service sont appelées des *Domaine Name Server*.

Port de communication :

Plusieurs applications sur une même machine connectée à un réseau peuvent utiliser en même temps les services du protocole TCP/IP. Pour pouvoir distinguer les données échangées par les différentes applications, ces dernières doivent se voir attribuer une adresse logique unique sur la machine. Cette adresse, codée sur 16 bits est appelée port de communication.
La combinaison adresse IP + Port constitue une adresse unique sur une machine. Dans ce cadre l'adresse IP sert à identifier de façon unique un ordinateur sur le réseau tandis que le numéro de port indique l'application à laquelle les données sont destinées. De cette manière, lorsque l'ordinateur reçoit des informations destinées à un port, les données sont envoyées vers l'application correspondante.

URL :

Acronyme de *Uniform Resource Locator*, l'URL est une chaîne de caractères ayant un format universel normalisé permettant de désigner une ressource sur Internet :

Protocole	UserName + Password (facultatif)	Nom du serveur	Port (facultatif si 80)	Chemin
http://	User:PassWord@	www.NomSite.net	80	/cours/index.htm

Serveur Web :

- Un serveur Web est un logiciel qui joue le rôle d'intermédiaire (middleware) entre un client (navigateur) et une page Web demandée par ce client et située sur l'ordinateur serveur.
- Un serveur Web reçoit la requête http (url) émise par le client, l'analyse, trouve la page correspondante sur le serveur et la renvoie toujours à l'aide du protocole http au client.
- Le serveur reçoit les requêtes des clients sur un port de communication. La valeur par défaut de ce port est 80.
- Les principaux serveurs Web sur le marché sont entre autres : Apache, Microsoft IIS (Internet Information Server), Microsoft PWS (Personal Web Server), …

Navigateur :

Un navigateur est un logiciel qui permet de :
- Envoyer des requêtes http sous forme d'URL à un serveur qui héberge des sites Web.
- Récupérer la réponse du serveur Web et l'afficher à l'écran.

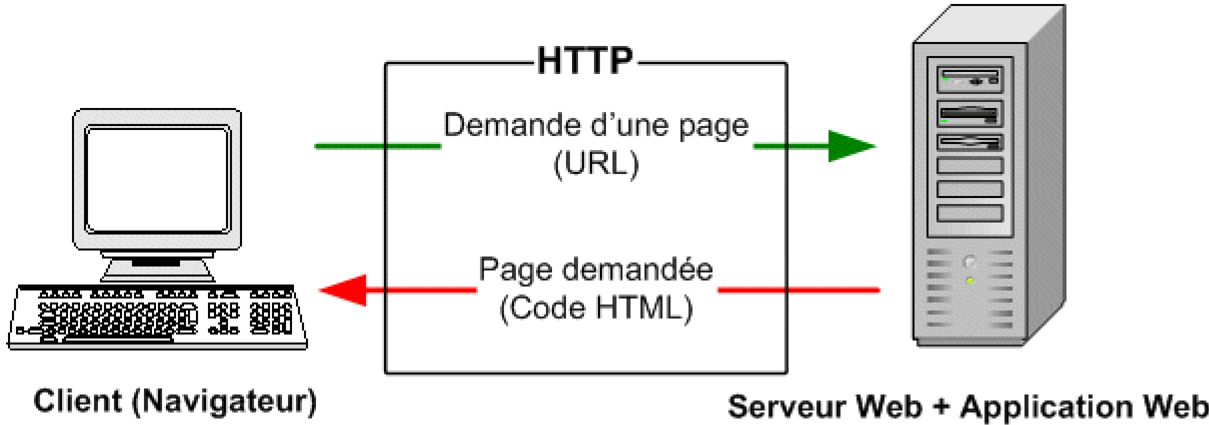
© Karim Kalti

Protocole http :

HTTP est un protocole qui permet d'assurer le transfert de fichiers (essentiellement au format HTML), localisés grâce à une chaîne de caractères appelée URL, entre un navigateur (le client) et un serveur Web.

Fonctionnement d'une application Web statique

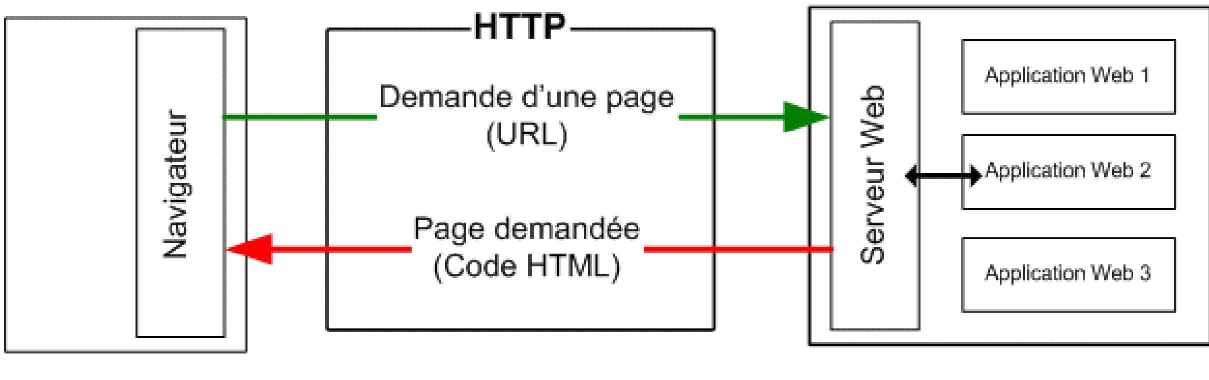
Les applications Web emploient une architecture client/serveur.



Logiciels nécessaires sur chaque machine

Client : Navigateur Web.
Serveur : Serveur Web (middleware) + les pages qui composent l'application Web.

Déroulement des échanges de données entre le client et le serveur



On considère ici une application Web simple, c'est-à-dire une application dont les pages sont codées en HTML seulement sans aucun autre langage de script supplémentaire.

1- Le navigateur (client) demande une page Web en envoyant son URL au serveur à l'aide du protocole HTTP.
2- Le serveur Web intercepte la requête du client et recherche la page demandée.
3- Le serveur renvoie le code source (HTML) de la page au client à l'aide du protocole HTTP.
4- Le navigateur du client reçoit la réponse du serveur, interprète le code HTML reçu et affiche la page qui résulte de cette interprétation.

Cette description montre le rôle important qu'assure le navigateur dans ce processus d'échange à savoir le rôle d'interpréteur de code HTML.

© Karim Kalti

Page Web statique et page Web dynamique

Page Web statique

- Une page Web statique est une page dont le contenu est figé et est identique pour chaque consultation de la page.
- Le contenu d'une page Web statique est connu au moment de la création de la page.
- Le contenu statique ne peut être modifié que suite à une intervention sur le code source de la page Web par le programmeur (le créateur de la page).

Page Web dynamique

- Une page Web dynamique est une page dont le contenu change d'une consultation à une autre.
- Ce contenu ne peut pas être explicitement défini au moment de la création de la page. Il est plutôt généré au moment de l'exécution (à la volée).
- L'aspect dynamique d'un contenu peut être dû entre autres à :
  - L'évolution des données à afficher dans le temps (Exemple : affichage de la date, affichage en temps réel des données de la bourse, …).
  - Interaction de l'utilisateur (Exemple : affichage des résultats suivant la requête de l'utilisateur).

Langage pour la génération du contenu dynamique

Limites du HTML

- Le HTML est un langage permettant de spécifier la mise en forme (formatage) du contenu d'une page Web.
- Le HTML ne propose pas de moyens de :
  - Faire de la programmation structurée (pas de structures de contrôle pas de fonctions, …).
  - Interaction avec l'utilisateur (Récupération et traitement des données d'un formulaire, etc.).
  - De génération dynamique du contenu.

Contenu dynamique côté client et contenu dynamique côté serveur

- La génération du contenu dynamique nécessite l'utilisation de langages complémentaires au HTML.
- Cette génération peut se faire à deux niveaux : au niveau du client (navigateur) et au niveau du serveur (serveur Web).
- Il existe des langages spécifiques pour chaque niveau de génération dynamique de contenu.

Contenu dynamique côté client

Motivations

La création dynamique du contenu côté client peut être motivée par plusieurs raisons.

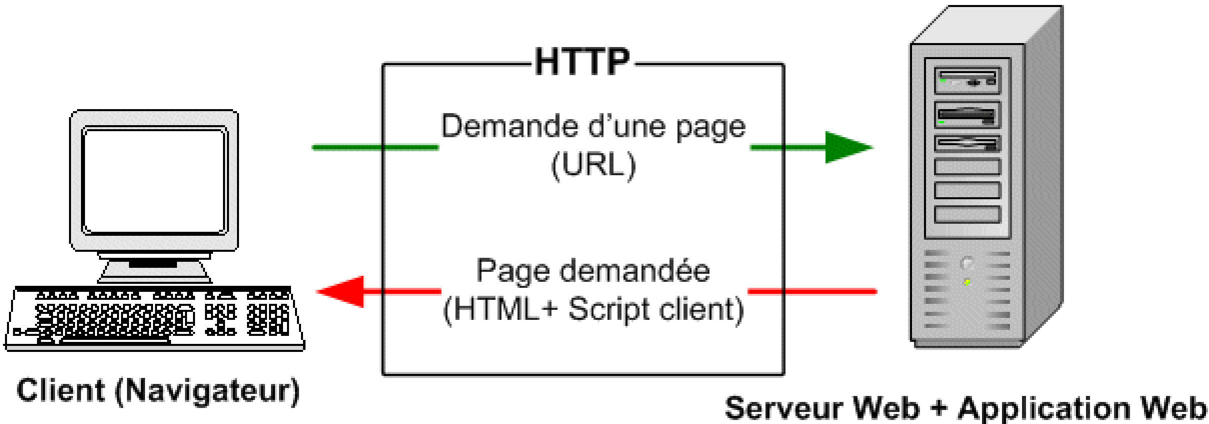
- Alléger la charge de traitement du serveur en déléguant au client certaines opérations qui peuvent s'effectuer sur ce dernier. (Exemple : vérification des données saisies par l'utilisateur lors du remplissage d'un formulaire avant leur insertion dans une base de données).
- Certaines données dynamiques, de part leur nature, ne peuvent être générées que sur le poste client. (Exemple: affichage de l'heure courante sur le poste du client : le serveur peut être situé dans une région qui n'appartient pas au même faisceau horaire que celui de la région du client).

Langages de script côté client :

- Il existe essentiellement deux langages pour la création du contenu Web dynamique côté client : JavaScript et VBScript.
- Le script côté client est inséré dans le même fichier source que le HTML.
- Un fichier qui contient du HTML et du script côté client garde l'extension HTML ou HTM.
- Le script côté client est exécuté par le navigateur.

© Karim Kalti

Echanges entre client et serveur pour une page qui contient un script client



Support des langages de script par les navigateurs

JavaScript	VbScript
Toutes les dernières versions des navigateurs (IE, Netscape Navigator, firefox,…)	IE seulement : Les scripts VB sont tout simplement ignorés par les autres navigateurs (Pas d'erreur signalée).

Remarque :
Il existe d'autres langages qui permettent d'écrire des modules qui sont téléchargés avec les pages Web et exécutés par les machines clientes comme par exemple les applettes Java et les contrôles ActiveX (C++, VB,…).

Génération côté serveur

Motivations

- Les scripts côté serveur peuvent être utilisés pour effectuer tout genre d'opérations d'interaction ou de création de contenu dynamique mais côté serveur (traitement des formulaires, mise en page dynamique, accès aux données, etc.).
- L'utilisation la plus fréquente reste celle de l'accès aux données (génération dynamique du contenu en se basant sur des données qui résultent de l'interrogation d'une base de données).
- Ce genre d'opérations ne peut pas être effectué côté client (nécessité dans ce cas d'avoir une copie locale de la base sur chaque client).

Intégration de script serveur dans une page Web

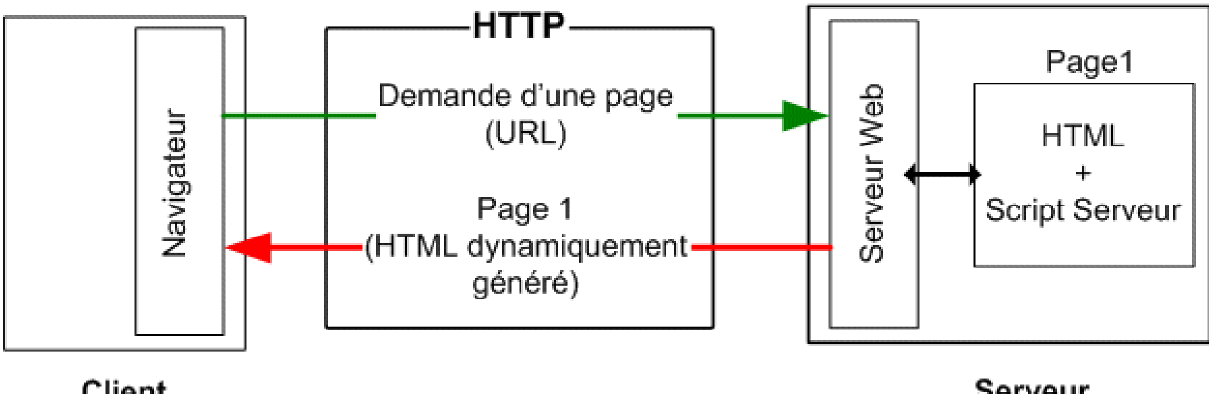
- Un script serveur est généralement inséré dans le code HTML avec des balises spéciales (balises qui dépendent du langage de script utilisé).
- Une page qui contient du HTML et un script serveur ne peut pas avoir l'extension HTML ou HTM. Son extension dépend du langage du script serveur qu'elle utilise.
- Une même page Web ne peut utiliser qu'un seul langage de script serveur (pas de possibilité d'utiliser plusieurs langages de script serveur au sein d'une même page).

Interprétation des scripts côté serveur

L'interprétation des scripts côté serveur est effectué par le serveur Web.

© Karim Kalti

Déroulement des échanges entre le client et le serveur



1- Le navigateur (client) demande une page Web dynamique en envoyant son URL au serveur à l'aide du protocole HTTP.
2- Le serveur recherche la page et reconnaît qu'elle contient un script serveur (d'après son extension).
3- Le serveur analyse le code de la page :
    a. Il laisse intacte les parties codées en HTML.
    b. Il interprète le script serveur et génère dynamiquement un code HTML équivalent.
4- Le serveur envoie la page (qui contient du HTML seulement) au client via le protocole HTTP.
5- Le navigateur du client reçoit la réponse du serveur, interprète le code HTML reçu et affiche la page qui résulte de cette interprétation.

Langages de scripts côté serveur

Il existe une panoplie de langages de script côté serveur. Mais actuellement trois sont les plus utilisés : ASP.NET, PHP, JSP.

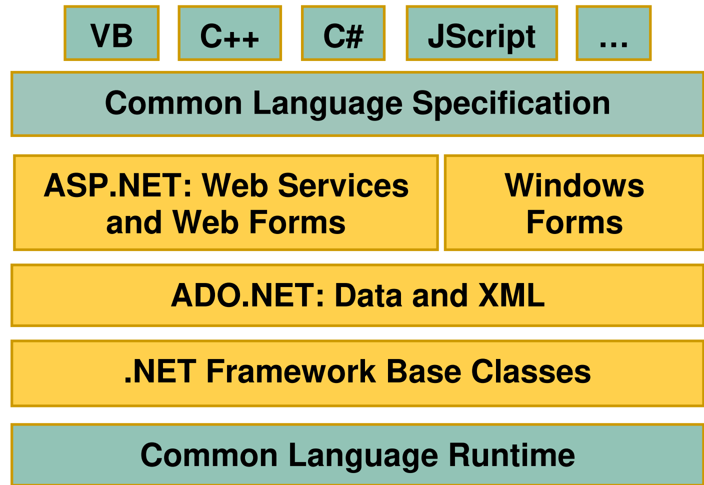
	Acronyme de	Propriétaire	Langage de programmation	Plateforme	Serveur Web
ASP	Active Server Pages	Microsoft	C#, VB, C++ , ...	Windows, (Linux projet en cours)	IIS
JSP	Java Server Pages	Sun	Java	Linux, Solaris, Windows, …	IIS, Apache, Tomcat, …
PHP	Pre Hypertext Processor	Open source	PHP proche du C	Linux, Solaris, Windows, …	IIS, Apache,…

© Karim Kalti

ASP.NET

Introduction

- ASP.NET est la nouvelle technologie proposée par MS pour le développement d'applications Web dynamiques.
- ASP.NET est différent de ASP (une nouvelle conception⇒ Architecture différente).
- ASP.NET fait partie du .NET FrameWork.
- Les langages de développement : VB, C#, C++, …



Caractéristiques de ASP.NET

- Permet le développement côté serveur d'applications Web dynamiques.
- Programmation orientée objet.
- Accès à toute l'API du .NET FrameWork.
- Introduction des contrôles Web côté serveur. (Facilité de programmation et adaptation dynamique aux navigateurs clients).
- Possibilité de programmer avec différents langages.
- Intégration avec ADO.NET pour l'accès aux données.
- Prise en charge complète de XML.
- Mécanisme intégré pour mettre en cache les pages Web les plus fréquemment réclamées au serveur.
- ASP permet de développer des pages interprétées alors que ASP.NET permet de développer des applications compilés (à l'aide de la technique du code behind) donc plus rapides que celles développées avec ASP.

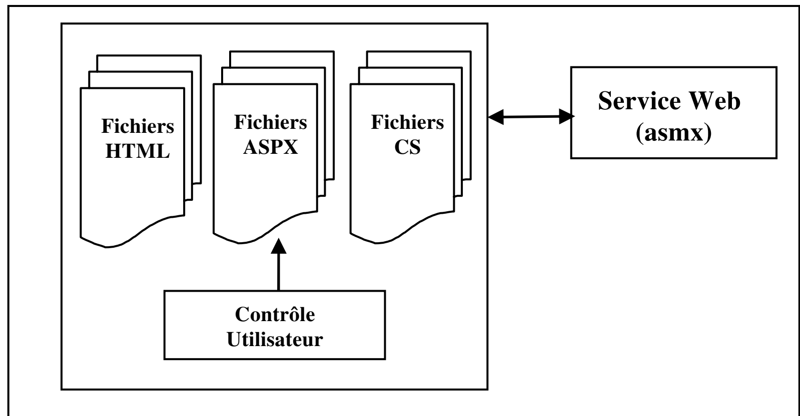
Structure et fichiers d'une application ASP.NET

Une application Web ASP.NET se compose :
- D'un ensemble de pages Web dynamiques (Fichiers *.aspx) comportant généralement des formulaires Web. A la manière des formulaires Windows, les formulaires Web constituent la partie visuelle de la page.

Elle peut comporter également :
- Des pages Web simples (Fichiers *.html).
- Des contrôles Web personnalisés construits à partir d'autres contrôles serveur (Fichiers *.ascx).
- Une application Web peut faire appel également à des services Web distants (Fichiers *.asmx).

© Karim Kalti

- ASP.NET a été conçu de façon à permettre la séparation entre le code de traitement et l'interface. Ce code de traitement peut être placé dans des fichiers sources C# (*.cs) ou Visual Basic (*.vb) liés aux fichiers des interfaces (*.aspx).



Fichiers d'une application Web ASP.NET

Outils pour développer en ASP.NET

- Un éditeur de texte pour écrire le code (Exemple : Notepad).
- Un Serveur Web : IIS ou Cassini.
- La CLR : (elle doit être installée après l'installation du serveur Web).

Il existe des environnements de développement intégrés (EDI) plus élaborés :
- Visual Studio.NET (payant)
- WebMatrix (gratuit).

Structure générale d'une page ASP.NET

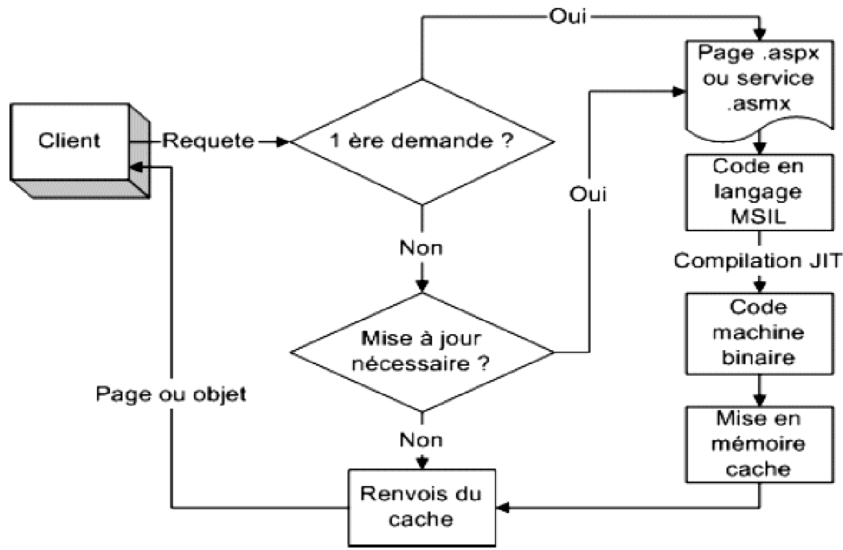
La structure générale d'une page ASP.NET (fichier aspx) est de la forme :

```
<head>
  <script language= "c#" runat="server">
    type fct (type i) { … }
  </script>
  <script language= "javascript">
    function f ;
  </script>
</head>
<body>
   <table>
    …
   </table>
   <asp :Label [Propriétés]></asp :Label>
   <% =fct(5) %>
</body>
```

Une page aspx comporte :
- Des scripts ASP.Net écrits en VB ou en C# ou en tout autre langage supporté par le .NET.
- Du HTML pour le formatage du contenu statique et éventuellement du javascript pour effectuer des traitements côté client.

© Karim Kalti

Echanges client serveur lors de la consultation d'une page ASP.NET



Commentaires :

- Suite à sa première demande une page ASP.NET est compilée puis mise en cache. Toute nouvelle demande de cette page va engendrer l'envoi direct du cache sans nécessiter une recompilation sauf si une mise à jour de la page est nécessaire.
- La mise à jour dont il est question ici concerne le code de traitement et de génération associé à la page.
- La saisie des données d'un formulaire ne demande pas une recompilation, le code de traitement et de génération étant en effet toujours le même (c'est comme la saisie des données pour un formulaire Windows, elle ne demande pas la recompilation et la génération d'un nouvel exécutable).

Remarque :

- Si le fichier consulté est un fichier aspx alors le client (le navigateur dans ce cas) reçoit de la part du serveur une page.
- Si le fichier consulté est un fichier asmx alors le client (une application web dans ce cas) reçoit de la part du serveur un objet.

Insertion des scripts ASP.NET dans une page Web dynamique

L'insertion des scripts serveur ASP.NET dans une page Web se fait à l'aide de balises spéciales. Les balises utilisées dans ce cadre diffèrent suivant l'endroit d'insertion de ces scripts.

Balise	Utilisation
`<% = … %>`	Pour l'affichage de la valeur d'une expression. Cette expression doit renvoyer une chaîne de caractères. Ces expressions peuvent être des variables simples ou des appels à des fonctions (fonctions de l'API .NET ou personnalisées). La valeur de l'expression est concaténée avec le contenu statique formaté en HTML
`<% #… %>`	Pour l'affichage du contenu des variables seulement (variables chaînes de caractères).
`<script > … </script>`	Pour l'insertion des scripts. Les scripts doivent figurer dans la partie entête de la page.
`<asp:NomControl … > </asp:NomControl>`	Pour l'insertion des contrôles Web dans un formulaire serveur.

© Karim Kalti

Affichage de la valeur d'une expression

Exemple :

```
<html>
<head>
<title>Exemple 1 ASP.NET </title>
</head>
<body>
   Il est <% =DateTime.Now.ToString("T") %>
</body>
</html>
```

Commentaires :

- L'expression en question ici est un appel d'une fonction.
- Ce code montre comment appeler une fonction prédéfinie du .NET et comment concaténer sa valeur de retour avec un contenu statique.
- La méthode *ToString* de cet objet est surchargée. La version utilisée permet de spécifier le format d'affichage de la date.
- Il n'est pas nécessaire de spécifier le langage car le code de l'appel s'écrit de la même manière en VB et en C# (Fonction .NET).

Résultat : (heure sur le serveur)

```
Il est 23:39:33
```

Code HTML reçu par le navigateur :

Suite à la réception d'une demande de la page, le serveur Web interprète le script ASP.NET et génère du HTML qu'il insère avec le flux HTML renvoyé au client.

```
<html>
 <head>
   <title>Exemple 1 ASP.NET </title>
 </head>
 <body>
   Il est 23:39:33
 </body>
</html>
```

Insertion de modules de scripts

- Si le contenu dynamique nécessite la définition de modules de scripts (fonctions) alors ces modules doivent être insérés dans la page dans une zone à part, définie par les balises <script> … </script>, et se trouvant dans la partie entête de la page.
- L'appel de ces fonctions dans le corps de la page peut toujours se faire en utilisant la balise <% =… %>.
- Cette technique permet une meilleure organisation du code source de la page en séparant les scripts de génération de contenu, du code HTML permettant le formatage de ce contenu.

Exemple :

```
<html>
<head>
 <script language= "c#" runat= "server">
  string Demain()
  {
    DateTime aujourdhui=DateTime.Now ;
    DateTime demain=aujourdhui.AddDays(1) ;
    return demain.ToString() ;
  }
  </script>
</head>
<body>
  Date du jour : <% =DateTime.Now.ToString() %><br>
  Date de demain : <% =Demain() %>
</body>
</html>
```


© Karim Kalti

Commentaires :

- L'attribut *language* désigne le langage utilisé pour écrire les scripts ASP.NET. Les deux langages les plus utilisés sont C# et VB.
- Il n'est pas possible d'utiliser deux langages différents dans la même page.
- La valeur *server* de l'attribut *runat* indique au serveur Web que le script doit être exécuté sur le serveur. Le client recevra seulement du HTML.

Résultat :

Date du jour : 24/01/2003 23:51:45
Date de demain : 25/01/2003 23:51:45

Affichage du contenu d'une variable.

- La valeur d'une variable peut être insérée dans le flux HTML à l'aide de la balise <%#   %>.
- La variable doit être de type chaîne ou convertie en chaîne.
- Il faut faire un Binding lors du chargement de la page.

Exemple :

```
<html>
<script language= "c#" runat= "server">
   string s = "bonjour";
   void Page_Load(Object sender, EventArgs e)
   { DataBind();}
</script>

<body>
 le contenu de la variable est : <%# s %>
</body>
</html>
```

Insertion de contenu dynamique dans la réponse adressée au navigateur

L'ASP.NET dispose d'un objet *Response* qui représente la réponse (code HTML de la page) qui sera envoyée par le serveur au navigateur. Cet objet possède une méthode *Write* qui permet d'insérer un contenu généré dynamiquement dans le flux de données envoyé au navigateur.

Exemple 1 : Insertion d'une chaîne de caractères simple :

```
<html>
<script language= "c#" runat= "server">
</script>
<body>
 <% Response.Write("Hello Word"); %>
</body>
</html>
```

Exemple 2 : Insertion d'un flux formaté

```
<html>
<script language= "c#" runat= "server">
</script>
<body>
 <%
 Response.Write("Bonjour <b>tout <i>le monde</i><b>");
 %>
</body>
</html>
```


© Karim Kalti

Résultat :

Bonjour **tout** ***le monde***

Exemple 3 : Insertion d'un flux formaté

```
<html>
<script language= "c#" runat= "server">
</script>
<body>
 <%
  for(int i=6;i>1;i--)
   {
       string s= "<h"+i+">"
       + "Nous sommes le : "
       + DateTime.Now.ToString()
       + "</h"+i+">";
       Response.Write(s);
   }
 %>
</body>
</html>
```

Résultat :

Nous sommes le : 25/01/2003 00:10:41

Nous sommes le : 25/01/2003 00:10:41

Nous sommes le : 25/01/2003 00:10:41

Nous sommes le : 25/01/2003 00:10:41

Nous sommes le : 25/01/2003 00:10:41

L'ajout de commentaires dans un code ASP.NET

Une page ASP.NET peut contenir des balises HTML, des balises ASP et du code C# ou VB. L'insertion des commentaires dépend alors du langage utilisé :

Dans du code C#	// le reste de la ligne est en commentaire. /* … */ Commentaire multi lignes.
Dans du code HTML	<!-- début des commentaires --> Fin des commentaires
Dans une balise ASP.NET	<%-- début des commentaires --%> Fin des commentaires

La technique du code behind

- Le code behind est une manière d'organiser le code d'une page web dynamique qui vise à séparer la partie interface de la partie traitement (couche présentation et couche métier).

- En ASP.NET, cela consiste à séparer dans deux fichiers différents le code nécessaire à la création de l'interface (fichier aspx) et le code qui contient la partie traitement (fichier cs).

Fichier Page1.cs

```
using System;
public class LaDate : System.Web.UI.Page
{
 protected DateTime aujourdhui;
 protected DateTime demain;
```


© Karim Kalti

```
 protected string Demain()
  {
    aujourdhui=DateTime.Now;
    demain=aujourdhui.AddDays(1);
    return demain.ToString() ;
  }
}
```

Fichier Test.aspx

```
<%@ Page Language="C#" Src="Page1.cs" Inherits="LaDate"%>
<html>
<body>
  Date de demain : <% =Demain() %>
</body>
</html>
```

Commentaires :

Le fichier Page1.cs

- Ce fichier contient la définition d'une classe *LaDate* qui contient le code permettant de déterminer la date recherchée (script de la partie traitement).
- Ce fichier est entièrement écrit en C# (aucune référence à l'interface).
- La classe *Page* est la classe de base de toute page Web écrite en ASP.NET.
- La classe *LaDate* hérite de la classe *Page*.

Le fichier Test.aspx

- Le fichier *Test.aspx* contient le code permettant de générer le contenu de la page (la partie interface). Ce fichier contient un appel à la fonction *Demain( )* qui n'est pas définie dans ce fichier mais dans le fichier associé nommé *Page1.cs*.
- La directive @*Page* au début du fichier *Test.aspx* indique au serveur que la page *Test* en tant qu'objet est basée sur le modèle de page (héritage) définie par la classe *LaDate*.
- Cette directive spécifie également le langage de programmation (*Language="C#"*) et le fichier source dans lequel est définie la classe *LaDate* (*Src="Page1.cs"*).
- La définition de l'objet qui représente la page *Test* est transparente pour le programmeur. Grâce au lien d'héritage avec la classe *LaDate*, cet objet peut utiliser directement les méthodes et propriétés de cette classe et des classes sur lesquelles elle se base.
- Ceci explique la possibilité de l'appel directe de la fonction *Demain( )* dans *Test.aspx*.

© Karim Kalti

Documents d'une application Web

Une application Web est une application regroupant divers documents :

- Des fichiers aspx : contenant des scripts serveur.
- Des fichiers de code .NET (partie métier).
- Des fichiers HTML.
- Des contrôles utilisateurs.
- Des fichiers spéciaux (*global.asax* et *web.config*).
- Des ressources (images, sons, vidéo, …).

Organisation des documents d'une application Web

Organisation physique

Une application Web ASP.NET comprend généralement :

- Un dossier Racine : Les documents d'une application Web doivent être sous une même racine appelée la racine de l'application Web.
- Un dossier [bin] : Ce dossier sert à placer les modules pré-compilés de l'application. Il doit être un sous-dossier du dossier racine.
- Un fichier [global.asax] qui permet d'initialiser l'application Web dans son ensemble ainsi que l'environnement d'exécution de chacun de ses utilisateurs.
- Un fichier [web.config] qui permet de paramétrer le fonctionnement de l'application.
- Un fichier [default.aspx] qui joue le rôle de porte d'entrée de l'application (page de démarrage).

Dossier virtuel

Le dossier racine d'une application Web doit être associée à un chemin virtuel au sein du serveur Web pour que ce dernier puisse reconnaître l'application et servir ses demandeurs.

Cycle de vie d'une application Web

- Une application Web démarre dès q'une page située dans le dossier virtuel qui lui est associé est demandée par un client.
- Chaque client (navigateur) qui demande une page de l'application engendre la création d'une nouvelle session.
- Une application Web dure aussi longtemps qu'elle a des sessions actives.
- Une application Web est terminée par le serveur Web lorsqu'elle reste inactive pendant un délai donné après la fermeture de la dernière session active. Ce délai d'inactivité est configurable dans le fichier [web.config] associé à l'application.

Remarque :

Le client du serveur Web n'est pas l'utilisateur mais plutôt le navigateur. Si un utilisateur ouvre sur sa machine deux instances du navigateur pour consulter une application Web alors chaque instance représente un client.

Modèle objet d'une application Web

ASP.NET fournit une panoplie d'objets permettant de manipuler par programmation les différentes composantes d'une application Web. Les objets les plus importants et qui constituent le cœur de toute application Web sont :

- ***Application* :** qui représente une application Web et permet de manipuler ses propriétés.
- ***Session* :** qui représente une session ouverte par un client.
- ***Page* :** qui représente une page Web.
- ***Les contrôles Web* :** Ces contrôles sont similaires aux contrôles Windows et permettent de créer des formulaires Web.
- ***Request* :** qui représente la requête émise par le client au serveur.
- ***Response* :** qui représente la réponse envoyée par le serveur Web au client.

© Karim Kalti

Espaces de noms des objets ASP.NET

ASP.NET n'est pas un langage à proprement dit. Il se présente plutôt comme une bibliothèque de classes du FrameWork .NET qui offre toutes les fonctionnalités nécessaires pour la création d'une application Web. Ces classes sont regroupées dans les espaces de noms présentés par le tableau suivant :

Espaces de noms	Classes
`System.Web`	Contient la majorité des objets qui composent une application Web comme les objets : *Application, Browser, Cookies, Exception, Request, Response*, *Server* et *Trace*.
`System.Web.SessionState`	Contient les objets permettant de gérer l'état d'une session comme l'objet *Session*.
`System.Web.Services`	Contient les objets qui permettent de créer et d'exploiter des services Web.
`System.Web.UI`	Contient les objets qui permettent de contrôler l'interface comme les objets *Page* et *Control*.
`System.Web.UI.WebControls`	Contient les objets contrôles serveur utilisés dans les formulaires Web.
`System.Web.Caching`	Contient essentiellement la classe *Cache* qui sert à mettre en œuvre un cache côté serveur afin d'améliorer les performances de l'application.
`System.Web.mail`	Contient les objets *MailMessage*, *MailAttachement* et *SmtpMail* qui servent à envoyer des courriers électroniques.
`System.Web.Security`	Contient les objets et les modules d'authentification. Ils servent à authentifier les utilisateurs afin de renforcer la sécurité des applications.

L'objet Page

- Un fichier *aspx* représente une page Web qui comporte généralement un formulaire Web généré et traité côté serveur. L'accès par programmation à cette page se fait à l'aide de l'objet *Page* de ASP.NET.
- Chaque demande d'une page *aspx* par un client engendre la création sur le serveur d'une instance de l'objet *Page* basée sur la page demandée.

Cycle de vie d'une page aspx

Le cycle de vie d'une page Web est comme suit :

- Le client demande une page (fichier aspx).
- Le serveur Web :
    - localise le fichier *aspx* sur le serveur.
    - Crée en mémoire une instance *Page* à partir du fichier *aspx*.
    - Génère la réponse sous forme de code HTML.
    - Transmet cette réponse au navigateur.
    - Détruit l'instance de l'objet *Page* sur le serveur (pas le fichier aspx).

Espace de noms : `System.Web.UI`

Constructeurs : `public Page();`

Principales propriétés :

Propriété	Type	Signification
`Application`	`HttpApplicationState`	Donne accès à l'état de l'application en cours.
`Cache`	`Cache`	Donne accès à l'objet Cache associé à l'application.
`Controls`	`ControlCollection`	Contient la collection des contrôles Web de la page.
`Request`	`HttpRequest`	Donne accès à l'objet *Request* de la requête en cours.
`Response`	`HttpResponse`	Donne accès à l'objet *Response* de la réponse en cours.
`Server`	`HttpServerUtility`	Donne accès à l'objet *Server*.
`Session`	`HttpSessionState`	Donne accès à l'objet *Session*.
`Visible`	`bool`	Obtient ou définit une valeur indiquant si l'objet *Page* est rendu.
`EnableViewState`	`bool`	Obtient ou définit une valeur indiquant si la page conserve son état d'affichage, ainsi que celui de tous les contrôles serveur qu'elle contient, à la fin de la demande de la page en cours.
`IsPostBack`	`bool`	Obtient une valeur indiquant si la page est en cours de chargement en réponse à une publication du client ou en réponse à une première demande d'accès.

© Karim Kalti

Principales méthodes :

Méthode	Signification
`void DataBind()`	Lie une source de données au contrôle serveur appelé et à tous ses contrôles enfants.
`string MapPath(string)`	Retourne le chemin physique d'un chemin virtuel passé comme argument.

Principaux événements :

Les principaux événements d'un objet *Page* sont cités dans le tableau suivant :

Evénement	Type de délégation	Gestionnaire	Signification
`Init`	`EventHandler`	`Page_Init`	Cet événement se déclenche lorsqu'un contrôle est chargé en mémoire et initialisé à partir de ses propriétés spécifiées au moment de la définition de l'objet qui représente ce contrôle. Il n'est pas possible d'accéder aux autres contrôles durant cet événement (ces derniers ne sont pas nécessairement déjà créés). Pour cette raison, il n'est pas possible d'agir sur les propriétés des contrôles pour définir la disposition par exemple.
`Load`	`EventHandler`	`Page_Load`	Cet événement se déclenche lorsque le contrôle courant et tous ses contrôles enfants (La page et tous ses contrôles serveur) ont été chargés en mémoire (après l'événement *Init*). Il est possible dans ce cas d'accéder à ces contrôles et d'agir sur leurs propriétés pour faire une initialisation dynamique par exemple.
`Unload`	`EventHandler`	`Page_Unload`	Cet événement se déclenche lorsque la page se décharge de la mémoire. Il est exploité pour libérer les ressources utilisées par le contrôle (fermeture des fichiers et des connexions aux bases de données, …). L'accès aux contrôles enfants est encore possible à ce niveau.
`Disposed`	`EventHandler`	`Page_Disposed`	Cet événement se déclenche lorsque la page est détruite (l'accès aux contrôles enfants n'est pas sûr). C'est le dernier événement dans le cycle de vie d'une page.
`Error`	`EventHandler`	`Page_Error`	Se déclenche lorsqu'une exception non gérée se produit dans le code de l'objet qui représente la page.

- Sur le plan pratique l'événement *Load* est l'événement le plus traité.
- A part l'événement *Error*, l'ordre d'occurrence des autres événements suit l'ordre de leur apparition dans le tableau ci-dessus.

Les formulaires et les contrôles Web côté serveur

- ASP.NET apporte à travers son modèle objet une nouveauté qui consiste en une bibliothèque de classes permettant de créer côté serveur des formulaires Web avec des contrôles similaires à ceux utilisés dans les applications Windows classique (les WinForms).

- Les formulaires Web sont créés dans les pages aspx. Ils sont insérés dans ces dernières à l'aide de la balise **`<Form runat="server"> . . .  </Form>`**.

- Un contrôle Web côté serveur est inséré dans un formulaire Web à l'aide de la balise :
  `<asp:Nomcontrôle> … </asp:Nomcontrôle>`.

© Karim Kalti

- Les contrôles Web les plus utilisés fournis par le FrameWork .NET sont résumés dans le tableau suivant :

Composant	Fonctionnalité	Evénements fréquemment utilisés	Code d'ajout du contrôle
Label	Affichage de texte. Le texte peut inclure des balises html.	Les événements sont rarement utilisés.	<asp:Label id=Label1 runat="server">Label</asp:Label>
TextBox	Zone d'édition.	TextChanged	<asp:TextBox id=TextBox1 runat="server"></asp:TextBox>
Image	Affichage d'une image.	Les événements sont rarement utilisés.	<asp:Image id=Image1 runat="server"></asp:Image>
Button	Bouton de commande.	Click, Command (click avec argument).	<asp:Button id=Button1 runat="server" Text="Button"></asp:Button>
ImageButton	Bouton de commande avec image.	Click	<asp:ImageButton id=ImageButton1 runat="server"></asp:ImageButton>
HyperLink	Lien hypertexte pour la navigation Web. Le transfert vers la page de destination est direct à partir du client sans passage par le serveur.	Les événements sont rarement utilisés.	<asp:HyperLink id=HyperLink1 runat="server">HyperLink </asp:HyperLink>
Panel	Panneau permettant de déposer d'autres composants.	Les événements sont rarement utilisés.	<asp:Panel id=Panel1 runat="server">Panel</asp:Panel>
CheckBox	Case à cocher.	CheckChanged	<asp:CheckBox id=CheckBox1 runat="server"></asp:CheckBox>
RadioButton	Case à options.	CheckChanged	<asp:RadioButton id=RadioButton1 runat="server"></asp:RadioButton>
ListBox	Boîte de liste.	SelectedIndexChanged	<asp:ListBox id=ListBox1 runat="server"></asp:ListBox>
DropDownList	Boîte combo (Liste déroulante).	SelectedIndexChanged	<asp:DropDownList id=DropDownList1 runat="server"></asp:DropDownList>
CheckBoxList	Boîte de liste avec case à cocher pour chaque ligne.	CheckChanged	<asp:CheckBox id=CheckBox1 runat="server"></asp:CheckBox>
DataGrid	Grille de données.	CancelCommand, EditCommand, DeleteCommand, ItemCommand, SelectedIndexChanged, PageIndexChanged, SortCommand, UpdateCommand, ItemCreated, ItemDataBound	<asp:DataGrid id=DataGrid1 runat="server"></asp:DataGrid>

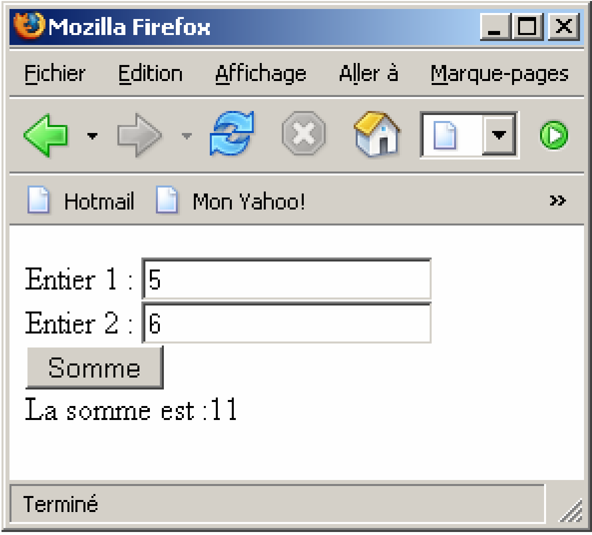
Premier formulaire Web

```
<%@ Page Language="C#"  %>
<script runat="server">

    void BtnSomme_Click(object sender, EventArgs e)
    {
        int Somme;
        if(TxtBE1.Text!="" && TxtBE2.Text!="")
        {
            try
            {
              Somme = Int32.Parse(TxtBE1.Text)+Int32.Parse(TxtBE2.Text);
              LbResultat.Text = "La somme est :" + Somme.ToString();
            }
            catch(Exception)
            { LbResultat.Text = "Prière de vérifier les valeurs saisies";}
        }

    }

</script>
```


© Karim Kalti

```
<html>
<head>
</head>
<body>
    <form runat="server">
        <p>
            <asp:Label id="LbE1" runat="server">Entier 1 : </asp:Label>
            <asp:TextBox id="TxtBE1" runat="server"></asp:TextBox>
            <br/>
            <asp:Label id="LbE2" runat="server">Entier 2 : </asp:Label>
            <asp:TextBox id="TxtBE2" runat="server"></asp:TextBox>
            <br />
            <asp:Button id="BtnSomme" onclick="BtnSomme_Click" runat="server"
             Text="Somme"></asp:Button>
            <br />
            <asp:Label id="LbResultat" runat="server">La somme est : </asp:Label>
        </p>
    </form>
</body>
</html>
```

Résultat de l'exécution



Organisation générale du code

- La première ligne de code du fichier *<%@ Page Language="C#" %>* est une directive qui indique qu'il s'agit d'une page aspx et que le langage utilisé par cette page est le C#.
- Le reste du code source du fichier est divisé en deux parties :
  - La première partie, définie dans l'entête de la page et délimitée par les balises *<script runat="server">* ... *</script>*, représente un script C# qui définit les traitements effectués par le formulaire.
  - La deuxième partie définie dans le corps de la page, représente le code nécessaire à la création de la présentation de la page (Interface). Cette partie comporte essentiellement un formulaire Web.

Ajout du formulaire à la page

- Le formulaire Web est ajouté à la page à l'aide de la paire de balises *<form runat="server">* ... *</form>*.
- L'attribut *runat="server"* indique que le formulaire Web est un formulaire qui doit être traité sur le serveur.

© Karim Kalti

Ajout d'un contrôle au formulaire

- L'ajout des contrôles Web au formulaire se fait à l'aide de la balise :
  <asp:Nomcontrôle . . .> </asp:Nomcontrôle.>.

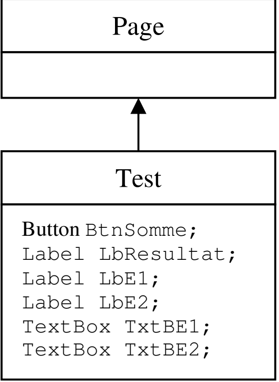
Exemple :
Cet exemple montre l'ajout d'un bouton au formulaire :

```
<asp:Button id="BtnSomme" onclick="BtnSomme_Click" runat="server" Text="Somme"></asp:Button>
```

- *asp:Button* : indique que le contrôle est un bouton.
- L'attribut *Id* permet de spécifier l'identifiant du bouton.
- L'attribut *onclick* permet de spécifier le gestionnaire de l'événement clic sur le bouton. Dans l'exemple courant ce gestionnaire est *BtnSomme_Click.*
- L'attribut *Runat="Server"* : indique que le bouton est géré côté serveur.
- L'attribut *Text* permet de spécifier le texte qui sera affiché sur le bouton.

Modèle de l'objet instancié en mémoire lors de la demande de la page

- La demande d'une page *aspx* comportant un formulaire Web par un client va engendrer la création côté serveur d'une instance d'un objet basé sur cette page et d'un ensemble d'instances des objets représentant les contrôles serveur composant le formulaire.
- L'objet qui représente la page hérite de l'objet *Page* de ASP.NET. Les instances des contrôles Web sont associées à cet objet sous forme d'attributs. Les identifiants des contrôles spécifiés par l'attribut *Id* permettent de référencer les instances de ces contrôles.
- Pour l'exemple courant la structure de l'objet instancié en mémoire suite à la demande de la page *Test.aspx* ressemble à ceci :



Cycle de vie d'une page ASP.NET contenant un formulaire Web

- Le client demande la page pour la première fois.
- Le serveur :
  - Localise la page.
  - Crée en mémoire une instance de l'objet représentant cette page (objet semblable à celui décrit ci-dessus).
  - Traite l'objet (exécution des gestionnaires d'événements, *Page_Init* puis *Page_Load*) et génère la réponse HTML.
  - Renvoie la réponse au client.
  - Détruit l'instance de la page côté serveur.
- Le client :
  - Reçoit la page contenant le formulaire vide.
  - Remplit les champs du formulaire et renvoie les valeurs saisies au serveur suite au clic sur un bouton de soumission. Cette opération s'appelle un *Postback* ou également "*server round-trip*".
- Le serveur :
  - crée en mémoire une nouvelle instance de l'objet représentant cette page.
  - Traite l'objet (exécution des gestionnaires d'événements, *Page_Init* puis *Page_Load* et enfin *ButtonClic*).

© Karim Kalti

La propriété IsPostBack d'une page

- L'objet *Page* sur lequel sont basés les pages ASP.NET possède une propriété intéressante appelée *IsPostBack* qui permet de vérifier si la page en cours de chargement est envoyée pour la première fois au client ou s'il s'agit d'un renvoi de la page suite à un *server round-trip*.

- Cette propriété est surtout utilisée dans le gestionnaire *Page_Load* pour décider de la manière avec laquelle il faut initialiser la page.

Exemple :

- Supposons que l'on souhaite que lors de l'affichage de la page *Test* pour la première fois pour un client, les zones de texte contiennent par défaut la valeur 0. Pour cela il suffit d'affecter la valeur 0 aux deux contrôles *TextBox* lors du chargement de la page (dans le gestionnaire *Page_Load*).

- Toutefois avec cette approche, à chaque fois qu'une instance de la page est créée en mémoire les deux entiers saisis par l'utilisateur seront écrasés et remplacés par 0 et la somme vaudra toujours 0 (car sur le serveur l'événement *Load* se déclenche avant l'événement *Click* associé au bouton). Pour remédier à cela il suffit de faire cette initialisation seulement lors du premier chargement de la page.

Code permettant de gérer l'initialisation

```
<%@ Page Language="C#"  %>
<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
     if( IsPostBack == false)
     {
        TxtBE1.Text="0";
        TxtBE2.Text="0";
     }
   }
  void BtnSomme_Click(object sender, EventArgs e)
   {
      int Somme;
      if(TxtBE1.Text!="" && TxtBE2.Text!="")
        {
          try
           {
             Somme = Int32.Parse(TxtBE1.Text)+Int32.Parse(TxtBE2.Text);
             LbResultat.Text = "La somme est :" + Somme.ToString();
           }
        catch(Exception)
         { LbResultat.Text = "Prière de vérifier les valeurs saisies";}
        }
   }
```


© Karim Kalti

```
</script>
<html>
<head>
</head>
<body>
  <form runat="server">
     <p>
     <asp:Label id="LbE1" runat="server">Entier 1 : </asp:Label>
     <asp:TextBox id="TxtBE1" runat="server"></asp:TextBox>
     <br />
     <asp:Label id="LbE2" runat="server">Entier 2 : </asp:Label>
     <asp:TextBox id="TxtBE2" runat="server"></asp:TextBox>
     <br />
     <asp:Button id="BtnSomme" onclick="BtnSomme_Click" runat="server" Text="Somme">
     </asp:Button>
     <br/>
     <asp:Label id="LbResultat" runat="server">La somme est : </asp:Label>
     </p>
   </form>
</body>
</html>
```

Remarque :

Un fichier *aspx* inclut par défaut les espaces de noms les plus fréquemment utilisés lors de la création d'une page Web. Ces espaces de noms sont : *System*, *System.Web*, *System.Web.UI*, *System.Web.UI.WebControls*. Ceci explique d'ailleurs la possibilité d'utilisation directe des objets "Contrôles Web", des exceptions, etc.

Version "code behind" du code précédent

Fichier Test.aspx

```
<%@ Page Language="C#" Src="Page1.cs" Inherits="Test.Page1" %>
<html>
<head>
</head>
<body>
    <form runat="server">
        <p>
            <asp:Label id="LbE1" runat="server">Entier 1 : </asp:Label>
            <asp:TextBox id="TxtBE1" runat="server"></asp:TextBox>
            <br />
            <asp:Label id="LbE2" runat="server">Entier 2 : </asp:Label>
            <asp:TextBox id="TxtBE2" runat="server"></asp:TextBox>
            <br />
            <asp:Button id="BtnSomme" onclick="BtnSomme_Click" runat="server"
             Text="Somme"></asp:Button>
            <br />
            <asp:Label id="LbResultat" runat="server">La somme est : </asp:Label>
        </p>
    </form>
</body>
</html>
```

Fichier Page1.cs

```
namespace Test
{
    using System;
    using System.Web;
    using System.Web.UI;
    using System.Web.UI.WebControls;

    public class Page1 : Page
    {
        // Références aux contrôles définis dans le fichier aspx
        public Label LbE1;
        public Label LbE2;
        public Label LbResultat;
        public TextBox TxtBE1;
        public TextBox TxtBE2;
```


© Karim Kalti

```
        public Button BtnSomme;

        protected void Page_Load(object sender, EventArgs e)
        {
            if( IsPostBack == false)
                {
                    TxtBE1.Text="0";
                   TxtBE2.Text="0";
               }
        }

        public void BtnSomme_Click(object sender, EventArgs e)
        {
            int Somme;
            if(TxtBE1.Text!="" && TxtBE2.Text!="")
            {
                try
                {
                    Somme = Int32.Parse(TxtBE1.Text)+Int32.Parse(TxtBE2.Text);
                    LbResultat.Text = "La somme est :" + Somme.ToString();
                }
                catch(Exception)
                { LbResultat.Text = "Prire de vrifier les valeurs saisies";}
            }
        }
    }
}
```

Remarque :
Aucun espace de noms n'est inclut par défaut dans un fichier .cs. C'est pourquoi, il est nécessaire de mentionner explicitement les espaces de noms utilisés dans ce type de fichier.

La classe Control

Cette classe est la classe de base de tous les contrôles. (Web Controls et HTML Controls).

Principales propriétés :

Propriété	Type	Signification
ID	string	Tout composant peut porter un nom, ce nom est l'Id. C'est à partir de ce nom qu'il est possible de modifier les attributs du contrôle dans du code c# . Il est interdit d'avoir deux fois le même Id dans une même page asp.NET.
Page	Page	Obtient une référence à l'instance de Page qui contient le contrôle serveur.
Parent	Control	Obtient une référence au contrôle parent du contrôle serveur dans la hiérarchie des contrôles de la page.
Controls	ControlCollection	Obtient un objet ControlCollection qui représente les contrôles enfants d'un contrôle serveur spécifié dans la hiérarchie de l'interface utilisateur.
EnableViewState	Bool	Obtient ou définit une valeur indiquant si le contrôle serveur rend persistant son état d'affichage, ainsi que celui de tous les contrôles enfants qu'il contient, sur le client à l'origine de la demande.
Visible	bool	Obtient ou définit une valeur qui indique si un contrôle serveur est rendu sous la forme d'une interface utilisateur sur la page.

Principales méthodes :

Méthode	Signification
void DataBind()	Lie une source de données au contrôle serveur appelé et à tous ses contrôles enfants.
void Dispose()	Permet à un contrôle serveur d'effectuer le nettoyage final avant qu'il soit libéré de la mémoire.
bool HasControls()	Détermine si le contrôle serveur contient des contrôles enfants.

Principaux événements :

Evénement	Type de délégation	Signification
Init	EventHandler	Cet événement se déclenche lorsqu'un contrôle est chargé en mémoire et initialisé à partir de ses propriétés spécifiées au moment de la définition de l'objet qui représente ce contrôle. Il n'est pas possible d'accéder aux autres contrôles durant cet événement

© Karim Kalti

		(ces derniers ne sont pas nécessairement déjà créés). Pour cette raison, il n'est pas possible d'agir sur les propriétés des contrôles pour définir la disposition par exemple.
`Load`	`EventHandler`	Cet événement se déclenche lorsque le contrôle courant et tous ses contrôles enfants ont été chargés en mémoire (après l'événement *Init*). Il est possible dans ce cas d'accéder à ces contrôles et d'agir sur leurs propriétés pour faire une initialisation dynamique par exemple.
`Unload`	`EventHandler`	Cet événement se déclenche lorsque le contrôle se décharge de la mémoire. Il est exploité pour libérer les ressources utilisées par le contrôle.
`Disposed`	`EventHandler`	Cet événement se déclenche lorsque le contrôle est détruit.

La classe WebControl

Cette classe est la classe de base de tous les contrôles Web côté serveur. Elle définit les méthodes, propriétés et événements communs à tous ces contrôles.

Espace de noms : `System.Web.UI.WebControls`

Classes dérivées

```
System.Object
   System.Web.UI.Control
      System.Web.UI.WebControls.WebControl
         System.Web.UI.WebControls.AdRotator
         System.Web.UI.WebControls.BaseDataList
         System.Web.UI.WebControls.Button
         System.Web.UI.WebControls.Calendar
         System.Web.UI.WebControls.CheckBox
         System.Web.UI.WebControls.DataListItem
         System.Web.UI.WebControls.HyperLink
         System.Web.UI.WebControls.Image
         System.Web.UI.WebControls.Label
         System.Web.UI.WebControls.LinkButton
         System.Web.UI.WebControls.ListControl
         System.Web.UI.WebControls.Panel
         System.Web.UI.WebControls.Table
         System.Web.UI.WebControls.TableCell
         System.Web.UI.WebControls.TableRow
         System.Web.UI.WebControls.TextBox
         System.Web.UI.WebControls.ValidationSummary
```

Principales propriétés communes aux Web contrôles :

Propriété	Type	Signification
`AccessKey`	`string`	C'est la touche servant d'accélérateur (en combinaison avec ALT) pour exécuter l'événement click sur le contrôle. (Exemple AccessKey='A', ALT+A).
`Backcolor`	`enum Color`	Couleur d'arrière plan du contrôle.
`BorderWidth`	`Uint`	Epaisseur de la bordure.
`BorderStyle`	`enum BorderStyle`	Type de contour. Les valeurs possibles sont celles de l'énumération *BorderStyl*e.
`Enabled`	`bool`	Indique si le contrôle est activé ou non.
`ForeColor`	`enum Color`	Couleur d'avant plan du texte.
`Height`	`Uint`	Hauteur du contrôle en pixels.
`TabIndex`	`short`	Ordre du passage du focus d'entrée par les touches de tabulation.
`Width`	`Uint`	Largeur du contrôle en pixels.

Spécification des différents contrôles Web côté serveur

La spécification complète des contrôles Web côté serveur est donnée dans la documentation du FrameWork .NET (Consulter MSDN).

© Karim Kalti

L'objet HttpApplication

- L'objet *HttpApplication* se situe au sommet de la hiérarchie des objets d'une application Web ASP.NET.
- Il permet la configuration et la sauvegarde des informations d'état au niveau de l'application et au niveau des sessions.
- L'objet *HttpApplication* permet à travers ses propriétés d'accéder aux différents autres objets composant une application Web. Le tableau suivant présente quelques exemples de ces objets :

Propriété	Type	Signification
`Application`	`HttpApplicationState`	Permet de sauvegarder des données ayant une portée au niveau "Application".
`Session`	`HttpSessionState`	Permet de sauvegarder des données ayant une portée au niveau "Session".
`Request`	`HttpRequest`	Donne accès à l'objet *Request* qui représente la requête courante de l'utilisateur.
`Response`	`HttpResponse`	Donne accès à l'objet *Response* qui représente la réponse courante envoyée par le serveur au client.
`Server`	`HttpServerUtility`	Cette propriété donne accès à l'objet Server qui fournit des méthodes très pratiques pour l'encodage et le décodage des URL et pour la récupération d'informations sur le serveur.

- Chaque application Web est représentée par un objet qui dérive de la classe *HttpApplication.*
- Cet objet appelé généralement *Global* est automatiquement instancié au moment du démarrage de l'application Web et est libéré lorsque cette dernière se termine.
- L'objet *Global* est défini dans un fichier particulier appelé *Global.asax* de la manière suivante :

```
public class Global : System.Web.HttpApplication
{ . . .}
```

L'objet HttpRequest

- L'objet *HttpRequest* contient les informations renvoyées par le client (navigateur) lorsque ce dernier demande une page de l'application.
- L'objet *HttpRequest* possède des propriétés qui permettent d'accéder à d'autres objets et informations susceptibles d'accompagner la requête du client. Le tableau suivant présente quelques exemples de ces propriétés :

Propriété	Type	Signification
`Browser`	`HttpBrowserCapabilities`	Permet de récupérer des informations sur le browser du client ayant émis la requête.
`Cookies`	`HttpCookieCollection`	Permet de récupérer une collection de cookies à partir du client.
`Files`	`HttpFileCollection`	Reçoit les fichiers téléchargés du client vers le serveur.
`InputStream`	`Stream`	Permet d'avoir accès au contenu du corps de la requête http adressée au serveur.
`ApplicationPath`	`String`	Retourne le chemin virtuel de l'application sur le serveur
`URL`	`classe Uri`	Retourne l'url de la requête courante. La classe Uri permet d'avoir une représentation objet de l'url.
`UserHostAdress`	`string`	Retourne l'adresse IP d'un client distant.
`Params`	`NameValueCollection`	Contient la collection de paramètres qui accompagne la requête. L'accès à un paramètre donné se fait à l'aide d'un indexeur de la collection qui prend comme argument le nom du paramètre et qui retourne sa valeur.

- L'accès par programmation à l'instance de la classe *HttpRequest* qui représente la requête courante adressée au serveur se fait à travers la propriété *Request* de l'objet *Global* automatiquement instancié par dérivation de la classe *HttpApplication* lors du démarrage de l'application.
- Le fait que l'objet *Global* soit global, rend également l'objet *Request* global et directement accessible à partir de n'importe quelle page de l'application Web.

© Karim Kalti

Exemple 1 : Lecture de la valeur d'un cookie

```
protected void Page_Load(object sender, EventArgs e)
  {
    // ce code s'exécute lorsque la page est affichée pour la première fois
    if( IsPostBack == false)
     {
      // Vérification si le navigateur accepte les cookies
      if(Request.Browser.Cookies)
       {
        // Vérification de l'existance d'un cookie appelé UName
        if(Request.Cookies["UName"]!=null)
         // Lecture de la valeur du cookie
         Session["UName"] = Request.Cookies["UName"].Value;
       }

     }
  }
```

Exemple 2 : Récupérer les données d'un GET

Pour rappel, Il existe deux méthodes permettant d'envoyer les données d'un formulaire d'une page Web vers le serveur.

- La méthode GET : qui engendre un envoi des données avec l'url. L'url ressemble alors à ceci :
  `http://NomDomaine/PageScript?champ1=valeur1&champ2=valeur2&champ3=valeur3`
- La méthode POST : qui engendre un envoi des données dans le corps de la requête.

Le choix de la méthode d'envoi est spécifié dans la balise d'ajout du formulaire comme suit :

```
<form id="Form1" method="GET" runat="server">
```

On considère l'url : `http://NomDomaine/PageScript?champ1=valeur1&champ2=valeur2&champ3=valeur3`
La récupération de la valeur du champ1 du formulaire se fait comme suit :

```
Request.Params["champ1"]
```

L'objet HttpResponse

- L'objet *HttpResponse* sert à constituer le flux HTML envoyé par le serveur au client (navigateur) en réponse à une requête de ce dernier.

- L'objet *HttpResponse* possède des propriétés permettant d'accéder à d'autres objets qui peuvent accompagnés ou être utilisés par la réponse. Le tableau suivant présente quelques exemples de ces propriétés :

Propriété	Type	Signification
`Cache`	`HttpCachePolicy`	Détermine la manière avec laquelle le serveur met en cache la réponse (la page) avant de l'envoyer au client.
`Cookies`	`HttpCookieCollection`	Définit le contenu du cookie qui est envoyé au client.
`OutputStream`	`Stream`	Permet d'avoir accès au flux qui contient les données brutes qui composent la réponse envoyée au client.

La classe *HttpResponse* possède quelques méthodes intéressantes qui sont présentées dans le tableau suivant :

Méthode	Signification
`void Clear()`	Efface le contenu de la réponse (le flux OutputStream).
`void Redirect(string)`	Redirige le client vers l'url spécifiée comme argument.
`void Write(string s)`	Ecrit la chaîne s dans le flux constituant la réponse.

- L'accès par programmation à l'instance de la classe *HttpResponse* qui représente la réponse courante du serveur se fait à travers la propriété *Response* de l'objet *Global* automatiquement instancié par dérivation de la classe *HttpApplication* lors du démarrage de l'application.
- Le fait que l'objet *Global* soit global, rend également l'objet *Response* global et directement accessible à partir de n'importe quelle page de l'application Web.

© Karim Kalti

Exemple 1 : Ecriture d'un texte dans une page Web.
ASP.NET fournit un moyen d'insérer directement du texte dans la réponse renvoyée au client et ce à l'aide de la méthode *Write* de la classe *HttpResponse*. Cette méthode est similaire à la fonction *echo* de PHP ou à la méthode *Response.Write* de l'ASP (la version avant .NET).

La méthode *Write* est utilisée comme suit :
```
Resposne.Write ("Ceci est un programme ASP.NET");
```

La chaîne passée comme argument peut être formatée avec du HTML.
```
Resposne.Write ("Ceci est un programme <b>ASP.NET</b>");
```

Toutefois ASP.NET fournit une autre alternative pour afficher le texte dans une page web et qui consiste dans l'utilisation des contrôles Web côté serveur et notamment le contrôle *Label*.

Exemple 2 : Redirection

L'objet *Response* fournit le moyen d'effectuer une redirection par programmation et ce à l'aide de la méthode *Redirect*.
```
Response.Redirect("NouvellePage.aspx");
```

Exemple 3 : Création d'un cookie

```
protected void Page_Load(object sender, EventArgs e)
{
  if( IsPostBack == false)
     {
      // Vérification si le navigateur accepte les cookies
      if(Request.Browser.Cookies)
       {
        // Création d'un cookie
        HttpCookie MyCookie = new HttpCookie("LastVisit");
        MyCookie.Value= DateTime.Now.ToString();
        // cookie valuable 30 jours
        DateTime dt=DateTime.Now;
        TimeSpan ts = new TimeSpan(30,0,0);
        MyCookie.Expires = dt.Add(ts);
        // Ajout du cookie à la réponse
        Response.Cookies.Add(MyCookie);
       }
    }
}
```
Exemple d'application : contrôle d'accès (passage d'arguments à travers l'url)

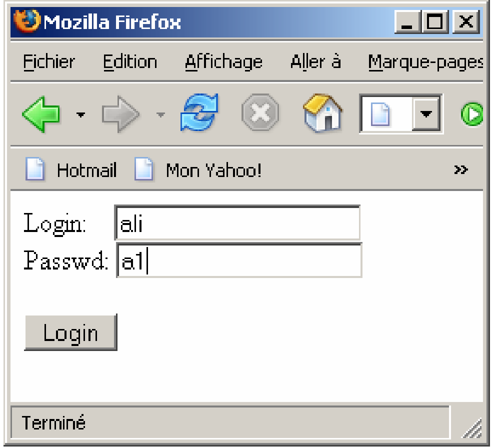
Fichier LogPw.aspx

```
<html>
<head>
<%@ Page Language="C#" %>
<script runat="server">
public void Check(object sender, EventArgs e)
{ string[,] tab={{"ali","a1"},{"salah","s1"}};
  int i=0;
  while (i<2)
  {
    if (TbLogin.Text == tab[i,0] && TbPasswd.Text == tab[i,1])
       Response.Redirect("MesInfos.aspx?Lg=" + TbLogin.Text +"&Pw=" + TbPasswd.Text);
    i++;
  }
  Response.Write("Veuillez saisir une autre fois le login et le password!!");
}
</script>
</head>
<body>
<form runat="server">
Login:    <asp:TextBox id="TbLogin" Runat="server"></asp:TextBox><br/>
Passwd: <asp:TextBox id="TbPasswd" Runat="server"></asp:TextBox><br/><br/>
<asp:Button id="BuLogin" onclick="Check" Runat="server"
Text="Login"></asp:Button>
</form>
</body>
</html>
```

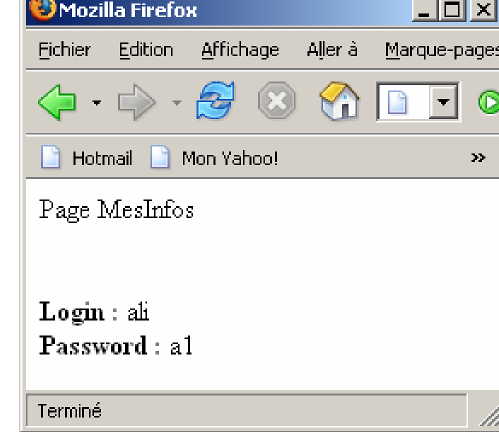

© Karim Kalti

Fichier MesInfos.aspx

```
<html>
<head>
<%@ Page Language="C#" %>
<script runat="server">
void Page_Load(object sender, EventArgs e)
{
  if (Request.Params["Lg"]!=null)
     La.Text = "<br><b>Login :</b> " + Request.Params["Lg"].ToString();
  if (Request.Params["Pw"]!=null)
     La.Text += "<br><b>Password :</b> " + Request.Params["Pw"].ToString();
}
</script>
</head>
<body>
<form runat="server">
Page MesInfos<br>
<asp:Label Runat=server ID=La></asp:Label>
</form>
</body>
</html>
```



LogPw.aspx



MesInfos.aspx

L'objet HttpApplicationState

Rappel :

- Une application Web démarre lors de l'ouverture d'une première session par un client.
- Une application Web peut avoir plusieurs sessions ouvertes en même temps.
- Une session se termine après l'écoulement d'une période d'attente paramétrable (Timeout) après la fermeture du navigateur ou l'annulation de la session par programmation.
- Une application Web se termine suite à l'écoulement d'une période d'attente paramétrable (Timeout) après la terminaison de la dernière session ouverte de l'application.

Les variables d'application

- Lors du développement d'une application Web, il est souvent utile de déclarer des variables ou des objets qui ont une portée globale (visibles dans toute l'application Web). Ces objets sont partagés par toutes les pages de toutes les sessions d'une application. Ils sont appelés des ***variables d'application***.
- La classe *HttpApplicationState* permet de gérer les variables d'application.

Propriété	Type	Signification
`Count`	`Int`	Donne le nombre d'objets stockés dans l'état Application (le nombre de variables d'application).
`Item`	`Indexeur []`	Indexeur permettant d'accéder aux différents objets stockés dans l'état Application. L'index peut être numérique ou une chaîne qui représente le nom de l'objet auquel on veut accéder.

© Karim Kalti

Méthode	Signification
`void Add(string name, object value);`	Ajoute l'objet dont le nom est "name" et la valeur est "value" à la collection des variables d'application.
`void Clear()`	Supprime tous les objets de la collection de variables d'application.
`void Remove(string name)`	Supprime l'objet dont le nom est passé comme argument de la collection de variables d'application.
`void RemoveAt(int index)`	Supprime l'objet dont l'index numérique est passé comme argument de la collection de variables d'application.
`void Lock()`	Effectue un verrouillage de l'état d'application afin de faciliter la synchronisation des accès aux variables d'application.
`void Unlock()`	Effectue un déverrouillage de l'état d'application afin de faciliter la synchronisation des accès aux variables d'application.

- L'accès par programmation à l'instance de la classe *HttpApplicationState* qui représente l'état de l'application courante se fait à travers la propriété *Application* de l'objet *Global* automatiquement instancié par dérivation de la classe *HttpApplication* lors du démarrage de l'application.
- Le fait que l'objet *Global* soit global, rend également l'objet *Application* global et directement accessible à partir de n'importe quelle page de l'application Web.

Utilisation des variables d'application

- La classe *HttpApplicationState* offre la possibilité de déclarer des variables qui sont accessibles à partir de n'importe quel endroit d'une application Web par toutes les sessions ouvertes. Ces variables sont dites des variables d'application.
- La création d'une variable d'application se fait au vol sans déclaration explicite, ni spécification de type.
- La syntaxe utilisée pour faire cette création est :

```
Application["NomVariable"] = Valeur;
```

- La création d'une variable d'application peut se faire n'importe où dans une application Web. Toutefois l'endroit qui est généralement utilisé pour faire cette création est le gestionnaire *Application_Start* dans le fichier *Global.asax*.

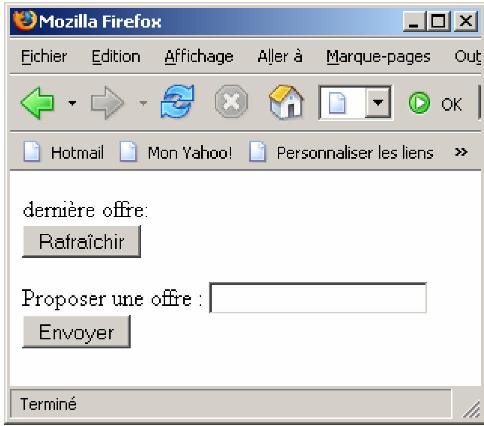
Exemple :

```
<%@ Page Language="C#" %>
<script runat="server">

    void Page_Load(object sender, EventArgs e)
    {
      if(Application["DerniereOffre"]!=null)
        LbDerniereOffre.Text ="La dernière offre est :"+Application["DerniereOffre"];
    }

    void BtnRafraichir_Click(object sender, EventArgs e)
    {
      Response.Redirect("Offre.aspx");
    }

    void BtnEnvoyer_Click(object sender, EventArgs e)
    {
      if(TxtBProposer.Text!="")
      {
        Application["DerniereOffre"]=TxtBProposer.Text;
        LbDerniereOffre.Text ="La dernière offre est :"+Application["DerniereOffre"];
      }
    }
</script>
<html>
<head>
</head>
<body>
 <form runat="server">
 <p>
 <asp:Label id="LbDerniereOffre" runat="server" width="265px">dernière offre:
 </asp:Label> <br />
 <asp:Button id="BtnRafraichir" onclick="BtnRafraichir_Click" runat="server" Width="100px"
   Text="Rafraîchir" Height="25px"></asp:Button>
  </p>
 <p>
```


© Karim Kalti

```
 <asp:Label id="LbProposer" runat="server">Proposer une offre :</asp:Label>
 <asp:TextBox id="TxtBProposer" runat="server"></asp:TextBox><br />
 <asp:Button id="BtnEnvoyer" onclick="BtnEnvoyer_Click" runat="server" Width="100px"
  Text="Envoyer" Height="25px"></asp:Button>
 </p>
 </form>
 </body>
</html>
```

Commentaire :
Ce code crée une interface qui permet à différents utilisateurs de proposer des offres (des entiers numériques) et de voir à chaque fois la dernière offre proposée (bouton Rafraîchir).



Synchronisation des accès à une variable d'application :

Le fait que les variables d'application soient partagées par toutes les sessions peut engendrer des problèmes lors des tentatives d'accès simultanés à ces variables par plusieurs utilisateurs. Par exemple un utilisateur accède à une variable d'application et commence à faire des traitements avec et entre temps un autre utilisateur accède à cette variable et modifie son contenu. Pour éviter ce genre de problème, il faut synchroniser les opérations d'accès à l'aide des méthodes *Lock* et *Unlock*.

Exemple :

```
. . .
Application.Lock();
Application["DerniereOffre"]=TxtBProposer.Text;
LbDerniereOffre.Text ="La Dernière offre est :"+Application["DerniereOffre"];
Application.UnLock();
. . .
```

L'objet HttpSessionState

Les variables de session

- Outre les variables d'application, il existe une autre catégorie de variables globales qui peut être très utile lors du développement d'une application Web. Il s'agit des variables qui ont pour portée une session et qui sont appelées de ce fait les ***variables de session***.
- Une variable de session est partagée par toutes les pages d'une application Web mais pour une session donnée.
- La classe *HttpSessionState* permet de gérer les variables et les paramètres relatifs à une session donnée.

Propriété	Type	Signification
Count	Int	Donne le nombre d'éléments stockés dans l'état d'une session.
Item	Indexeur []	Indexeur permettant d'accéder aux différents éléments stockés dans l'état d'une session. L'index peut être numérique ou une chaîne qui représente le nom de l'élément auquel on veut accéder.
SessionID	String	Récupère un ID permettant d'identifier d'une manière unique une session.
TimeOut	Int	Détermine le temps d'attente (en minutes) qu'il ne faut pas dépasser entre deux requêtes avant qu'une session ne soit terminée.

Méthode	Signification
void Add(string name, object value);	Ajoute l'objet dont le nom est "name" et la valeur est "value" à la

© Karim Kalti

	collection des variables de session.
`void Abandon()`	Annule la session en cours.
`void Clear()`	Supprime tous les objets de la collection de variables de session.
`void Remove(string name)`	Supprime l'objet dont le nom est passé comme argument de la collection de variables de session.
`void RemoveAt(int index)`	Supprime l'objet dont l'index numérique est passé comme argument de la collection de variables de session.

- L'accès par programmation à l'instance de la classe *HttpSessionState* qui représente l'état de la session courante se fait à travers la propriété *Session* de l'objet *Global*.

- Le fait que l'objet *Global* soit global, rend également l'objet *Session* global et directement accessible à partir de n'importe quelle page de l'application Web.

Utilisation des variables de session :

- La création d'une variable de session se fait au vol sans déclaration explicite, ni spécification de type.
- La syntaxe utilisée pour faire cette création est :

```
Session["NomVariable"] = Valeur;
```

- La création d'une variable de session peut se faire n'importe où dans une application Web. Cette variable est alors accessible à partir de n'importe quelle page mais pour une session donnée.
- L'endroit qui est généralement utilisé pour créer les variables de session est le gestionnaire *Session_Start* défini dans le fichier *Global.asax*.

Exemple : Programme "contrôle d'accès" utilisant les variables de session

Il s'agit du même programme de contrôle d'accès présenté précédemment. Cette fois ce programme utilise les variables de session au lieu du passage d'arguments à travers l'url.

Fichier LogPw.aspx

```
<html>
<head>
<%@ Page Language="C#" %>
<script runat="server">
public void Check(object sender, EventArgs e)
{
  string[,] tab={{"ali","a1"},{"salah","s1"}};
  int i=0;
  while (i<2)
  {
    if (TbLogin.Text == tab[i,0] && TbPasswd.Text == tab[i,1])
     {
        Session["Lg"]=TbLogin.Text;
        Session["Pw"]=TbPasswd.Text;
        Response.Redirect("MesInfos.aspx");
     }
     i++;
  }
  Response.Write("Veuillez saisir une autre fois le login et le password!!");
}
</script>
</head>
<body>
<form runat="server">
Login:    <asp:TextBox id="TbLogin" Runat="server"></asp:TextBox><br/>
Passwd: <asp:TextBox id="TbPasswd" Runat="server"></asp:TextBox><br/><br/>
<asp:Button id="BuLogin" onclick="Check" Runat="server"
Text="Login"></asp:Button>
</form>
</body>
</html>
```

Fichier MesInfos.aspx

© Karim Kalti

```
<html>
<head>
<%@ Page Language="C#" %>
<script runat="server">
void Page_Load(object sender, EventArgs e)
{
  if (Session["Lg"]!=null)
     La.Text = "<br><b>Login :</b> " + Session["Lg"];
  if (Session["Pw"]!=null)
     La.Text += "<br><b>Password :</b> " + Session["Pw"];
}
</script>
</head>
<body>
<form runat="server">
Page MesInfos<br><br>
<asp:Label Runat=server ID=La></asp:Label>
</form>
</body>
</html>
```

Remarque:
Les variables de session et d'application sont stockées en mémoire sur le serveur. Il peut y avoir autant de variables que l'on le désire, cependant il ne faut pas perdre de vue qu'avec l'accroissement du nombre de variables utilisées, ce sont souvent les performances qui sont en chute libre. Il faut donc trouver le juste milieu entre performances et facilité. Les informations utilisées de manière répétée, si elles n'ont pas un trop haut coût en mémoire, peuvent donc être stockée dans ce type de variables (par exemple : le UserName de l'utilisateur pour les variables de session, les chaînes et les objets de connexion à la base de données pour les variables d'application).

Le fichier Global.asax

- Le fichier *Global.asax*, connu également comme le fichier d'application, est un fichier optionnel qui contient le code permettant de gérer les événements déclenchés par une application Web ainsi que par ses sessions et ses modules.
- Le fichier *Global.asax* doit résider toujours dans le répertoire racine de l'application Web.
- Le fichier *Global.asax* est configuré de façon à ce que toute requête qui lui est adressée (url) soit rejetée. Ce fichier ne peut ni être téléchargé ni édité par un utilisateur externe.
- Lors du lancement d'une application Web, le fichier *Global.asax* est compilé et ses données sont utilisées pour générer dynamiquement une classe dérivée de la classe *HttpApplication* appelée *Global*.

Remarque :
Le fichier *Global.asax* est optionnel. S'il n'est pas présent pour une application Web alors cela suppose que cette dernière ne gère pas les événements au niveau *Application* et au niveau *Session*. La classe *Global* est dans ce cas implicitement définie et instanciée.

Evénements gérés dans le Global.asax

- Le fichier *Global.asax* permet d'écrire le code de gestion des événements déclenchés par une application ASP.NET et par ses modules (éventuellement des modules personnalisés).
- Toutefois les événements les plus traités dans ce fichier restent ceux liés au niveau *Application* et au niveau *Session*. Les noms des gestionnaires d'événements pour ces deux objets sont prédéfinis et automatiquement reconnus par *Global.asax*. (Si l'attribut *AutoEventWireup* de l'objet Page vaut "*true*", *true* est la valeur par défaut pour cet attribut).
- Le tableau suivant donne quelques exemples de gestionnaires les plus utilisés dans ce cadre ainsi qu'une description des événements qu'ils traitent.

Gestionnaire d'événement	Type de délégation	Description de l'événement traité
`Application_Start` ou également `Application_OnStart`	`EventHandler`	Ce gestionnaire s'exécute lors du lancement de l'application (quant un premier client effectue une première requête à une ressource située dans le répertoire virtuel de l'application). Ce gestionnaire est exploité pour faire les initialisations nécessaires au fonctionnement de l'application comme par exemple la création des variables d'application.

© Karim Kalti

Application_End ou également Application_OnEnd	EventHandler	Ce gestionnaire s'exécute lorsque l'application est terminée. . A toute application est associé un délai d'inactivité, configurable dans [web.config], au bout duquel l'application est considérée comme terminée. C'est donc le serveur web qui prend cette décision en fonction du paramétrage de l'application. Le délai d'inactivité d'une application est défini comme le temps pendant lequel aucun client n'a fait une demande pour une ressource de l'application.
Application_BeginRequest	EventHandler	Ce gestionnaire s'exécute au début de chaque requête envoyée par un client au serveur (demande d'une page). Ce gestionnaire peut être exploité pour examiner la requête avant de l'envoyer à la page demandée. Il est même possible de rediriger la requête vers une autre page.
Application_EndRequest	EventHandler	Ce gestionnaire s'exécute à la fin de chaque requête envoyée par un client au serveur
Application_Error	EventHandler	Ce gestionnaire s'exécute à chaque fois que se produit une erreur non explicitement interceptées dans le code du fichier *Global.asax* ou dans les fichiers *.aspx* et *.cs*. Il constitue un bon moyen pour centraliser le processus de gestion des erreurs pouvant se produire dans toute l'application.
Session_Start ou également Session_OnStart	EventHandler	Ce gestionnaire s'exécute à chaque ouverture d'une nouvelle session.
Session_End	EventHandler	Ce gestionnaire s'exécute à la fermeture de chaque session. (la fermeture d'une session peut être déclenchée par la fermeture du navigateur, ou suite à l'écoulement d'un délai d'inactivité de la session. Une page Web peut proposer également une commande de fermeture explicite de la session à l'utilisateur).

Structure du fichier Global.asax

```
<%@ Application language="C#" %>
<script runat="server">

    public void Application_Start(Object sender, EventArgs e){
        // Code qui s'exécute lors du démarrage de l'application.
    }
    public void Application_End(Object sender, EventArgs e){
        // Code qui s'exécute lorsque l'application se termine
    }
    public void Application_BeginRequest(Object sender, EventArgs e){
        // Code qui s'exécute au début de chaque demande de page
    }
    public void Application_EndRequest(Object sender, EventArgs e){
        // Code qui s'exécute à la fin de chaque demande de page
    }
    public void Application_Error(Object sender, EventArgs e) {
        // Code qui s'exécute lorsqu'une erreur non gérée se produit
    }
    public void Session_Start(Object sender, EventArgs e) {
        // Code qui s'exécute lors du démarrage d'une session
    }
    public void Session_End(Object sender, EventArgs e) {
        // Code qui s'exécute lorsqu'une session se termine
    }
</script>
```

Commentaire :

- La directive utilisée au début du fichier n'est plus *Page* comme cela a été le cas pour les exemples précédents mais *Application* : `<%@ Application language="C#" %>`.
- Avec cette directive, toutes les méthodes qui figurent dans la partie script du fichier seront compilées et ajoutées à la classe *Global* basée sur *HttpApplication.*

© Karim Kalti

Version "Code behind" du fichier Global.asax

Comme les fichiers aspx, le fichier *Global.asax* peut être organisé suivant le modèle "Code Behind" et être ainsi réparti sur deux fichiers *Global.asax* et *Global.asax.cs*.

Contenu du fichier Global.asax

```
<%@ Application src="Global.asax.cs" Inherits="Global" %>
```

Contenu du fichier Global.asax.cs

```
public class Global : System.Web.HttpApplication
{
  public Global() { }
  protected void Application_Start(Object sender, EventArgs e) { }
  protected void Session_Start(Object sender, EventArgs e) { }
  protected void Application_BeginRequest(Object sender, EventArgs e){  }
  protected void Application_EndRequest(Object sender, EventArgs e){  }
  protected void Application_Error(Object sender, EventArgs e){  }
  protected void Session_End(Object sender, EventArgs e){  }
  protected void Application_End(Object sender, EventArgs e){  }
}
```

Exemple d'application 1

L'exemple suivant montre les différents moments d'appel des gestionnaires d'événements *Application_Start*, *Session_Start* et *Application_BeginRequest* définis dans le fichier *Global.asax*.

Fichier Global.asax :

```
<%@ Application language="C#" %>

<script runat="server">

    public void Application_Start(Object sender, EventArgs e) {
        Application["StartApp"] = DateTime.Now.ToString("T");
    }

    public void Session_Start(Object sender, EventArgs e) {
        Session["StartSess"] = DateTime.Now.ToString("T");
    }

    public void Application_BeginRequest(Object sender, EventArgs e) {
        Context.Items["StartReq"] = DateTime.Now.ToString("T");
    }

</script>
```

Fichier principal.aspx

```
<html>
<head>
<%@ Page Language="C#" Debug="true" %>
<script runat="server">

    string jeton;
    string startApplication;
    string startSession;
    string startRequest;

    void Page_Load(object sender, EventArgs e)
    {
        jeton=Session.SessionID;
        startApplication = Application["StartApp"].ToString();
        startSession = Session["StartSess"].ToString();
        startRequest = Context.Items["StartReq"].ToString();
    }

</script>
```


© Karim Kalti

```
</head>
<body>
    jeton de session : <% =jeton %>
    <br />
    début Application : <% =startApplication %>
    <br />
    début Session : <% =startSession %>
    <br />
    début Requête : <% =startRequest %>
    <br />
</body>
</html>
```

Pour voir les différents moments d'exécution des gestionnaires d'événements lancer le fichier *principal.aspx* à partir de plusieurs instances du navigateur sur une même machine ou sur des machines différentes.

Exemple d'application 2

Cet exemple montre comment il est possible en utilisant les variables d'application de déterminer le nombre d'utilisateurs connectés.

Fichier Global.asax :

```
<%@ Application language="C#" %>

<script runat="server">

    public void Application_Start(Object sender, EventArgs e) {
        Application["NbUtilisateurs"] = 0;
    }

    public void Session_Start(Object sender, EventArgs e) {
        Application["NbUtilisateurs"] = Convert.ToInt32(Application["NbUtilisateurs"])+1;
    }

    public void Session_End(Object sender, EventArgs e) {
        Application["NbUtilisateurs"] = Convert.ToInt32(Application["NbUtilisateurs"])-1;
    }

</script>
```

Fichier principal.aspx

```
<%@ Page Language="C#" %>
<html>
<head>
</head>
<body>
    Nombre d'utilisateurs connectés : <% =Application["NbUtilisateurs"] %>
    <br />
</body>
</html>
```

<u>Commentaires :</u>

- La fermeture du navigateur n'engendre pas la terminaison immédiate de la session. Cette dernière se termine en effet après l'écoulement d'un Timeout paramétrable dans le fichier *Web.config*. (fichier de configuration d'une application Web ASP.NET)
- Les variables d'application et de session sont par défaut de type *object*. Ceci explique d'une part la possibilité d'y placer directement des valeurs et d'autre part la nécessiter de les convertir à travers un casting lorsqu'elles sont utilisées dans des expressions.

© Karim Kalti

Accès aux données dans les applications Web ASP.NET

Introduction

- L'accès aux données dans les applications Web ASP.NET se fait à l'aide de ADO.NET comme c'est le cas pour toutes les applications développées en .NET.
- Dans une page Web dynamique qui fait des accès à une base de données, le rôle de ASP se réduit alors au développement de la couche "Présentation" (interface). L'accès et le traitement des données se font en effet à l'aide de ADO.NET en utilisant comme langage C# ou VB.
- Les deux modes d'accès (connecté et déconnecté) supportés par ADO.NET peuvent être utilisés en ASP.NET.

Les paragraphes suivants présentent des exemples de pages Web dynamiques faisant des accès à une base de données pour effectuer différentes opérations de mise à jour et de sélection. La base de données considérée possède les caractéristiques suivantes :

- Nom :"BIBLIO ".
- Format : MS ACCESS.
- Tables : une seule table appelée "OUVRAGE".
- La structure de la table OUVRAGE est comme suit :

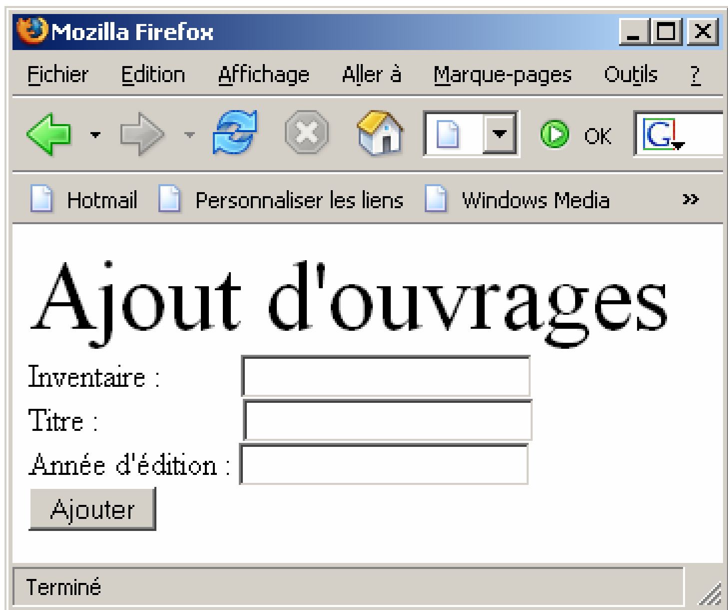
Ouvrage		
Inventaire	Titre	AnnéeEdition
1	Programmation C++	2003
2	Les réseaux	2004
3	Les bases de données	1995
4	Java	1999
5	UML	2004

Exemple 1 : Page d'ajout d'un ouvrage à la base

```
<%@ Page Language="C#" Debug="true" %>
<%@ import Namespace="System.Data" %>
<%@ import Namespace="System.Data.OleDb" %>
<script runat="server">
 void ClickAjouter(object sender, EventArgs e) {

  // Création de la chaîne de connexion
  string StrCnn = "Provider = Microsoft.Jet.OLEDB.4.0; Data Source ="+  MapPath("Biblio.mdb");

 // Création de l'objet de connexion
 OleDbConnection cnn = new OleDbConnection(StrCnn);

 // Ouverture de la connexion
 cnn.Open();

 // Texte de la commande
 string StrSql = "INSERT INTO Ouvrage VALUES ("+
 TxtBInventaire.Text+",'"+TxtBTitre.Text+"',"+ TxtBEdition.Text+")";

 // Création de l'objet Command
 OleDbCommand cmd=new OleDbCommand(StrSql,cnn);
```


© Karim Kalti

```
 // Exécution de la commande
 try
 { cmd.ExecuteNonQuery(); }
 catch(Exception)
 { Response.Write("Erreur d'insertion"); }
 finally
 {
   // Fermeture de la connexion
   cnn.Close();
 }
}

</script>
<html>
<head>
</head>
<body>
    <form runat="server">
        <div align="left"><font size="7">Ajout d'ouvrages</font>
        </div>
        <div align="left">
        </div>
        <div align="left"><asp:Label id="LbInventaire" runat="server">Inventaire :
</asp:Label>         <asp:TextBox
id="TxtBInventaire" runat="server" Width="162px"></asp:TextBox>
            <br />
            <asp:Label id="LbTitre" runat="server">Titre :
</asp:Label>             &nbs
p;  
            <asp:TextBox id="TxtBTitre" runat="server" Width="159px"></asp:TextBox>
            <br />
            <asp:Label id="LbEdition" runat="server">Année d'édition : </asp:Label>
            <asp:TextBox id="TxtBEdition" runat="server" Width="160px"></asp:TextBox>
        </div>
        <div align="left">
        </div>
        <div align="left">
            <asp:Button id="BtnAjouter" onclick="ClickAjouter" runat="server" Width="86px"
Text="Ajouter"></asp:Button>
        </div>
        <!-- Insert content here -->
    </form>
</body>
</html>
```



© Karim Kalti

Exemple 2 : Affichage de la liste des ouvrages

```
<%@ Page Language="C#" Debug="true" %>
<%@ import Namespace="System.Data" %>
<%@ import Namespace="System.Data.OleDb" %>
<script runat="server">

    void Page_Load(object sender, EventArgs e) {

        string    StrCnn    =    "Provider    =    Microsoft.Jet.OLEDB.4.0;    Data    Source
="+MapPath("Biblio.mdb");
        OleDbConnection cnn = new OleDbConnection(StrCnn);
         string StrSql = "SELECT * FROM Ouvrage";
        OleDbDataAdapter DA;
        DataSet DS;
        cnn.Open();
        DA = new OleDbDataAdapter(StrSql,cnn);
        DS = new DataSet();
        DA.Fill(DS,"LesInventaires");
        DG.DataSource = DS;
        DG.DataMember = DS.Tables["LesInventaires"].ToString();
        DG.DataBind();

    }

</script>
<html>
<head>
</head>
<body>
    <form runat="server">
        <p>
            <font size="7">Liste des ouvrages</font>
        </p>
        <p>
            <font size="7">
            <asp:DataGrid id="DG" runat="server" EnableViewState="False" OnLoad="Page_Load">
                <HeaderStyle   font-bold="True"   horizontalalign="Center"   forecolor="Black"
verticalalign="Middle" backcolor="Silver"></HeaderStyle>
                <AlternatingItemStyle                               horizontalalign="Center"
backcolor="WhiteSmoke"></AlternatingItemStyle>
                <ItemStyle          horizontalalign="Center"          verticalalign="Middle"
backcolor="Beige"></ItemStyle>
            </asp:DataGrid>
            </font>
        </p>
    </form>
</body>
</html>
```


© Karim Kalti

Inventaire	Titre	AnnéeEdition
1	Programmation C++	2003
2	Les réseaux	2004
3	Les bases de données	1995
4	Java	1999
5	UML	2004

Exemple 3 : Page de suppression d'un ouvrage de la base

```
<%@ Page Language="C#" Debug="true" %>
<%@ import Namespace="System.Data" %>
<%@ import Namespace="System.Data.OleDb" %>
<script runat="server">

void Page_Load(object sender, EventArgs e) {
 if(IsPostBack==false)
 {
    string StrCnn = "Provider = Microsoft.Jet.OLEDB.4.0; Data Source ="+MapPath("Biblio.mdb");
    OleDbConnection cnn = new OleDbConnection(StrCnn);
    cnn.Open();
    string StrSql = "SELECT * FROM Ouvrage";
    OleDbDataAdapter DA;
    DataSet DS;
    DA = new OleDbDataAdapter(StrSql,cnn);
    DS = new DataSet();
    DA.Fill(DS,"LesInventaires");
    CmbInventaire.DataSource=DS.Tables["LesInventaires"];
    CmbInventaire.DataTextField="Inventaire";
    CmbInventaire.DataBind();
    TxtBTitre.Text=
    DS.Tables["LesInventaires"].Rows[CmbInventaire.SelectedIndex]["Titre"].ToString();
    TxtBEdition.Text=
    DS.Tables["LesInventaires"].Rows[CmbInventaire.SelectedIndex]["AnnéeEdition"].ToString();
    cnn.Close();
  }
}

void ClickSupprimer(object sender, EventArgs e) {
    string StrCnn = "Provider = Microsoft.Jet.OLEDB.4.0; Data Source ="+MapPath("Biblio.mdb");
    OleDbConnection cnn = new OleDbConnection(StrCnn);
    OleDbDataAdapter DA;
    DataSet DS;
    cnn.Open();
    string StrSql = "DELETE FROM Ouvrage WHERE Inventaire = "+ CmbInventaire.SelectedValue;
    OleDbCommand cmd=new OleDbCommand(StrSql,cnn);
```


© Karim Kalti

```
    try
    { cmd.ExecuteNonQuery(); // Suppression
      DA = new OleDbDataAdapter("SELECT * FROM Ouvrage",cnn);
      DS = new DataSet();
      DA.Fill(DS,"LesInventaires");
      CmbInventaire.DataSource=DS.Tables["LesInventaires"];
      CmbInventaire.DataTextField="Inventaire";
      CmbInventaire.DataBind();
      TxtBTitre.Text=
      DS.Tables["LesInventaires"].Rows[CmbInventaire.SelectedIndex]["Titre"].ToString();
      TxtBEdition.Text=
     DS.Tables["LesInventaires"].Rows[CmbInventaire.SelectedIndex]["AnnéeEdition"].ToString();
    }
    catch(Exception)
    {  Response.Write("Erreur de suppression"); }
    finally
    {  cnn.Close();  }
}
void Index_Changed(object sender, EventArgs e) {
   string StrCnn = "Provider = Microsoft.Jet.OLEDB.4.0; Data Source ="+MapPath("Biblio.mdb");
   OleDbConnection cnn;
   OleDbDataReader rd;
   OleDbCommand cmd;
   cnn = new OleDbConnection(StrCnn);
   cnn.Open();
   string StrSql = "SELECT * FROM Ouvrage WHERE Inventaire = "+CmbInventaire.SelectedValue;
   try
   {
     cmd = new OleDbCommand(StrSql,cnn);
     rd = cmd.ExecuteReader();
     if(rd!=null)
      {
        if(rd.Read())
         {
           TxtBTitre.Text=rd["Titre"].ToString();
           TxtBEdition.Text=rd["AnnéeEdition"].ToString();
         }
      }
   }
   catch(Exception)
   { Response.Write("erreur");}
   finally
   { cnn.Close();}
}
</script>
<html>
<head>
</head>
<body>
    <form runat="server">
     <div align="left"><font size="6">Suppression d'ouvrages</font>
     </div>
     <div align="left">
     </div>
     <div align="left"><asp:Label id="LbInventaire" runat="server">Inventaire :
     </asp:Label>          
     <asp:DropDownList id="CmbInventaire" runat="server" Width="160px"
          OnSelectedIndexChanged="Index_Changed" AutoPostBack="True"></asp:DropDownList>
     <br/>
     <asp:Label id="LbTitre" runat="server">Titre : </asp:Label>    
                       
           <asp:TextBox id="TxtBTitre" runat="server" Width="161"></asp:TextBox>
     <br/>
     <asp:Label id="LbEdition" runat="server">Année d'édition : </asp:Label> 
     <asp:TextBox id="TxtBEdition" runat="server" Width="161"></asp:TextBox>
     </div>
     <div align="left">
     </div>
     <div align="left">
     <asp:Button id="BtnSupprimer" onclick="ClickSupprimer" runat="server" Width="86px"
          Text="Supprimer"></asp:Button>
     </div>
   </form>
</body>
</html>
```


© Karim Kalti

- Le code de l'exemple de suppression a été délibérément écrit de façon à montrer la possibilité d'utiliser les modes connecté et déconnecté de ADO.NET dans ASP.NET. Ainsi dans certains gestionnaires on trouve parfois des accès à la base à l'aide des objets *Command* d'une part et à l'aide du *DataAdpeter et du DataSet* d'autre part. Il serait plus judicieux d'utiliser un seul mode par gestionnaire.

- Il est à noter par ailleurs que chaque gestionnaire a utilisé ses propres objets de connexion à la base. Il est possible d'optimiser encore plus le code en proposant une autre conception qui fait partager tous ces objets par les différents gestionnaires.

Exercices

Exercice d'application 1 :

Proposer une version "code behind" des trois pages d'ajout, de suppression et de consultation d'ouvrages.

Exercice d'application 2 :

Proposer une nouvelle conception du code de la page de suppression d'ouvrages utilisant seulement un mode de travail (connecté ou déconnecté) et permettant de faire partager les objets de connexion à la base par tous les gestionnaires.

Exercice d'application 3 :

Proposer une nouvelle conception du code de l'application (ajout, suppression et consultation d'ouvrages) de façon à permettre le partage des mêmes objets de connexion par toutes les pages de l'application.

© Karim Kalti

Le fichier de configuration Web.config

Vue d'ensemble

- ASP.Net propose un modèle de configuration des applications Web basé sur l'utilisation de fichiers de configuration séparés de l'application.
- Ce système fournit une infrastructure de configuration hiérarchique permettant la définition et l'utilisation de données de configuration extensibles dans une application, un site et/ou un ordinateur.
- Ce modèle présente plusieurs avantages :
  - Les données de configuration sont enregistrées dans des fichiers de texte brut. Les administrateurs et les développeurs peuvent utiliser n'importe quel éditeur de texte, analyseur XML ou langage de script standard pour interpréter et mettre à jour ces données de configuration.
  - Les modifications apportées aux fichiers de configuration ASP.NET sont automatiquement détectées par le système et appliquées sans intervention de la part de l'utilisateur (en d'autres termes, un administrateur ne doit pas redémarrer le serveur Web ou réamorcer l'ordinateur pour qu'elles entrent en vigueur).

Modèle hiérarchique des fichiers de configuration

- Les fichiers de configuration ASP.NET sont des fichiers texte basés sur XML, portant chacun le nom ***Web.config***, et pouvant apparaître dans n'importe quel répertoire d'un serveur d'application Web ASP.NET.
- Chaque fichier *Web.config* applique des paramètres de configuration au répertoire dans lequel il se trouve et à tous ses répertoires virtuels enfants. Les paramètres des répertoires enfants peuvent éventuellement substituer ou modifier les paramètres spécifiés dans les répertoires parents.
- Le fichier de configuration racine : **WinNT\Microsoft.NET\Framework\<version>\config\ machine.config**, fournit les paramètres de configuration par défaut pour la totalité de l'ordinateur.
- Au moment de l'exécution, ASP.NET utilise ces fichiers de configuration *Web.config* pour calculer de manière hiérarchique une collection unique de paramètres pour chaque demande cible d'URL entrante (ces paramètres ne sont calculés qu'une seule fois, avant d'être mis en cache pour les demandes suivantes. ASP.NET recherche automatiquement les modifications du fichier et invalide le cache si un des fichiers de configuration est modifié).

<u>Exemple :</u>

Les paramètres de configuration de l'URL http://myserver/myapplication/mydir/page.aspx sont calculés en appliquant les paramètres du fichier *Web.config* dans l'ordre suivant :

```
Base configuration settings for machine.
C:\WinNT\Microsoft.NET\Framework\v.1.00\config\machine.config

Overridden by the configuration settings for the site (or the root application).
C:\inetpub\wwwroot\Web.config

Overridden by application configuration settings.
D:\MyApplication\Web.config

Overridden by subdirectory configuration settings.
D:\MyApplication\MyDir\Web.config
```

Si un fichier *Web.config* est présent dans le répertoire racine d'un site, par exemple « Inetpub\wwwroot », ses paramètres de configuration s'appliquent à toutes les applications de ce site. Certains de ces paramètres peuvent être redéfinis à l'intérieur d'une application et ce à l'aide d'un fichier *Web.config* spécifique à cette application et situé à la racine de cette dernière.

<u>Remarque 1 :</u>

La présence d'un fichier *Web.config* à la racine d'une application ou dans un répertoire donné est complètement facultative. Si aucun fichier *Web.config* n'est présent, tous les paramètres de configuration du répertoire sont automatiquement hérités du répertoire parent.

© Karim Kalti

Remarque 2 :
ASP.NET configure IIS pour éviter que le navigateur puisse accéder directement aux fichiers *Web.config* afin de garantir que leurs valeurs ne risquent pas d'être publiques. En cas de tentative d'accès à ces fichiers, ASP.NET retourne l'erreur 403: Accès interdit.

Structure d'un fichier de configuration

Un fichier *Web.config* est un fichier texte basé sur XML, contenant des éléments de document XML standard, notamment des balises au format adéquat, des commentaires, du texte, etc.

La racine du document :

L'élément racine d'un fichier *Web.config* est toujours une balise **<configuration>**. Les paramètres ASP.NET et d'utilisateur final sont ensuite encapsulés dans la balise de la manière suivante :

```
<configuration>
    <!- les paramètres de configurations doivent être ici -->
</configuration>
```

Les groupes de sections :

Le corps du fichier de configuration est organisé d'une manière hiérarchique en groupes de sections. Chaque groupe de sections est défini par une balise qui porte le nom du groupe.

Exemple :
System.Web est un groupe de section qui contient les sections de configuration les plus fréquemment utilisées.

```
<configuration>
    <System.Web>
       <!- les paramètres de configurations doivent être ici -->
    </System.Web>
</configuration>
```

Un groupe de sections peut comporter d'autres sous-groupes de sections.

Remarque :
Les groupes de sections sont facultatifs. Ils ont essentiellement un intérêt organisationnel permettant de regrouper les sections de configuration d'une manière thématique.

Section de configuration

Une section de configuration est une zone qui comprend des paramètres de configuration. Cette zone est définie par une paire de balises qui porte le nom de la section. Les paramètres sont définis sous forme d'attributs de cette balise.

Les sections de configuration les plus utilisées dans le groupe de sections *system.web* sont :

Section	Description
sessionState	Permet de gérer les sessions au sein de l'application Web.
compilation	Permet de gérer les paramètres de compilation pour ASP.NET.
trace	Permet de gérer le suivi ASP.NET.

Exemple :

```
<configuration>
    <System.Web>
        <SessionState>
               <!- les paramètres de configurations doivent être ici -->
        </SessionState>
        <Compilation>
               <!- les paramètres de configurations doivent être ici -->
        </Compilation>
    </System.Web>
</configuration>
```


© Karim Kalti

Centralisation de la configuration

- Grâce à l'organisation hiérarchique des fichiers de configuration et à la possibilité d'héritage des paramètres, il est possible de centraliser tous les paramètres de toutes les applications Web dans un seul fichier *Web.config* placé à la racine du serveur Web.
- L'accès aux paramètres spécifiques à chaque application se fait alors à l'aide d'une balise **Location** à laquelle on passe le chemin de l'application comme suit :

```
<configuration>
  <location path="WebApp1">
    <System.Web>
       <!- les sections de configuration de WebApp1 -->
    </System.Web>
  <location path="WebApp1">
  <location path="WebApp2">
    <System.Web>
       <!- les sections de configuration de WebApp2 -->
    </System.Web>
  <location path="WebApp2">
</configuration>
```

Verrouillage des paramètres de configuration

Outre la spécification des informations de chemin d'accès à l'aide de la balise **<location>**, il est également possible de configurer la sécurité afin que les paramètres ne puissent pas être substitués par un autre fichier de configuration situé à un niveau inférieur de la hiérarchie de la configuration. Pour verrouiller un groupe de paramètres, il faut spécifier un attribut **allowOverride** sur la balise **<location>** qui l'entoure et lui affecter la valeur *false*. Le code suivant verrouille les paramètres d'emprunt d'identité pour deux applications différentes.

```
<configuration>
  <location path="app1" allowOverride="false">
    <system.web>
        <identity impersonate="false" userName="app1" password="app1pw" />
    </system.web>
  </location>

  <location path="app2" allowOverride="false">
    <system.web>
        <identity impersonate="false" userName="app2" password="app2pw" />
    </system.web>
  </location>
</configuration>
```

Si un utilisateur tente de substituer ces paramètres dans un autre fichier de configuration, le système de configuration affiche une erreur :

```
<configuration>
  <system.web>
      <identity userName="developer" password="loginpw" />
  </system.web>
</configuration>
```

Paramètres personnalisés

Il est possible d'ajouter dans le fichier de configuration des paramètres personnalisés comme la chaîne de connexion à une base de données par exemple.
Les paramètres personnalisés sont placés dans une section de configuration spéciale définie par la balise **appSettings**.

Exemple :

```
<configuration>
  <appSettings>
    <add key="ConnectionString" value="c:\Data\..."/>
    <add key="Nom" value="Dupond"/>
  </appSettings>
</configuration>
```


© Karim Kalti

- L'ajout d'un paramètre personnalisé se fait à l'aide de la balise <add> comme suit :

```
<add key="identificateur" value="valeur"/>
```

Accès aux paramètres de configuration par programmation

ASP.NET fournit des classes qui permettent de récupérer les valeurs des paramètres de configuration par programmation à l'intérieur de l'application. Il fournit même grâce à la classe *ConfigurationSettings* un moyen pour récupérer les valeurs des paramètres personnalisés définis par l'utilisateur.

L'accès aux paramètres se fait à l'aide de la propriété *AppSettings* de la classe *ConfigurationSettings*.

Propriété	Type	Signification
AppSettings	Collection NameValueCollection	Permet de récupérer la liste des paramètres définis dans la section appSettings sous forme d'une collection de paires (paramètre/valeur). La collection NameValueCollection possède un indexeur qui prend comme argument le nom du paramètre et qui retourne sa valeur.

Exemple :

```
using System.Configuration;
. . .
string strcnn =ConfigurationSettings.AppSettings["ConnectionString"];
```


© Karim Kalti

Les services Web

Introduction

La notion de composant logiciel

Un composant logiciel est un module informatique qui effectue une tâche bien donnée et qui s'intègre par programmation dans une application de plus grande taille.

Caractéristiques des composants logiciels

- Les composants exposent généralement un certain nombre de classes instanciables et de méthodes invocables par programmation par les applications qui intègrent ces composants. Ces classes et méthodes sont appelées les interfaces du composant.
- Un composant cache aux applications qui l'intègrent ses membres privés.
- Les composants sont compilés. Ils cachent par conséquent les détails de leur implémentation.
- Un composant logiciel classique doit se trouver dans la même machine que l'application qui l'appelle.

Exemples de technologies utilisées pour la création de composants

- Il existe plusieurs technologies permettant la création de composants logiciels comme par exemple : les *dll*, les *ActiveX*, les objets *COM*, et les *Java Beans*.
- Certaines de ces technologies sont indépendantes des langages comme par exemple la technologie COM et les ActiveX. Un composant issu de ces technologies peut être écrit dans un langage et appelé par un programme écrit dans un autre langage. D'autres technologies sont spécifiques à un langage bien donné (les Beans pour le langage JAVA).

Avantages des composants

- Réutilisation.
- Accélération du processus de développement et réduction des coûts.
- Facilité de maintenance des applications.

Les composants distribués

Aujourd'hui l'informatique de l'entreprise est résolument devenue distribuée. Les "gros" systèmes informatiques sont le plus souvent répartis sur plusieurs sites physiquement distribués. Par ailleurs, grâce à l'évolution des technologies Client/Serveur, de plus en plus d'applications offrent la possibilité de consultation de leurs services à distance (exemple : consultation des comptes bancaires à partir des mobiles ou de l'Internet). Le développement de ce genre d'applications a engendré la nécessite de l'utilisation de composants logiciels invocables à distance.

Technologies utilisées

- La première technologie qui a été utilisée pour créer des composants invocables à distance est la technologie RPC (*Remote Procedure Call*).
- D'autres technologies orientées objet ont également vu le jour par la suite : DCOM, CORBA, RMI.
- La technologie des services Web est le dernier né dans ce genre de technologies d'accès distants aux composants logiciels programmables.

Qu'est ce qu'un service Web

- Un Service Web est une unité logique applicative programmable accessible via les protocoles standards de l'Internet.
- En d'autres mots, c'est une «librairie» fournissant des données et des services à d'autres applications à travers un réseau intranet ou extranet en utilisant les technologies de l'Internet.

© Karim Kalti

Objectifs

- Accès rapide, intégré et généralisé à l'information en interne (Intranet) ou en externe (Internet).
- Interopérabilité : Les services Web sont indépendants de la plateforme (unix, windows, …) et du langage de programmation (C#, VB.NET, Java, …). Leur capacité de faire converser entre elles des applications et des composants hétérogènes est remarquable. On peut très bien réaliser un service Web fonctionnant sous GNU/Linux en Java et l'interroger depuis une page Web ASP.NET en même temps que depuis une application Perl.
- Réutilisation : l'utilisation des composants distribués permet d'accélérer le processus de développement d'applications et réduit par conséquent les coûts.

Exemples de services Web

- Diffusion d'informations : (horaires des vols, des trains, cours de la bourse, états de stocks, incidents, …).
- Commerce électronique : (présentation, transactions, paiement, recherche, …)
- Authentification d'accès.

Les Architectures Orientées Services (SOA)

Présentation

- Une architecture orientée services (notée *SOA* pour **S**ervices **O**riented **A**rchitecture) est une architecture logicielle s'appuyant sur un ensemble de services simples. La notion de service désignant ici une fonction encapsulée dans un composant que l'on peut interroger à l'aide d'une requête composée d'un ou plusieurs paramètres et fournissant une ou plusieurs réponses. Idéalement chaque service doit être indépendant des autres afin de garantir sa réutilisabilité et son interopérabilité.

- L'idée sous-jacente derrière les *SOA* est de cesser de construire les applications de l'entreprise sous forme d'un système tout-en-un pour faire en sorte de construire une architecture logicielle globale décomposées en services correspondant chacun à un des processus métiers de l'entreprise. Le rôle de ces architectures globales revient alors à décrire finement le schéma d'interaction entre ces services.

- Lorsque l'architecture SOA s'appuie sur des services Web, on parle alors de *WSOA*, pour **W**eb **S**ervices **O**riented **A**rchitecture).

Avantages d'une architecture orientée service

Une architecture orientée services permet d'obtenir tous les avantages d'une architecture client-serveur et notamment :

- Une modularité permettant de remplacer facilement un composant (service) par un autre.
- Une réutilisation possible des composants (par opposition à un système tout-en-un fait sur mesure pour une organisation).
- De meilleures possibilités d'évolution (il suffit de faire évoluer un service ou d'ajouter un nouveau service).
- Une plus grande tolérance aux pannes et une maintenance facilitée

Les intervenants dans le processus d'utilisation d'un service Web

- ***Le client (Service Requester)*** : le demandeur de service. C'est une application cliente qui se connecte à un service et invoque ses fonctions.
- ***L'annuaire de services (Service Registry)*** : c'est l'annuaire des services publiés par les fournisseurs de services. L'annuaire est géré sur un serveur au niveau Intranet ou Internet. L'annuaire permet de retrouver le service et de récupérer les informations qui le concernent.
- ***Le fournisseur de service (Service Provider)*** : Composant s'exécutant sur un serveur d'applications et qui fournit le service demandé.

© Karim Kalti

Le scénario complet d'utilisation d'un service Web

Le scénario d'utilisation d'un service comporte deux grande phases : la publication et la consommation du service.

La publication du service

Etape 1: **Définition et description du service :**

- il s'agit d'une description d'un point de vue informatique du service Web. La description est faite en WSDL au sein du fournisseur de services.

Etape 2 : **Publication du service**

- Une fois le service défini et décrit en termes de mise en œuvre, il peut être déclaré dans un annuaire. Il s'agit alors de la publication du service afin de le rendre accessible aux clients.
- La publication est faite au sein d'un annuaire dédié.

La Consommation du service

Etape 1 : **Recherche du service** (1 et 2 sur la figure)
Le client se connecte à un annuaire pour effectuer une recherche de service.

Etape 2 : **Etablissement de contrats** (3 et 4 sur la figure)
Une fois le service trouvé par le client, ce dernier doit établir un contrat avec le fournisseur. Ce contrat consiste en un document WSDL transmis par le fournisseur au client et indiquant les règles d'utilisation que doit respecter le client s'il veut exploiter le service.

Etape 3 : **Mise en œuvre du service** (5 et 6 sur la figure)
Le client peut invoquer le service suivant les conditions du contrat mentionnées dans le document WSDL et le fournisseur renvoie le résultat.

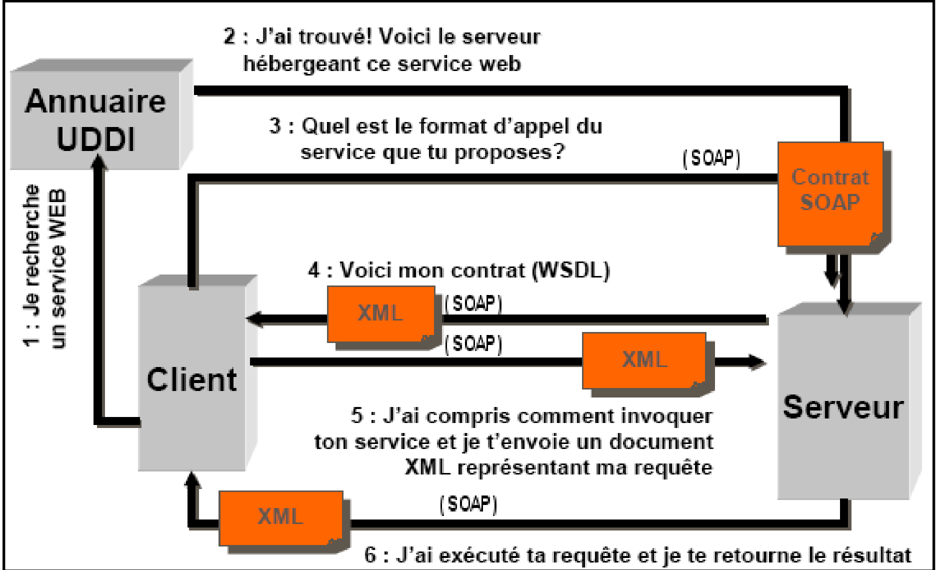


Figure 1 : scénario complet d'utilisation d'un service

Remarque : la composition de services

Un service Web peut jouer le rôle de client à un autre service Web. On parle dans ce cas de composition de services.

© Karim Kalti

Pile de protocoles standards utilisés par les services Web

Les protocoles standards utilisés par les services Web sont : UDDI, WSDL, SOAP, XML, HTTP. Tous ces protocoles sont disposés en couches comme le montre la figure ci-dessous :

UDDI (standard) ou **DISCO** (MS)	Publication et recherche de services
WSDL (basé sur XML et XSD)	Description des services
SOAP (basé sur XML)	Communication (échange de messages)
Protocole de transport (http, SMTP, …)	N'importe quel protocole (généralement le HTTP)

Le WDSL : Web Service Description Language

Le WSDL est un langage de description de services Web basé sur XML. La description est faite en deux niveaux d'abstraction :

- Le niveau interface du service. Il s'agit d'une description indépendante des détails de l'implémentation du service. Elle permet de garantir l'interopérabilité du service (neutralité par rapport à la plateforme et au langage). La description concerne dans ce cas :
  - Les types : il s'agit des types de données manipulés par le service. La description est faite en XSD.
  - Les messages : un message décrit la nature des données échangées entre le service et le client. Cette description indique le type de ces données, et le rôle qu'elles jouent dans l'échange. Par exemple l'interaction avec une méthode du service nécessite deux messages, un décrivant l'appel de la méthode (*Request*) et les données à passer comme arguments et l'autre décrivant la réponse de la méthode (*Response*). Les données échangées dans chaque message sont définies avec les éléments "*Part*".
  - Les types de ports : le type de port décrit l'ensemble des opérations abstraites d'un service. Une opération abstraite est une description d'une méthode du service en terme de paramètres en entrée et de valeur de retour. En d'autres mots une opération abstraite est une description de la signature d'une méthode du service sous forme de messages (en fait c'est une séquence de deux messages un pour l'appel de la méthode et l'autre pour le retour).
  - Les liaisons : une liaison décrit les détails nécessaires à l'appel effectif d'une opération abstraite. Ces détails concernent la technique d'invocation distante de l'opération, le protocole utilisé pour recevoir les données en entrée et pour renvoyer la valeur de retour (http, RPC, …).
- Le niveau implémentation du service : cette description s'intéresse aux informations permettant de localiser le service sur un serveur donné. Elle concerne :
  - Les ports : un port (appelé également *endpoint*) représente une adresse *url* qui est associée à l'implémentation de l'une des liaisons d'un service par un fournisseur.
  - Le service : Le service est décrit comme une collection de ports. Chaque port désigne l'adresse d'une des liaisons du service.

Remarque :
Un même service Web spécifié par son interface peut être implémentée par plusieurs fournisseurs. Les niveaux d'abstraction qu'offre le WSDL (notamment la distinction entre la description de l'interface et de l'implémentation) permettent de décrire d'une manière unique un même service et de référencer séparément ses différents fournisseurs (implémentations).

© Karim Kalti

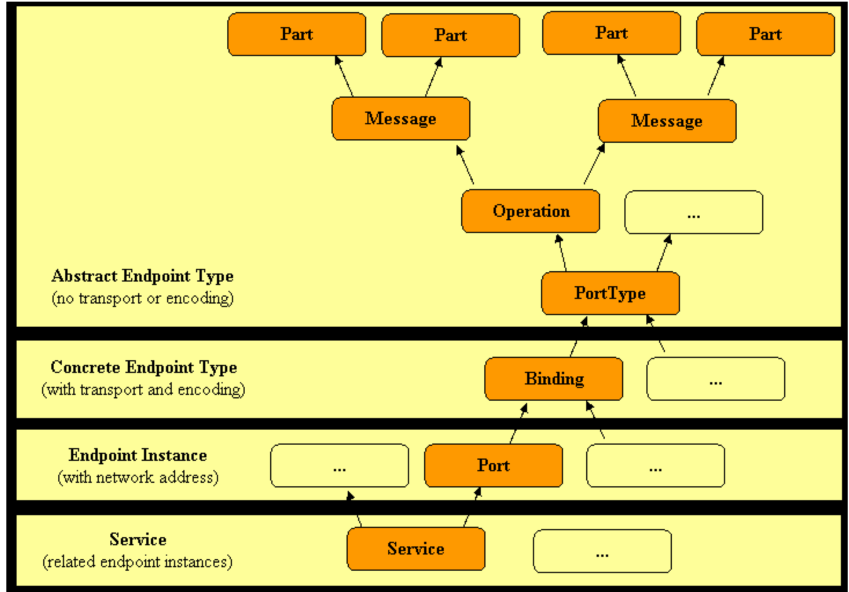


Figure 2 : Eléments de la grammaire XML définissant le WSDL

Le SOAP : Simple Object Access Protocol

SOAP est un protocol permettant d'échanger les informations dans un environnement distribué et hétérogène. Ce protocole basé sur XML permet de définir :

- Le format des messages d'informations échangés entre le demandeur et le fournisseur du service. Les informations en question concernent essentiellement les noms des méthodes à invoquer, les valeurs des paramètres à leur faire passer et les valeurs renvoyées par ces méthodes.
- La manière avec laquelle sont envoyées les données.
- La manière avec laquelle sont reçues les réponses.
- L'encodage des données.

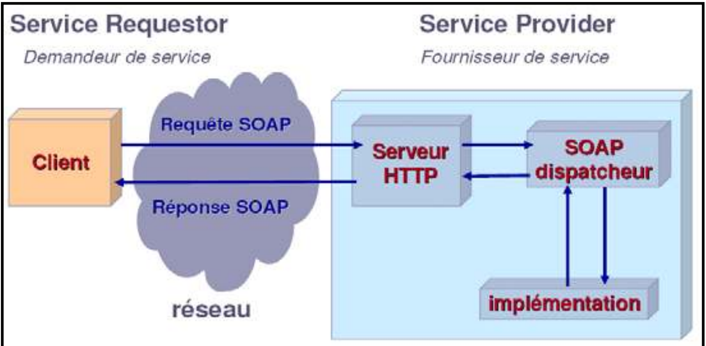


Figure 3 : Communication SOAP entre le client et le service

Eléments d'un message SOAP

- Enveloppe : C'est l'enveloppe du message. Elle encapsule les sous éléments du message.
- L'entête (*header*) : c'est sous élément optionnel de l'enveloppe. Il peut contenir des extensions d'informations telles que l'authentification, les sessions, …
- Le corps (*body*) : c'est un sous élément obligatoire de l'enveloppe. Il contient des informations sur la méthode à invoquer ainsi que ses paramètres (*Request message*) ou sur la valeur de retour (*Response message*). Il peut contenir un élément "*Fault*" en cas d'erreur.

© Karim Kalti

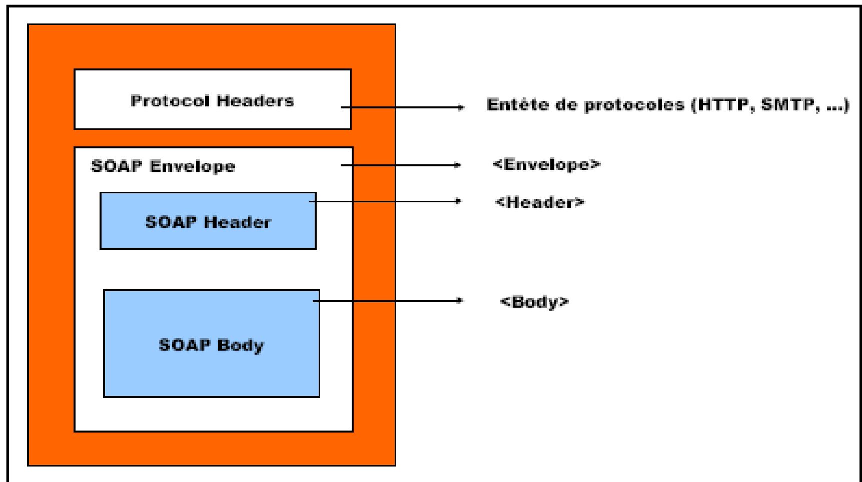


Figure 4 : Architecture d'un message SOAP

Caractéristiques de SOAP

- SOAP est basé sur le XML. Les données échangées entre le demandeur et le fournisseur du service sont donc au format texte.
- SOAP est indépendant des systèmes d'exploitation et des langages de programmation. Il permet par conséquent l'interopérabilité des systèmes.
- SOAP est indépendant du protocole de transport (il utilise souvent le protocole HTTP).

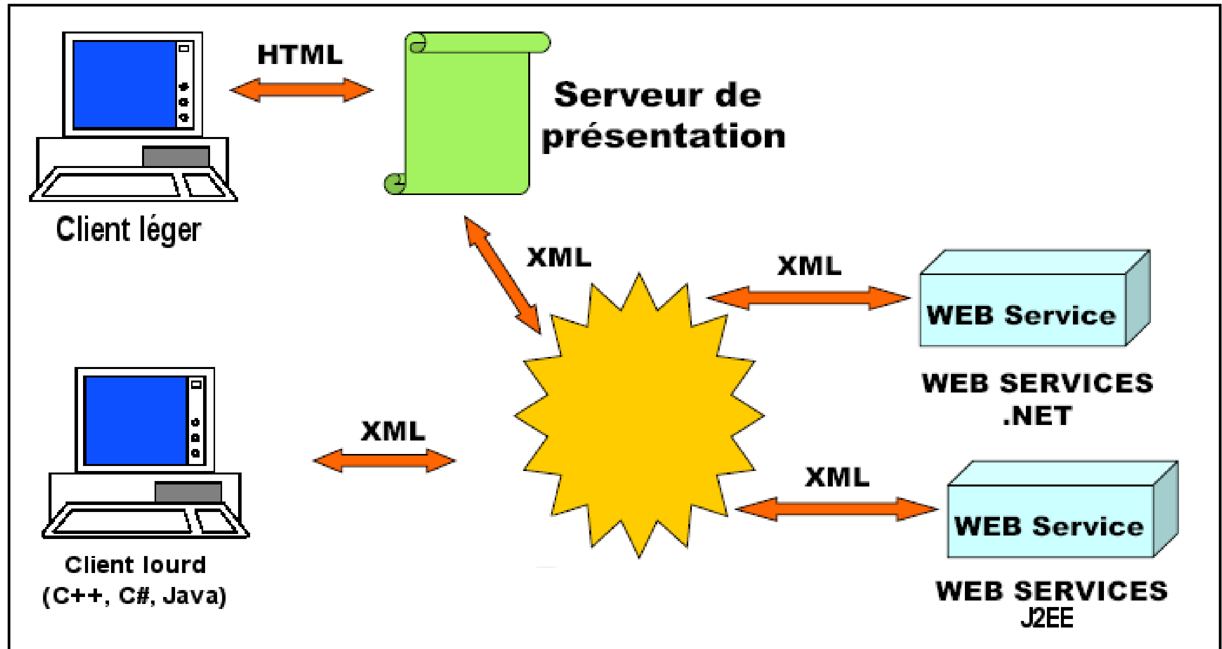


Figure 5 : Interopérabilité des services Web

© *Karim Kalti*

DISCO

- DISCO (*Discovery*) est une technologie de Microsoft pour la publication et la recherche de services Web.
- Il s'agit d'un algorithme qui permet de rechercher les services Web exposés sur un serveur donné. Il permet également de découvrir les capacités de chaque service et la manière d'interagir avec lui et ce en analysant le document WSDL qui accompagne ce service.

Publication d'un service à l'aide de DISCO

- Pour publier un service Web à l'aide de DISCO, il suffit de créer un fichier *.disco* et de le placer dans le répertoire racine du service

Exemple :
pour un service appelé math, il faut ajouter le fichier *.disco* comme suit dans le répertoire du service.

```
\inetpub
  \wwwroot
    \math  (vroot)
       math.asmx
       web.config
       math.disco
      \bin
         simpleMath.dll
         complexMath.dll
```

- Le fichier *.disco* est un document XML qui contient des liens aux ressources qui décrivent le service (document WSDL).

```
<disco:discovery
  xmlns:disco="http://schemas.xmlsoap.org/disco/"
  xmlns:scl="http://schemas.xmlsoap.org/disco/scl/">
  <!-- reference to other DISCO document -->
  <disco:discoveryRef
    ref="related-services/default.disco"/>
  <!-- reference to WSDL and documentation -->
  <scl:contractRef ref="math.asmx?wsdl"
    docRef="math.asmx"/>
</disco:discovery>
```

Récupération des informations à l'aide de DISCO

La récupération des informations à travers un fichier *.disco* se fait à l'aide d'un utilitaire qui fonctionne en ligne de commande appelé *disco.exe*. Cette commande prend comme argument l'url du fichier *.disco* du service à consulter.

Exemple :

```
c:\temp> disco.exe http://localhost/math/math.disco
```

Cette commande génère en sortie un fichier appelé *result.discomap* qui contient des informations sur le service Web situé à cette *url*.

Remarques :

- Il est possible d'utiliser au lieu des fichiers *.disco*, des fichiers *.vsdisco* pour récupérer d'une manière dynamique la liste de tous les services Web qui se trouvent sur un serveur donné spécifié par son url. (c'est spécifique pour MS VS).

- DISCO est une technologie de microsoft. Elle utilisée pour la publication et la recherche à petite échelle. Le client doit connaître au préalable l'url du service Web qu'il veut consommer (ce n'est pas toujours le cas en pratique).

- L'alternative à DISCO est UDDI qui est un standard universel pour la publication et la recherche à grande échelle de services Web.

© Karim Kalti

UDDI : Universal Description, Discovery and Integration

- UDDI est un annuaire qui donne des informations sur les entreprises et les services qu'elles publient.
- Il spécifie un mécanisme pour les fournisseurs de services Web pour publier leurs services et pour les consommateurs de services de localiser les services qui les intéressent.

Interaction avec un annuaire UDDI

- L'interaction avec un annuaire peut être manuelle (via des moteurs de recherche) ou à l'aide d'une API de programmation qui repose sur SOAP.
- Il existe deux API permettant de dialoguer avec un annuaire :
  - Une API d'interrogation de l'annuaire : permet de recherche dans les points d'entrée de l'annuaire et de lire les descriptions des services.
  - Une API de publication : Elle permet de publier ou de supprimer des services au sein d'un annuaire. Elle impose des autorisations d'accès.

Remarque :

L'existence d'une API d'interrogation d'annuaires ouvre la porte aux applications consommatrices de découvrir d'une manière dynamique les services qui les intéressent. Le service qui sera utilisé peut ne pas être connu au moment du développement de l'application cliente (son interface doit être connue mais pas son implémentation).

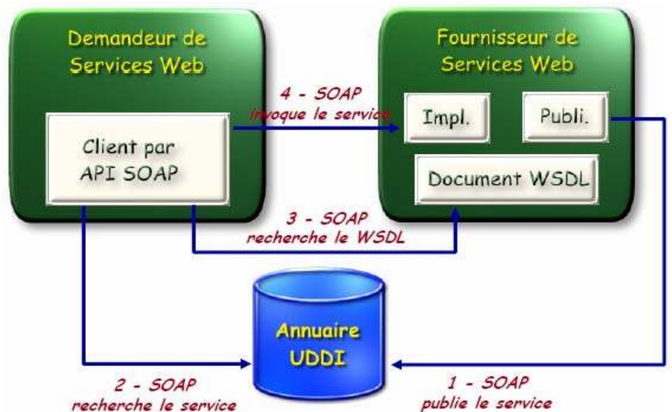


Figure 6 : Processus de découverte d'un service

Organisation des données dans l'annuaire

L'annuaire est organisé sous forme de pages de trois catégories selon les informations qu'elles renferment :

- *Les pages blanches* : elles contiennent des informations qui concernent les entreprises (noms des entreprises, leurs coordonnées, leurs domaines d'activité, des descriptions supplémentaires, etc.).
- *Les pages jaunes* : Elles donnent la liste des services par catégorie selon leurs natures ou selon les entreprises qui les proposent.
- *Les pages vertes* : elles contiennent les spécifications techniques des services, classées par société.

Figure 7 : Types de pages d'un annuaire UDDI

© Karim Kalti

Exemples d'entité de données utilisées par UDDI

Le modèle UDDI comporte cinq structures de données principales décrites sous formes de schémas XML :

- ***BusinessEntity*** : Ensemble d'informations sur l'entreprise qui publie le service (domaine d'activité, nom, description, coordonnées).
- ***BusinessService*** : Ensemble d'informations sur le service publié par l'entreprise (nom du service, sa description, son code unique).
- ***tModel*** : ensemble d'informations décrivant le modèle du service. C'est l'interface qui définie la spécification du service (Description WSDL).
- ***BindingTemplate*** : ensemble d'informations concernant le lieu d'hébergement du service (url de ses points d'accès). Il s'agit de l'url de l'implémentation de la spécification décrite dans le *tModel.* La spécification d'un service peut avoir plusieurs implémentations proposées par différents industriels sous forme de plusieurs *BindingTemplate* qui référencent donc le même *tModel.*
- ***PublisherAssertion*** : ensembles d'informations nécessaires à l'établissement de contrats entre partenaires dans le cadre d'échanges commerciaux.

Annexe : Discovery-driven Applications

Cette annexe est une partie d'un article rédigé par Aaron Skonnard dans lequel il discute des avantages et des inconvénients du processus de découverte dynamique des services publiés dans les annuaires UDDI par les applications qui les consomment.

Discovery-driven Applications

Auteur : Aaron Skonnard
Source : MSDN Magazine - Faburary 2002

One of the main benefits of UDDI's richer data model and more advanced inquiry API is that developers can build their applications around certain types of Web Services, each with the same technical specifications, interfaces, and protocols that can be discovered at runtime (just like COM component categories). This allows applications to automatically take advantage of new and improved Web Services (of that specific type) that are published after the application ships. Building applications with this functionality requires the lookup and categorization features offered today only by UDDI.

However, even though it's technically possible to build discovery-driven applications, most developers won't do it. Developers want to choose their business partners a priori so they can establish strong relationships based on quality assurance and future legal liabilities. No serious developer is going to allow the inclusion of some hacker's completely untested service to debilitate his or her aggregate application.

This is exactly why COM component category-based applications never gained wide acceptance in the industry. And since most of today's UDDI hype revolves around this type of discovery-driven application, it's unclear as to how much better this beefed-up API will be compared to something that is more primitive like DISCO.

If the information in the existing UDDI repositories is a sign of future acceptance, it's not looking good. If you browse some of the test UDDI sites, you'll notice that there is plenty of yellow page information (like business name, business details, and so forth), but very little technical information (such as contract information or WSDL documents).

Developers using DISCO today have found it simple and effective, and they retain complete control over the discovery process. Moving to UDDI increases the complexity by an order of magnitude and forces developers to either relinquish control to a centralized UDDI operating site or implement their own. It's hard to imagine that many will rush to implement that solution.

© Karim Kalti

Mise en œuvre des Services Web en .NET

Le FrameWork .NET à travers ASP.NET fournit l'infrastructure nécessaire pour la création et la consommation de services Web XML.

Outils pour développer et déployer un service Web XML

Outils pour le développement :

- Un éditeur de texte pour écrire le code (Exemple : Notepad).
- Le SDK du FrameWork .NET.

Il existe des environnements de développement intégrés (EDI) plus élaborés :
- Visual Studio.NET (payant)
- WebMatrix avec son serveur Web Cassini (gratuit).

Outils pour le déploiement

- Un Serveur Web : IIS ou Cassini.
- La CLR : (elle doit être installée après l'installation du serveur Web).

Implémentation d'un service Web XML

Fichier d'implémentation

Un service Web XML est implémenté sous forme d'une classe placée dans un fichier qui porte l'extension (*.asmx*). Cette classe comporte les méthodes exposées par le service. Toutefois un service peut éventuellement utiliser en interne d'autres ressources placées dans des fichiers ayant d'autres extensions que (*.asmx*). Par exemple des fichiers de bases de données si le service en fait accès, des fichiers de code compilé, etc.

Langage d'implémentation

En .NET un service Web peut être programmé dans n'importe quel langage supporté par ASP.NET (C#, VB, JavaScript).

Déclaration d'un service Web

La déclaration d'un service Web se fait de la manière suivante :

```
<%@ WebService Language="C#" Class="NomDuService"%>
```

- Cette déclaration est faite dans un fichier (*asmx*).
- La directive ***@WebService*** indique que le fichier contient l'implémentation d'un service Web.
- L'attribut ***Language*** indique le langage de programmation utilisé (C#, VB, JavaScript).
- L'attribut ***Class*** permet de spécifier le nom de la classe qui implémente le service Web.

© *Karim Kalti*

Définition de la classe du service Web

La définition de la classe qui implémente le service Web se fait de la manière suivante :

```
using System.Web.Services;
public class NomDuService
{
  // Définition des méthodes du service
  [WebMethod]
  …… Méthode1( … … )
  {…………}
   … … … … …
  [WebMethod]
   …… MéthodeN( … … )
  {…………}
}
```

NomDuService est le nom de la classe qui implémente le service Web. Cette classe publique comporte généralement les méthodes qui seront exposées par le service.

Méthodes exposées par un service

- Une méthode exposée par un service est une méthode qui peut être appelée par le client du service. Une telle méthode :
  - doit être membre de la classe qui implémente le service.
  - doit être déclarée publique.
  - doit être précédée dans sa définition par l'attribut [WebMethod]. C'est cet attribut, qui va rendre la méthode appelables par les clients du service. Une méthode de la classe du service Web qui n'est pas définie en tant que [WebMethod] ne peut pas être invoquée par les clients même si elle est déclarée publique.

- [WebMethod] est un attribut prédéfini en .NET dans l'espace de noms *System.Web.Services*. Ceci explique l'utilisation de la directive *using System.Web.Services;* avant la définition de la classe.

Endroit de définition de la classe du service Web

<u>Définition dans le fichier (asmx)</u>

La classe qui implémente le service peut être définie dans le fichier (*asmx*). Dans ce cas elle doit succéder à la déclaration du service.

```
<%@ WebService Language="C#" Class="NomDuService"%>
 using System.Web.Services;
 public class NomDuService
 {
   // Définition des méthodes du service
   [WebMethod]
    …… Méthode1( … … )
   {…………}
    … … … … …
   [WebMethod]
    …… MéthodeN( … … )
   {…………}
 }
```

<u>Définition selon le modèle "codebehind"</u>

La classe du service peut être définie dans un fichier de code pur (.cs ou .vb) séparé du fichier (*asmx*) selon le modèle "*codebehind*" utilisé par les WebForms (pages aspx).

Contenu du fichier de déclaration du service (*.asmx*)

```
<%@ WebService Language="C#" CodeBehind="NomFichierClass.cs" Class="EspaceNoms.NomDuService"%>
```

La déclaration du service inclut dans ce cas une référence au fichier qui contient la définition de la classe. Cette référence est spécifiée à l'aide de l'attribut ***CodeBehind***.

© Karim Kalti

Contenu du fichier d'implémentation de la classe du service (.cs)

```
using System.Web.Services;
namespace EspaceNoms
{
      public class NomDuService
       {
         // Définition des méthodes du service
         [WebMethod]
          …… Méthode1( … … )
          {………..}
            … … … … …
         [WebMethod]
          …… MéthodeN( … … )
         {………..}

       }
}
```

Remarque :

Il est possible d'appeler dans la déclaration d'un service Web une classe déjà compilée au lieu du fichier contenant le code source de la classe. Dans ce cas, la déclaration du service doit être effectuée comme suit :

```
<%@ WebService Language="C#"  Class="EspaceNoms.NomDuService, NomAssemblage"%>
```

NomAssemblage désigne le nom de l'assemblage qui contient la classe compilée. Ce dernier doit figurer dans le répertoire *\bin* de l'application qui représente le service Web.

Ajout d'informations supplémentaires à un service Web

Il est possible d'ajouter des informations supplémentaires à un service Web. La spécification de ces informations est optionnelle. Elle se fait à l'aide de l'attribut optionnel *WebService*. Cet attribut possède deux propriétés intéressantes :

- *Namespace* : cette propriété permet d'associer un espace de noms XML au service. Cet espace de noms permet aux applications clientes d'identifier d'une manière unique le service dans le cas où ce dernier porte le même nom qu'un autre service publié sur le NET. Il est à remarquer dans ce cadre que :
    - L'espace de noms par défaut associé au service en cours de construction est http://tempuri.org/.
    - Si le fournisseur du service désire publier ce dernier en intranet dans un domaine dont il contrôle les noms des services alors la spécification de l'espace de noms n'est pas nécessaire et l'espace de noms par défaut suffit. Toutefois si ce fournisseur désire publier son service sur le NET alors l'attribution d'un espace de nom unique devient inéluctable pour éviter tout risque de conflit.
    - L'attribut *Namespace* peut accepter comme valeur n'importe quelle chaîne de caractères unique. Cependant la valeur qui lui est communément donnée se présente souvent sous la forme de l'url de l'entreprise qui fournit le service.
    - L'espace de noms dont il est question est l'espace XML. Il ne faut pas le confondre avec l'espace de noms de la classe qui implémente le service.

- *Description* : Cette propriété permet de fournir des informations supplémentaires pour décrire avec plus de détails le service.

Le contenu générique d'un fichier (*asmx*) peut se présenter comme suit :

```
using System.Web.Services;
namespace EspaceNoms
{
      [WebService (Namespace ="www.MonEspace.com") (Description = "Ce service permet ……")]
      public class NomDuService
       {
         // Définition des méthodes du service
         [WebMethod]
          …… Méthode1( … … )
          … … … … …
         [WebMethod]
          …… MéthodeN( … … )
       }
}
```


© Karim Kalti

Remarque :

- Il est également possible d'ajouter des informations supplémentaires pour chaque méthode du service et ce à l'aide de la propriété *Description* de l'attribut [WebMethod].

```
[WebMethod (Description = "Cette méthode effectue … … …")]
…… Méthode1( … … )
```

Gestion des états Application et Session dans un service Web

Un service Web peut bénéficier de la panoplie de possibilités qu'offre l'ASP.NET notamment en ce qui concerne la gestion des états *Application*, *Session*, etc. Pour cela il faut faire dériver la classe qui implémente le service de la classe `System.Web.Services.WebService` du FrameWork .NET. Le contenu générique d'un fichier (*asmx*) peut se présenter comme suit :

```
using System.Web.Services;
namespace EspaceNoms
{
      [WebService (Namespace ="www.MonEspace.com", Description = "Ce service permet ……")]
      public class NomDuService : WebService
        {
          // Définition des méthodes du service
          [WebMethod]
           …… Méthode1( … … )
           … … … … …
          [WebMethod]
           …… MéthodeN( … … )
        }
}
```

Premier exemple de service Web

L'exemple suivant montre l'implémentation d'un service Web simple qui effectue la somme de deux entiers :

```
<%@ WebService Language="C#" Class="MathOps.Operations"%>
  using System.Web.Services;
  namespace MathOps
  {
        [WebService (Namespace="www.Test.com", Description = "Premier service Web")]
        public class Operations
          {
            [WebMethod (Description ="Cette méthode effectue la somme de deux entiers")]
             public int Somme(int a, int b)
               {return a+b;}
          }
  }
```

Test du service Web

Pour tester le service Web que l'on vient de créer, il n'est pas nécessaire de développer une application cliente mais il suffit de placer le fichier *Operations.aspx* dans le répertoire de publication (*wwwroot* pour IIS) et de taper son *url* dans le navigateur.
http://localhost/Operations.asmx

Le navigateur affiche une page créée automatiquement par IIS avec la collaboration des services du FrameWork qui décrit le service Web demandé. Cette description comporte :

- Le nom du service et sa description mentionnée dans la propriété *Description* de l'attribut *WebService*.
- Les noms des méthodes exposées par le service et leurs descriptions mentionnées dans la propriété *Description* de l'attribut *WebMethod*.

© Karim Kalti

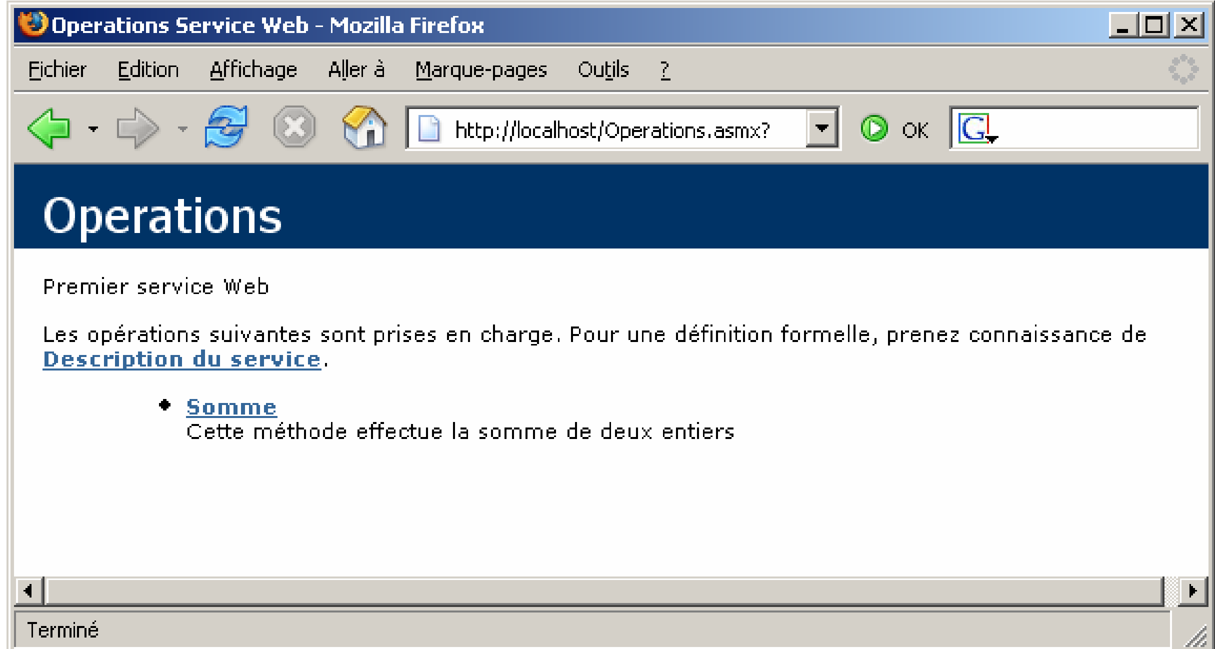


Figure 1 : Interface de présentation du service Web

Le clic sur le lien vers la méthode *Somme* engendre l'affichage de la page suivante :

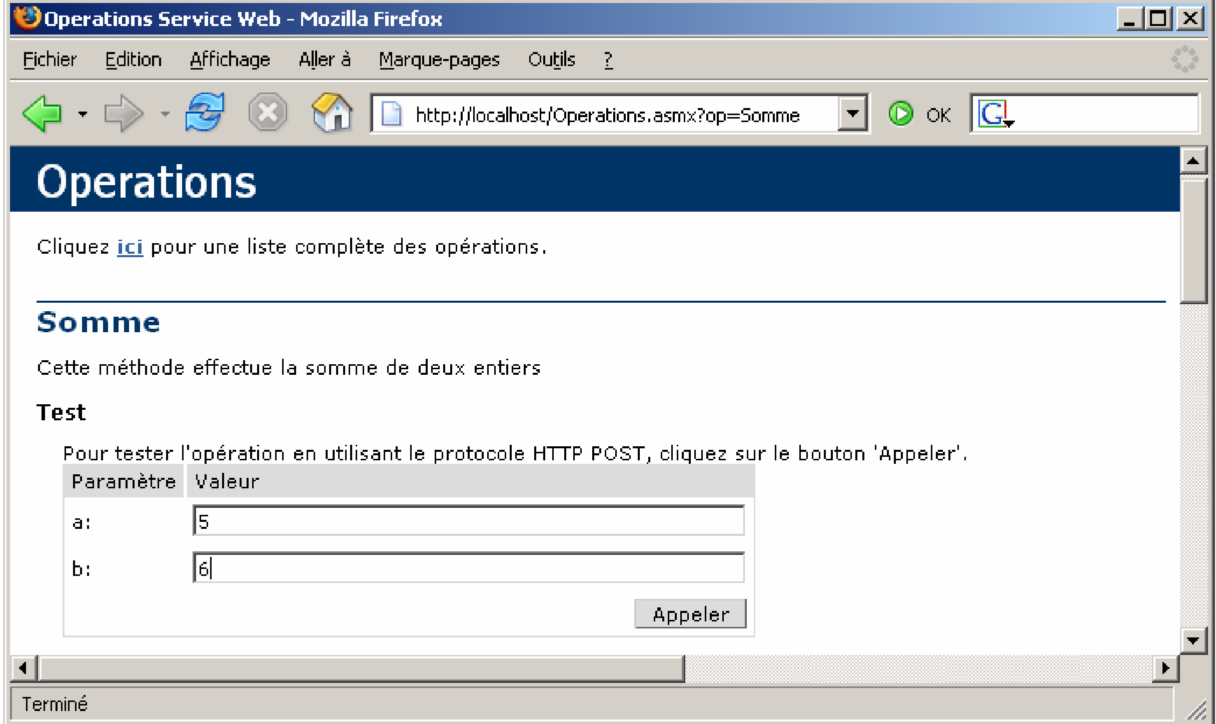


Figure 2 : Interface de test de la méthode *Somme*

- Cette page permet de tester la méthode *Somme* du service Web. L'appel de la méthode du service aurait pu être effectué directement en tapant l'url suivante : http://localhost/Operations.asmx?op=Somme

© Karim Kalti

- Le navigateur affiche également :
    - le texte des deux messages SOAP utilisés par cette méthode pour recevoir les paramètres et pour renvoyer le résultat.
    - le texte montrant l'appel de la méthode à l'aide de HTTP POST.

```
SOAP
Le texte suivant est un exemple de demande et de réponse SOAP. Les espaces réservés affichés doivent être remplacés par des valeurs réelles.
POST /Operations.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "www.Test.com/Somme"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope                           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Somme xmlns="www.Test.com">
      <a>int</a>
      <b>int</b>
    </Somme>
  </soap:Body>
</soap:Envelope>
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope                           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SommeResponse xmlns="www.Test.com">
      <SommeResult>int</SommeResult>
    </SommeResponse>
  </soap:Body>
</soap:Envelope>



HTTP POST
Le texte suivant est un exemple de demande et de réponse HTTP POST. Les espaces réservés affichés doivent être remplacés par des valeurs réelles.
POST /Operations.asmx/Somme HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

a=string&b=string
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<int xmlns="
```


© Karim Kalti

La saisie des valeurs pour les deux paramètres a et b et le clic sur le bouton *Appeler* engendre l'affichage de la page suivante qui contient le résultat :

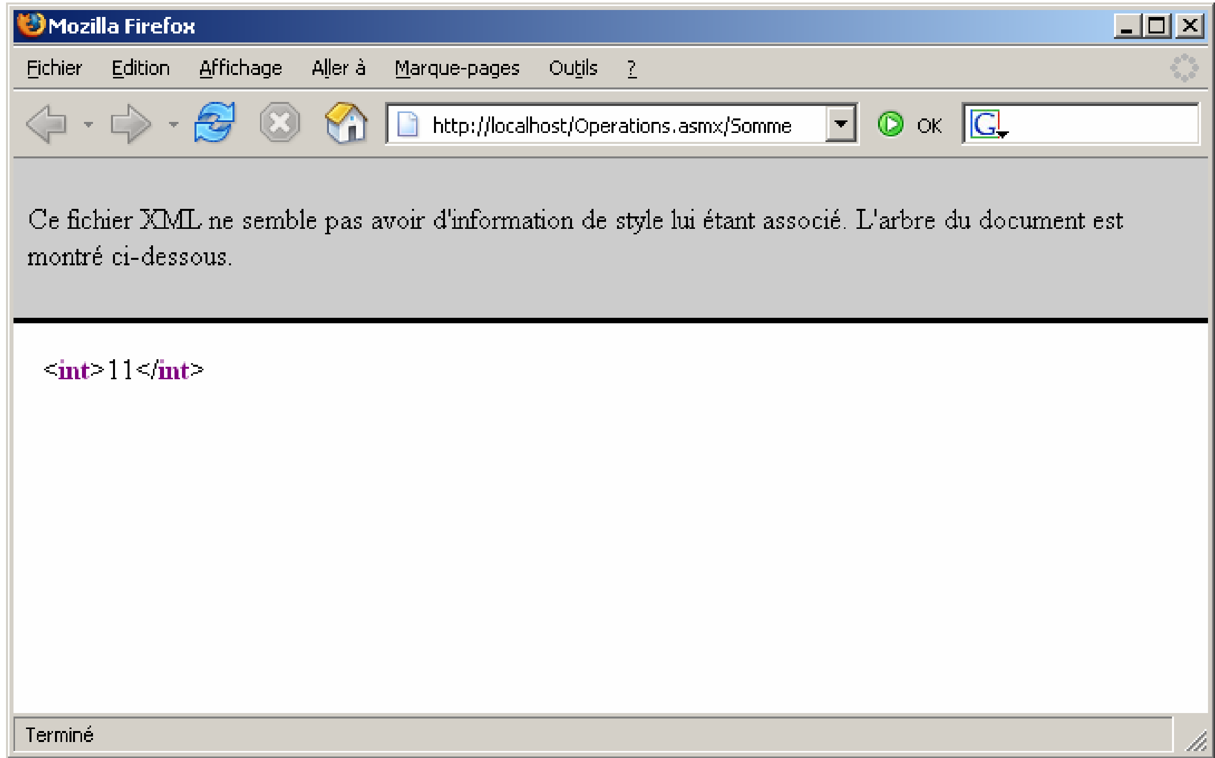


Figure 3 : Page affichant le résultat de la méthode *Somme*

Génération de la description WSDL pour un service

La génération de la description WSDL pour un service peut être faite d'une manière automatique en adjoignant *?wsdl* à l'url du service. Pour l'exemple du service *Operations*, cela se traduit comme suit :
http://localhost/Operations.asmx?wsdl

La description générée dans le navigateur est alors la suivante :

```
<?xml version="1.0" encoding="utf-8" ?>
-<definitions xmlns:http=http://schemas.xmlsoap.org/wsdl/http/
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:s0="www.Test.com" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" targetNamespace="www.Test.com"
xmlns="http://schemas.xmlsoap.org/wsdl/">
- <types>
- <s:schema elementFormDefault="qualified" targetNamespace="www.Test.com">
- <s:element name="Somme">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="a" type="s:int" />
  <s:element minOccurs="1" maxOccurs="1" name="b" type="s:int" />
  </s:sequence>
  </s:complexType>
  </s:element>
- <s:element name="SommeResponse">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="SommeResult" type="s:int" />
  </s:sequence>
  </s:complexType>
  </s:element>
  </s:schema>
  </types>
```


© Karim Kalti

```
- <message name="SommeSoapIn">
  <part name="parameters" element="s0:Somme" />
  </message>
- <message name="SommeSoapOut">
  <part name="parameters" element="s0:SommeResponse" />
  </message>
- <portType name="OperationsSoap">
- <operation name="Somme">
  <documentation>Cette méthode effectue la somme de deux entiers</documentation>
  <input message="s0:SommeSoapIn" />
  <output message="s0:SommeSoapOut" />
  </operation>
  </portType>
- <binding name="OperationsSoap" type="s0:OperationsSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
- <operation name="Somme">
  <soap:operation soapAction="www.Test.com/Somme" style="document" />
- <input>
  <soap:body use="literal" />
  </input>
- <output>
  <soap:body use="literal" />
  </output>
  </operation>
  </binding>
- <service name="Operations">
  <documentation>Premier service Web</documentation>
- <port name="OperationsSoap" binding="s0:OperationsSoap">
  <soap:address location="http://localhost/Operations.asmx" />
  </port>
  </service>
  </definitions>
```

Module *proxy* nécessaire pour le développement d'un client du service Web

Le client d'un service Web est un programme qui fait appel aux méthodes exposées de ce service. De ce fait le programme client doit avoir une référence à la classe qui implémente le service ainsi qu'à ses méthodes pour pouvoir faire cet appel. L'obtention de la référence en question est assurée par une classe appelée *proxy*.

Rôle de la classe *proxy*

La classe *proxy* joue le rôle d'intermédiaire entre le client et le service Web demandée. Dans ce cadre :

- Elle fournit aux programmes clients des références aux méthodes exposées par le service. Ces références servent à appeler ces méthodes distantes (situées sur le serveur où est hébergé le service).
- Elle réalise l'empaquetage du nom de la méthode ainsi que de ses arguments dans un message SOAP de type *Request*.
- Elle envoie ce message au serveur pour qu'il exécute la méthode.
- Elle reçoit un message SOAP de type *Response* contenant la valeur de retour de la méthode du service.
- Elle décortique le message afin d'en extraire la valeur renvoyée (au format XML) et de la traduire vers le type de données approprié.

Rôle du serveur (IIS avec la CLR) dans la médiation entre le proxy et le service Web

Le serveur :

- Reçoit un message SOAP de type *Request* contenant un appel à une méthode du service.
- Décortique le message, en extrait le nom de la méthode et les paramètres et effectue l'appel.
- Récupère la valeur de retour et effectue son empaquetage dans un message SOAP de type *Response*.
- Envoie le message au client.

© Karim Kalti

Définition de la classe proxy

- La classe ***System.Web.Services.Protocols.SoapHttpClientProtocol*** dispose d'un ensemble de méthodes qui implémentent une grande partie des opérations qui sont à la charge du *proxy* client notamment l'invocation distante de méthodes et la construction et l'analyse des messages SOAP.
- Cette classe contient trois méthodes intéressantes qui sont :
  - *Invoke* : Appelle une méthode de service Web XML de manière synchrone à l'aide de SOAP.
  - *BeginInvoke* : Débute un appel asynchrone d'une méthode de service Web XML au moyen de SOAP.
  - *EndInvoke* : Met fin à un appel asynchrone d'une méthode de service Web XML au moyen de SOAP.

- La classe ***proxy*** à ajouter au niveau de l'application cliente doit être dérivée de la classe *SoapHttpClientProtocol.*
- Cette classe peut être défini par le programmeur ou générée d'une manière automatique à l'aide d'un utilitaire du FrameWork .NET appelé ***WSDL.exe***.

Génération de la classe *proxy*

- L'utilitaire *WSDL* prend comme argument l'url du service Web pour lequel on veut créer un proxy.
- Pour l'exemple précédemment présenté il faut taper :
  WSDL http://localhost/Operations.asmx

L'utilitaire génère alors un fichier *Operations.cs* qui contient la définition du *proxy*.

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.1.4322.573
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

//
// Ce code source a été automatiquement généré par wsdl, Version=1.1.4322.573.
//
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;


[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Web.Services.WebServiceBindingAttribute(Name="OperationsSoap",
 Namespace="www.Test.com")]
public class Operations : System.Web.Services.Protocols.SoapHttpClientProtocol
{

    public Operations()
    {
        this.Url = "http://localhost/operations.asmx";
    }

    [System.Web.Services.Protocols.SoapDocumentMethodAttribute("www.Test.com/Somme",
     RequestNamespace="www.Test.com", ResponseNamespace="www.Test.com",
     Use=System.Web.Services.Description.SoapBindingUse.Literal,
     ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]

    public int Somme(int a, int b)
    {
        object[] results = this.Invoke("Somme", new object[] {a,b});
        return ((int)(results[0]));
    }
```


© Karim Kalti

```
public System.IAsyncResult BeginSomme(int a, int b, System.AsyncCallback callback, object
    asyncState)
    {
        return this.BeginInvoke("Somme", new object[] {a,b}, callback, asyncState);
    }

    public int EndSomme(System.IAsyncResult asyncResult)
    {
        object[] results = this.EndInvoke(asyncResult);
        return ((int)(results[0]));
    }
}
```

- Le nom de la classe *proxy* est celui du service.
- Cette classe comprend trois méthodes pour chaque méthode exposée du service : une pour l'appel directe (*Somme*) de la méthode du service et deux pour un appel asynchrone (*BeginSomme* et *EndSomme*).
- Pour pouvoir exploiter la classe *proxy* il suffit de l'ajouter au code de l'application cliente. Cette application en question peut représenter n'importe quelle application qui peut faire usage du protocole HTTP.

Développement d'une application cliente de type Console

L'exemple suivant montre une application de type console qui exploite le service *Operations* pour effectuer une opération d'addition. Le code de la classe *proxy* est copié dans le même fichier que celui de la classe de la console.

```
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;

namespace ClientConsole
{

 [System.Diagnostics.DebuggerStepThroughAttribute()]
 [System.ComponentModel.DesignerCategoryAttribute("code")]
 [System.Web.Services.WebServiceBindingAttribute(Name="OperationsSoap",
Namespace="www.Test.com")]

// La classe proxy
public class Operations : System.Web.Services.Protocols.SoapHttpClientProtocol
{

    public Operations()
    {
        this.Url = "http://localhost/operations.asmx";
    }

     [System.Web.Services.Protocols.SoapDocumentMethodAttribute("www.Test.com/Somme",
      RequestNamespace="www.Test.com", ResponseNamespace="www.Test.com",
      Use=System.Web.Services.Description.SoapBindingUse.Literal,
      ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]

    public int Somme(int a, int b)
    {
        object[] results = this.Invoke("Somme", new object[] {a,b});
        return ((int)(results[0]));
    }

    public System.IAsyncResult BeginSomme(int a, int b, System.AsyncCallback callback, object
    asyncState)
    {
        return this.BeginInvoke("Somme", new object[] {a,b}, callback, asyncState);
    }

    public int EndSomme(System.IAsyncResult asyncResult)
    {
        object[] results = this.EndInvoke(asyncResult);
        return ((int)(results[0]));
    }
}
```


© Karim Kalti

```
// La classe qui fait appel au service
class MainClass
    {
        public static void Main(string[] args)
         {
              // Instanciation du service
              Operations op=new Operations();
              int i,j,s;
              Console.WriteLine("Donner un premier entier : ");
              i=Int32.Parse(Console.ReadLine());
              Console.WriteLine("Donner un deuxième entier : ");
              j=Int32.Parse(Console.ReadLine());
              // Appel de la méthode du service
              s= op.Somme(i,j);
              Console.WriteLine("La somme est : "+s);

         }
    }
}
```

Le résultat de l'exécution du programme est le suivant :

```
 Donner un premier entier : 5
 Donner un deuxième entier : 6
 La somme est : 11
```

Développement d'un client de type application Web (WebForm)

L'exemple suivant montre le développement d'un client de type application Web qui fait appel au service *Operations*.

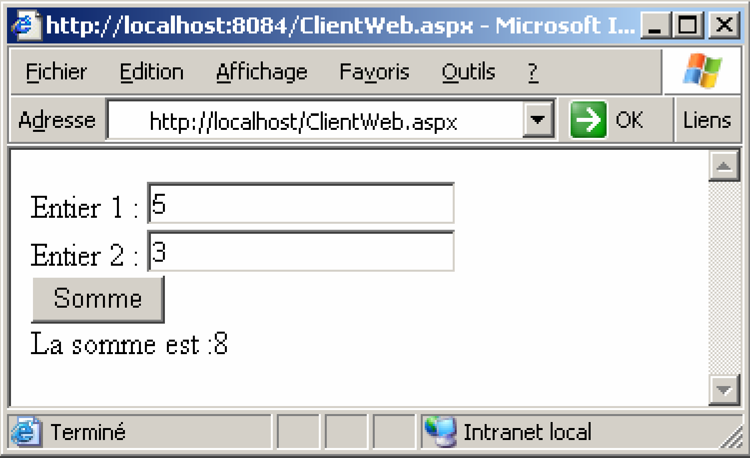


Figure 4 : Interface du client Web

Le code de la classe *proxy* est placé dans un fichier appelé *Operations.cs*. Le contenu de ce fichier est le suivant :

```
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;


[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Web.Services.WebServiceBindingAttribute(Name="OperationsSoap",
 Namespace="www.Test.com")]
```


© Karim Kalti

```
public class Operations : System.Web.Services.Protocols.SoapHttpClientProtocol
{
    public Operations()
    {
        this.Url = "http://localhost/operations.asmx";
    }
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute("www.Test.com/Somme",
     RequestNamespace="www.Test.com", ResponseNamespace="www.Test.com",
     Use=System.Web.Services.Description.SoapBindingUse.Literal,
     ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public int Somme(int a, int b)
    {
        object[] results = this.Invoke("Somme", new object[] {a,b});
        return ((int)(results[0]));
    }

    public System.IAsyncResult BeginSomme(int a, int b, System.AsyncCallback callback, object
    asyncState)
    {
        return this.BeginInvoke("Somme", new object[] {a,b}, callback, asyncState);
    }

    public int EndSomme(System.IAsyncResult asyncResult)
    {
        object[] results = this.EndInvoke(asyncResult);
        return ((int)(results[0]));
    }
}
```

Le code de l'interface qui utilise cette classe *proxy* pour faire appel au service Web se présente comme suit :

```
<%@ Page Language="C#" Src="operations.cs" %>
<script runat="server">

    void Page_Load(object sender, EventArgs e)
       {
          if( IsPostBack == false)
          {
            TxtBE1.Text="0";
            TxtBE2.Text="0";
          }
        }
    void BtnSomme_Click(object sender, EventArgs e)
       {
           int i,j, Somme;
           if(TxtBE1.Text!="" && TxtBE2.Text!="")
           {
               Operations op = new Operations();
              i= Int32.Parse(TxtBE1.Text);
              j= Int32.Parse(TxtBE2.Text);
              Somme = op.Somme(i,j);
              LbResultat.Text = "La somme est :" + Somme.ToString();
           }
       }
</script>
<html>
<head>
</head>
<body>
    <form runat="server">
        <p>
            <asp:Label id="LbE1" runat="server">Entier 1 : </asp:Label>
            <asp:TextBox id="TxtBE1" runat="server"></asp:TextBox>
            <br />
            <asp:Label id="LbE2" runat="server">Entier 2 : </asp:Label>
            <asp:TextBox id="TxtBE2" runat="server"></asp:TextBox>
            <br />
            <asp:Button id="BtnSomme" onclick="BtnSomme_Click" runat="server"
                        Text="Somme"></asp:Button>
            <br />
            <asp:Label id="LbResultat" runat="server">La somme est : </asp:Label>
        </p>
    </form>
</body>
</html>
```


© Karim Kalti

Développement d'un client de type application Windows (WinForm)

L'exemple suivant montre le développement d'un client de type application Windows qui fait appel au service *Operations*.

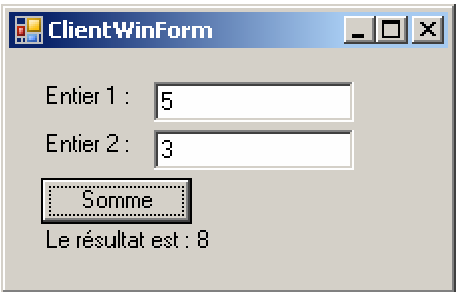


Figure 5 : Interface du client Windows

Le code de l'application cliente est le suivant :

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Web.Services;
using System.Web.Services.Protocols;

namespace ClientWinForm
{
        [System.Diagnostics.DebuggerStepThroughAttribute()]
        [System.ComponentModel.DesignerCategoryAttribute("code")]
        [System.Web.Services.WebServiceBindingAttribute(Name="OperationsSoap",
Namespace="www.Test.com")]

        // La classe proxy
        public class Operations : System.Web.Services.Protocols.SoapHttpClientProtocol
        {
                public Operations()
                {
                        this.Url = "http://localhost/operations.asmx";
                }


        [System.Web.Services.Protocols.SoapDocumentMethodAttribute("www.Test.com/Somme",
                         RequestNamespace="www.Test.com", ResponseNamespace="www.Test.com",
                         Use=System.Web.Services.Description.SoapBindingUse.Literal,
                ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]

                public int Somme(int a, int b)
                {
                        object[] results = this.Invoke("Somme", new object[] {a,b});
                        return ((int)(results[0]));
                }

                public System.IAsyncResult BeginSomme(int a, int b, System.AsyncCallback
                callback, object  asyncState)
                {
                  return this.BeginInvoke("Somme", new object[] {a,b}, callback, asyncState);
                }

                public int EndSomme(System.IAsyncResult asyncResult)
                {
                        object[] results = this.EndInvoke(asyncResult);
                        return ((int)(results[0]));
                }
        }
```


© Karim Kalti

```
	public class ClientWinForm : System.Windows.Forms.Form
	{
		private System.Windows.Forms.Label label1;
		private System.Windows.Forms.TextBox TxtBEntier1;
		private System.Windows.Forms.TextBox TxtBEntier2;
		private System.Windows.Forms.Label LbEntier2;
		private System.Windows.Forms.Button BtnSomme;
		private System.Windows.Forms.Label LbResultat;

		private System.ComponentModel.Container components = null;

		public ClientWinForm()
		{
			InitializeComponent();
		}

		protected override void Dispose( bool disposing )
		{
			if( disposing )
			{
				if (components != null)
				{
					components.Dispose();
				}
			}
			base.Dispose( disposing );
		}

		private void InitializeComponent()
		{
			this.label1 = new System.Windows.Forms.Label();
			this.TxtBEntier1 = new System.Windows.Forms.TextBox();
			this.TxtBEntier2 = new System.Windows.Forms.TextBox();
			this.LbEntier2 = new System.Windows.Forms.Label();
			this.BtnSomme = new System.Windows.Forms.Button();
			this.LbResultat = new System.Windows.Forms.Label();
			this.SuspendLayout();
			// label1
			this.label1.Location = new System.Drawing.Point(16, 16);
			this.label1.Name = "label1";
			this.label1.Size = new System.Drawing.Size(64, 16);
			this.label1.TabIndex = 0;
			this.label1.Text = "Entier 1 :";
			// TxtBEntier1
			this.TxtBEntier1.Location = new System.Drawing.Point(72, 16);
			this.TxtBEntier1.Name = "TxtBEntier1";
			this.TxtBEntier1.TabIndex = 1;
			this.TxtBEntier1.Text = "";
			// TxtBEntier2
			this.TxtBEntier2.Location = new System.Drawing.Point(72, 40);
			this.TxtBEntier2.Name = "TxtBEntier2";
			this.TxtBEntier2.TabIndex = 3;
			this.TxtBEntier2.Text = "";
			// LbEntier2
			this.LbEntier2.Location = new System.Drawing.Point(16, 40);
			this.LbEntier2.Name = "LbEntier2";
			this.LbEntier2.Size = new System.Drawing.Size(64, 16);
			this.LbEntier2.TabIndex = 2;
			this.LbEntier2.Text = "Entier 2 :";
			// BtnSomme
			this.BtnSomme.Location = new System.Drawing.Point(16, 64);
			this.BtnSomme.Name = "BtnSomme";
			this.BtnSomme.TabIndex = 4;
			this.BtnSomme.Text = "Somme";
			this.BtnSomme.Click += new System.EventHandler(this.BtnSomme_Click);
			// LbResultat
			this.LbResultat.Location = new System.Drawing.Point(16, 88);
			this.LbResultat.Name = "LbResultat";
			this.LbResultat.TabIndex = 5;
			// ClientWinForm
			this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
			this.ClientSize = new System.Drawing.Size(224, 133);
			this.Controls.Add(this.LbResultat);
			this.Controls.Add(this.BtnSomme);
			this.Controls.Add(this.TxtBEntier2);
			this.Controls.Add(this.LbEntier2);
```


© Karim Kalti

```
                    this.Controls.Add(this.TxtBEntier1);
                    this.Controls.Add(this.label1);
                    this.Name = "ClientWinForm";
                    this.Text = "ClientWinForm";
                    this.ResumeLayout(false);

            }
            [STAThread]
            static void Main()
            {
                    Application.Run(new ClientWinForm());
            }

            private void BtnSomme_Click(object sender, System.EventArgs e)
            {
                    int i,j,s;
                    // Instanciation de la classe du service
                    Operations op = new Operations();
                    if(this.TxtBEntier1.Text!="" && this.TxtBEntier2.Text!="")
                    {
                            i=Int32.Parse(this.TxtBEntier1.Text);
                            j=Int32.Parse(this.TxtBEntier2.Text);
                            // Appel de la méthode du service
                            s=op.Somme(i,j);
                            this.LbResultat.Text="Le résultat est : "+s.ToString();
                    }
            }
    }
}
```

La gestion des exceptions

- Lorsqu'une exception est levée dans un service Web, SOAP la remonte à travers un message vers l'application cliente. Cette exception est de type *System.Web.Services.Protocols.SoapException.*
- L'exemple suivant montre la levée et l'interception d'une exception générée par une division par zéro.

Exemple : Gestion d'une exception levée par une division par zéro

Une méthode de division est d'abord ajoutée au service Web *Operations*.

```
<%@ WebService Language="C#" Class="MathOps.Operations"%>
using System.Web.Services;
namespace MathOps
{
 [WebService(Namespace = "www.Test.com", Description = "Premier service Web")]
 public class Operations
 {
   [WebMethod(Description = "Cette méthode effectue la somme de deux entiers")]
   public int Somme(int a, int b)
   { return a + b; }

   [WebMethod(Description = "Cette méthode effectue la division du premier paramètre par le
    second")]
   public int Division(int a, int b)
    { return a / b; }
 }
}
```


© Karim Kalti

La nouvelle classe *proxy* générée par WSDL pour ce service se présente comme suit :

```
// la classe proxy
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Web.Services.WebServiceBindingAttribute(Name="OperationsSoap",
Namespace="www.Test.com")]
  public class Operations : System.Web.Services.Protocols.SoapHttpClientProtocol
  {
        public Operations()
        { this.Url = "http://localhost/operations.asmx";}

[System.Web.Services.Protocols.SoapDocumentMethodAttribute("www.Test.com/Somme",
RequestNamespace="www.Test.com", ResponseNamespace="www.Test.com",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
        public int Somme(int a, int b)
        {
                object[] results = this.Invoke("Somme", new object[] { a, b});
                return ((int)(results[0]));
        }

        public  System.IAsyncResult  BeginSomme(int  a,  int  b,  System.AsyncCallback  callback,
object asyncState)
        {
           return this.BeginInvoke("Somme", new object[] {a, b}, callback, asyncState);
        }

        public int EndSomme(System.IAsyncResult asyncResult)
        {
                object[] results = this.EndInvoke(asyncResult);
                return ((int)(results[0]));
        }

[System.Web.Services.Protocols.SoapDocumentMethodAttribute("www.Test.com/Division",
RequestNamespace="www.Test.com", ResponseNamespace="www.Test.com",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
        public int Division(int a, int b)
        {
                object[] results = this.Invoke("Division", new object[] {a,        b});
                return ((int)(results[0]));
        }
        public System.IAsyncResult BeginDivision(int a, int b, System.AsyncCallback callback,
object asyncState)
        {
                return this.BeginInvoke("Division", new object[] { a, b}, callback, syncState);
        }
        public int EndDivision(System.IAsyncResult asyncResult)
        {
                object[] results = this.EndInvoke(asyncResult);
                return ((int)(results[0]));
        }
}
```

Le test du service est effectué à travers un client de type application Windows dont l'interface se présente comme suit :

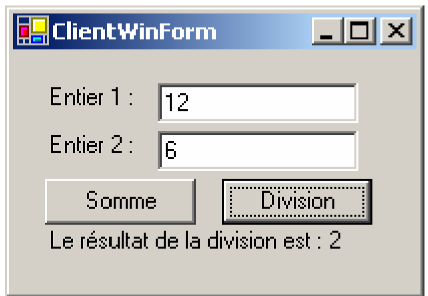


Figure 6 : Interface de test d'une gestion d'exception

© *Karim Kalti*

Le code du gestionnaire de l'événement clic du bouton *Division* est le suivant :

```
private void BtnDivision_Click(object sender, System.EventArgs e)
{
        try
        {
                int i,j,s;
                // Instanciation de la classe du service
                Operations op = new Operations();
                i=Int32.Parse(this.TxtBEntier1.Text);
                j=Int32.Parse(this.TxtBEntier2.Text);
                // Appel de la méthode du service
                s=op.Division(i,j);
                this.LbResultat.Text="Le résultat de la division est : "+s.ToString();
        }
        catch(SoapException)
        {
                MessageBox.Show("Une erreur s'est produite au niveau du serveur","Erreur");
        }
}
```

Une division par zéro engendre l'affichage d'une boîte de message indiquant l'occurrence d'une erreur.

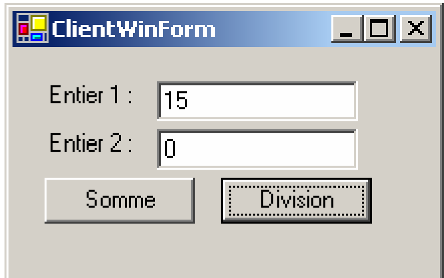


Figure 7 : Division par zéro

Figure 8 : Message d'erreur affiché

Traitement de l'exception au niveau du serveur

Il est également possible de traiter l'erreur au niveau du serveur dans le service Web afin de modifier le message de l'exception. Le code suivant montre une nouvelle version de la méthode *Division* du service Web *Operations* qui traite l'exception levée par la division par zéro.

```
[WebMethod(Description = "Cette méthode effectue la division du premier paramètre par le
second")]
public int Division(int a, int b)
{
    try
    { return a / b; }
    catch (System.DivideByZeroException e)
    {
     throw new System.DivideByZeroException("Attention : La division par zéro est interdite");
    }
}
```

La méthode *Division* du service Web lève une exception de type *System.DivideByZeroException*. Cette exception sera remontée par SOAP jusqu'au client mais toujours sous forme d'une exception de type *SoapException*.

© Karim Kalti

Le code suivant montre une nouvelle version de la fonction de traitement de l'événement *clic* du bouton *Division* qui traite l'exception et affiche le message d'erreur provenant du serveur.

```
private void BtnDivision_Click(object sender, System.EventArgs e)
{
        try
        {
                int i,j,s;
                // Instanciation de la classe du service
                Operations op = new Operations();
                i=Int32.Parse(this.TxtBEntier1.Text);
                j=Int32.Parse(this.TxtBEntier2.Text);
                // Appel de la méthode du service
                s=op.Division(i,j);
                this.LbResultat.Text="Le résultat de la division est : "+s.ToString();
        }
        catch(SoapException ex)
        {
                MessageBox.Show(ex.Message,"Erreur");
        }
}
```

Ceci est le message d'erreur affichée dans le cas d'une division par zéro.

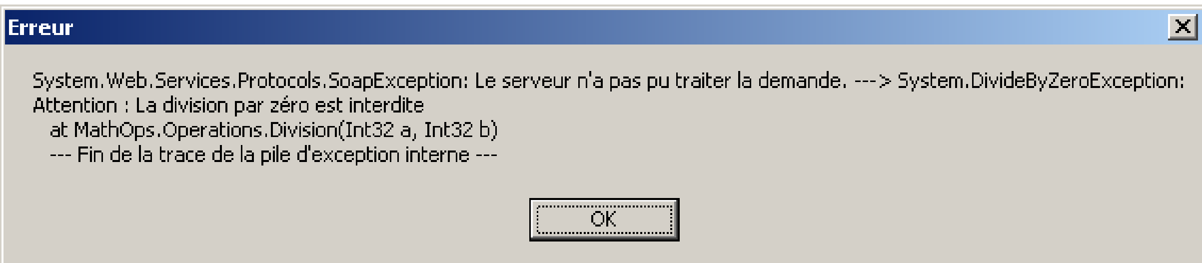


Figure 9 : Nouveau message d'erreur affiché

Communication asynchrone avec les services Web

- Dans un contexte Client/Serveur l'appel d'une méthode de service Web peut avoir une durée importante. Ceci peut être dû à des raisons diverses :
  - la nature même des traitements qu'effectue la méthode nécessite un temps d'exécution important comme par exemple les méthodes qui font des accès à des bases de données de très grande taille.
  - Le serveur sur lequel est hébergé le service peut être encombré s'il est sollicité par un très grand nombre de clients et surtout lorsqu'il héberge plusieurs applications différentes.
  - Le réseau peut être encombré s'il gère un trafic très important.

- Pour éviter les blocages dans les applications clientes lors de l'utilisation des services Web, il est plus intéressant d'effectuer des appels asynchrones aux méthodes de ces services.

- Un appel asynchrone est un appel qui lance l'exécution d'une méthode sans attendre un renvoi immédiat d'une valeur de retour. L'appel asynchrone se déroule en deux étapes :
  - Le client initialise l'appel de la méthode. Cette action étant rapide et non bloquante, l'application cliente peut alors effectuer d'autres actions pendant que le déroulement de l'exécution de la méthode.
  - La deuxième étape se produit au gré de l'application cliente qui décide d'aller chercher le résultat si celui-ci est disponible.

- La classe *proxy* générée par WSDL propose pour chaque méthode du service Web deux méthodes supplémentaires permettant d'effectuer l'appel asynchrone : *BeginNomMéthode* et *EndNomMéthode*.
  - *BeginNomMéthode* permet d'initialiser l'appel asynchrone d'une méthode.
  - *EndNomMéthode* permet de récupérer le résultat de l'appel.

© Karim Kalti

L'exemple suivant montre un appel asynchrone à la méthode *Somme* dans le gestionnaire de l'événement clic sur le bouton *Ajouter* dans le client de type application Windows.

```
// Ajout d'attributs à la classe
int s;
Operations op;

// Gestionnaire de l'événement clic sur le bouton Somme
private void BtnSomme_Click(object sender, System.EventArgs e)
{
	int i,j,s;
	// Instanciation de la classe du service
	op = new Operations();
	if(this.TxtBEntier1.Text!="" && this.TxtBEntier2.Text!="")
	{
		i=Int32.Parse(this.TxtBEntier1.Text);
		j=Int32.Parse(this.TxtBEntier2.Text);
		// Appel de la méthode du service
		op.BeginSomme(i,j, new AsyncCallback(FinSomme), null);
		this.LbResultat.Text="Le résultat est : "+s.ToString();
	}
}
// Fonction appelée lorsque le service Web retourne sa valeur
public void FinSomme(IAsyncResult ar)
{
	if(ar.IsCompleted)
		s = (int)op.EndSomme(ar);
	op = null;
}
```

- La méthode *BeginNomMéthode* (*BeginSomme* dans l'exemple courant) prend comme premiers arguments la liste des arguments de la méthode du service Web. L'avant dernier argument et un délégué de type *AsyncCallback* qui est appelé lorsque l'appel asynchrone est terminé. Si cet argument prend comme valeur *null* alors aucun délégué n'est appelé. Le dernier argument de type *object* représente des informations supplémentaires fournies par l'appelant.

- La méthode *BeginNomMéthode* renvoie une valeur de type *IAsyncResult*. Cette classe permet de donner l'état de l'opération asynchrone notamment à travers sa propriété *IsCompleted*.

© Karim Kalti